SEMESTER EXAM SERIES

COMPILER

IN 7 HOURS + NOTES

Video chapters

Chapter-1 (INTRODUCTION TO COMPILER): Phases and passes, Bootstrapping, Finite state machines and regular expressions and their applications to lexical analysis, Optimization of DFA-Based Pattern Matchers implementation of lexical analyzers, lexical-analyzer generator, LEX compiler, Formal grammars and their application to syntax analysis, BNF notation, ambiguity, YACC. The syntactic specification of programming languages: Context free grammars, derivation and parse trees, capabilities of CFG.

Chapter-2 (BASIC PARSING TECHNIQUES): Parsers, Shift reduce parsing, operator precedence parsing, top down parsing, predictive parsers Automatic Construction of efficient Parsers: LR parsers, the canonical Collection of LR(0) items, constructing SLR parsing tables, constructing Canonical LR parsing tables, Constructing LALR parsing tables, using ambiguous grammars, an automatic parser generator, implementation of LR parsing tables.

Chapter-3 (SYNTAX-DIRECTED TRANSLATION): Syntax-directed Translation schemes, Implementation of Syntax- directed Translators, Intermediate code, postfix notation, Parse trees & syntax trees, three address code, quadruple & triples, translation of assignment statements, Boolean expressions, statements that alter the flow of control, postfix translation, translation with a top down parser. More about translation: Array references in arithmetic expressions, procedures call, declarations and case statements.

Chapter-4 (SYMBOL TABLES): Data structure for symbols tables, representing scope information. Run-Time Administration: Implementation of simple stack allocation scheme, storage allocation in block structured language. Error Detection & Recovery: Lexical Phase errors, syntactic phase errors semantic errors.

Chapter-5 (CODE GENERATION): Design Issues, the Target Language. Addresses in the Target Code, Basic Blocks and Flow Graphs, Optimization of Basic Blocks, Code Generator. Code optimization: Machine-Independent Optimizations, Loop optimization, DAG representation of basic blocks, value numbers and algebraic laws, Global Data-Flow analysis.

Chapter-1

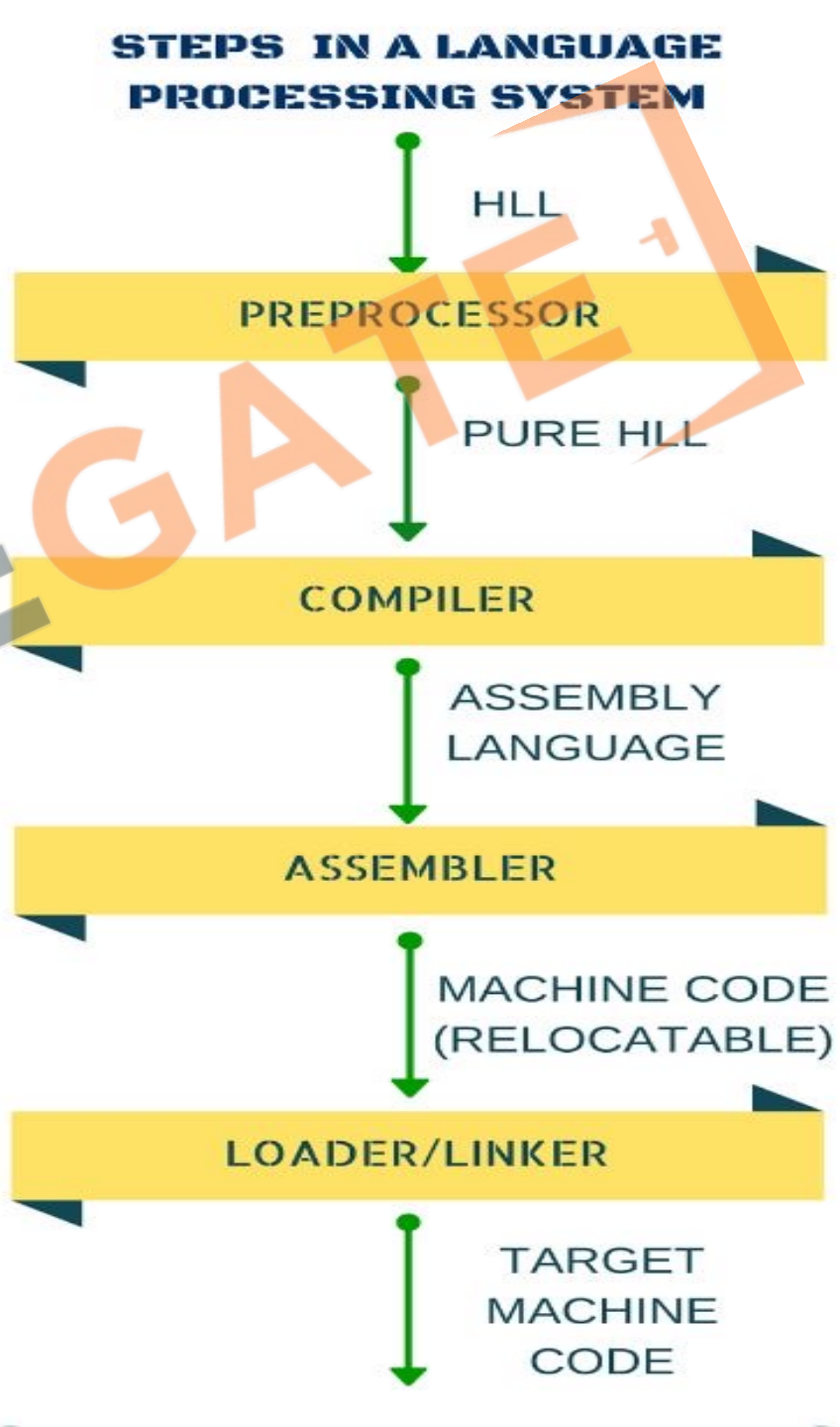
(INTRODUCTION TO COMPILER): Phases and passes, Bootstrapping, Finite state machines and regular expressions and their applications to lexical analysis, Optimization of DFA-Based Pattern Matchers implementation of lexical analyzers, lexical-analyzer generator, LEX compiler, Formal grammars and their application to syntax analysis, BNF notation, ambiguity, YACC. The syntactic specification of programming languages: Context free grammars, derivation and parse trees, capabilities of CFG.

Language Processing System

- We tend to write programs in high-level language, that is much less complicated for us to comprehend and maintain in thoughts. These programs go through a series of transformation so that they can readily be used in machines. This is where language processing systems come handy.

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



```c
#include <stdio.h>
#include <unistd.h>

int main ()
{
        printf("Running 'net join' with the following parameters: \n");
        char *domain="mydomain";
        char *user="domainjoinuser";
        char *pass="mypassword";
        char *vastool="/opt/quest/bin/vastool";
        char *ou="OU=test,DC=mtdomian,DC=local";
        char unjoin[512];

        sprintf(unjoin,"/opt/quest/bin/vastool -u %s -w '%s' unjoin -f",user,pass);

        printf("Domain: %s\n",domain);
        printf("User: %s\n",user);
        printf("-----------------\n");

        printf("\nUnjoin.....\n");
        system(unjoin);

        printf("\nJoin.....\n");
        execl("/opt/quest/bin/vastool", "vastool", "-u", user, "-w", pass, "join", "-c", ou,
"-f", domain, (char*)0);

}
```

Macros may be nested

- in definitions, e.g.:

```
#define  Pi          3.1416
#define  Twice_Pi    2*Pi
```

- in uses, e.g.:

```
#define  double(x)   x+x
#define  Pi  3.1416
...
if ( x > double(Pi) ) ...
```

Simple assignment operator	Shorthand operator
a = a+1	a + =1
a = a-1	a - =1
a = a* (m+n)	a * = m+n
a = a / (m+n)	a / = m+n
a = a %b	a %=b

Pre-Processor

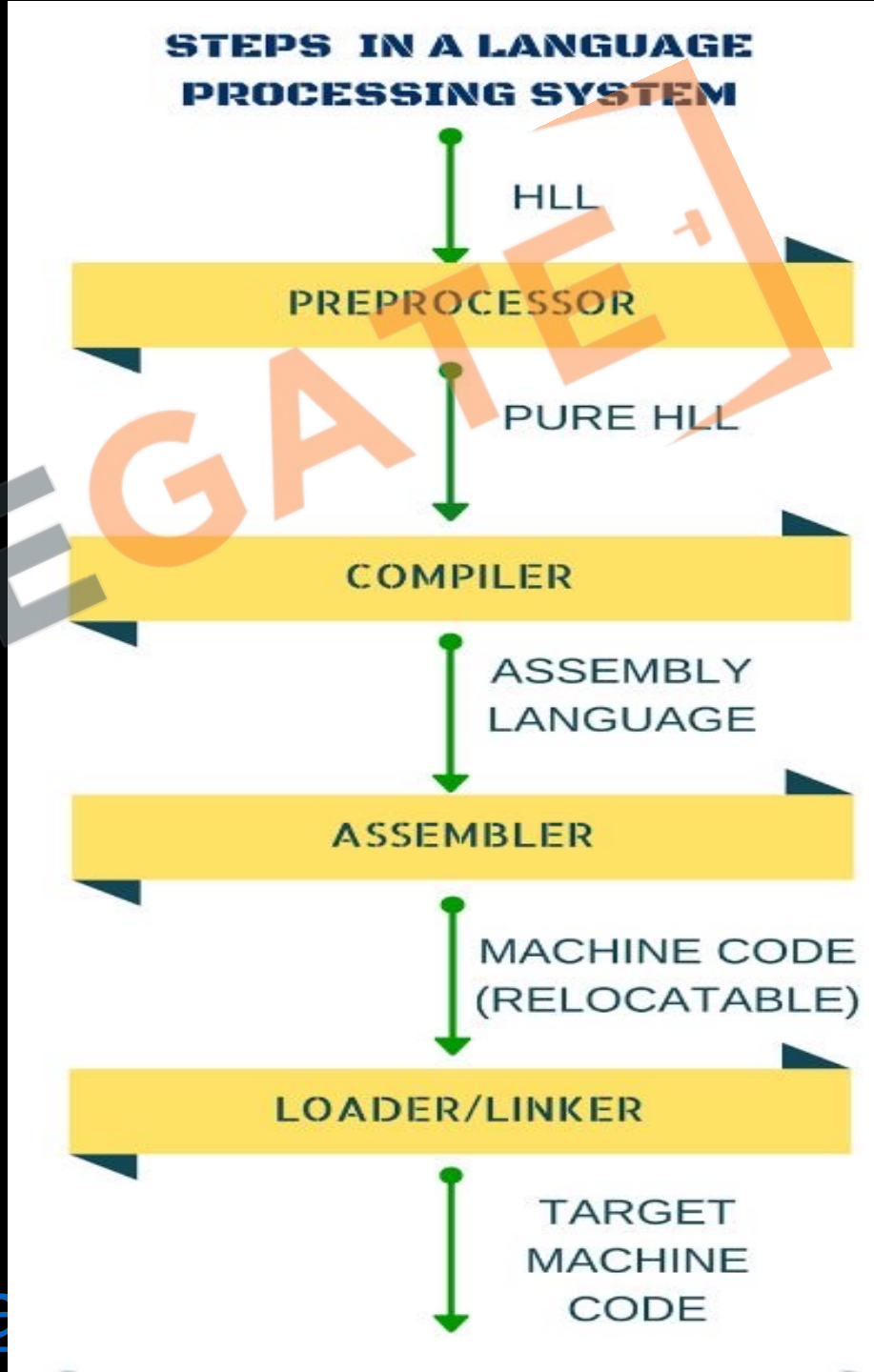
- The pre-processor removes all the #include directives by including the files called file inclusion.
- All the #define directives using macro expansion. It performs file inclusion, macro-processing, short hand operators etc.

Pure High-Level Language

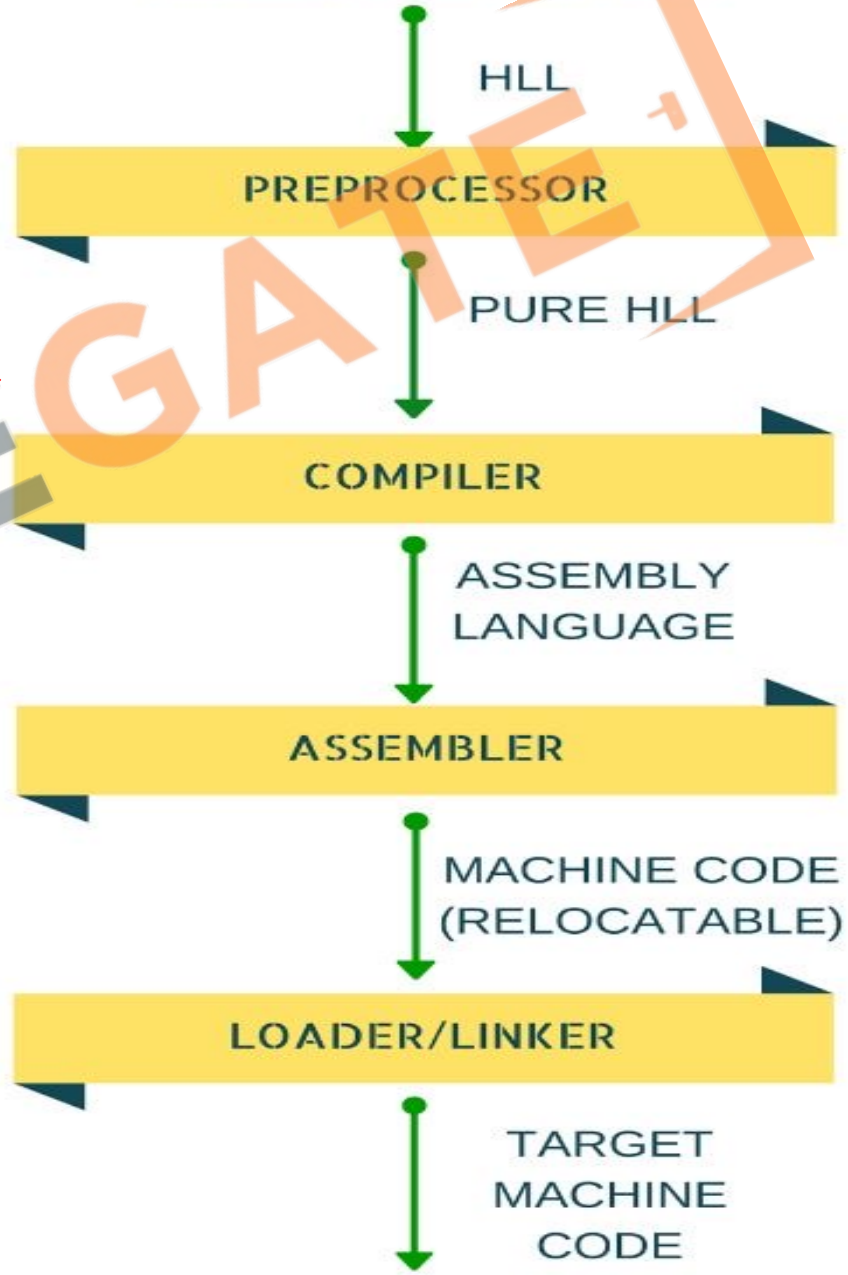
- That HLL which can be directly understood by the compiler.

डालडा

देसी

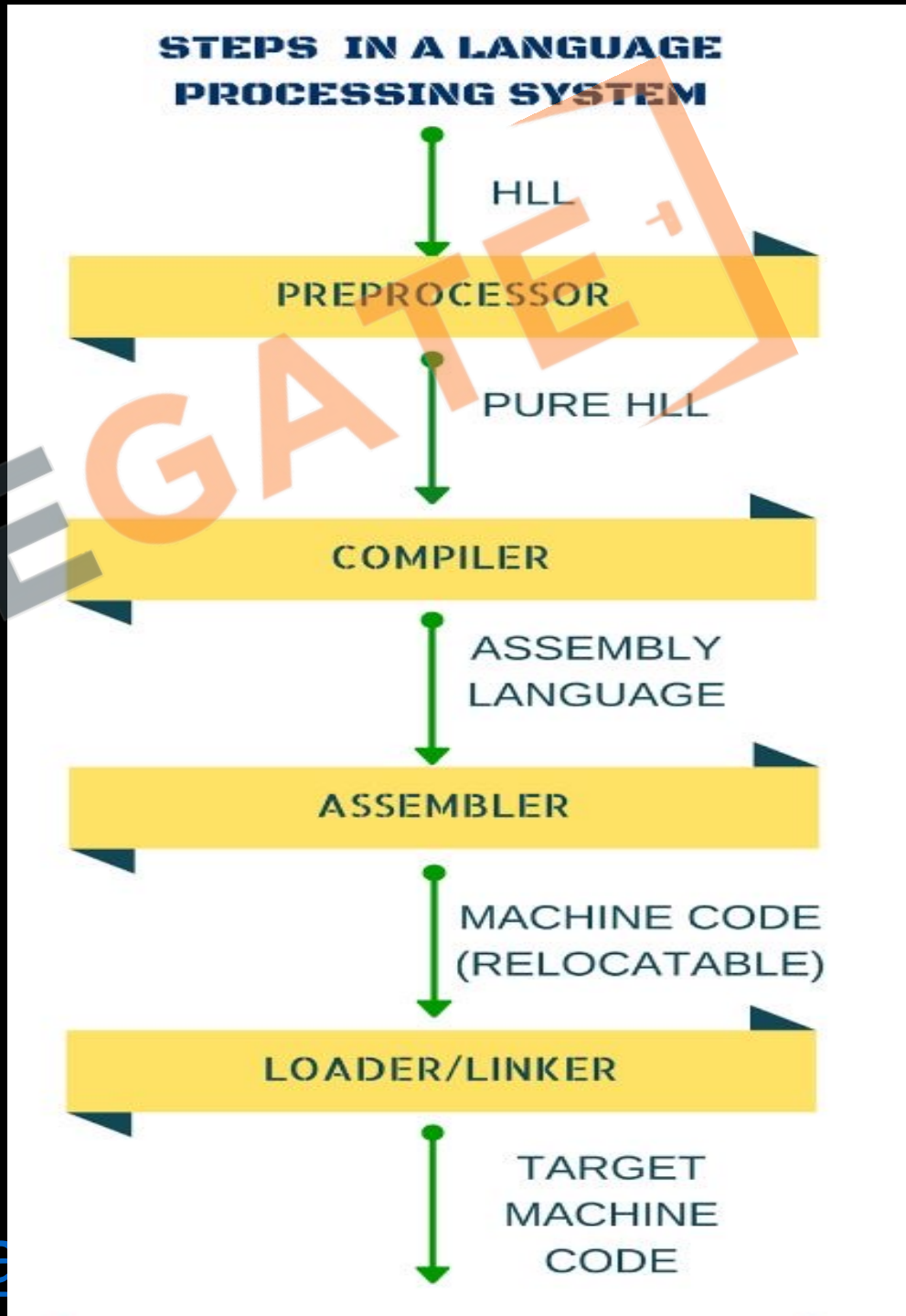


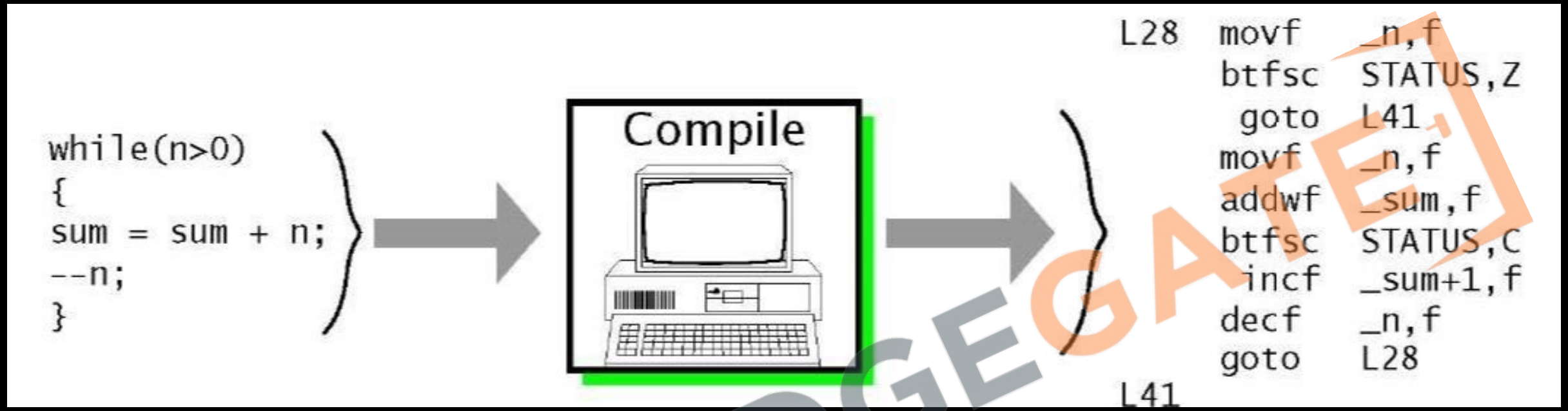
STEPS IN A LANGUAGE PROCESSING SYSTEM



Compiler

- A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language).
- The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language) to create an executable program.
- The reason is we are not comfortable in writing a low-level language there for we write a code which is easy and then convert it into low level language.

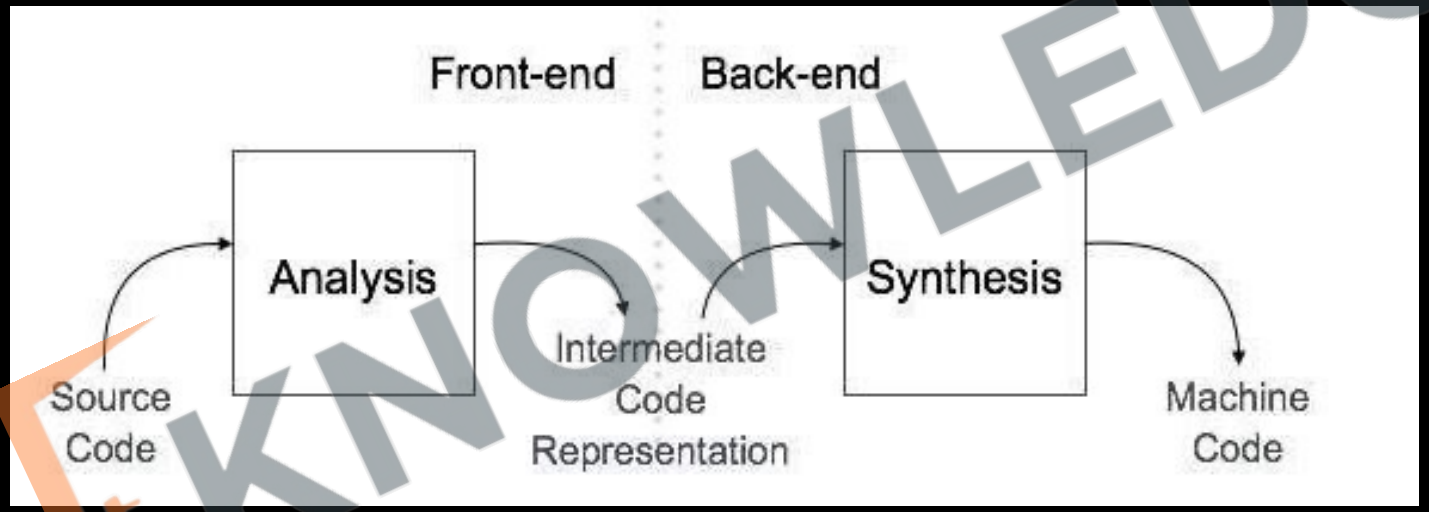


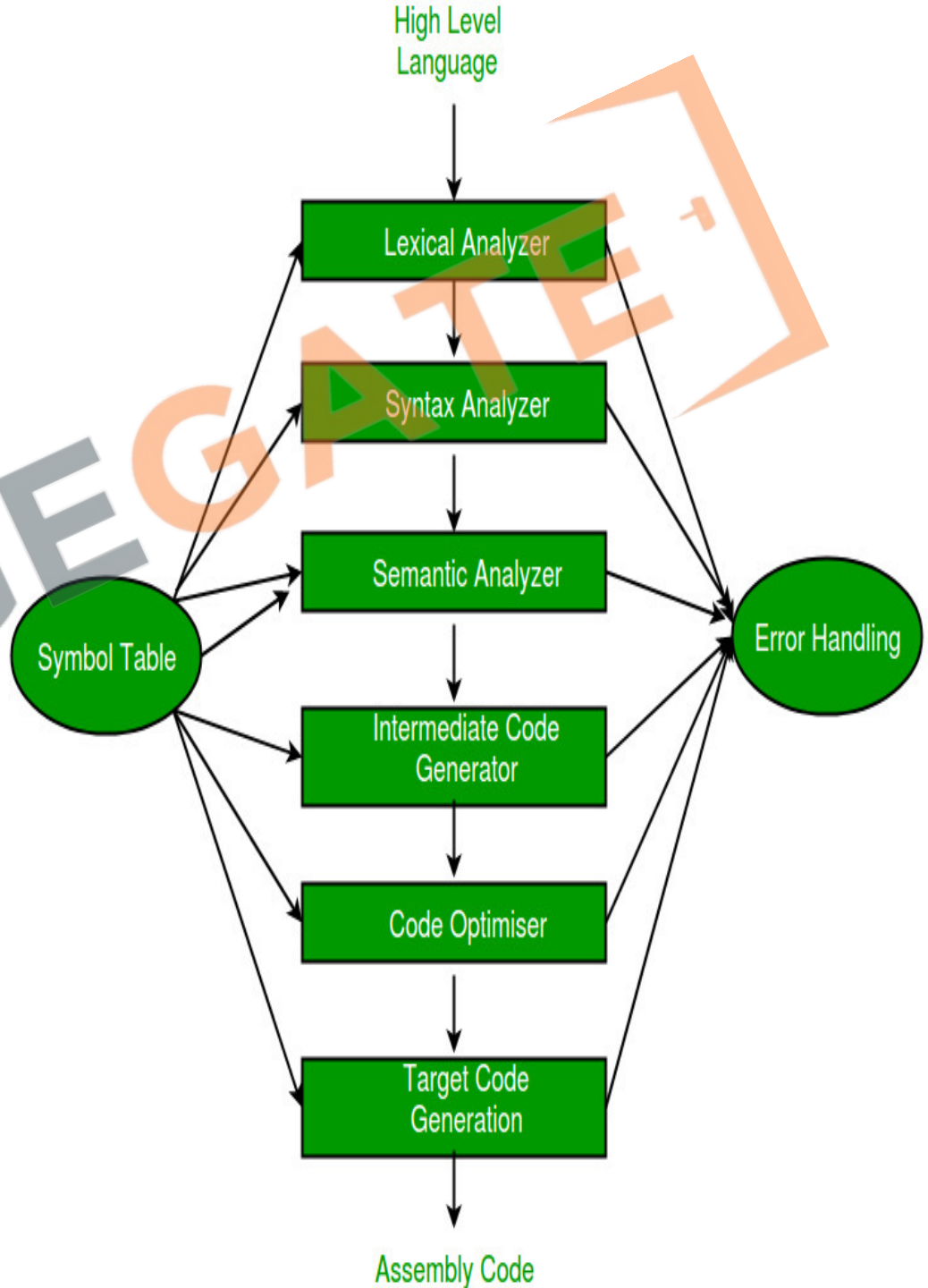


http://www.knowledgegate.in/gate

Compiler Design

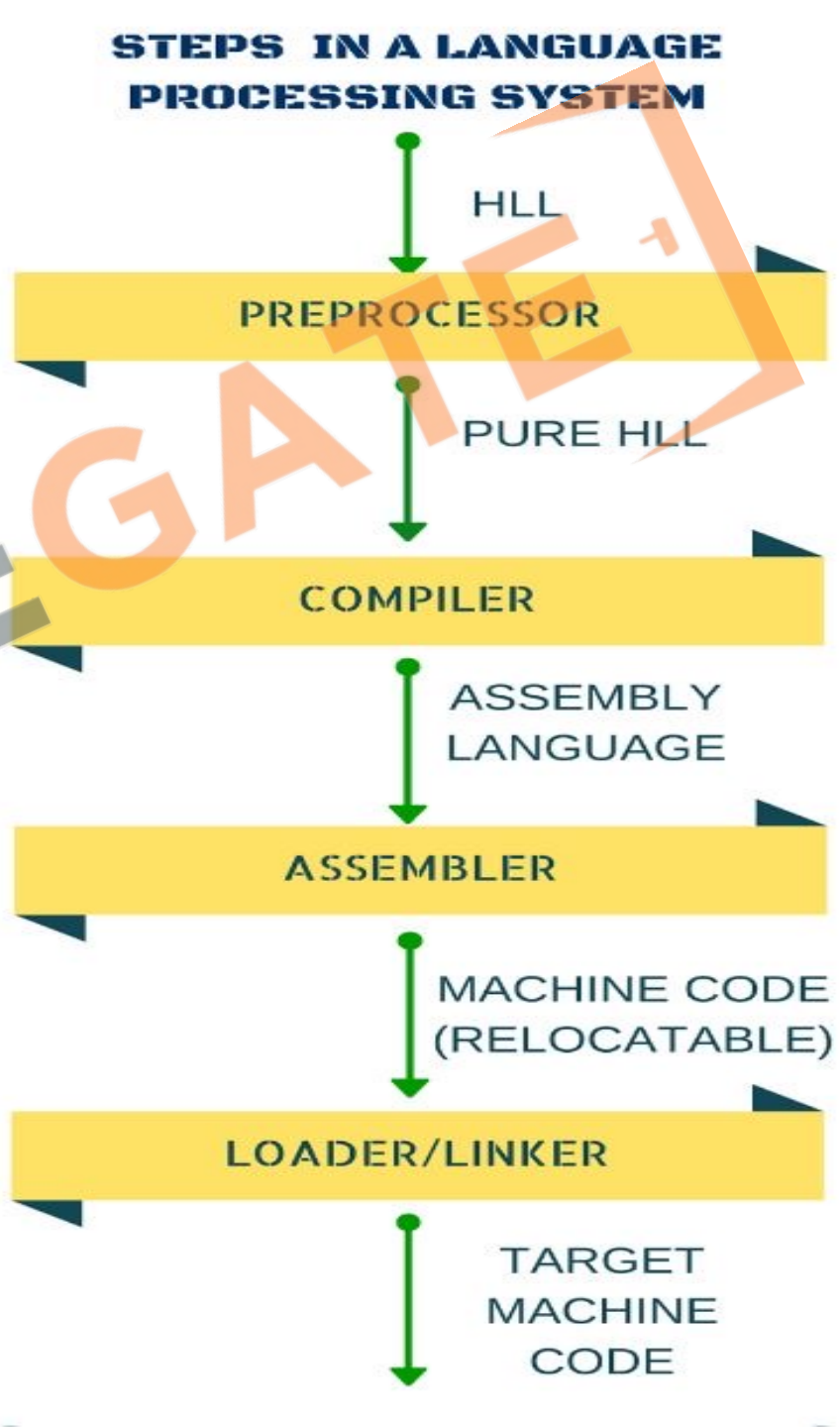
- We basically have two phases of compilers, namely Analysis phase and Synthesis phase.
  - **Analysis phase** creates an intermediate representation from the given source code.
  - **Synthesis phase** creates an equivalent target program from the intermediate representation.





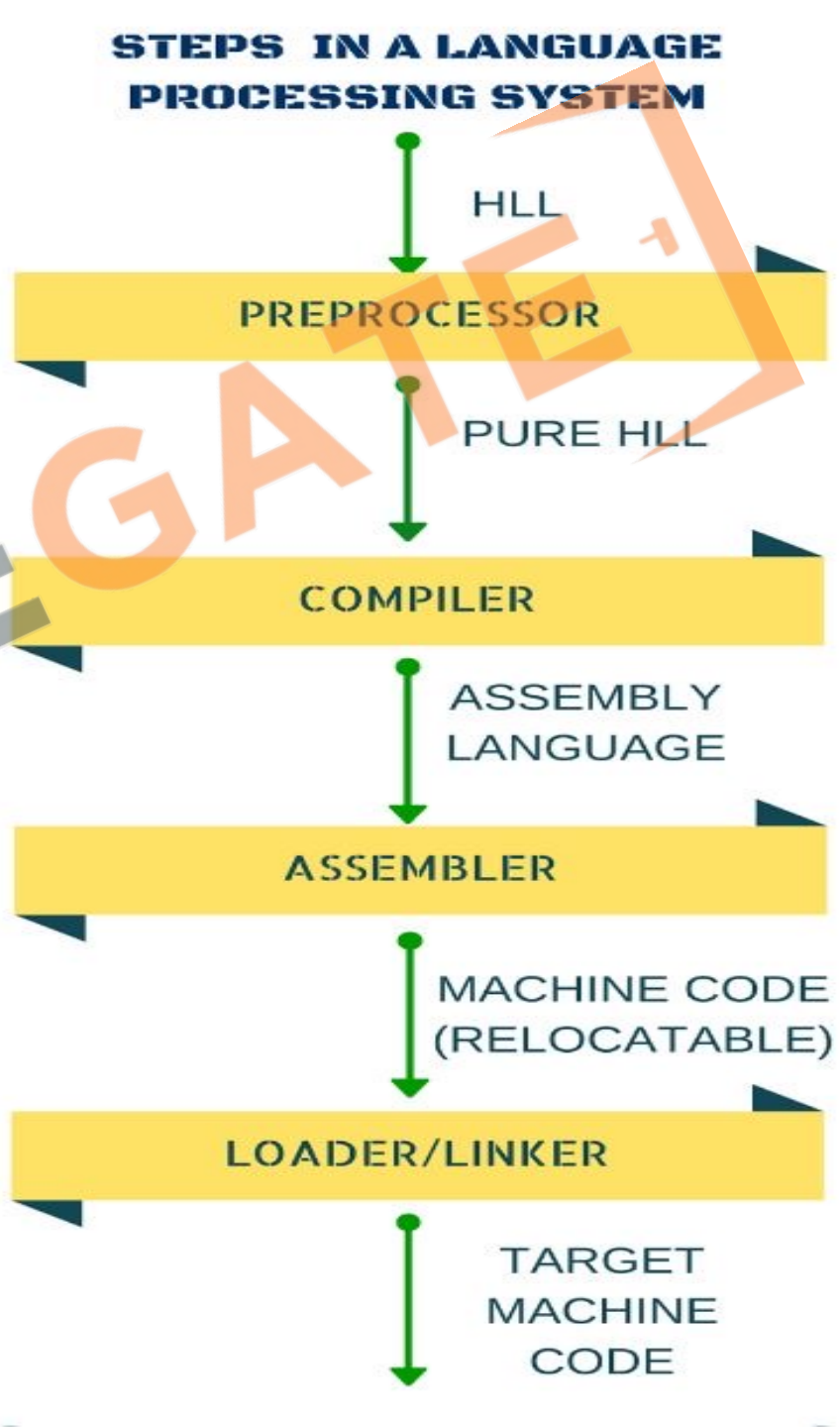
Assembly Language

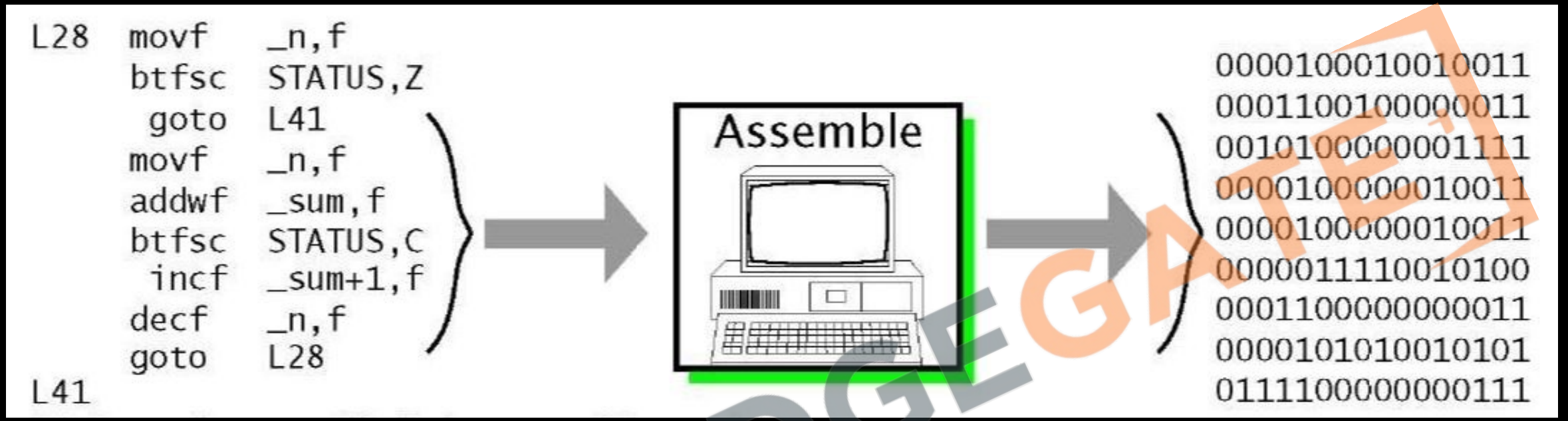
- Its neither in binary form nor high level.
- It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.



Assembler

- For every platform (Hardware + OS) we will have an assembler. They are not universal since for each platform we have one.
- The output of assembler is called object file. It translates assembly language to machine code.



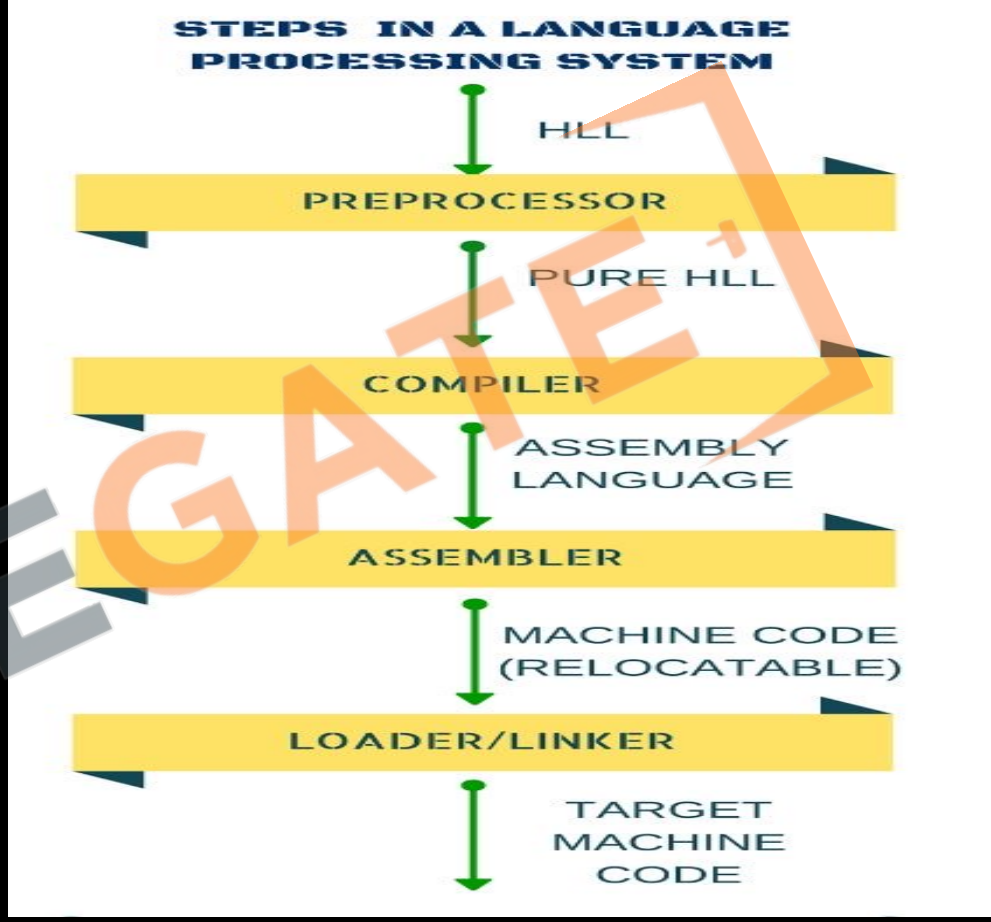


8085 Microprocessor

http://www.knowledgegate.in/gate

Relocatable Machine Code

- It can be loaded at any point and can be run.
- The address within the program will be in such a way that it will cooperate for the program movement.



if(a<b) then t=1
else t = 0

i) if (a<b) goto (i+3)
i+1) t=0
i+2) goto (i+4)
i+3) t=1
i+4) exit

while(C) do S

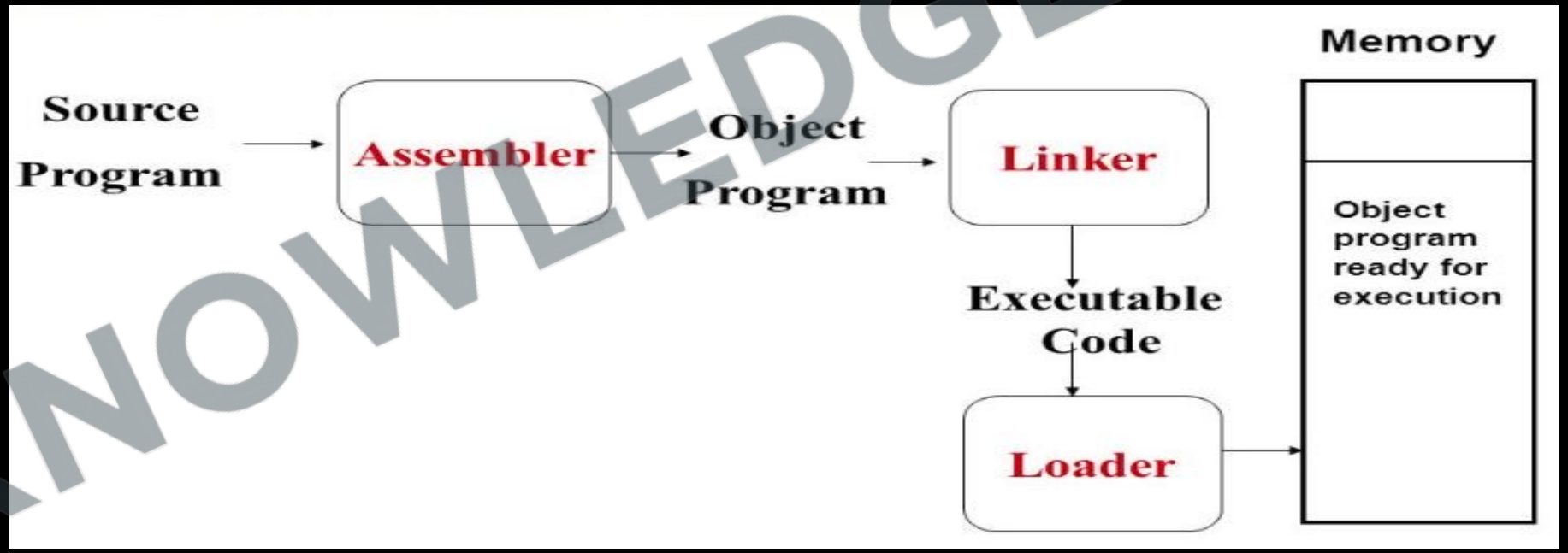
i) if (E) goto i+2
i+1) goto i+4
i+2) S
i+3) goto i
i+4) exit

```
for(1=0; i<10 ; i++)
    s
```

i) i = 0
i+1) if(i<10) goto i+3
i+2) goto i+6
i+3) S
i+4) i = i + 1
i+5) goto i+1
i+6) exit

Linker → Loader

- It converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message.
- Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.



- The first practical compiler was written by Corrado Böhm, in 1951, for his PhD thesis.

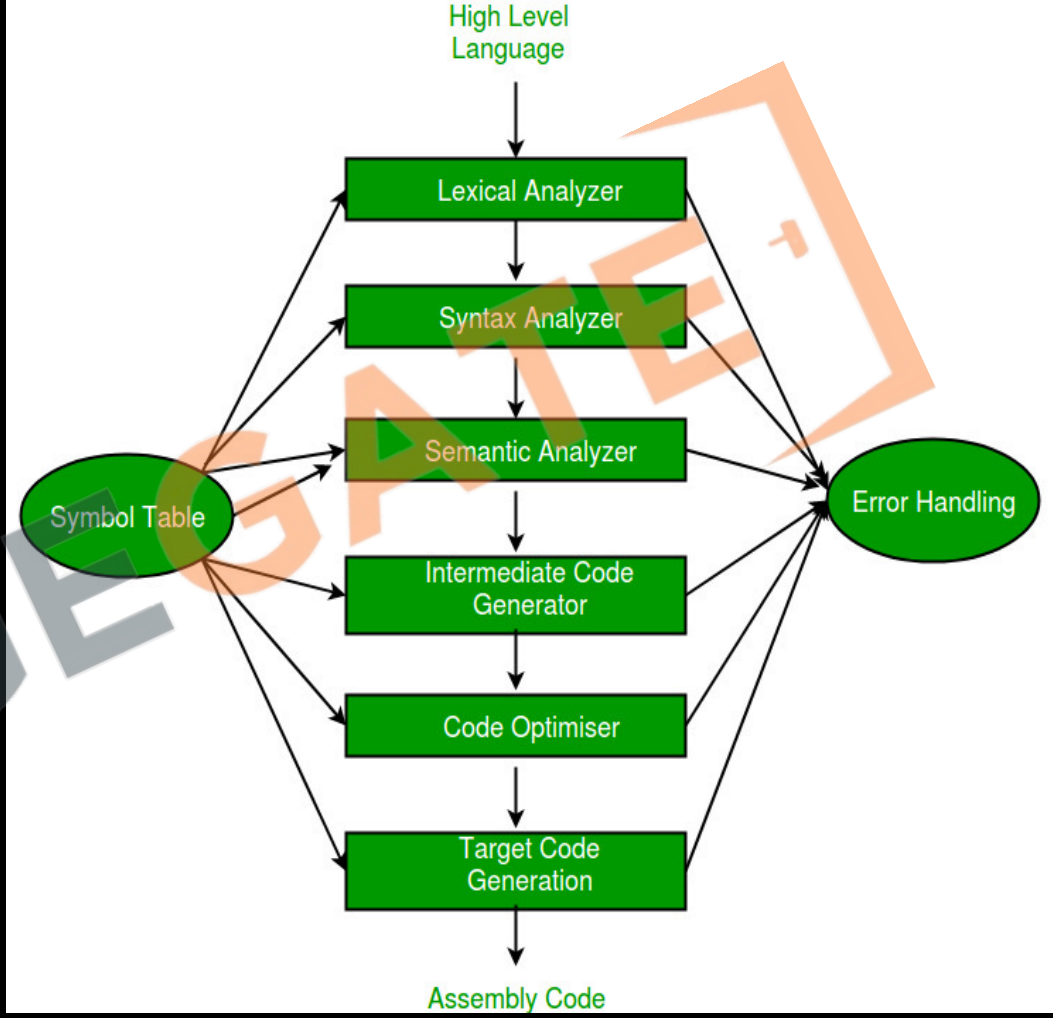
- The first implemented compiler was written by Grace Hopper, who also coined the term "compiler", referring to her A-0 system which functioned as a loader or linker, not the modern notion of a compiler.

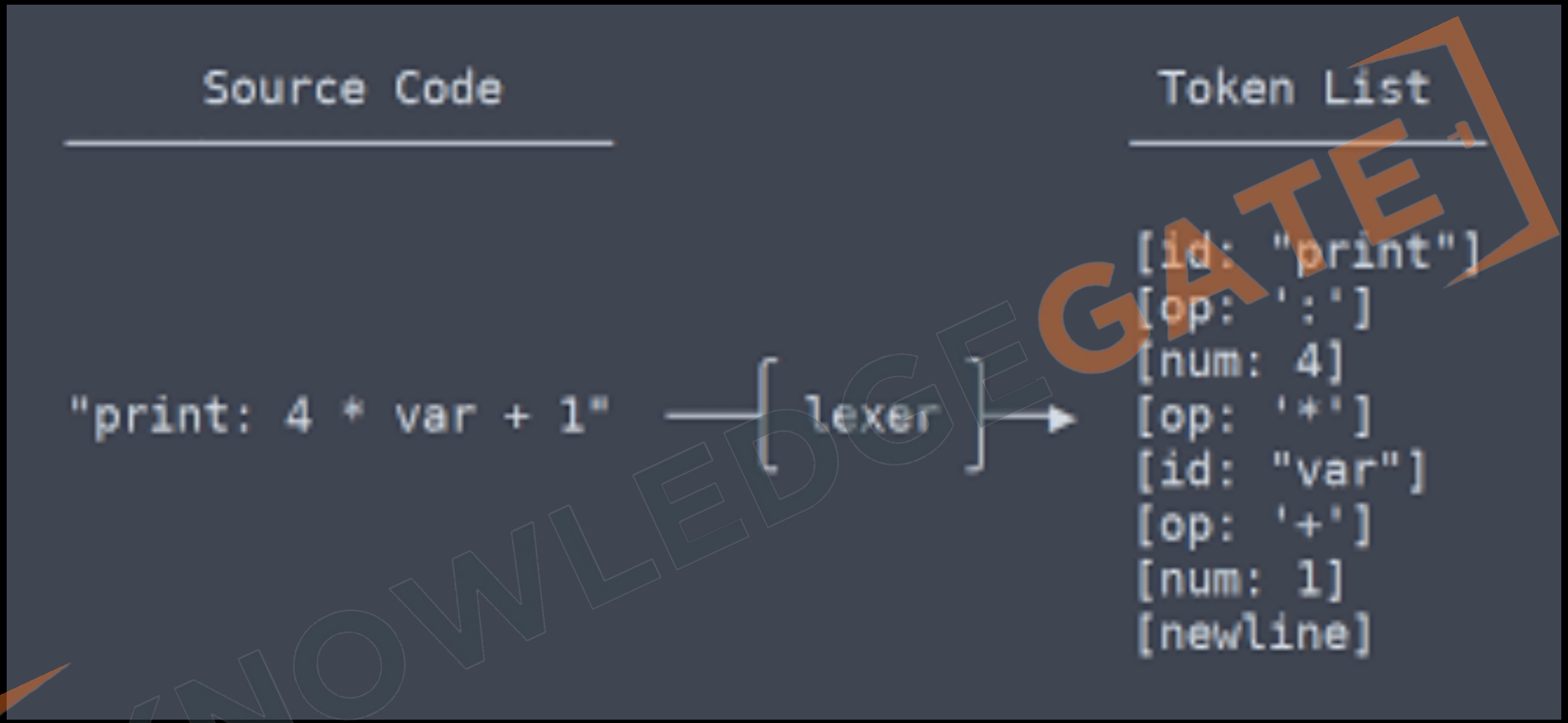
- The first Autocode and compiler in the modern sense were developed by Alick Glennie in 1952 at the University of Manchester for the Mark 1 computer.

- The FORTRAN team led by John W. Backus at IBM introduced the first commercially available compiler, in 1957, which took 18 person-years to create.

Lexical Analyzer

- It reads the program and converts it into Lexemes.
- A stream of lexemes into a stream of tokens.
- Tokens are defined by regular expressions which are understood by the lexical analyser. It also removes white-spaces and comments.



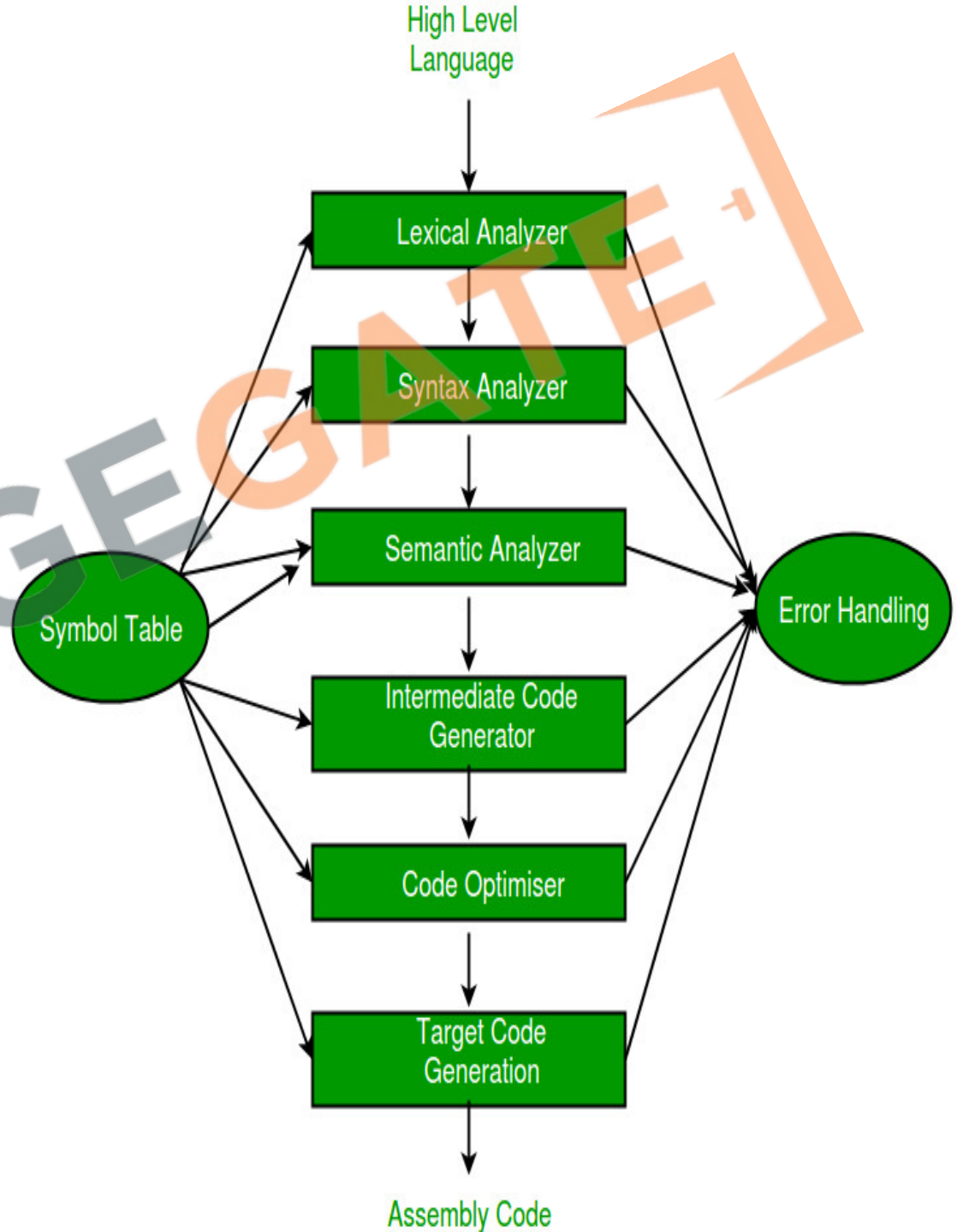


http://www.knowledgegate.in/gate

Float x, y, z;
x = y+ z*60

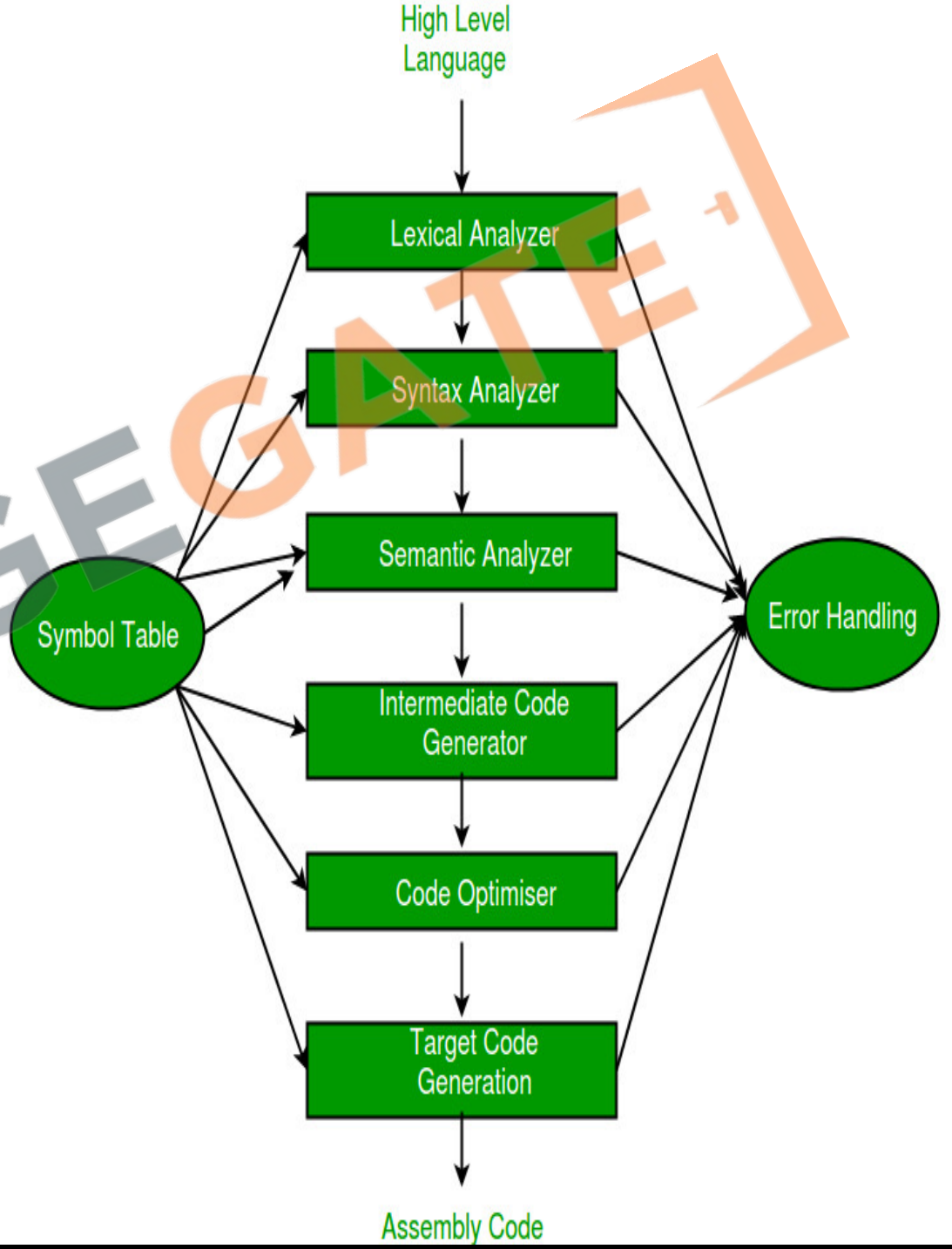
X → token → identifier
= → token → operator
Y → token → identifier
+ → token → operator
Z → token → identifier
* → token → operator
60 → token → constant

$id_1 = id_2 + id_3 * 60$



Syntax Analyzer

- It is sometimes called as parser.
- It constructs the Parse/Syntax tree.
- Uses productions of Context Free Grammar to construct the parse tree.
- It reads all the tokens one by one.



S → id = E

E → E + T / T

T → T * F / F

F → id / num

$id_1 = id_2 + id_3 * 60$

1. *program* → *declarationList*
2. *declarationList* → *declarationList declaration* | *declaration*
3. *declaration* → *varDeclaration* | *funDeclaration*

---

4. *varDeclaration* → *typeSpecifier varDeclList* **;**
5. *scopedVarDeclaration* → *scopedTypeSpecifier varDeclList* **;**
6. *varDeclList* → *varDeclList* **,** *varDeclInitialize* | *varDeclInitialize*
7. *varDeclInitialize* → *varDeclId* | *varDeclId* **:** *simpleExpression*
8. *varDeclId* → **ID** | **ID [NUMCONST]**
9. *scopedTypeSpecifier* → **static** *typeSpecifier* | *typeSpecifier*
10. *typeSpecifier* → **int** | **bool** | **char**

---

11. *funDeclaration* → *typeSpecifier* **ID** **(** *params* **)** *statement* | **ID** **(** *params* **)** *statement*
12. *params* → *paramList* | ϵ
13. *paramList* → *paramList* **;** *paramTypeList* | *paramTypeList*
14. *paramTypeList* → *typeSpecifier paramIdList*

15. *paramIdList* → *paramIdList* , *paramId* | *paramId*

16. *paramId* → **ID** | **ID** [ ]

---

17. *statement* → *expressionStmt* | *compoundStmt* | *selectionStmt* | *iterationStmt* | *returnStmt* | *breakStmt*

18. *expressionStmt* → *expression* ; | ;

19. *compoundStmt* → { *localDeclarations statementList* }

20. *localDeclarations* → *localDeclarations scopedVarDeclaration* | ϵ

21. *statementList* → *statementList statement* | ϵ

22. *elsifList* → *elsifList* **elsif** *simpleExpression* **then** *statement* | ϵ

23. *selectionStmt* → **if** *simpleExpression* **then** *statement elsifList* | **if** *simpleExpression* **then** *statement elsifList* **else** *statement*

24. *iterationRange* → **ID** = *simpleExpression* .. *simpleExpression* | **ID** = *simpleExpression* .. *simpleExpression* : *simpleExpression*

25. *iterationStmt* → **while** *simpleExpression* **do** *statement* | **loop forever** *statement* | **loop** *iterationRange* **do** *statement*

26. *returnStmt* → **return** ; | **return** *expression* ;

27. *breakStmt* → **break** ;

28. *expression* → *mutable* = *expression* | *mutable* += *expression* | *mutable* −= *expression* | *mutable* *= *expression* | *mutable* /= *expression* | *mutable* ++ | *mutable* −− | *simpleExpression*

29. *simpleExpression* → *simpleExpression* **or** *andExpression* | *andExpression*

30. *andExpression* → *andExpression* **and** *unaryRelExpression* | *unaryRelExpression*

31. *unaryRelExpression* → **not** *unaryRelExpression* | *relExpression*

32. *relExpression* → *sumExpression relop sumExpression* | *sumExpression*

33. *relop* → <= | < | > | >= | == | !=

34. *sumExpression* → *sumExpression sumop mulExpression* | *mulExpression*

35. *sumop* → + | −

36. *mulExpression* → *mulExpression mulop unaryExpression* | *unaryExpression*

37. *mulop* → * | / | %

38. *unaryExpression* → *unaryop unaryExpression* | *factor*

39. *unaryop* → − | * | ?

40. *factor* → *immutable* | *mutable*

41. *mutable* → **ID** | *mutable* [*expression*]

42. *immutable* → (*expression*) | *call* | *constant*

43. *call* → **ID** (*args*)

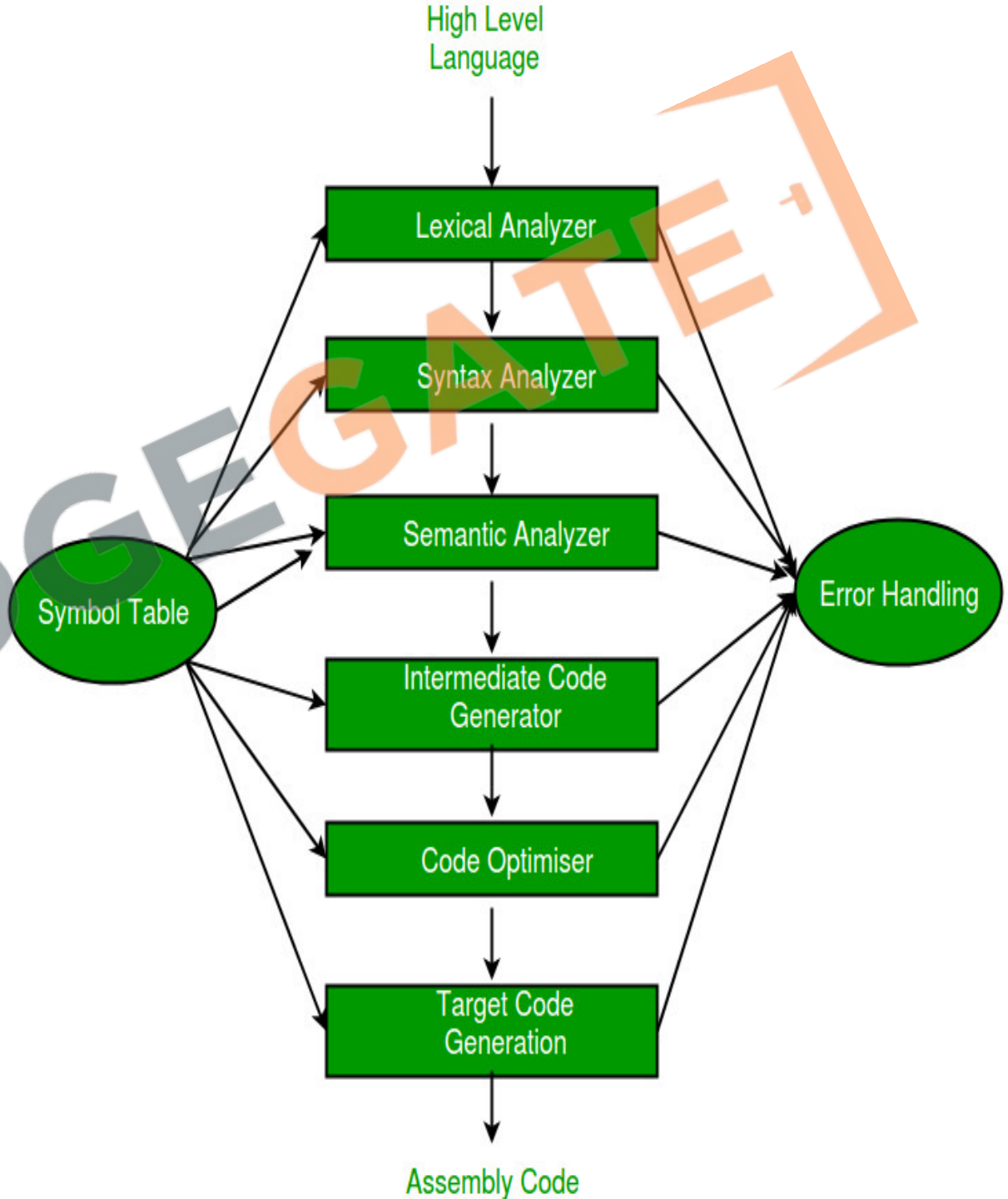
44. *args* → *argList* | **ϵ**

45. *argList* → *argList* , *expression* | *expression*

46. *constant* → **NUMCONST** | **CHARCONST** | **STRINGCONST** | **true** | **false**

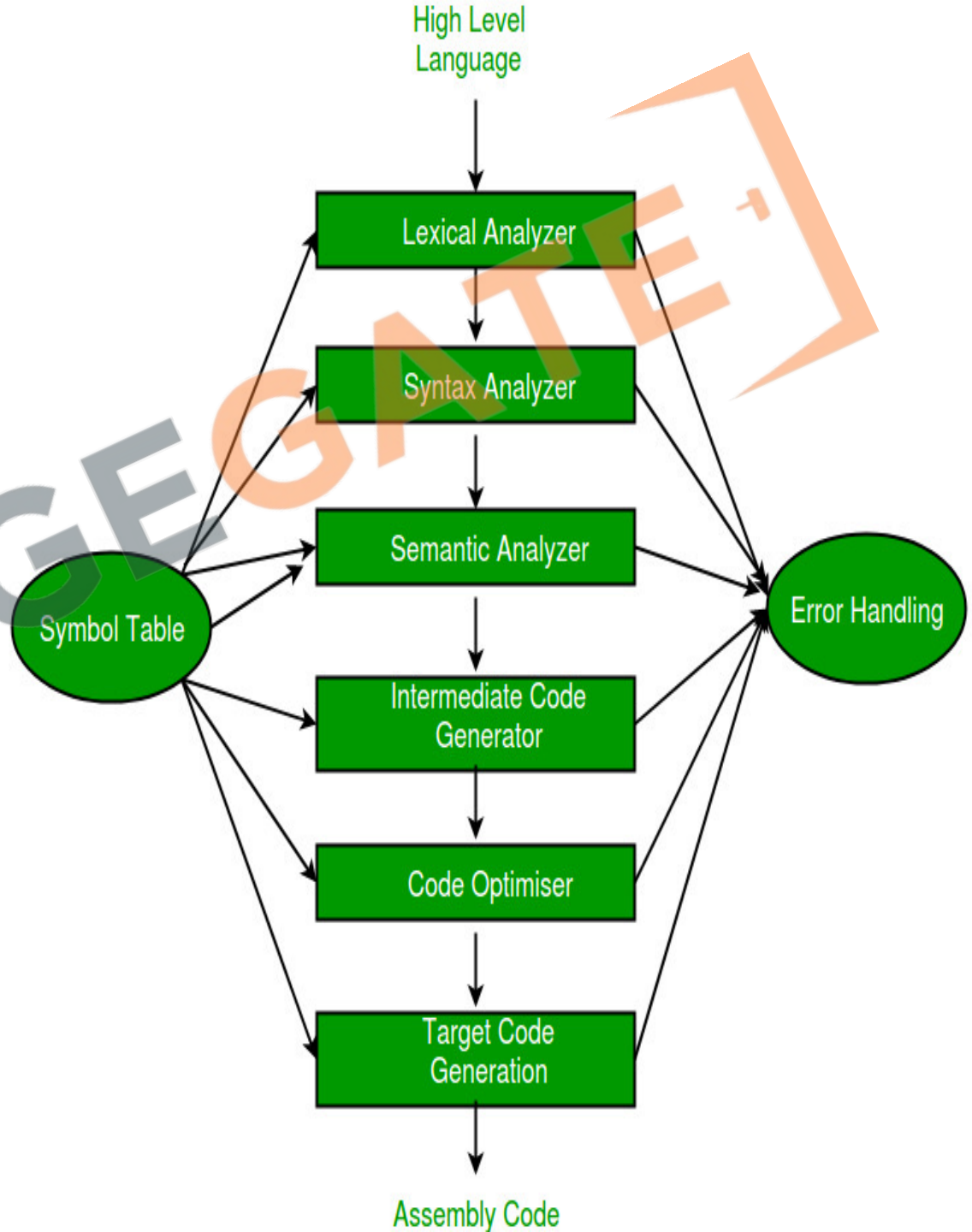
Semantic Analyzer

- It verifies the parse tree, whether it's meaningful or not.
- It furthermore produces a verified / Annotated parse tree.



Intermediate Code Generator

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

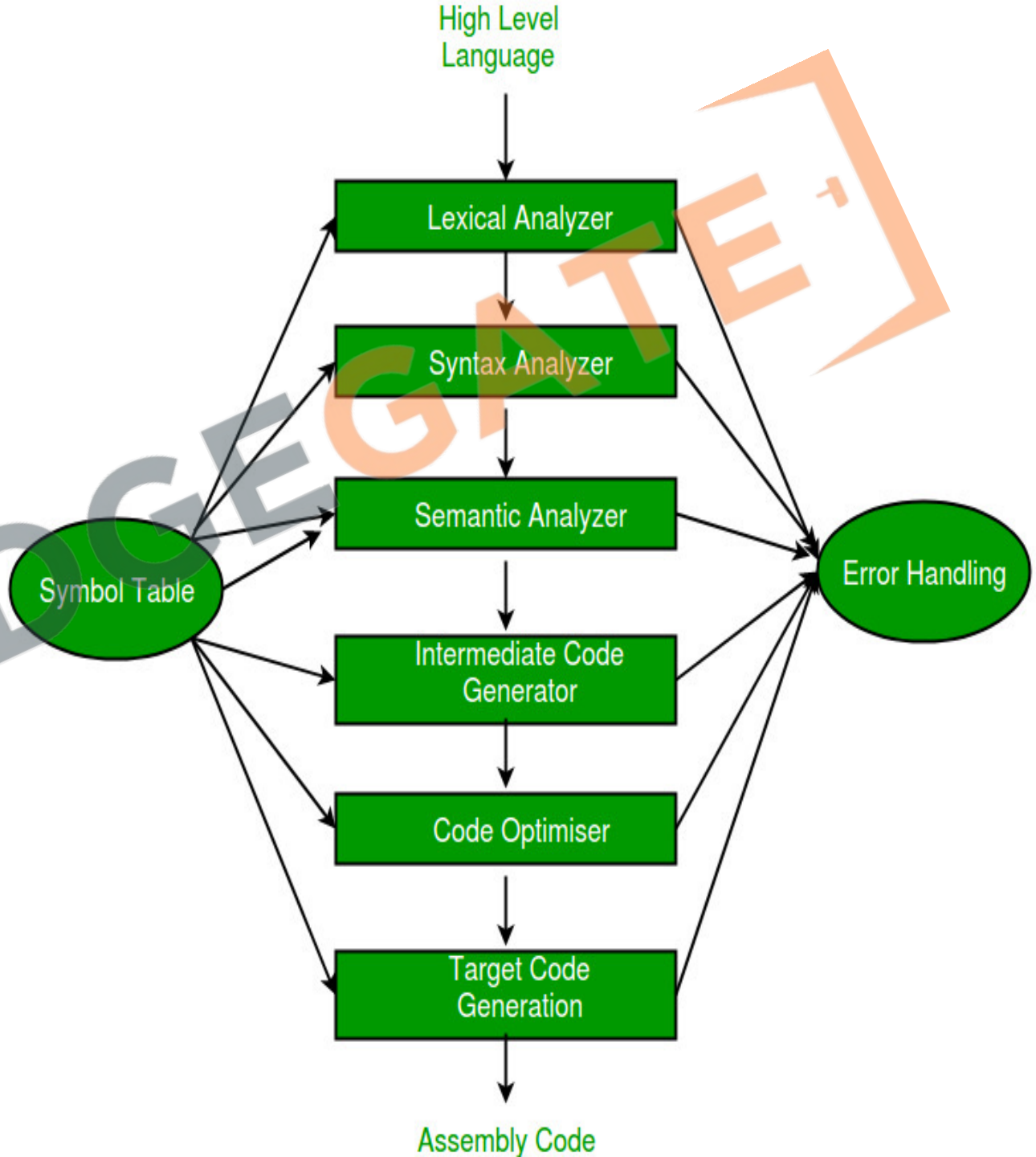


- $x = y + z*60$

  $t_1 = z * 60$

  $t_2 = y + t_1$

  $x = t_2$

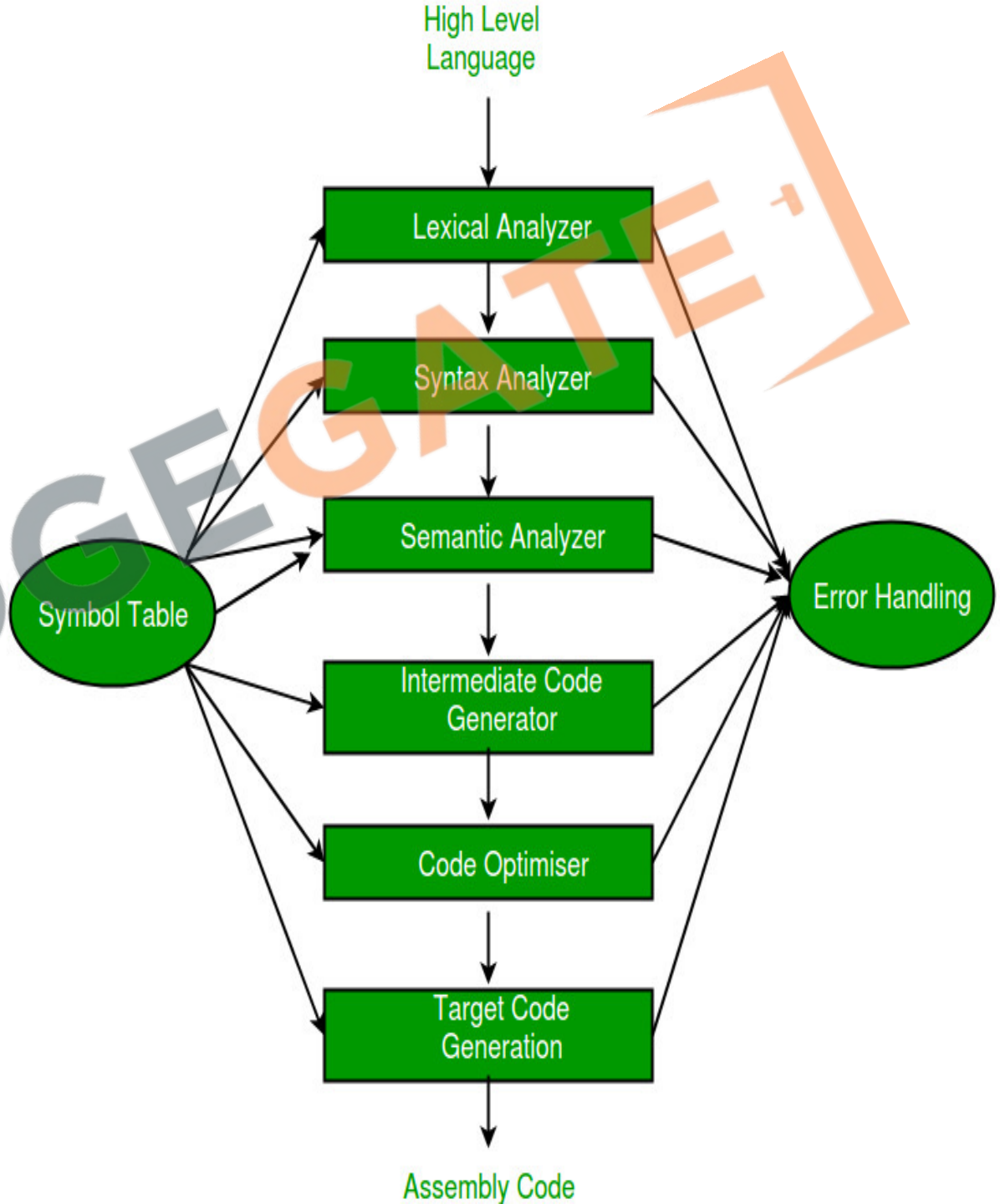


Code Optimizer

- It transforms the code so that it consumes fewer resources and produces more speed.
- The meaning of the code optimizer is code being transformed is not altered.
- Optimisation can be categorized into two types: machine dependent and machine independent.

$t_1 = z * 60$

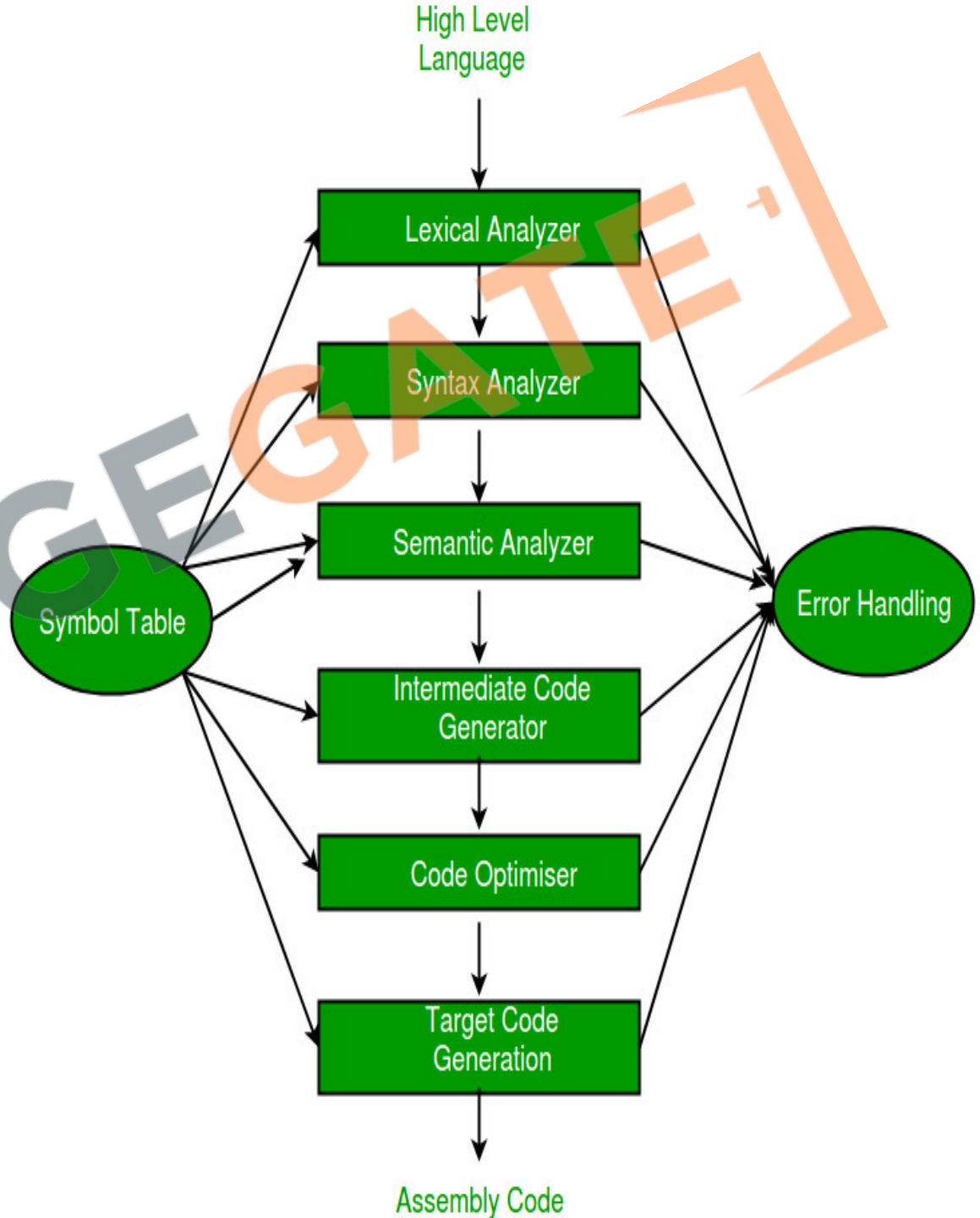
$x = y + t_1$



Target Code Generator

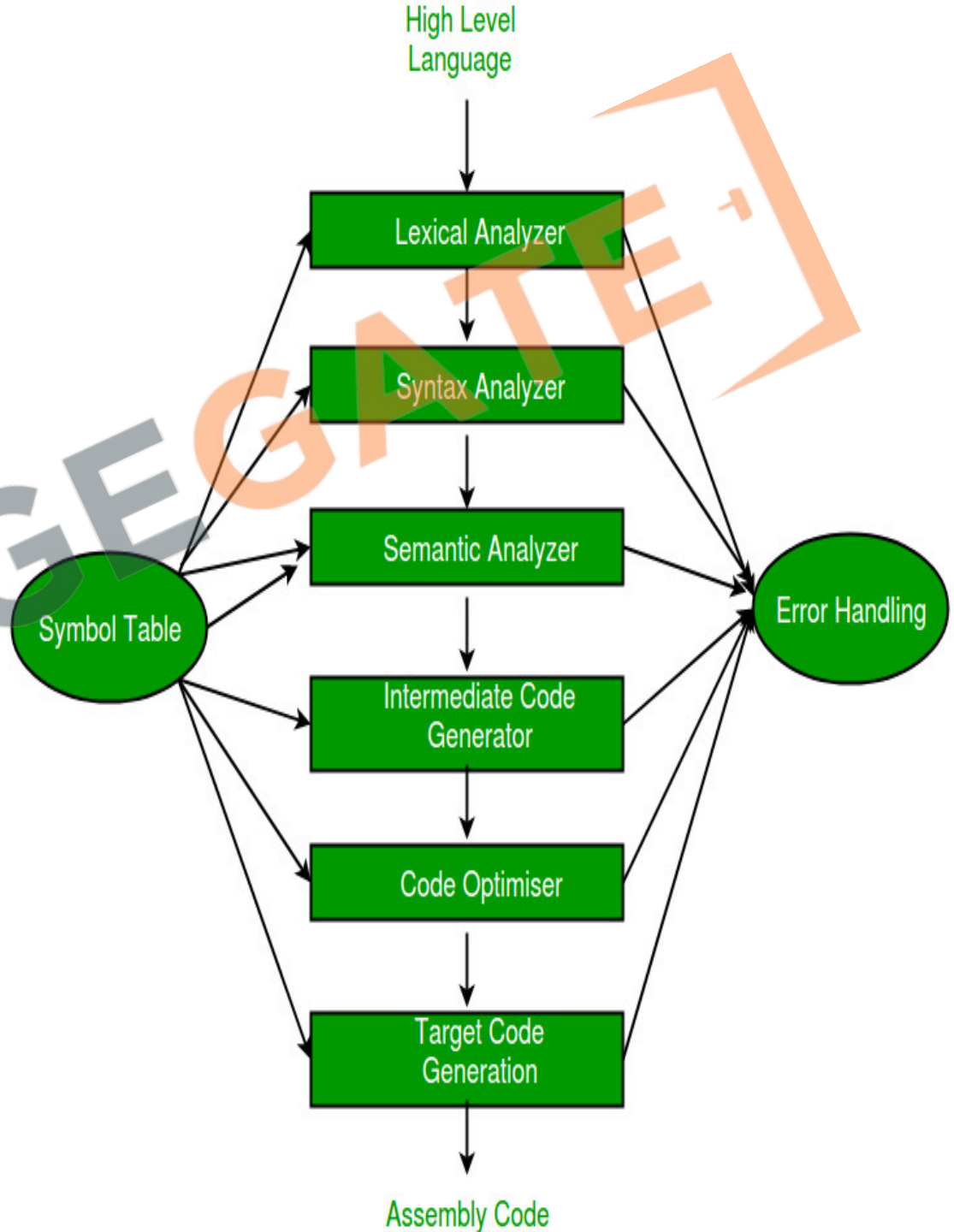
- The main purpose of Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection etc.
- The output is dependent on the type of assembler. This is the final stage of compilation.

MOV R_1, Z
MUL R_1, 60
ADD R_1, Y
STORE X, R_1



Symbol Table

- It is a data structure being used and maintained by the compiler, consists all the identifier's name along with their types.
- It helps the compiler to function smoothly by finding the identifiers quickly.



- The compiler has two modules namely front end and back end. Front-end constitutes of the Lexical analyser, semantic analyser, syntax analyser and intermediate code generator. And the rest are assembled to form the back end.

- In general Front end fill symbol table and back end uses it.

(1) int x = 10;

Line No	Keyword	identifier	Constant	Operator
1	int	x	10	;

- Lexical analysis in the first phase to communicate with the symbol and the compiler generate the symbol table during the lexical analysis phase.
- Compiler is responsible to provide the memory for symbol table. at every phase if any new variable occurs, then they will be stored in the symbol table.
- Every phase of the compiler will be interacting with the symbol table.
- In general, during the first two phases, we store the information in the symbol table and in the memory and in the later phases, we make use of the information available in symbol table.

- Information stored in the symbol table about identifier
  - name
  - type
  - scope
  - size
  - offset

- Other information in case of array, records and procedures etc.
  - array → size
  - records → column names
  - procedure → i/p parameter
  - functions → i/p, o/p parameter, actual, formal parameter

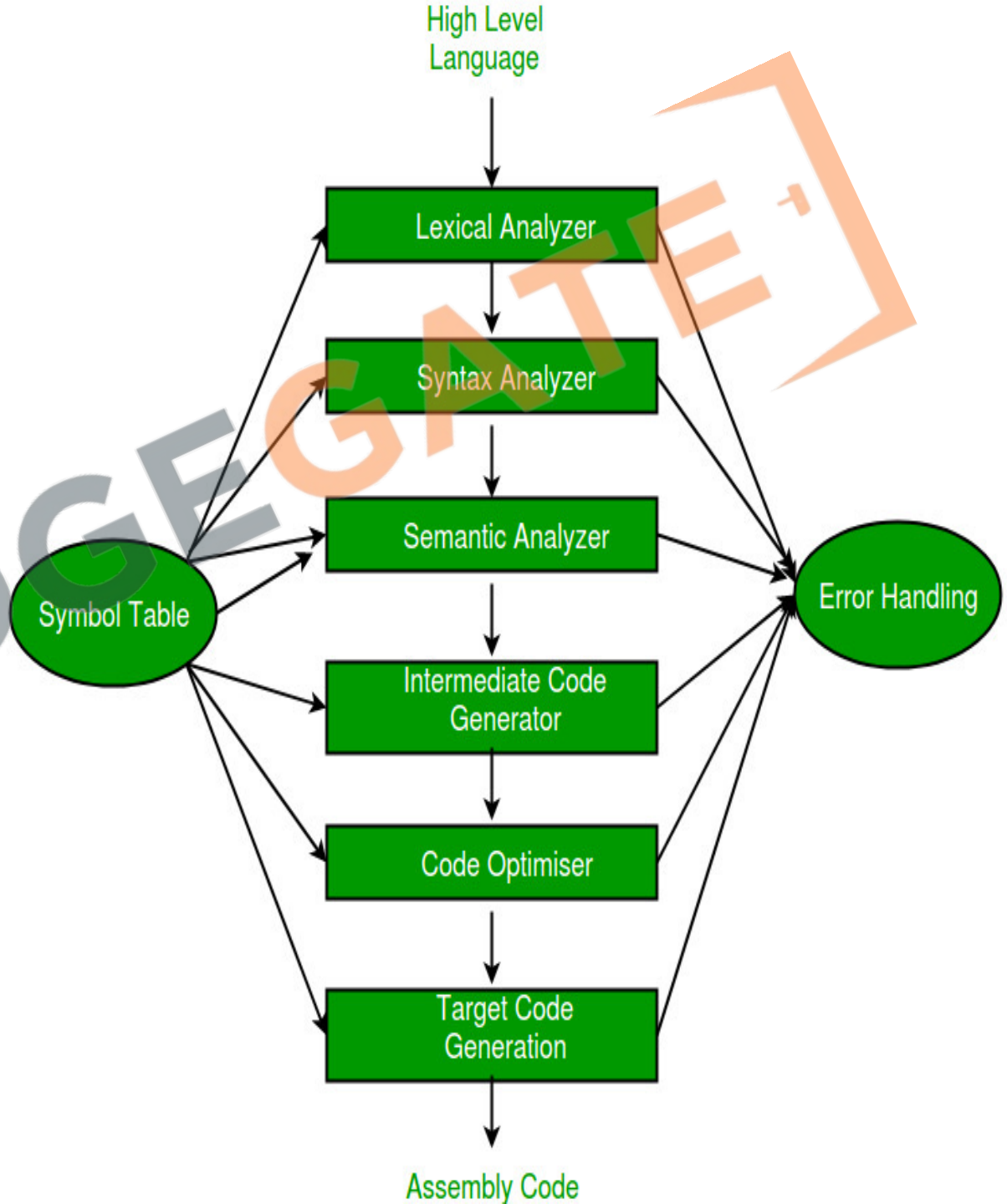
- Operation on the symbol table: - there are 4 operation function that can be performed on symbol table
  - insert
  - lookup/search
  - Modify
  - delete

- Implementation of symbol tables can be done using anyone of the data structure
  - liner table
  - Binary Search Tree
  - Linked List
  - Hash Table (most popular)

Error handler

- It is a sub routine to care take care of the continuation of compilation, even any error at any phase, i.e. error handler is responsible to continue the compilation process even any error occurs at phase 1 or phase 2 or phase 3.

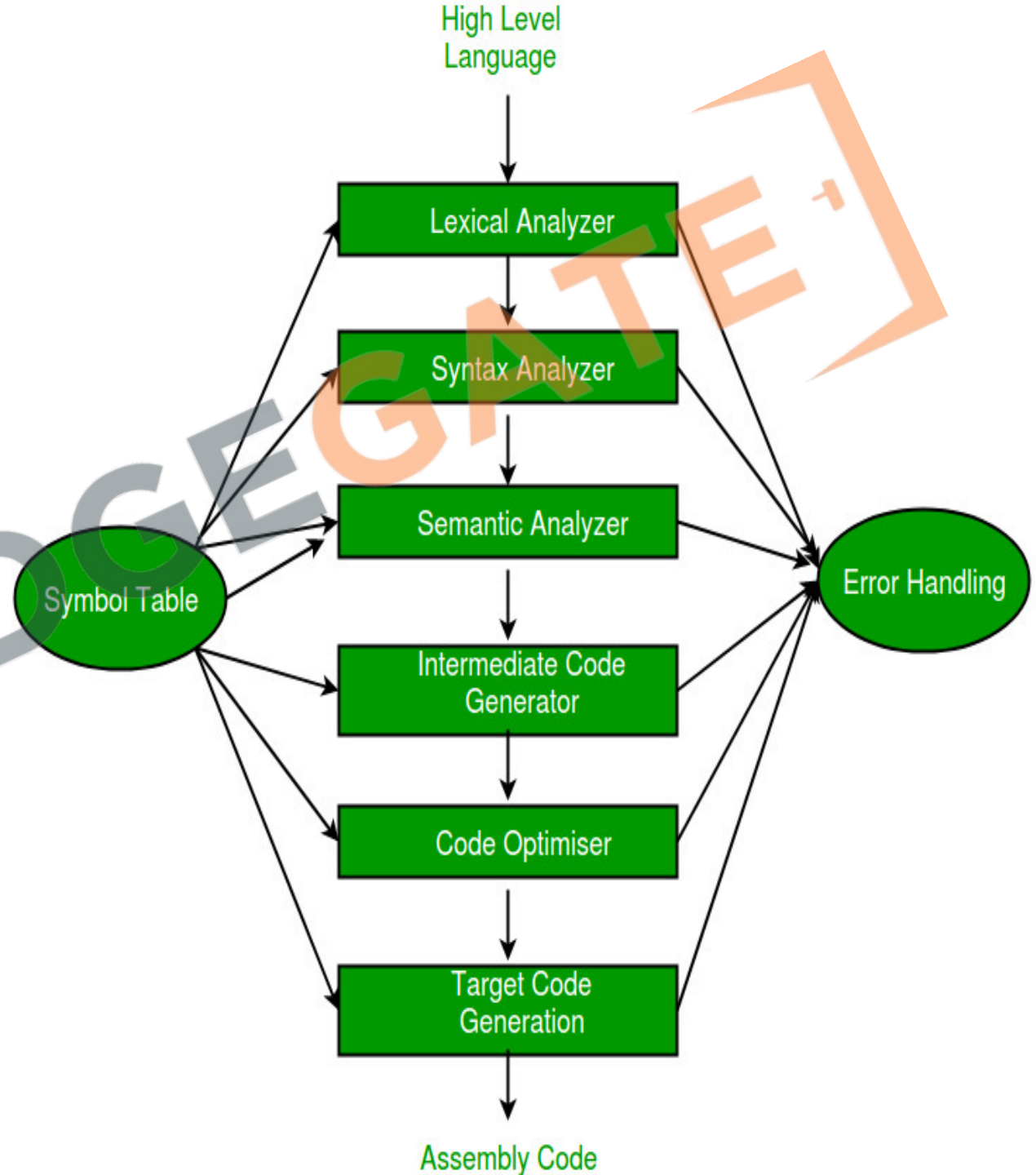
- After phase 3, if the error handler object is empty, then the source code is free error and it can be converted into target code. if the error handler object is not empty after phase 3 then there will be some error at phase 1,2 or 3 then the error will be displayed.



- Compiler can handle, there types of errors: -
  - lexical error
  - syntax error
  - semantic error

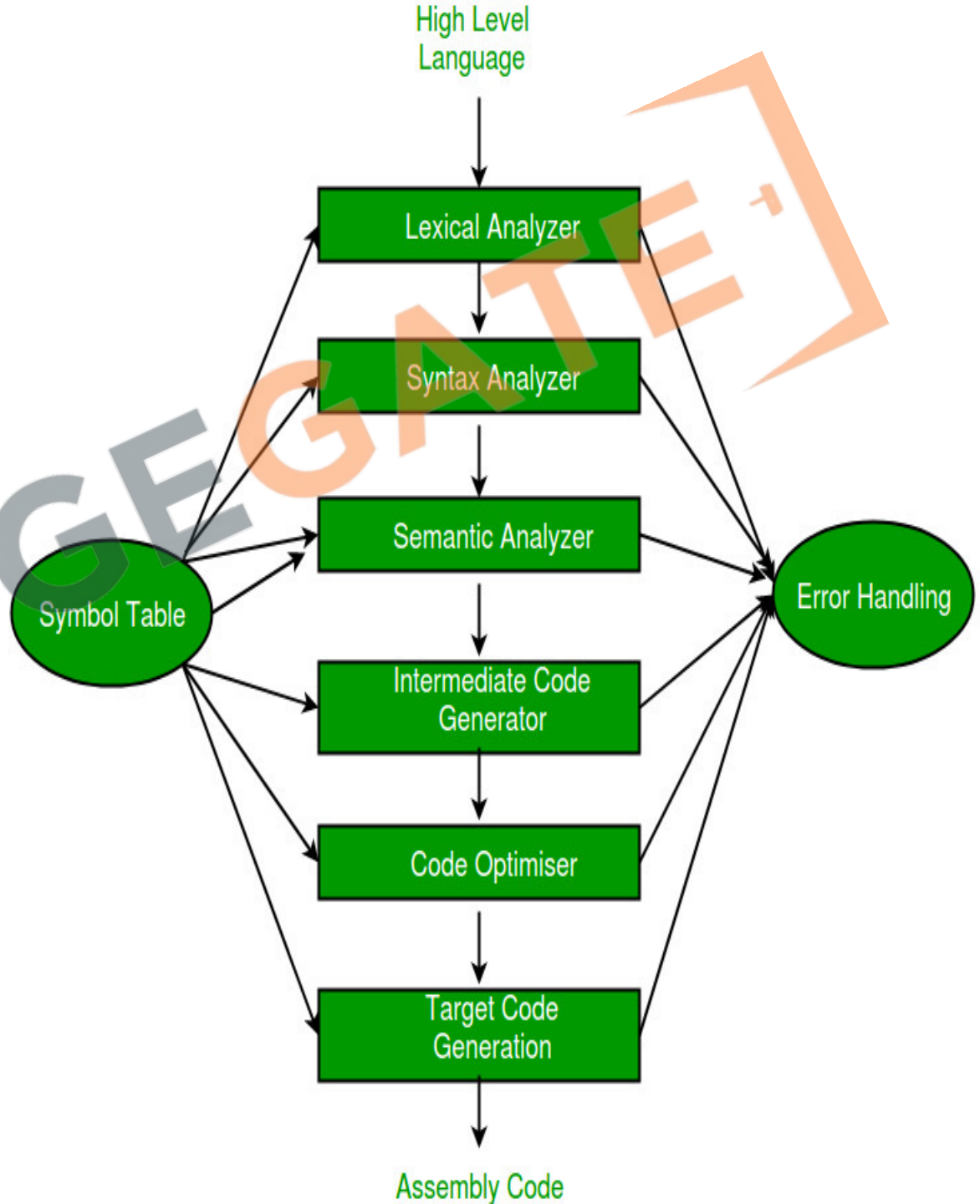
- The error handles by compiler are known as exception and programmer is responsible to handle the exception.

- At the time of execution also, we can get some error they are called fatal error and system admin is responsible to handle fatal error.



Phases and Passes

- A compiler can have many phases and passes.
- **Pass**: A pass refers to the traversal of a compiler through the entire program.
- **Phase**: A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.



Single Pass Compiler

- **One-Time Processing**: Reads and processes the source code in a single pass, translating it directly into machine code.
- **Speed Advantage**: Faster than multi-pass compilers due to the single traversal of the source code.
- **Memory Efficient**: Uses less memory as it doesn't store extensive intermediate state information.
- **Optimization Limitations**: Offers limited optimization capabilities due to less context understanding of the entire program.
- **Best for Simple Languages**: Ideal for simpler programming languages where complex analysis and deep optimization are not required.
- **Early FORTRAN Compilers, Pascal (Original Version), Tiny C Compiler (TCC)**

Multi-Pass Compiler

- A multi-pass compiler is a type of compiler that processes the source code multiple times before producing the final output. Here's a quick rundown in bullet points:
  - **Multiple Scans of Source Code**: It processes the source code in several passes, each for a specific analysis or transformation task.
  - **Advanced Optimization**: Multi-pass compilers perform complex optimizations by analyzing and re-analyzing the code, improving performance or minimizing resource usage.
  - **Context Awareness**: They have a deeper understanding of the program's context, which allows for more sophisticated error checking and optimization.
  - **Memory Usage**: They typically require more memory than single-pass compilers, as they need to maintain intermediate representations of the source code between passes.
  - **Examples**: Modern compilers like GCC (GNU Compiler Collection) and LLVM are multi-pass compilers that provide extensive optimization and support for complex language features.

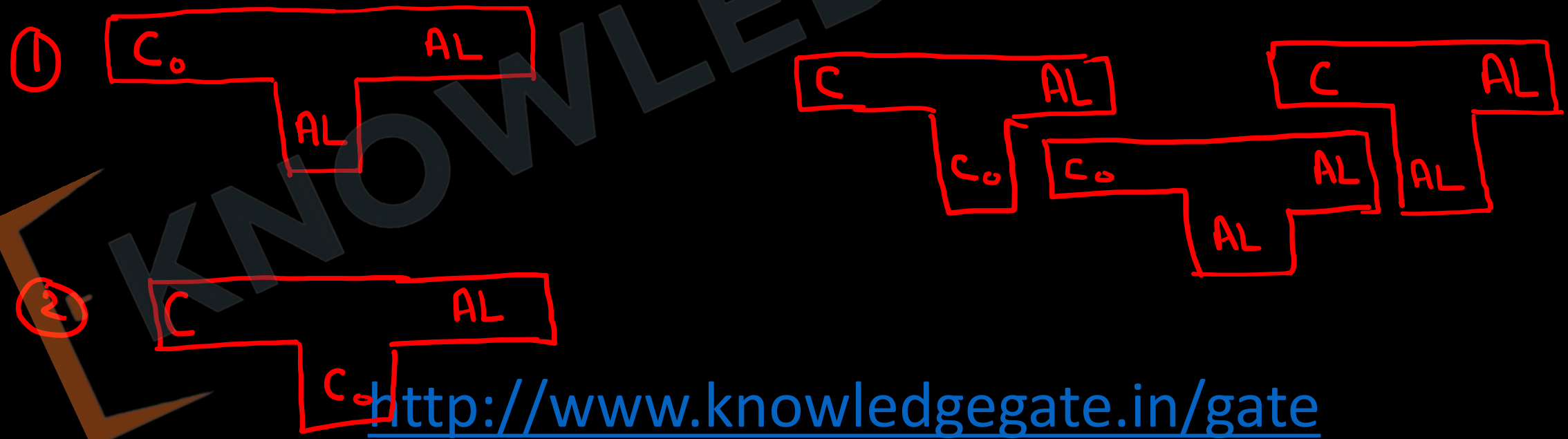
Aspect	Single-Pass Compiler	Multi-Pass Compiler
Compilation Speed	Generally faster as it compiles the source code in one pass.	Slower, as it goes through the source code multiple times.
Memory Usage	Less memory-intensive because it doesn't store much intermediate data.	More memory-intensive due to the need to store intermediate representations.
Error Detection	May not detect all errors in the first pass.	Better at error detection, as it analyzes the code in more depth over multiple passes.
Optimization	Limited optimization opportunities due to the single pass nature.	Better optimization, as multiple passes allow for more analysis and refinement.
Language Complexity	Suited for simpler languages with less complex syntax and semantics.	Better for complex languages, as it can handle intricate syntax and semantics more effectively.

Bootstrapping

- Bootstrapping in compiler design is a fascinating and critical concept.
- **Historical Context**
  - **Early Stages**: Initially, compilers were written in assembly language or a low-level language specific to the hardware.
  - **Evolution**: As programming languages evolved, the need for writing compilers in a higher-level language became evident.
  - **Bootstrapping Emergence**: Bootstrapping was introduced as a solution to this need. It refers to writing a compiler in the same language it intends to compile.

Concept

- **Initial Step**: A simple compiler is first written in a low-level language. This is often termed as the "bootstrap compiler."
- **Self-Compiling**: The compiler is then rewritten in its own higher-level language and compiled using the bootstrap compiler.
- **Iteration**: This process can be iterated, with each new version of the compiler used to compile its next version.



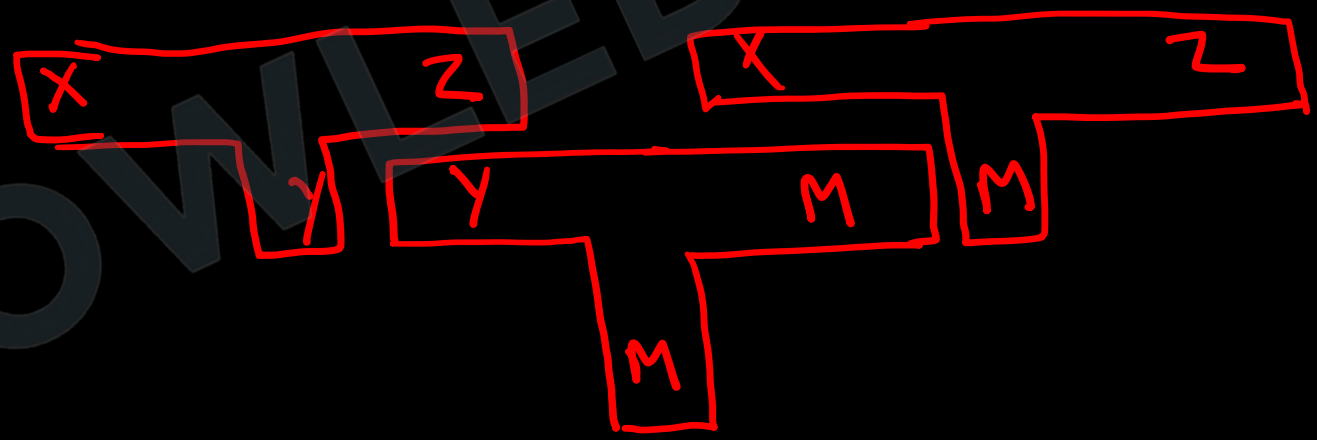
- Bootstrapping is the process of writing a compiler for a programming language using the language itself. In other words, it is the process of using a compiler written in a particular programming language to compile a new version of the compiler written in the same language.

- The process of bootstrapping typically involves several stages. In the first stage, a minimal version of the compiler is written in a different language, such as assembly language or C. This minimal version of the compiler is then used to compile a slightly more complex version of the compiler written in the target language. This process is repeated until a fully functional version of the compiler is written in the target language.

- One advantage is that it ensures that the compiler is compatible with the language it is designed to compile. so it is better able to understand and interpret the syntax and semantics of the language.
- Another advantage is that it allows for greater control over the optimization and code generation process.
- One disadvantage is that it can be a time-consuming process, especially for complex languages or compilers. It can also be more difficult to debug a bootstrapped compiler, since any errors or bugs in the compiler will affect the subsequent versions of the compiler.

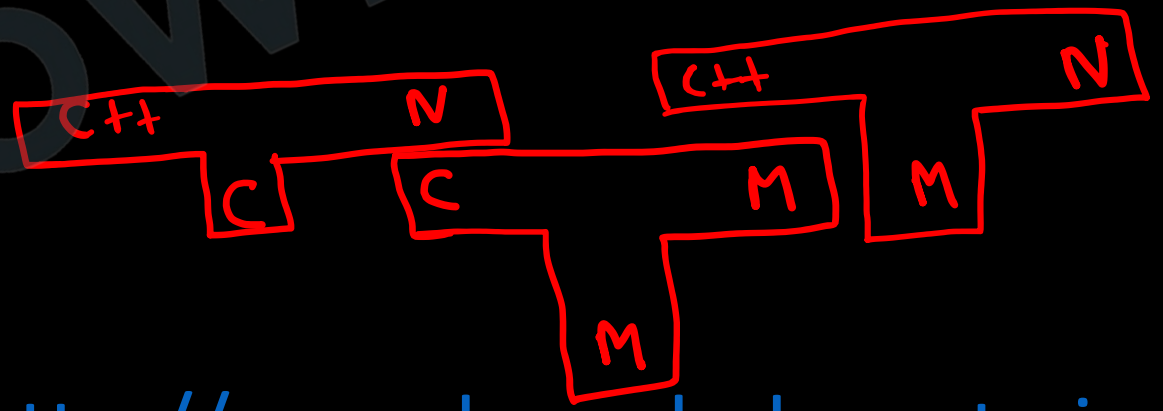
Cross Compiler

- Suppose we want to write a cross compiler for new language X. The implementation language of this compiler is say Y and the target code being generated is in language Z. That is, we create XYZ. Now if existing compiler Y runs on machine M and generates code for M then it is denoted as YMM. Now if we run XYZ using YMM then we get a compiler XMZ. That means a compiler for source language X that generates a target code in language Z and which runs on machine M.



Cross Compiler

- **Early Computing Era**: Initially, software was often developed and executed on the same type of machine. This limited software portability across different hardware.
- **Growth of Diverse Hardware**: With the proliferation of various computing systems, including microprocessors and embedded systems, the need for software compatibility across different platforms increased.
- **Development of Cross Compilers**: Cross compilers were developed to address this need. They allowed developers to write code on one machine (the host) and execute it on another (the target), which could have a completely different architecture.



DETERMINISTIC FINITE AUTOMATA

A **deterministic finite automaton (DFA)** is defined by 5-tuple (Q,Σ,δ,S,F) where:

- Q is a finite and non-empty set of states
- Σ is a finite non-empty set of finite input alphabet
- δ is a transition function, ($\delta: Q \times \Sigma \rightarrow Q$)
- S is **initial state** (always one) ($S \in Q$)
- F is a set of final states ($F \subseteq Q$) ($0<=|F|<=N$, where n is the number of states)

Q Construct the NFA for the regular expression $a|abb|a^*b^+$ by using Thompson's construction methodology**?**

Q Draw NFA for the regular expression $ab^*|ab$ **?**

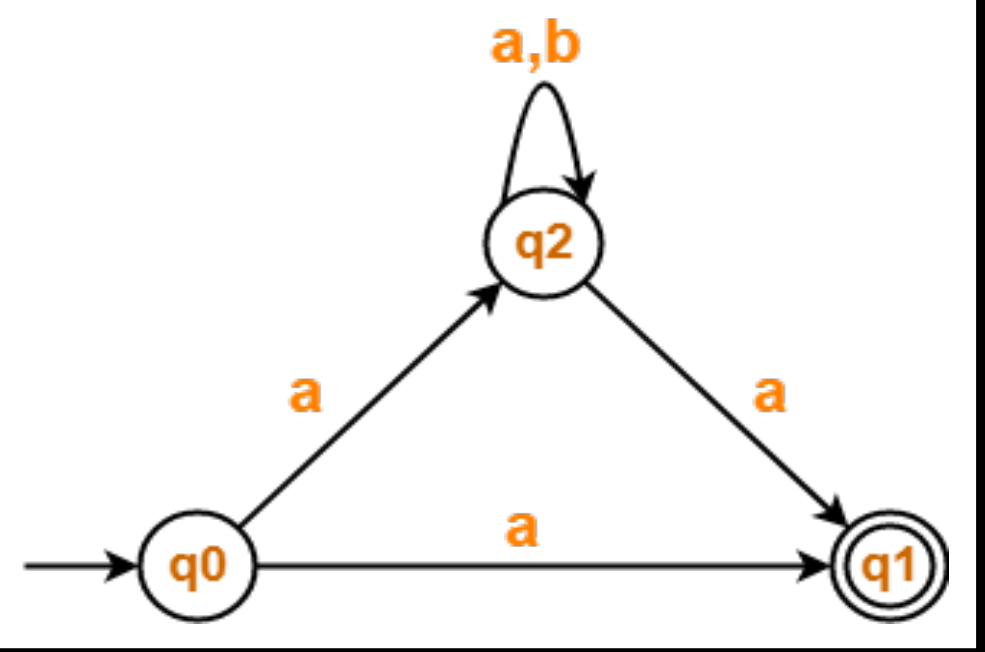
Q Construct the minimized DFA for the regular expression (0 + 1)*(0 + 1) 10 ?

NFA and DFA Equivalence

- In this topic we will be learning about the equivalence of NFA and DFA and how an NFA can be converted to equivalent DFA. Let us take an example and understand the conversion.
- Since every NFA and DFA has equal power that means, for every language if a NFA is possible, then DFA is also possible.
- So, every NFA can be converted to DFA.
- The process of conversion of an NFA into a DFA is called Subset Construction.
- If NFA have ‘n’ states which is converted into DFA which ‘m’ states than the relationship between n and m will be
- $1 <= m <= 2^n$

Procedure for Conversion

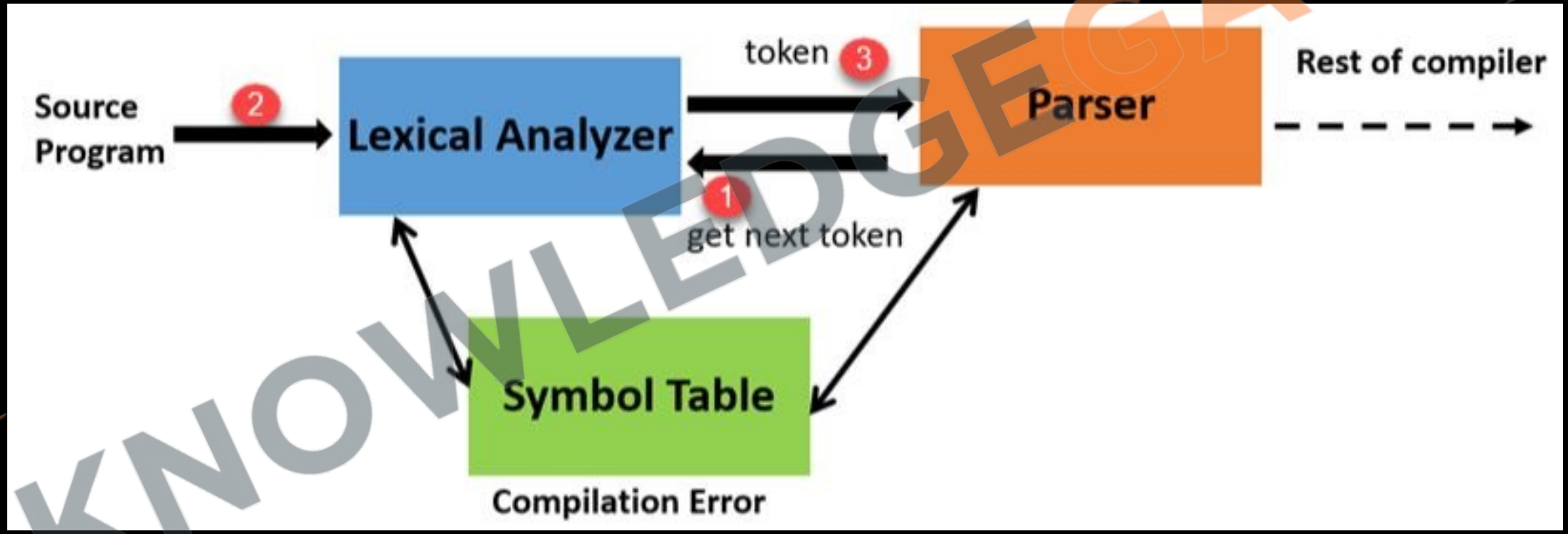
- There lies a fixed algorithm for the NFA and DFA conversion. Following things must be considered
  - Initial state will always remain same.
  - Start the construction of δ' with the initial state & continue for every new state that comes under the input column and terminate the process whenever no new state appears under the input column.
  - Every subset of states that contain the final state of the NFA is a final state in the resulting DFA.
  - $\delta'(q_0, q_1, q_2, q_{3\text{--------}}, q_{n-1}, a) = \bigcup_{i=0}^{i=n-1} \delta(q_i, a)$



Aspect	DFA (Deterministic Finite Automata)	NDFA (Nondeterministic Finite Automata)
State Transition	On each input symbol, transitions to exactly one state.	Can transition to multiple states or none on the same input symbol.
Determinism and Uniqueness	Each state has a unique transition for each input symbol.	A state can have multiple transitions for the same input symbol.
Computation Path	Always has a single, unique computation path for any input string.	May have multiple computation paths for the same input string.
Ease of Construction	Generally simpler and more straightforward to construct.	Can be more complex to construct due to non-determinism.
Acceptance of Input	Accepts an input if it reaches a final state after processing all input symbols.	Accepts an input if at least one computation path reaches a final state.

Lexical Analyzer

- **lexical analysis**, **lexing** or **tokenization** is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens.



Lexical Analyzer

- A *lexeme* is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token. Where a token is a strings with an assigned and thus identified meaning. Actual representation or stream of characters is called as Lexemes; Logical meaning of Lexemes is known as Tokens.

- Lexing can be divided into two stages: the *scanning*, which segments the input string into syntactic units called *lexemes* and categorizes these into token classes; and the *evaluating*, which converts lexemes into processed values.

Lexeme	token
while	while
(	lparen
y	identifier
>=	Comparison
t	identifier
)	Rparen
y	identifier
=	Assignment
y	identifier
-	Arithmetic
3	integer
;	Finish of a statement

- x = a + b * 2;

- [(identifier, x), (operator, =), (identifier, a), (operator, +), (identifier, b), (operator, *), (literal, 2), (separator, ;)]

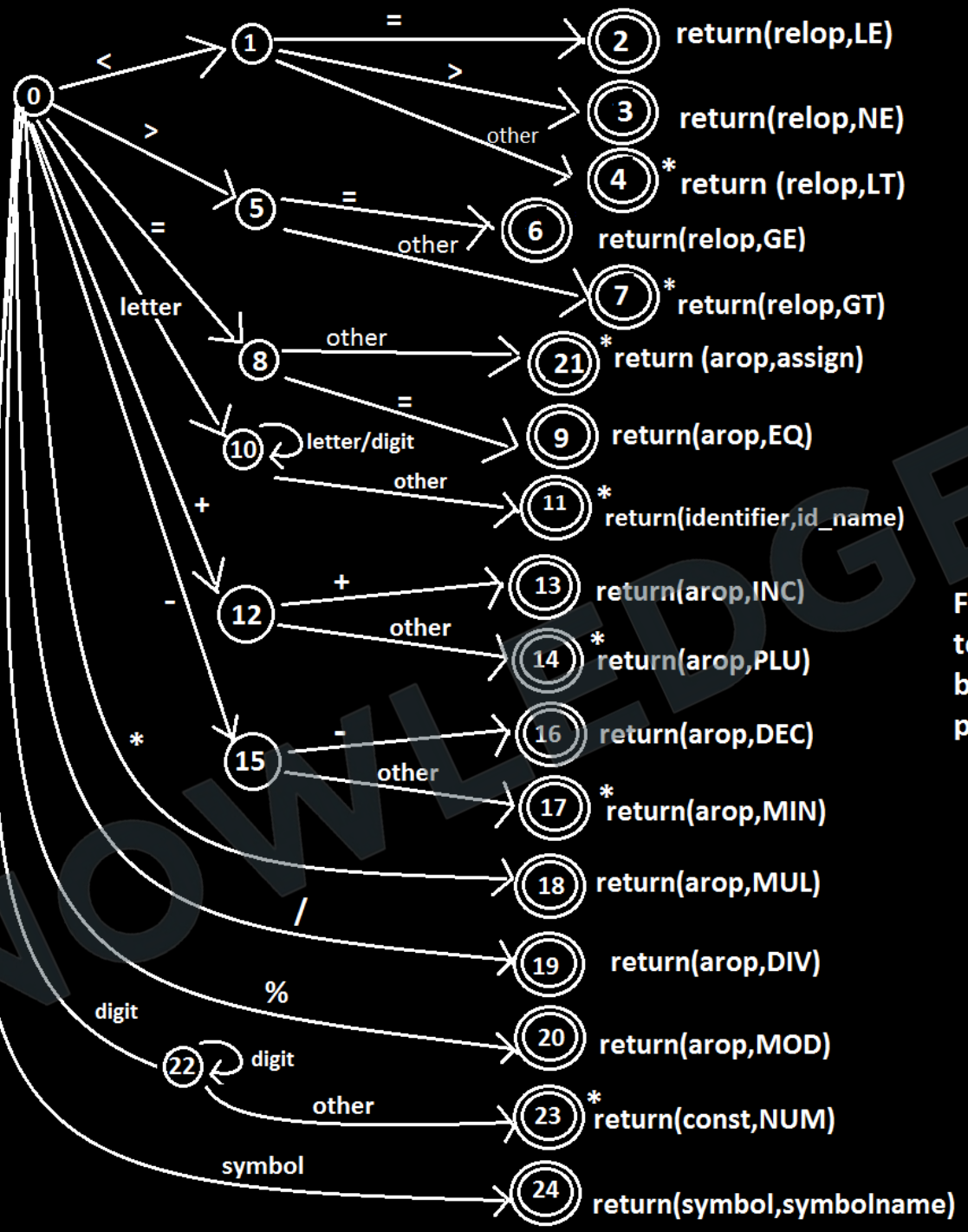
Q Identify the meaning of the rule implemented by Lexer using regular expression for identifying tokens of a password?

Regular Expression → $A(A+S+D)^3\,(A+S+D+\in)^4$

Implementation of Lexical analyzer

Lexical analyzer can be implemented in following step :

- Input to the lexical analyzer is a source program.
- By using input buffering scheme, it scans the source program.
- Regular expressions are used to represent the input patterns.
- Now this input pattern is converted into NFA by using finite automation machine.
- This NFA are then converted into DFA and DFA are minimized by using different method of minimization.
- The minimized DFA are used to recognize the pattern and broken into lexemes.
- Each minimized DFA is associated with a phase in a programming language which will evaluate the lexemes that match the regular expression.
- The tool then constructs a state table for the appropriate finite state machine and creates program code which contains the table, the evaluation phases, and a routine which uses them appropriately.



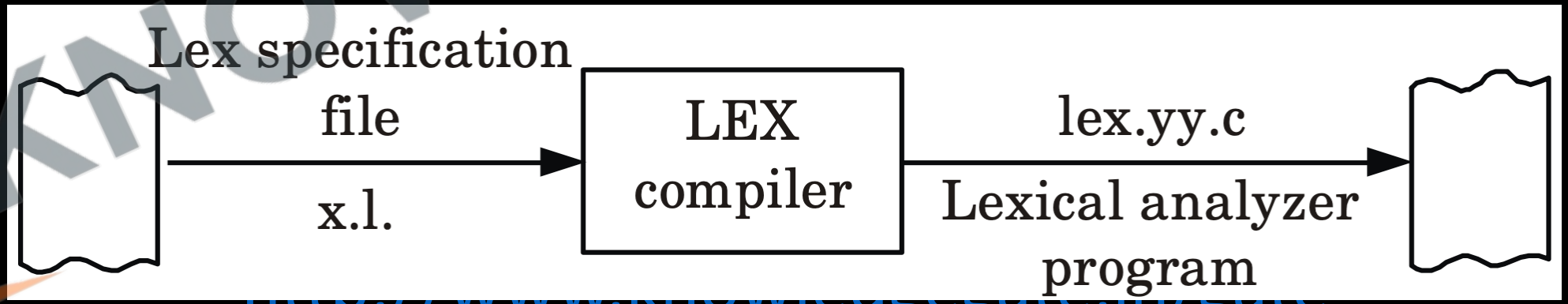
Final states marks with * requires to the forward pointer (fwdptr) to be retracted by one character position

lexical Analyzer Generator

- For efficient design of compiler, various tools are used to automate the phases of compiler. The lexical analysis phase can be automated using a tool called LEX.
- LEX is a Unix utility which generates lexical analyzer.
- The lexical analyzer is generated with the help of regular expressions.
- LEX lexer is very fast in finding the tokens as compared to handwritten LEX program in C.
- LEX scans the source program in order to get the stream of tokens and these tokens can be related together so that various programming structure such as expression, block statement, control structures, procedures can be recognized.

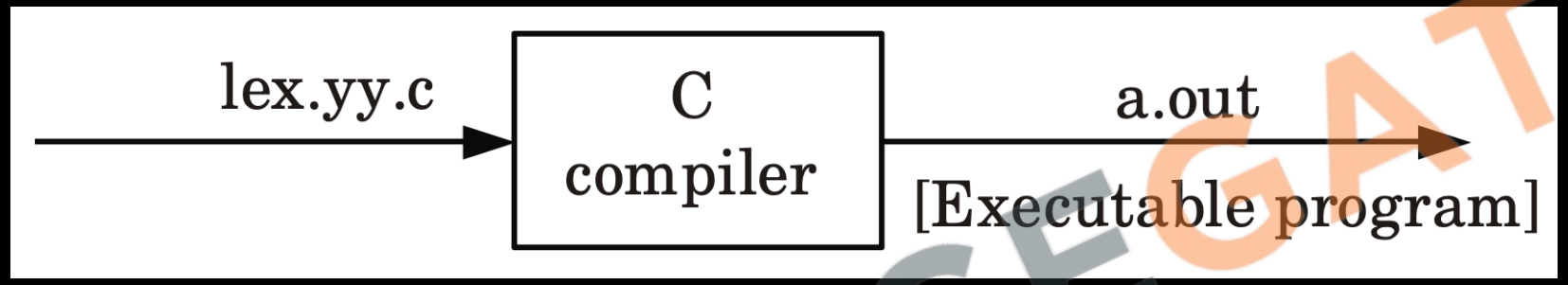
LEX compiler

- Automatic generation of lexical analyzer is done using LEX programming language.
- The LEX specification file can be denoted using the extension .l (often pronounced as dot L).
- For example, let us consider specification file as x.l.
- This x.l file is then given to LEX compiler to produce lex.yy.c . This lex.yy.c is a C program which is actually a lexical analyzer program.
- The LEX specification file stores the regular expressions for the token and the lex.yy.c file consists of the tabular representation of the transition diagrams constructed for the regular expression.
- In specification file, LEX actions are associated with every regular expression.
- These actions are simply the pieces of C code that are directly carried over to the lex.yy.c.

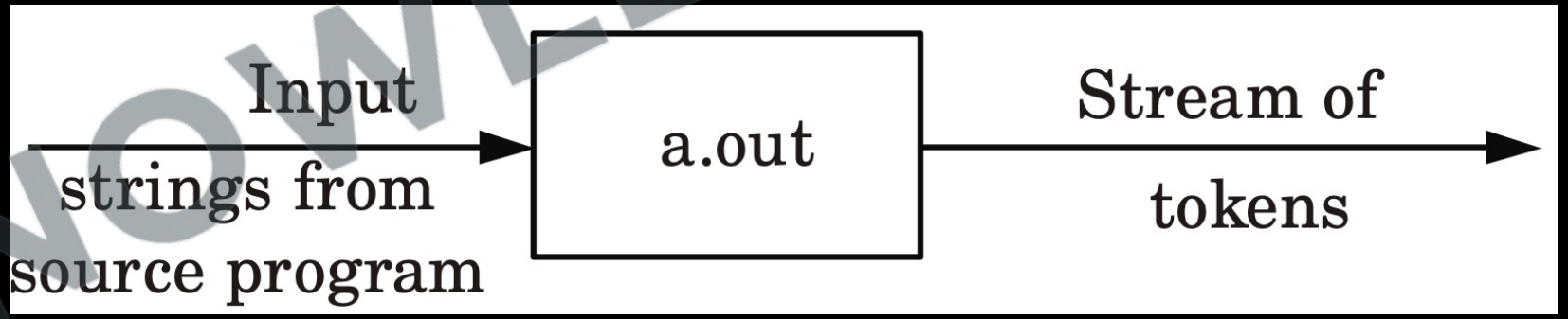


Generation of lexical analyzer using LEX

- Finally, the C compiler compiles this generated lex.yy.c and produces an object program a.out.



- When some input stream is given to a.out then sequence of tokens gets generated.



Components of LEX program

The LEX program consists of three parts :

Declaration section :

- In the declaration section, declaration of variable constants can be done.
- Some regular definitions can also be written in this section.
- The regular definitions are basically components of regular expressions.

Components of LEX program

- **Rule section :**
  - The rule section consists of regular expressions with associated actions. These translation rules can be given in the form as :
  - R_1 {action$_1$}
  - R_2 {action$_2$}
  - .
  - .
  - R_n {action$_n$}
  - Where each R_i is a regular expression and each action$_i$ is a program fragment describing what action is to be taken for corresponding regular expression.
  - These actions can be specified by piece of C code.

Components of LEX program

- **Auxiliary procedure section :**
- In this section, all the procedures are defined which are required by the actions in the rule section.
- This section consists of two functions :
  - main() function
  - yywrap() function

- Secondary Function of Lexical Analyser are: -
  - Removal of Comments lines

Start here | *main.c

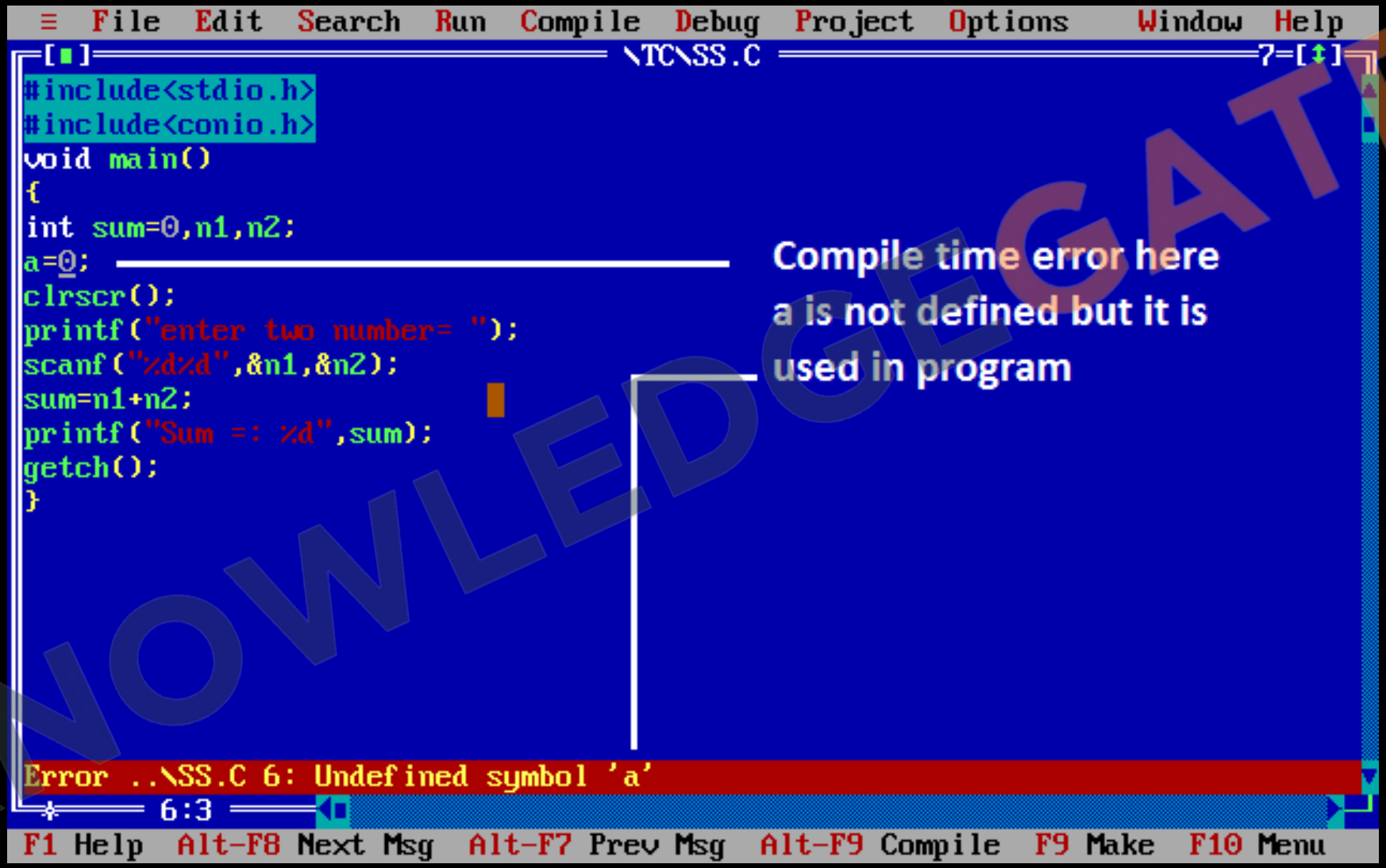
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a; // Declare two variables
    printf("Enter any number\n\n");
    scanf("%d",&a);
/* After declaration, we print our message on the screen*/
/* and in the 3rd step, save the address of a */
    return 0;
}
```

- Removal of White space characters

WHITE SPACE CHARACTERS

\b	blank space	\t	horizontal tab	\v	vertical tab
\\	Back slash	\'	Single quote	\"	Double quote
\r	carriage return	\f	form feed	\n	new line
\?	Question mark	\0	Null	\a	Alarm (bell)

- Co-relating with Error Messages along with line number.



Formal Grammar

A phrase-structure grammar (or simply a grammar) is a 4-tuple (V_N, ∑, P, S), where

- V_N is a finite nonempty set whose elements are called variables,
- ∑ is a finite nonempty set whose elements are called terminals, $V_N \cap \sum = \Phi$.
- S is a special variable (i.e., an element of V_N ($S \in V_n$)) called the start symbol. Like every automaton has exactly one initial state, similarly every grammar has exactly one start symbol.
- P is a finite set whose elements are $\alpha \rightarrow \beta$. where α and β are strings on $V_N \cup \sum$. α has at least one symbol from V_N, the element of P are called productions or production rules or rewriting rules. $\{\Sigma \cup V_n\}^*$ some writer refers it as total alphabet

For a formal valid production

$\alpha \rightarrow \beta$

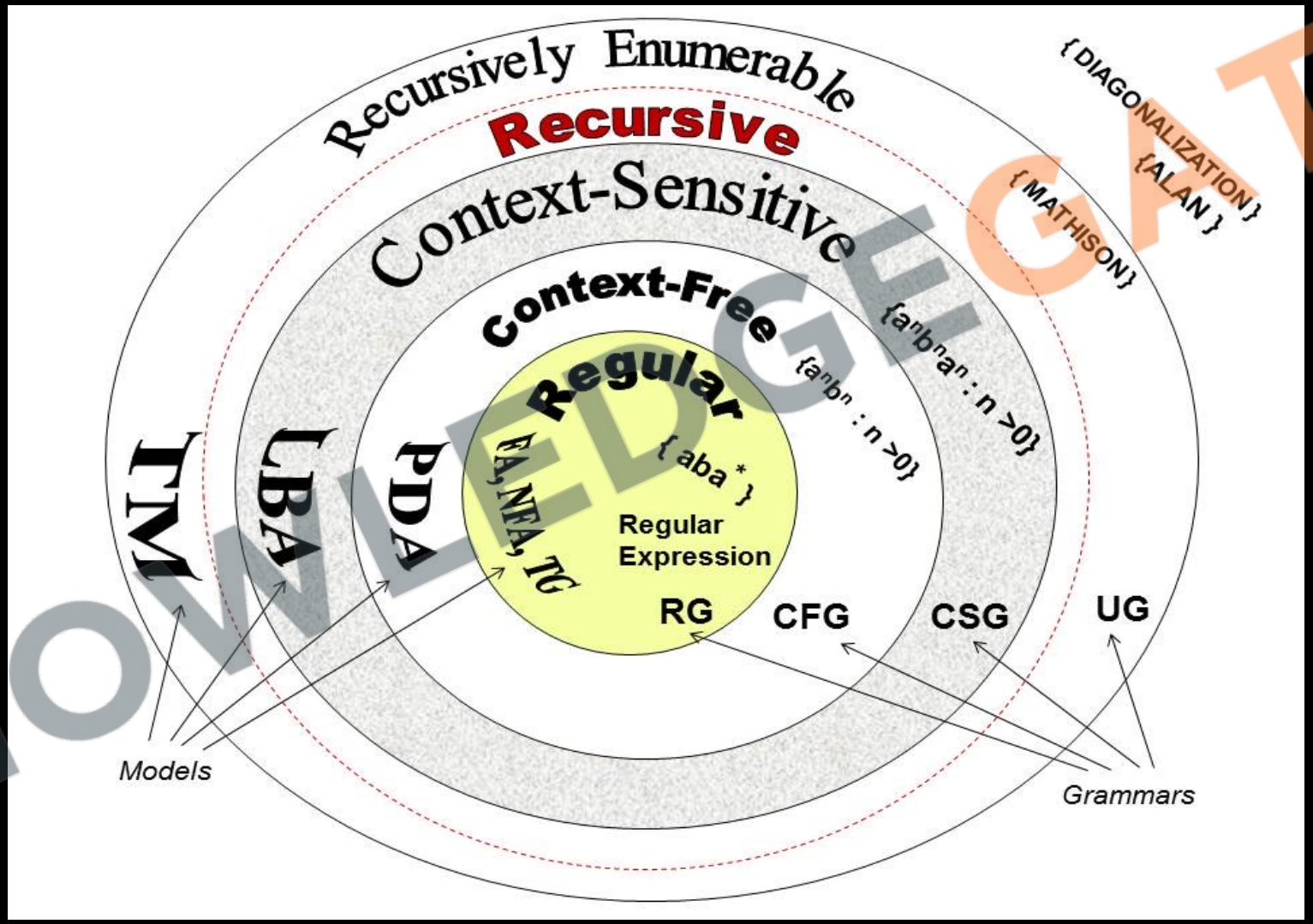
$\alpha \in \{\Sigma \cup V_n\}^* V_n \{\Sigma \cup V_n\}^*$

$\beta \in \{\Sigma \cup V_n\}^*$

If $G = (\{S\}, \{0, 1\}, \{S \rightarrow 0S1, S \rightarrow \Lambda\}, S)$, find $L(G)$.

LANGUAGES AND AUTOMATA

- Following are the machines that accepts the following grammars.



Type 2 Grammar

- Also known as Context Free Grammar, which will generate context free language that will be accepted by push down automata. (NPDA default case)

- if there is a production, from

$$\alpha \rightarrow \beta$$

$$\alpha \in V_n \qquad |\alpha| = 1$$

$$\beta \in \{\Sigma \cup V_n\}^*$$

- In other words, the L.H.S. has no left context or right context.

- A grammar is called a type 2 grammar if it contains only type 2 productions.

- Eg ALGOL 60, PASCAL

Formal grammar and its application to syntax analysis

- Formal grammar represents the specification of programming language with the use of production rules.
- The syntax analyzer basically checks the syntax of the language.
- A syntax analyzer takes the tokens from the lexical analyzer and groups them in such a way that some programming structure can be recognized.
- After grouping the tokens if at all any syntax cannot be recognized then syntactic error will be generated.
- This overall process is called syntax checking of the language.
- This syntax can be checked in the compiler by writing the specifications.
- Specification tells the compiler how the syntax of the programming language should be.

BNF notation

- Backus-Naur Form (BNF) is a notation technique used to express the grammar of a computer language. It's a formal mathematical way to describe a language, which is particularly useful in the context of programming languages. Here's a simple explanation:

- **Symbols**: BNF uses two types of symbols:
  - **Terminal Symbols**: These are the basic characters or strings in the language, like numbers, letters, or specific words.
  - **Non-terminal Symbols**: These represent a set of strings, and they're defined by writing rules in BNF.

- **Production Rules**: A language in BNF is defined by production rules. Each rule has a left-hand side (a non-terminal symbol) and a right-hand side, which is a sequence of terminal and/or non-terminal symbols. The rule shows how you can replace the non-terminal with that sequence.

- **Example**: Let's consider a simple arithmetic expression grammar:
  - Expression → Expression + Term | Term
  - Term → Term * Factor | Factor
  - Factor → (Expression) | Number
  - Number → Digit | Number Digit
  - Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- In this example, "Expression", "Term", "Factor", "Number", and "Digit" are non-terminal symbols, while '+', '*', '(', ')', and the digits are terminal symbols.

1. *program* → *declarationList*
2. *declarationList* → *declarationList declaration* | *declaration*
3. *declaration* → *varDeclaration* | *funDeclaration*

---

4. *varDeclaration* → *typeSpecifier varDeclList* **;**
5. *scopedVarDeclaration* → *scopedTypeSpecifier varDeclList* **;**
6. *varDeclList* → *varDeclList* **,** *varDeclInitialize* | *varDeclInitialize*
7. *varDeclInitialize* → *varDeclId* | *varDeclId* **:** *simpleExpression*
8. *varDeclId* → **ID** | **ID [NUMCONST]**
9. *scopedTypeSpecifier* → **static** *typeSpecifier* | *typeSpecifier*
10. *typeSpecifier* → **int** | **bool** | **char**

---

11. *funDeclaration* → *typeSpecifier* **ID** **(** *params* **)** *statement* | **ID** **(** *params* **)** *statement*
12. *params* → *paramList* | ϵ
13. *paramList* → *paramList* **;** *paramTypeList* | *paramTypeList*
14. *paramTypeList* → *typeSpecifier paramIdList*

15. *paramIdList* → *paramIdList* , *paramId* | *paramId*

16. *paramId* → **ID** | **ID** [ ]

---

17. *statement* → *expressionStmt* | *compoundStmt* | *selectionStmt* | *iterationStmt* | *returnStmt* | *breakStmt*

18. *expressionStmt* → *expression* ; | ;

19. *compoundStmt* → { *localDeclarations statementList* }

20. *localDeclarations* → *localDeclarations scopedVarDeclaration* | ϵ

21. *statementList* → *statementList statement* | ϵ

22. *elsifList* → *elsifList* **elsif** *simpleExpression* **then** *statement* | ϵ

23. *selectionStmt* → **if** *simpleExpression* **then** *statement elsifList* | **if** *simpleExpression* **then** *statement elsifList* **else** *statement*

24. *iterationRange* → **ID** = *simpleExpression* .. *simpleExpression* | **ID** = *simpleExpression* .. *simpleExpression* : *simpleExpression*

25. *iterationStmt* → **while** *simpleExpression* **do** *statement* | **loop forever** *statement* | **loop** *iterationRange* **do** *statement*

26. *returnStmt* → **return** ; | **return** *expression* ;

27. *breakStmt* → **break** ;

28. *expression* → *mutable* = *expression* | *mutable* += *expression* | *mutable* −= *expression* | *mutable* *= *expression* | *mutable* /= *expression* | *mutable* ++ | *mutable* −− | *simpleExpression*

29. *simpleExpression* → *simpleExpression* **or** *andExpression* | *andExpression*

30. *andExpression* → *andExpression* **and** *unaryRelExpression* | *unaryRelExpression*

31. *unaryRelExpression* → **not** *unaryRelExpression* | *relExpression*

32. *relExpression* → *sumExpression relop sumExpression* | *sumExpression*

33. *relop* → <= | < | > | >= | == | !=

34. *sumExpression* → *sumExpression sumop mulExpression* | *mulExpression*

35. *sumop* → + | −

36. *mulExpression* → *mulExpression mulop unaryExpression* | *unaryExpression*

37. *mulop* → * | / | %

38. *unaryExpression* → *unaryop unaryExpression* | *factor*

39. *unaryop* → − | * | ?

40. *factor* → *immutable* | *mutable*

41. *mutable* → **ID** | *mutable* [*expression*]

42. *immutable* → (*expression*) | *call* | *constant*

43. *call* → **ID** (*args*)

44. *args* → *argList* | **ϵ**

45. *argList* → *argList* , *expression* | *expression*

46. *constant* → **NUMCONST** | **CHARCONST** | **STRINGCONST** | **true** | **false**

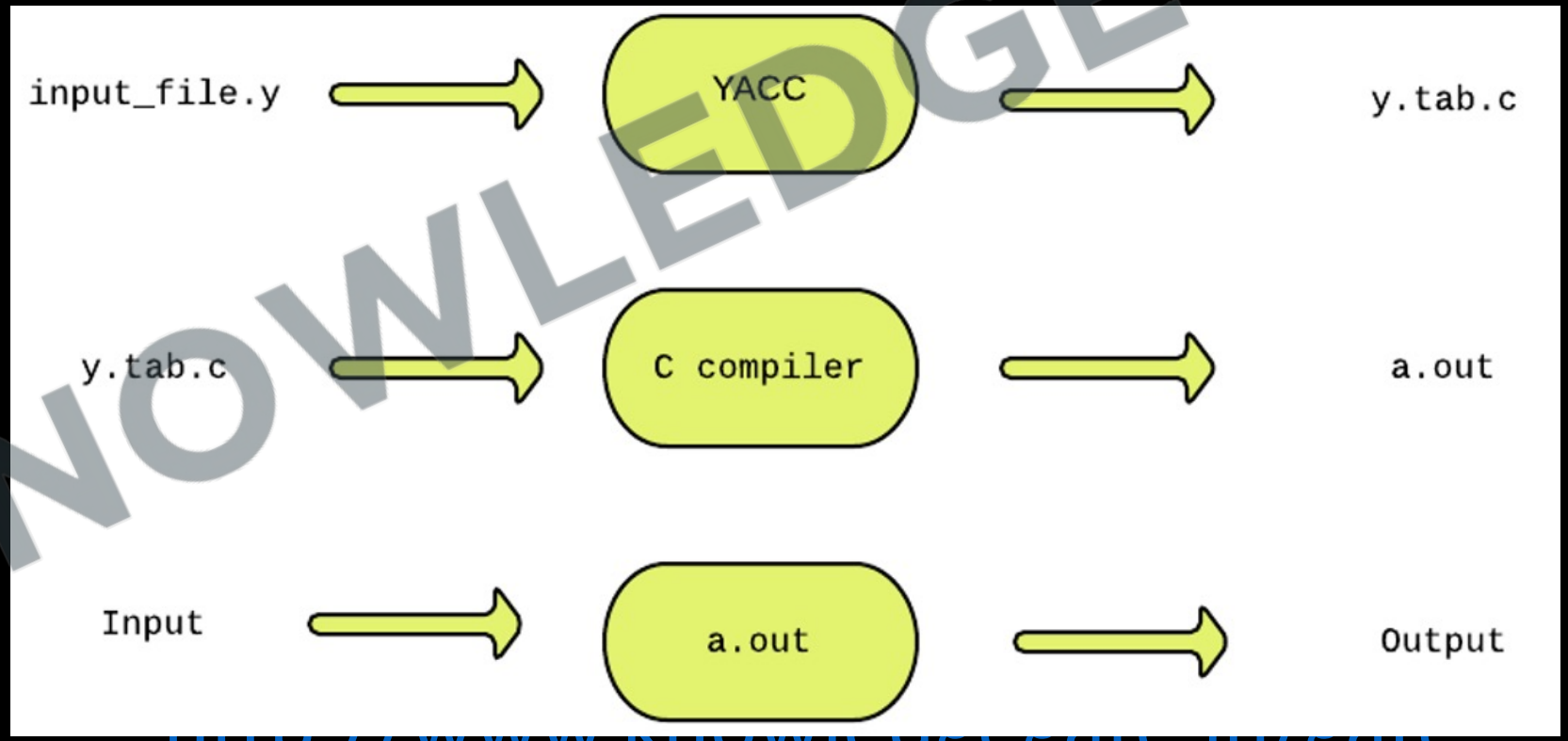
- **Use in Parsing**: BNF is widely used in the design and implementation of compilers and interpreters for programming languages. It helps in building the syntax tree of a program by parsing the source code according to the grammar rules defined in BNF.
- **Readability**: BNF provides a concise and readable way to specify the syntax of a language. It's easier to understand and modify compared to directly writing parser code.
- **Extensions**: There are also extensions to BNF, like Extended BNF (EBNF), which provides more expressive power by adding more constructs like optional elements, repetitions, and choices.

Ambiguous grammar: - The grammar CFG is said to be ambiguous if there are more than one derivation tree for any string i.e. if there exist more than one derivation tree (LMDT or RMDT), the grammar is said to be ambiguous.

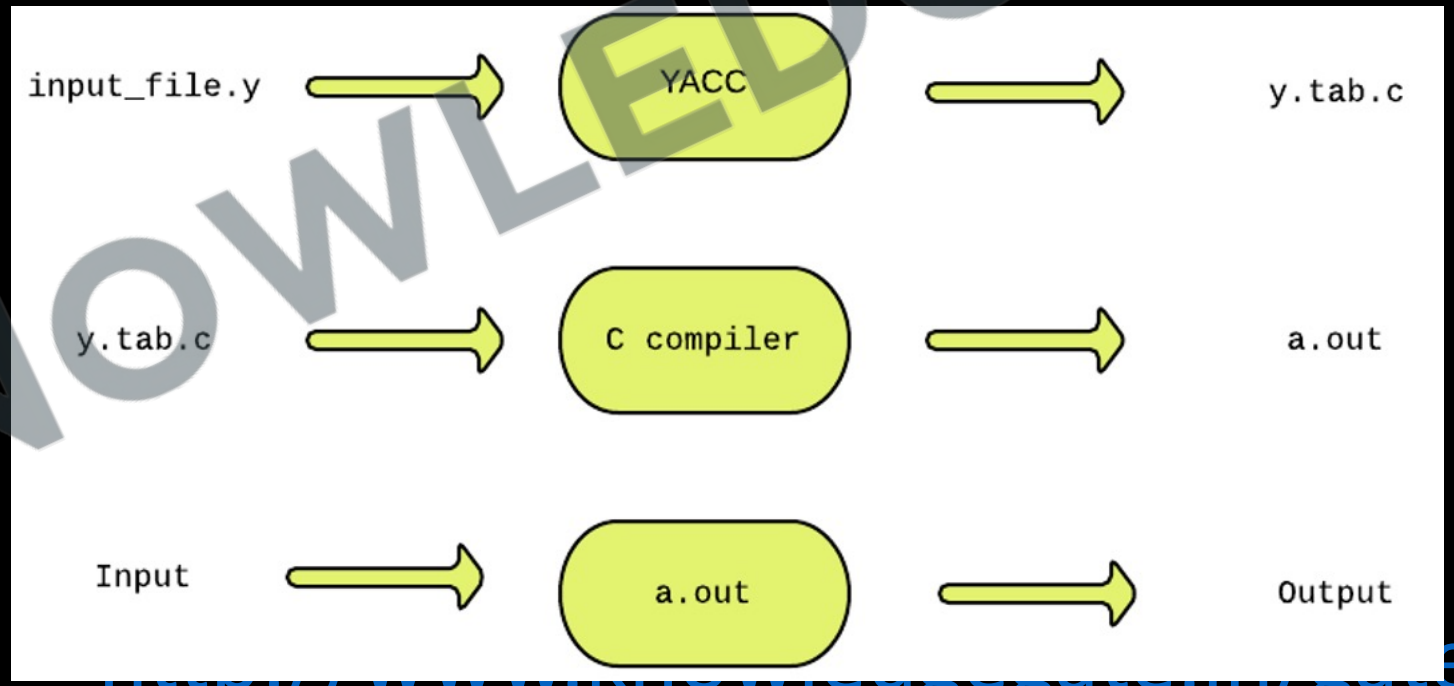
S → aS/Sa/a

YACC (Yet Another Compiler Compiler)

- YACC (Yet Another Compiler Compiler) is a tool used in compiler construction to generate a parser from a specified grammar. Here are the key aspects:
- **Function**: YACC is utilized for creating compilers or interpreters. It takes a language's grammar (in Backus-Naur Form) and produces C code for a parser.



- **Structure of YACC File**:
  - **Declarations**: Defines tokens and includes necessary C code.
  - **Rules**: Contains grammar rules with associated C actions.
  - **User Subroutines**: Additional C functions for the parser.
- **Integration with Lex**: YACC is commonly used with Lex, a lexical analyzer, for tokenization.
- **Parser Type**: The parser generated is a LALR(1) parser, a type of efficient bottom-up parser.



- **Error Handling**: YACC includes syntax error detection and recovery mechanisms.
- **Advantages**:
  - Automates parser creation.
  - Simplifies grammar modification.
  - Separates syntax rules from actions.
- **Applications and Alternatives**: It's widely used in both academia and industry for compiler and interpreter development. Tools like Bison offer similar functionality with additional features.
- Understanding YACC is important for computer science students, particularly for insights into compiler design and parsing techniques.

Defining a language by grammar

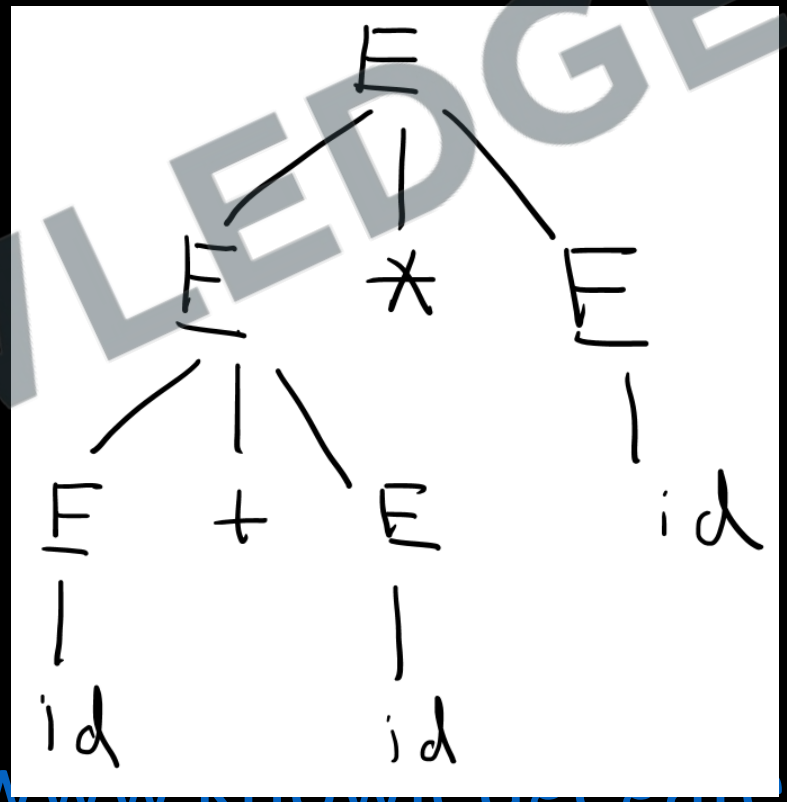
- The concept of defining a language using grammar is, starting from a start symbol using the production rules of the grammar any time, deriving the string. Here every time during derivation a production is used as its LHS is replaced by its RHS, all the intermediate stages(strings) are called sentential forms. The language formed by the grammar consists of all distinct strings that can be generated in this manner.

$$L(G) = \{w \mid w \in \sum^*, S \rightarrow^* W\}$$

- $\rightarrow^*$(reflexive, transitive closure) means from s we can derive w in zero or more steps

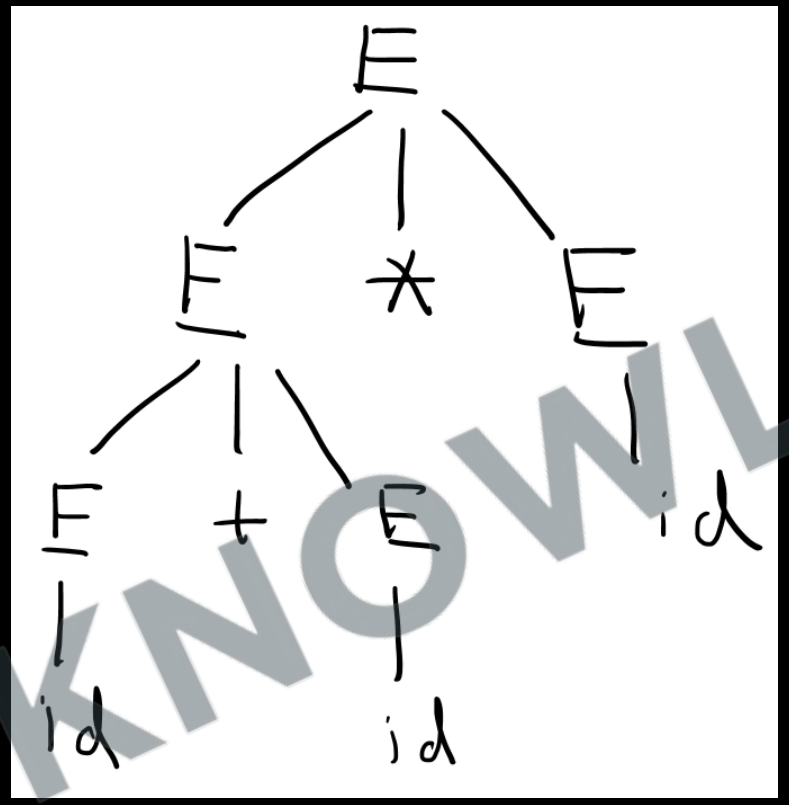
- **Derivation:** - The process of deriving a string is known as derivation.
- **Derivation/ Syntax/ Parse Tree**: - The graphical representation of derivation is known as derivation tree.

E → E + E / E * E / E = E / id



- **Sentential form:** - Intermediate step involve in the derivation is known as sentential form.

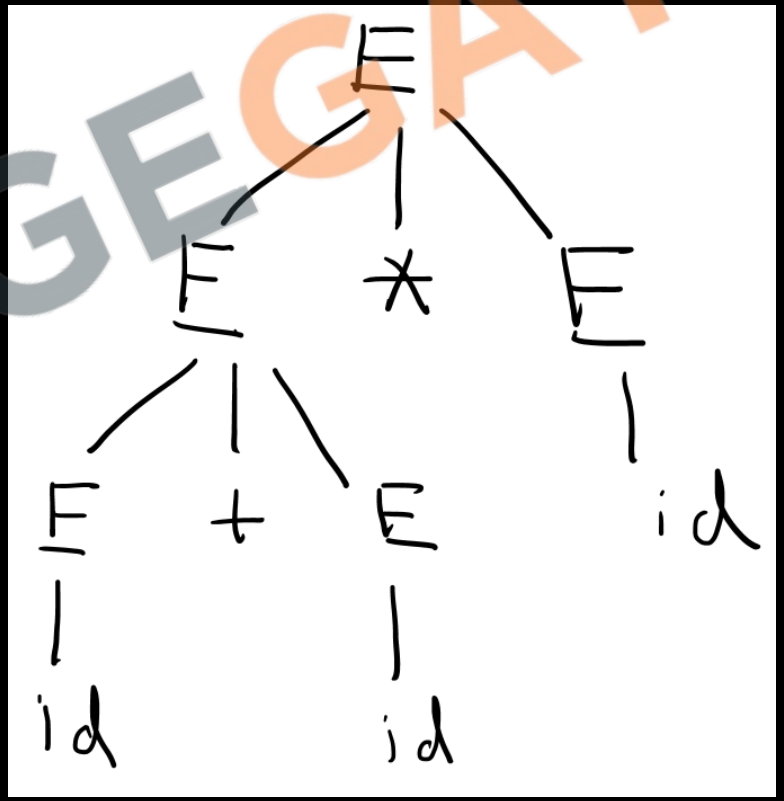
E → E + E / E * E / E = E / id



Sentential Form
E
E*E
E+E*E
ID+E*E
ID+ID*E
ID+ID*ID

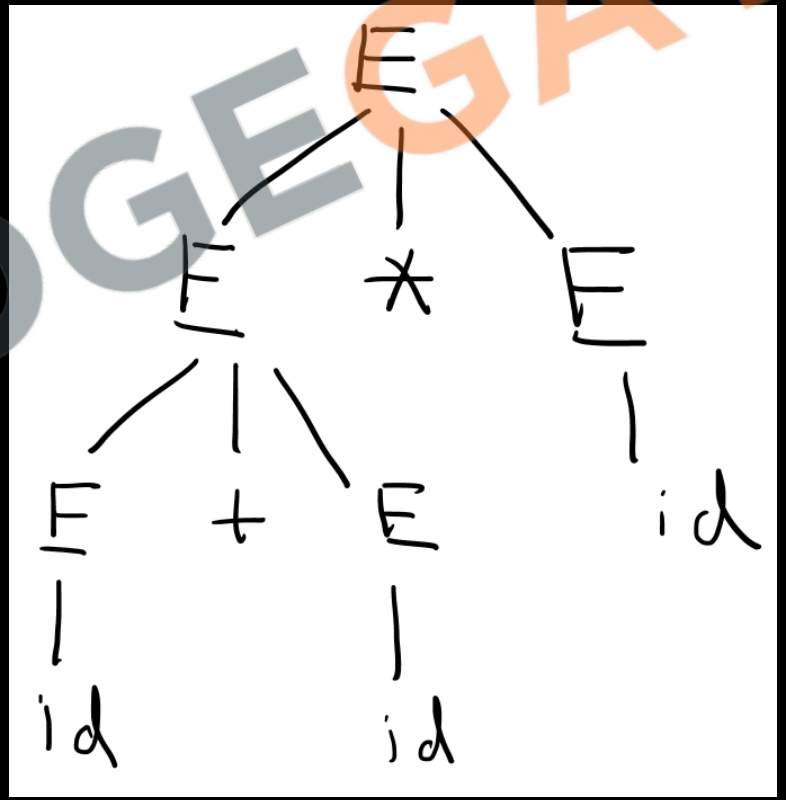
- **Left most derivation**: - the process of construction of parse tree by expanding the left most non terminal is known as LMD and the graphical representation of LMD is known as LMDT (left most derivation tree)

	LMD
E → E + E	E
E→ E * E	E*E
E→ E = E	E+E*E
E→ id	ID+E*E
E→ id	ID+ID*E
E→ id	ID+ID*ID



- **Right most derivation**: - the process of construction of parse tree by expanding the right most non terminal is known as RMD and the graphical representation of RMD is known as RMDT (right most derivation tree)

	RMD
E → E + E	E
E→ E * E	E+E
E→ E = E	E+E*E
E→ id	E+E*ID
E→ id	E+ID*ID
E→ id	ID+ID*ID



Capabilities of CFG

Context-Free Grammar (CFG) plays a vital role in computer science for describing programming language syntax and creating efficient parsers:

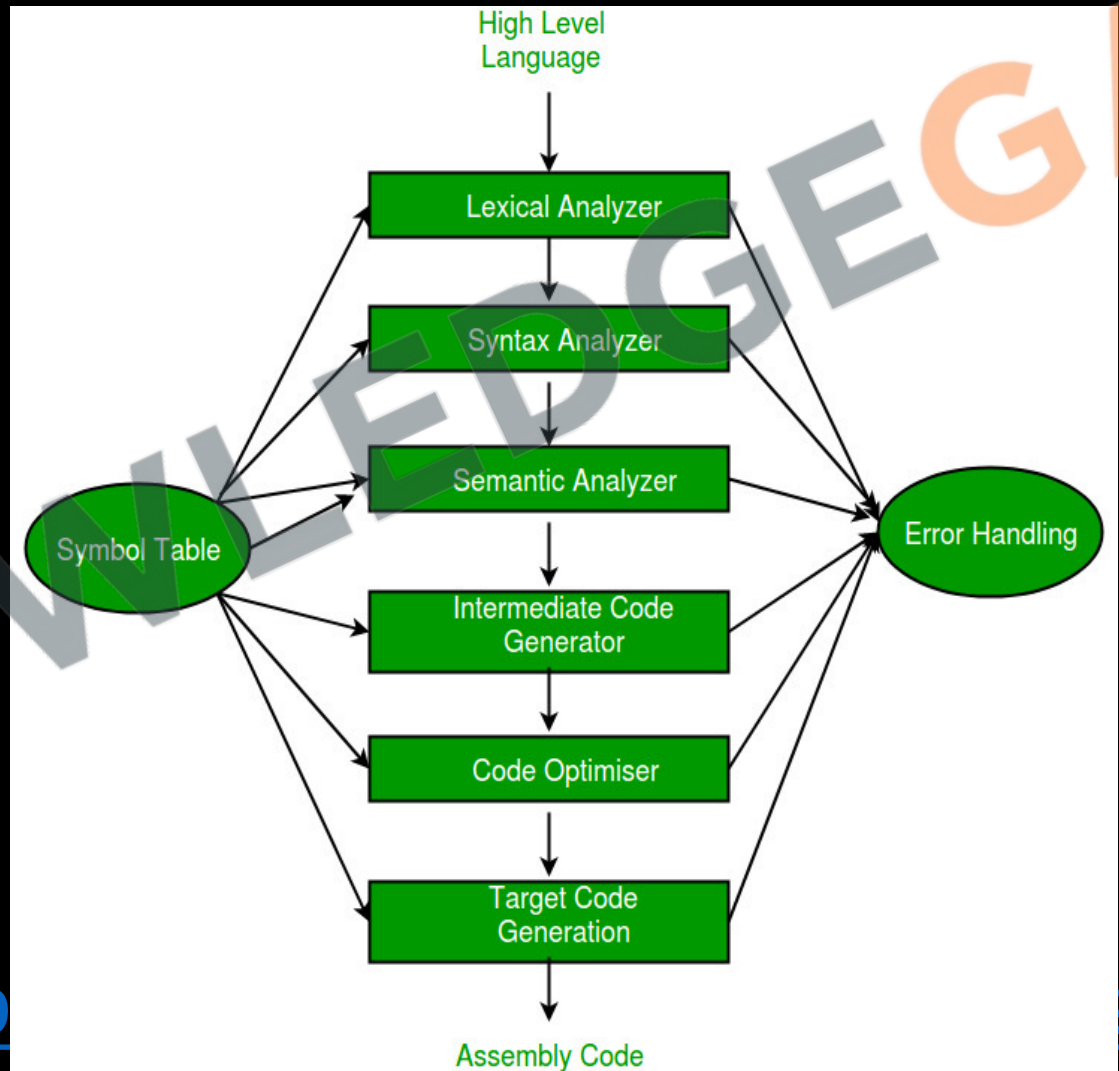
- **Programming Languages**: CFGs effectively describe the syntax of most programming languages, organizing elements like statements, expressions, and declarations.
- **Efficient Parser Construction**: A well-designed CFG enables the automatic generation of efficient parsers, essential for code analysis and interpretation.
- **Associativity and Precedence**: CFGs handle associativity and precedence in expressions, ensuring correct interpretation of operations and expression hierarchies.
- **Nested Structures**: CFGs excel in depicting nested structures common in programming languages, like balanced parentheses, matching begin-end blocks, and nested if-then-else statements.

Chapter-2

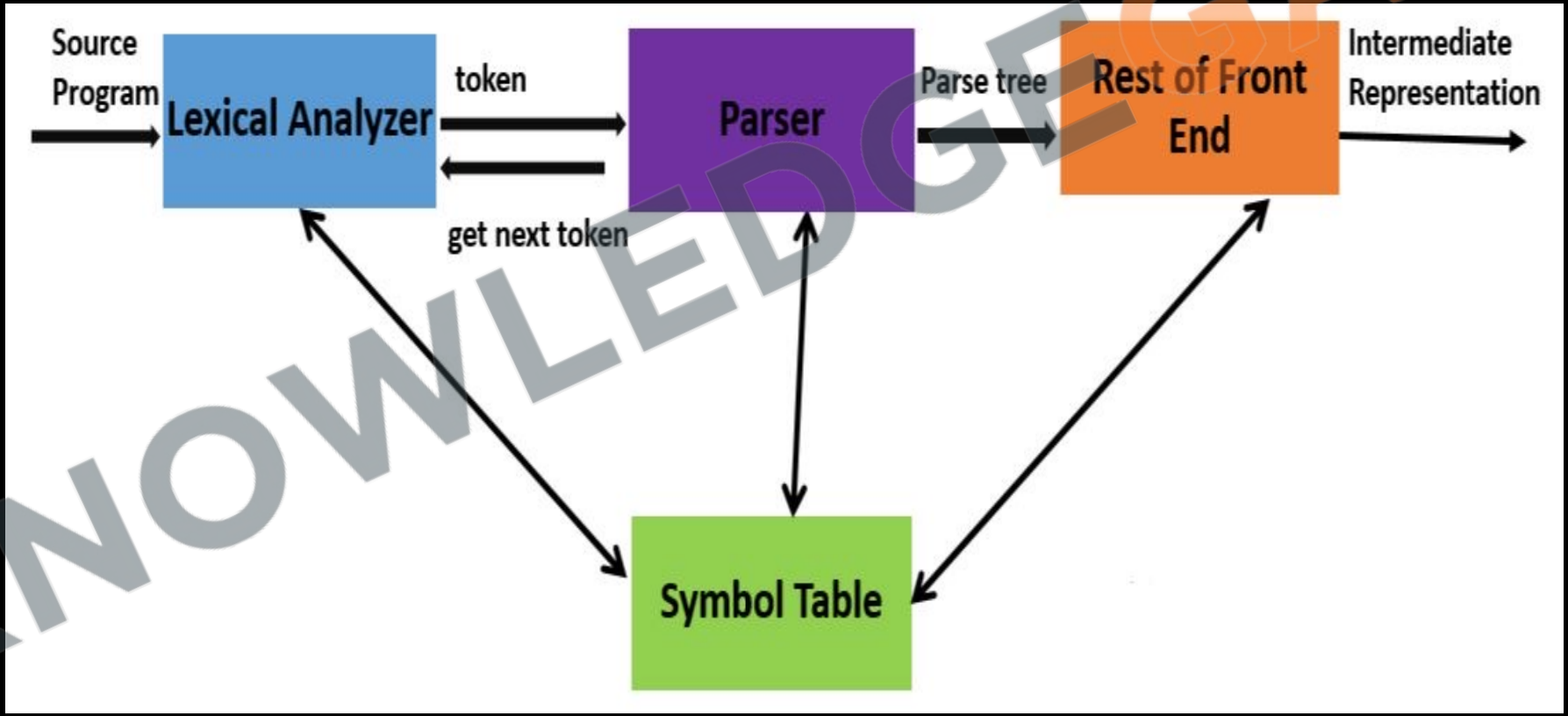
(BASIC PARSING TECHNIQUES): Parsers, Shift reduce parsing, operator precedence parsing, top down parsing, predictive parsers Automatic Construction of efficient Parsers: LR parsers, the canonical Collection of LR(0) items, constructing SLR parsing tables, constructing Canonical LR parsing tables, Constructing LALR parsing tables, using ambiguous grammars, an automatic parser generator, implementation of LR parsing tables.

Syntax analysis

- In Syntax analysis input is stream of tokens and output is syntax tree. The process of construction of the parse tree/ syntax tree/ derivation tree is called as parsing.



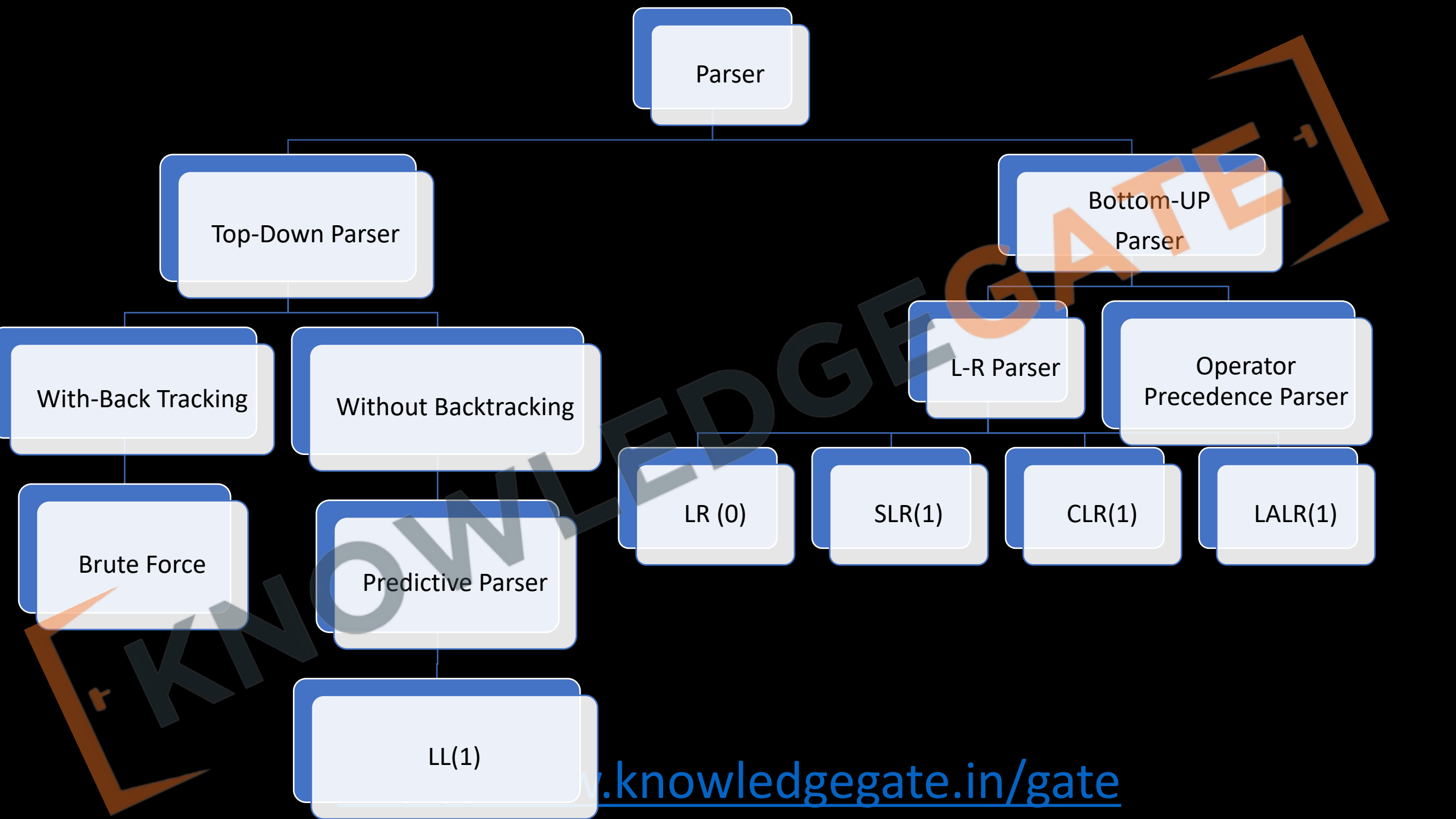
- For any input string which is given in the form of stream of tokens for the parser, if the derivation tree exists, then the input string is syntactically or grammatically correct.
- If the parser cannot generate the derivation tree from the i/p string, then there must be some grammatical mistakes in the string.



Classification of parser

- The program which perform parsing is known as parser or syntax analyzer.

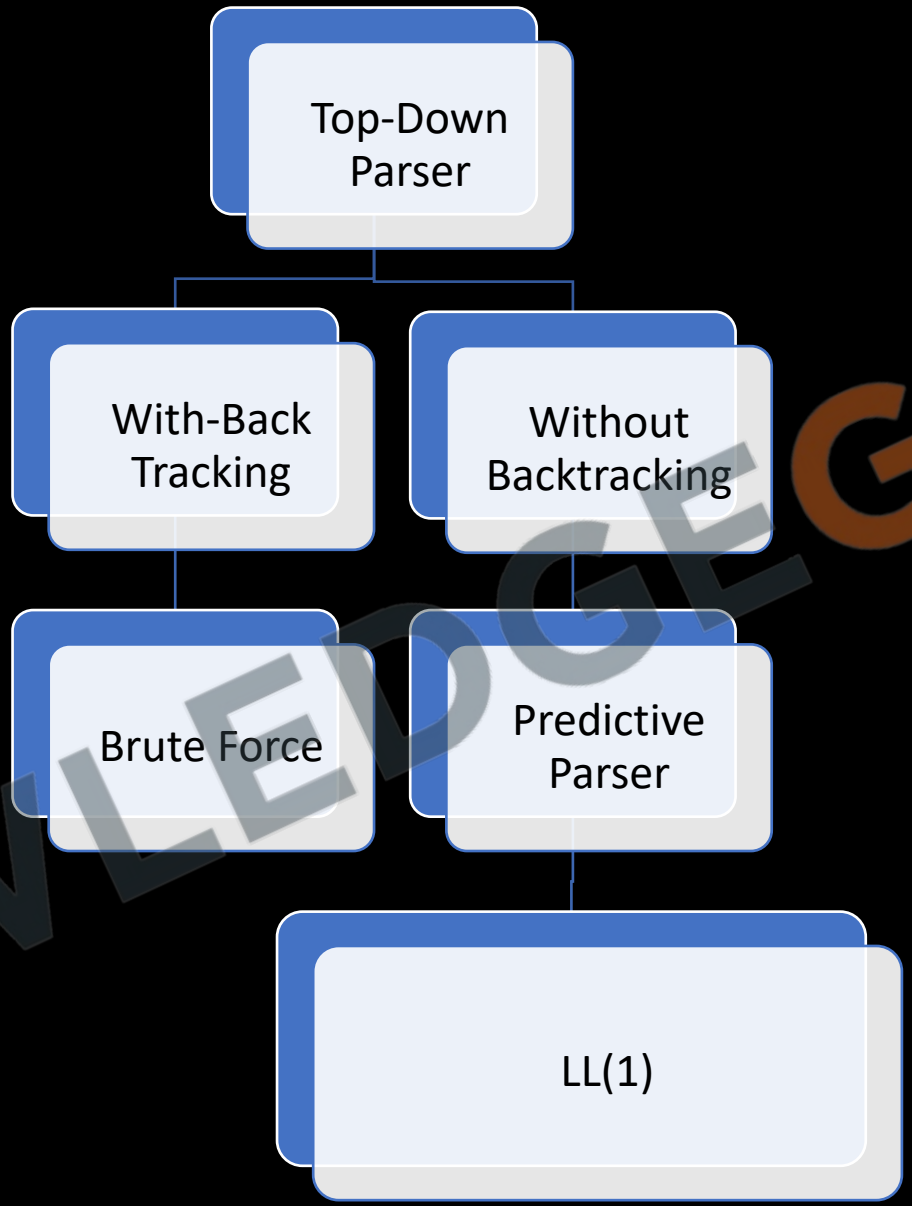
- There are two types of parser
  - Top-Down Parser
  - Bottom Up Parser



Top down parsing

- The process of construction of parse tree, starting from root and process to children, is known as TOP down parsing, i.e. getting the i/p string by starting with a start symbol of the grammar is top down parsing.

$A \rightarrow aA / \in$



Non-Deterministic Grammar

- The grammar with common prefix is known as Non-Deterministic Grammar.
  - $A \rightarrow \alpha\beta_1 / \alpha\beta_2$

- The grammar with common prefixes requires a lot of Back-tracking, back-tracking is very time consuming.

- To avoid the back-tracking, we need to remove the common prefixes, i.e. we need to convert the non-deterministic Grammar into Deterministic.

- **Left Factoring**: - The process of conversion of Non-Deterministic grammar into deterministic grammar is known as Left-Factoring.

- $A \rightarrow \alpha\beta_1 / \alpha\beta_2$
  - $A \rightarrow \alpha\beta$
  - $A \rightarrow \beta_1 / \beta_2$

Recursive production: - the production which has same variable both at left- and right-hand side of production is known as recursive production.

S→aSb

S→aS

S→Sa

Recursive grammar: - the grammar which contains at least one recursive production is known as recursive grammar.

S→aS / a

S→Sa / a

S→aSb / ab

Left Recursive Grammar: - The grammar G is said to be left recursive, if the Left most variable of RHS is same as the variable at LHS.

Right Recursive Grammar: - The grammar G is said to be right recursive, if the right most variable of RHS is same as the variable at LHS.

General recursion: - the recursion which is neither left nor right is called as general recursion. If a CFG generates infinite number of string then it must be a recursive grammar.

Non recursive grammar: - the grammar which is free from recursive production is called as non-recursive grammar.

S→AaB

A→a

B→b

- If a CFG contains left recursion then the compiler may go to infinite loop, hence to avoid the looping of the compiler, we need to convert the left recursive grammar into its equivalent right recursive production.

$A \rightarrow A\alpha / \beta_1 / \beta_2 \text{-------} / \beta_n$

$A \rightarrow A\alpha_1 / A\alpha_2 /----------A\alpha_n / \beta$

$A \rightarrow A\alpha_1 / A\alpha_2 /\text{---------}A\alpha_n / \beta_1 / \beta_2 \text{-------} / \beta_m$

Ambiguous grammar: - The grammar CFG is said to be ambiguous if there are more than one derivation tree for any string i.e. if there exist more than one derivation tree (LMDT or RMDT), the grammar is said to be ambiguous.

S → aS/Sa/a

- Grammar which is both left and right recursive is always ambiguous, but the ambiguous grammar need not be both left and right recursive.

- Top down parser uses, left most derivation.
- **Left most derivation**: - The process of construction of parse tree by expanding the left most non terminal is known as LMD and the graphical representation of LMD is known as LMDT (left most derivation tree).

$$A \rightarrow AaA \mid \in$$

- The TDP is constructed for the grammar, if it is free from left recursion.

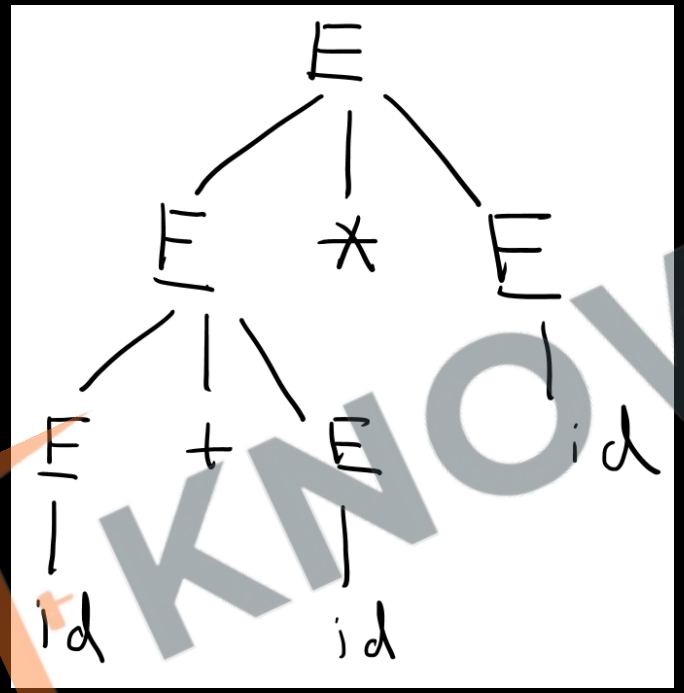
- **Left Recursive Grammar**: - The grammar G is said to be left recursive, if the Left most variable of RHS is same as the variable at LHS.

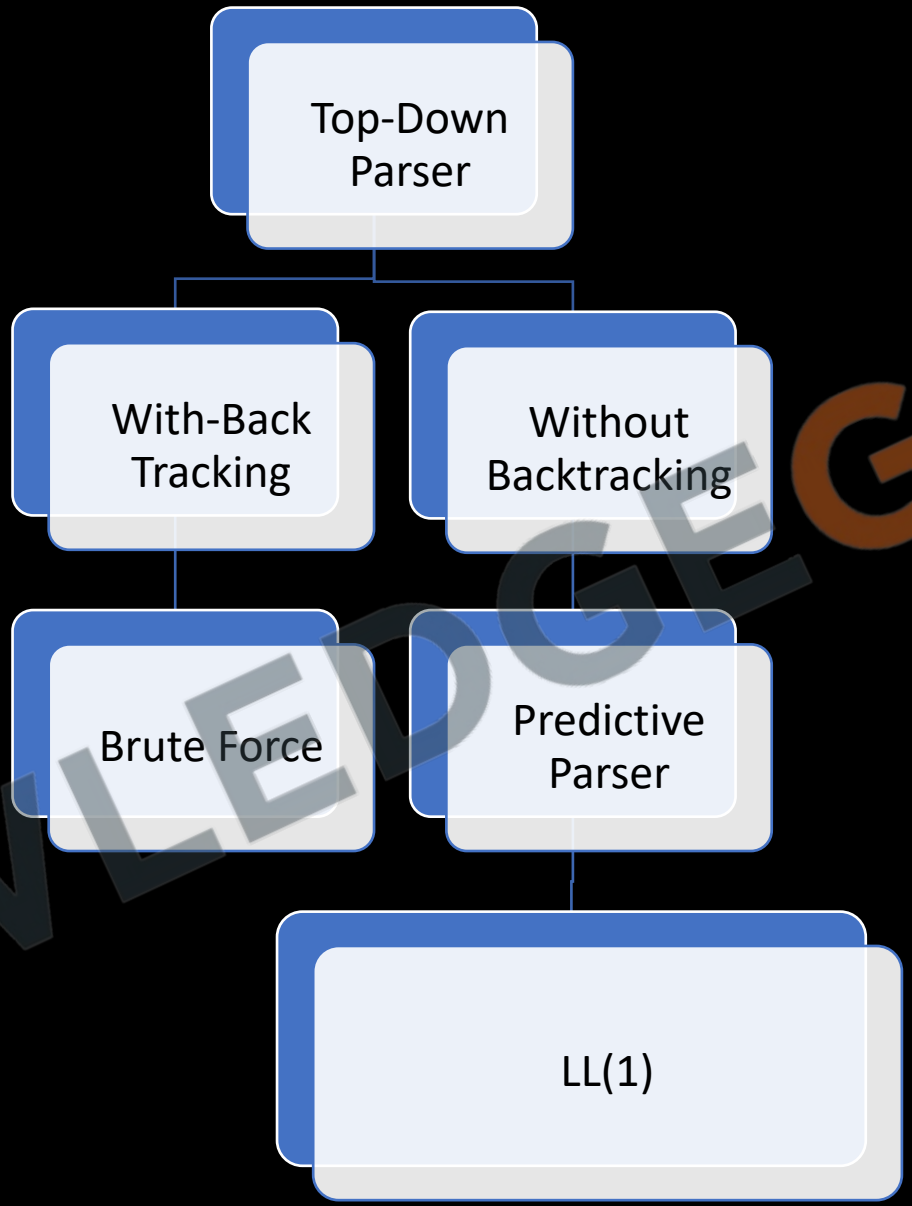
$$A \rightarrow Aa \,/\, \in$$

- The TDP is constructed for the grammar, if it is free from ambiguity.

- **Ambiguous grammar**: - The grammar CFG is said to be ambiguous if there are more than one derivation tree for any string i.e. if there exist more than one derivation tree (LMDT or RMDT), the grammar is said to be ambiguous.

E → E + E / E * E / E = E / id





Brute force technique

- Whenever a non- terminal is expanding first time, then go with the first alternative and compare with the i/p string. if does not matches, go for the second alternative and compare with i/p string, if does not matches go with the 3rd alternative and continue with each and every alternative.

- if the matching occurs for at least one alternative, then the parsing is successful, otherwise parsing fail.

S → cAd

A → ab / a

w_1 = cad

w_2 = cada

S → aAc / aB

A → b / c

B → ccd / ddc

w = addc

- TDP may be constructed for both left factor and non-left factor grammar.
- If the grammar is non- deterministic, then we use brute technique and if the grammar is deterministic, then we go with the predictive parser.
- Brute force requires lot of back-tracking, takes $O(2^n)$
- Back Tracking is very costly & reduces the performance of parser
- Debugging is very difficult.

- The TDP is constructed for a grammar, if there is less complexity, i.e. if the complexity of grammar is more than the parsing mechanism is very slow and performance is low.

- Worst case time complexity may go up to $O(n^4)$ in TDP except (Brute force).

- First(α) is a set of all terminals that may be in beginning in any sentential form, derived from α

- FIRST(α) is calculated for both Terminals and Non-Terminals

- **if** α is a terminal, then
  - **First(α) = {α}**

- **if** α is a string of terminal e.g. abc
  - **First(abc) = {a}**

- **if** α is a non - terminal, defined by α → ε, then
  - **First(α) = {ε}**

- **if** α is a non - terminal, defined by α → β, β ∈ T
  - **First(α) = {β}**

- **if** α is a non - terminal, defined by α → X_1 X_2 X_3, then
  - **First(α) = First(X_1)** **iff** X_1 →! ∈
  - **First(α) = [First(X_1) U First(X_2)] -** ∈ **iff** X_1 → ∈ && **iff** X_2 →! ∈
  - **First(α) = [First(X_1) U First(X_2) U First(X_3)] -** ∈ **iff** X_1 → ∈ && X_2 → ∈ && **iff** X_3 →! ∈
  - **First(α) = First(X_1) U First(X_2) U First(X_2)** **iff** X_1 → ∈ && X_2 → ∈ && **iff** X_3 → ∈

Q Consider the following Grammar find the First for each of them?

S → a / b / ε

First(S) =

Follow(S) =

S → aA / bB

A → ε

B → ε

First(S) =

First(A) =

First(B) =

Follow(S) =

Follow(A) =

Follow(B) =

S → AaB / BA

A → a / b

B → d / e

First(S) =

First(A) =

First(B) =

Follow(S) =

Follow(A) =

Follow(B) =

S → AaB

A → b / ε

B → c

First(S) =

First(A) =

First(B) =

Follow(S) =

Follow(A) =

Follow(B) =

S → AB

A → a / ε

B → b / ε

First(S) =

First(A) =

First(B) =

Follow(S) =

Follow(A) =

Follow(B) =

S → ABCDE

A → a / ε

B → b / ε

C → c / ε

D → d

E → e / ε

First(S) =

First(A) =

First(B) =

First(C) =

First(D) =

First(E) =

Follow(S) =

Follow(A) =

Follow(B) =

Follow(C) =

Follow(D) =

Follow(E) =

E → TE’

E’ → +TE’ / ε

T → FT’

T’ → *FT’ / ε

F → (E) / id

First(E) =

First(E’) =

First(T) =

First(T’) =

First(F) =

Follow(E) =

Follow(E’) =

Follow(T) =

Follow(T’) =

Follow(F) =

S → aBDh

B → cC

C → bC / ε

D → EF

E → g / ε

F → f / ε

First(S) =

First(B) =

First(C) =

First(D) =

First(E) =

First(F) =

Follow(S) =

Follow(B) =

Follow(C) =

Follow(D) =

Follow(E) =

Follow(F) =

Follow

Follow(A) is the set of all terminals that may follow to the right of (A) in any form of sentential Grammar.

Rules:

1) if A is the start symbol then Follow(A) = {$}

2) if A → αAβ, β →! ∈
   Follow(A) = First(β)

3) if S → αA
   Follow(A) = Follow(S)

4) S → αAβ, where β → ∈
   Follow(A) = First(β) U Follow(S) - ∈

$S \rightarrow aA$

$A \rightarrow bA / \varepsilon$

S → AaBb / BbAa

A → ε

B → ε

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

S → aAb

A → Ba / b

B → d

$S \rightarrow S0S1 / \varepsilon$

S → AaAb / BaBb

A → ε

B → ε

Follow

Follow(A) is the set of all terminals that may follow to the right of (A) in any form of sentential Grammar.

Rules:

1) if A is the start symbol then Follow(A) = {$}

2) if A → αAβ, β →! ∈
   Follow(A) = First(β)

3) if S → αA
   Follow(A) = Follow(S)

4) S → αAβ, where β → ∈
   Follow(A) = First(β) U Follow(S) - ∈

E → TE’

E’ → +TE’ / ∈

T → FT’

T’ → *FT’ / ∈

F → (E) / id

id	+	id	*	id	$

	+	*	(	)	id	$
E						
E’						
T						
T’						
F						

Q Consider a given Grammar LL(1) grammar design Parsing table and perform complete parsing table?

$E \rightarrow TE'$

$E' \rightarrow +TE' / \in$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \in$

$F \rightarrow (E) / id$

$w = id+id*id$

Stack	i/p	Action
$	id+id*id$	Push E
$ E	id+id*id$	Use Production E→TE' POP E and Push E'T
$ E'T	id+id*id$	Use Production T→FT' POP T and Push T'F
$ E'T'F	id+id*id$	Use Production F→id POP F and Push id
$ E'T'id	id+id*id$	Match Pop id and Increment Look Ahead Pointer
-	-	-
-	-	-
-	-	-
-	-	-
$	$	Accepted

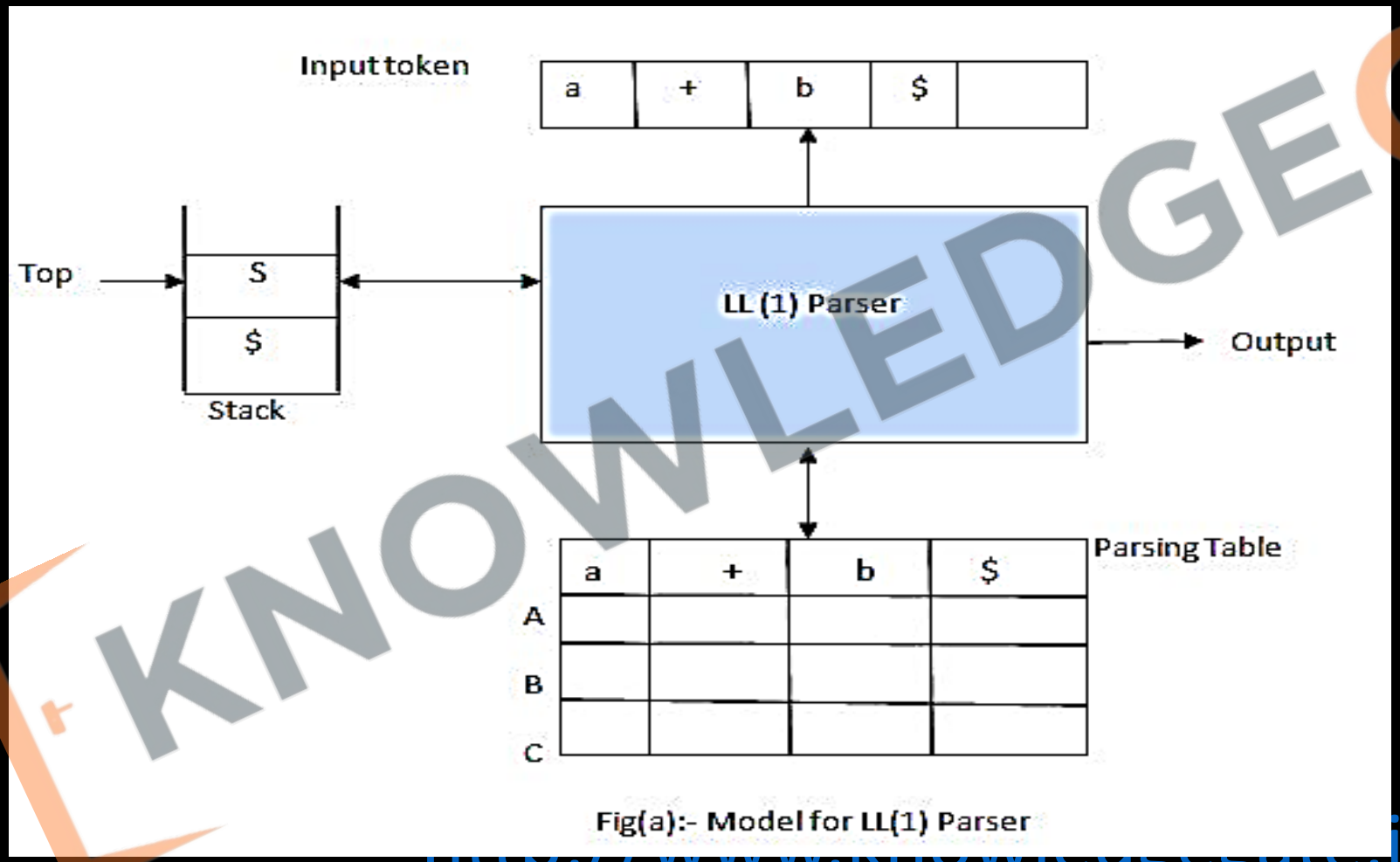
- LL(1)
  - First L means Left to right scanning
  - Second L means Left most derivation
  - 1 means no of look ahead symbol

- To predict the required production, to extend the parse tree, LL(1) parser depends on current processing symbol.

- The current processing symbol is called as Look-ahead-symbol.

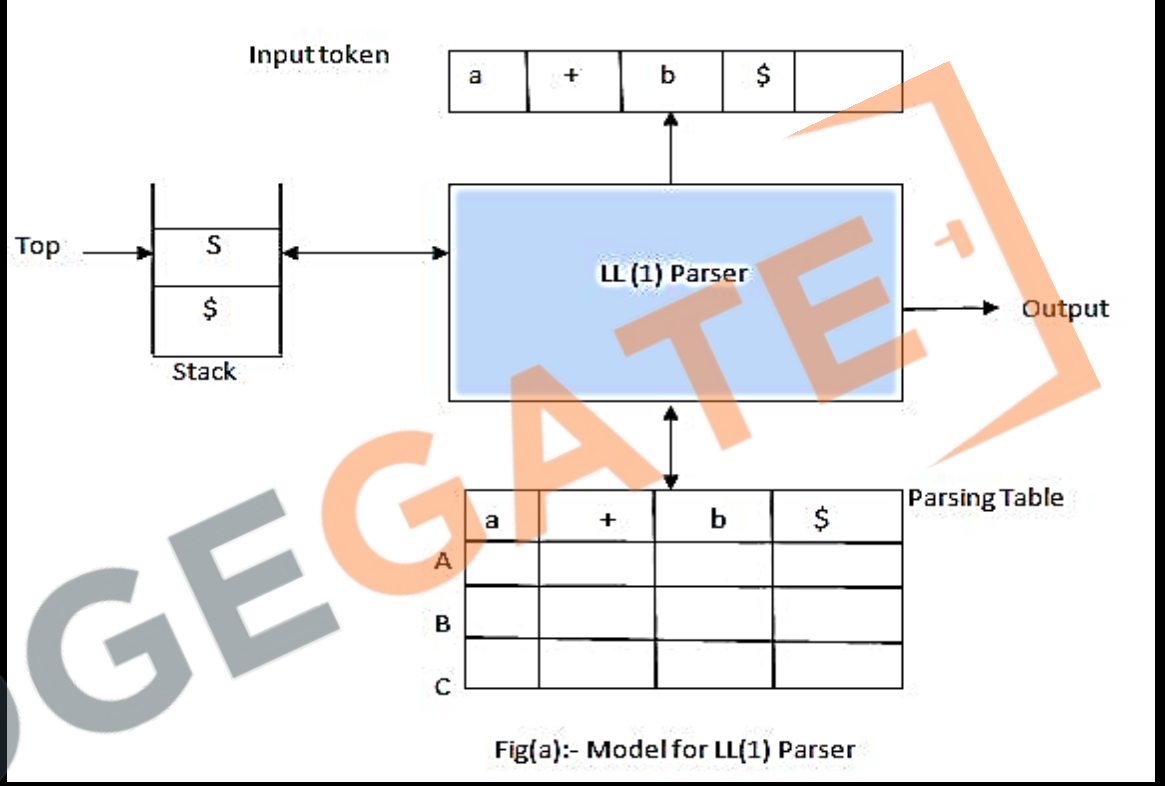
Block-Diagram of LL(1) Parser

- LL(1) parser consist of 3 components
  - Input Buffer
  - Parse Stack
  - Parse Table



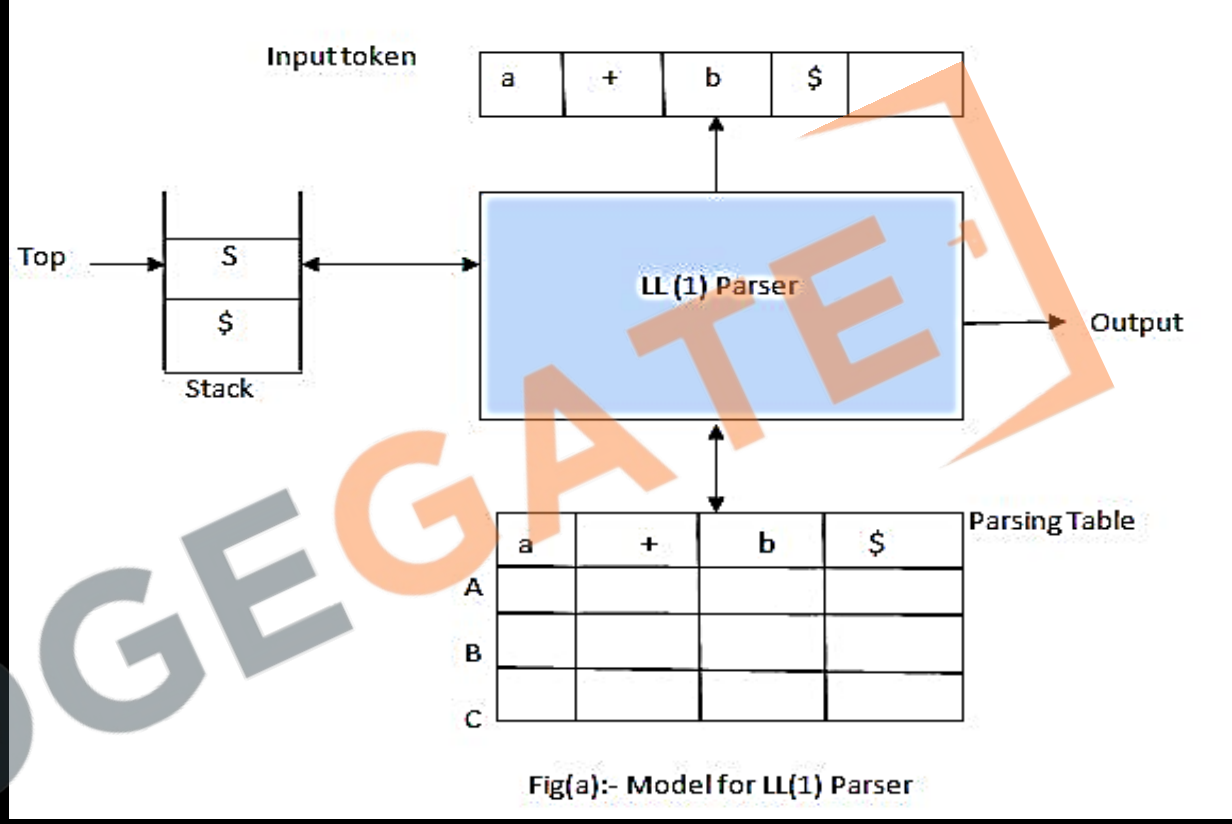
Fig(a):- Model for LL(1) Parser

- **Input Buffer**
  - it is divided into finite no of cells and each cell is capable of holding only one i/p symbol.
  - input buffer contains only i/p string at any point of time.
  - the tape header is always pointing only one look ahead symbol and after parsing the current look ahead symbol, the header moves to next cell towards right side.
  - End of the string is recognized by $



Fig(a):- Model for LL(1) Parser

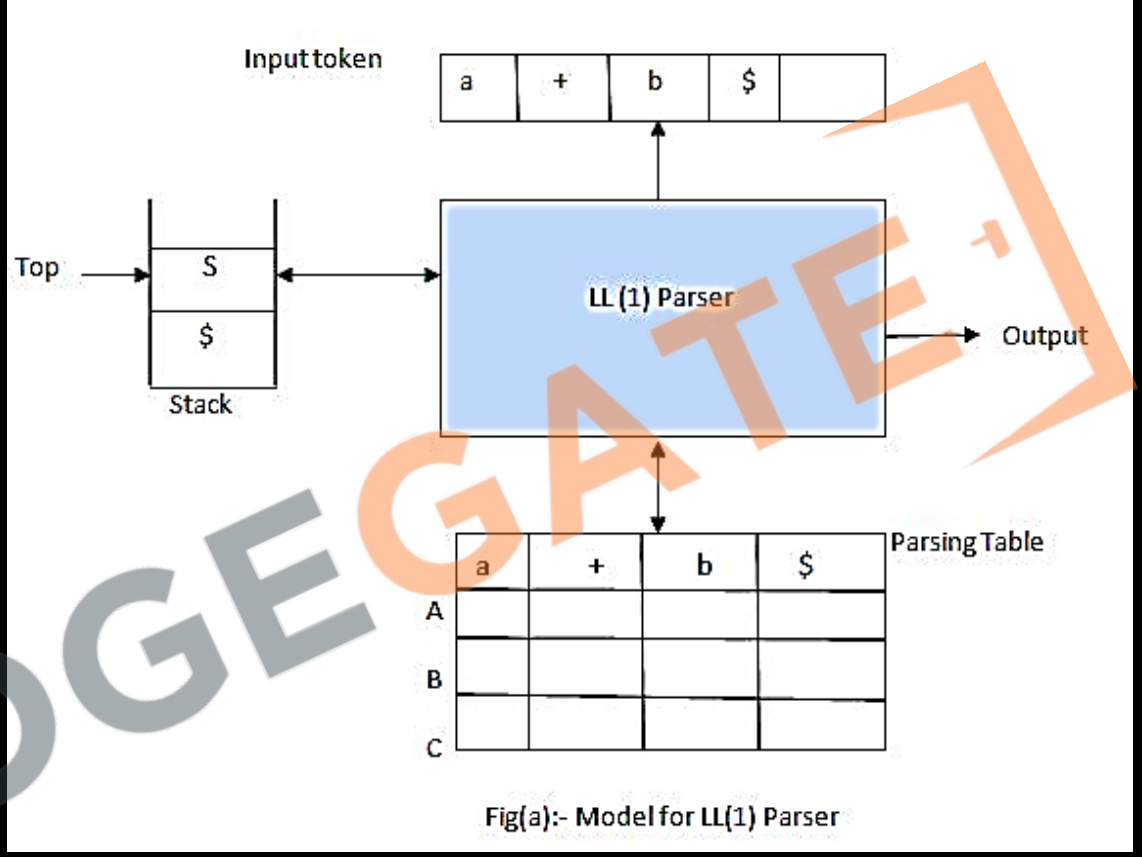
- **Parse Stack**
  - it contains the grammar symbol, the grammar symbol is pushed into stack or POP from the stack based on the occurrence of matching.
  - if the topmost symbol of the stack is matching with look ahead symbol, then the grammar symbol is POP out from the stack.
  - if the TOP most symbol of the stack is not matching with the look ahead symbol, then the grammar symbol is Pushed into stack.



Fig(a):- Model for LL(1) Parser

- **Parse table**
  - it is a two-dimensional array of order m*n where m = no of non-terminal and n = no of terminals + 1

  - parse table contains all the production which are used to contain the parse tree for that i/p string.



Fig(a):- Model for LL(1) Parser

- Procedure to constructed LL(1) parser Table:
  - for every production A → α, repeat the following steps
  - add A → α in M[A, α] for every terminal ‘α’ in first(A).
  - if First(α) contains ε, then add A → ε in M[A, b] for every symbol, b in Follow[A]

- **Parsing Process**
  - Push the start symbol into stack
  - Compare the top most symbol of the stack with the look ahead symbol
  - If matching occurs, then pop of the grammar symbol from the stack and increment the i/p pointer
  - O/p the production which is used for expanding a non- terminal, i.e. the result is a production which is used for push operation.

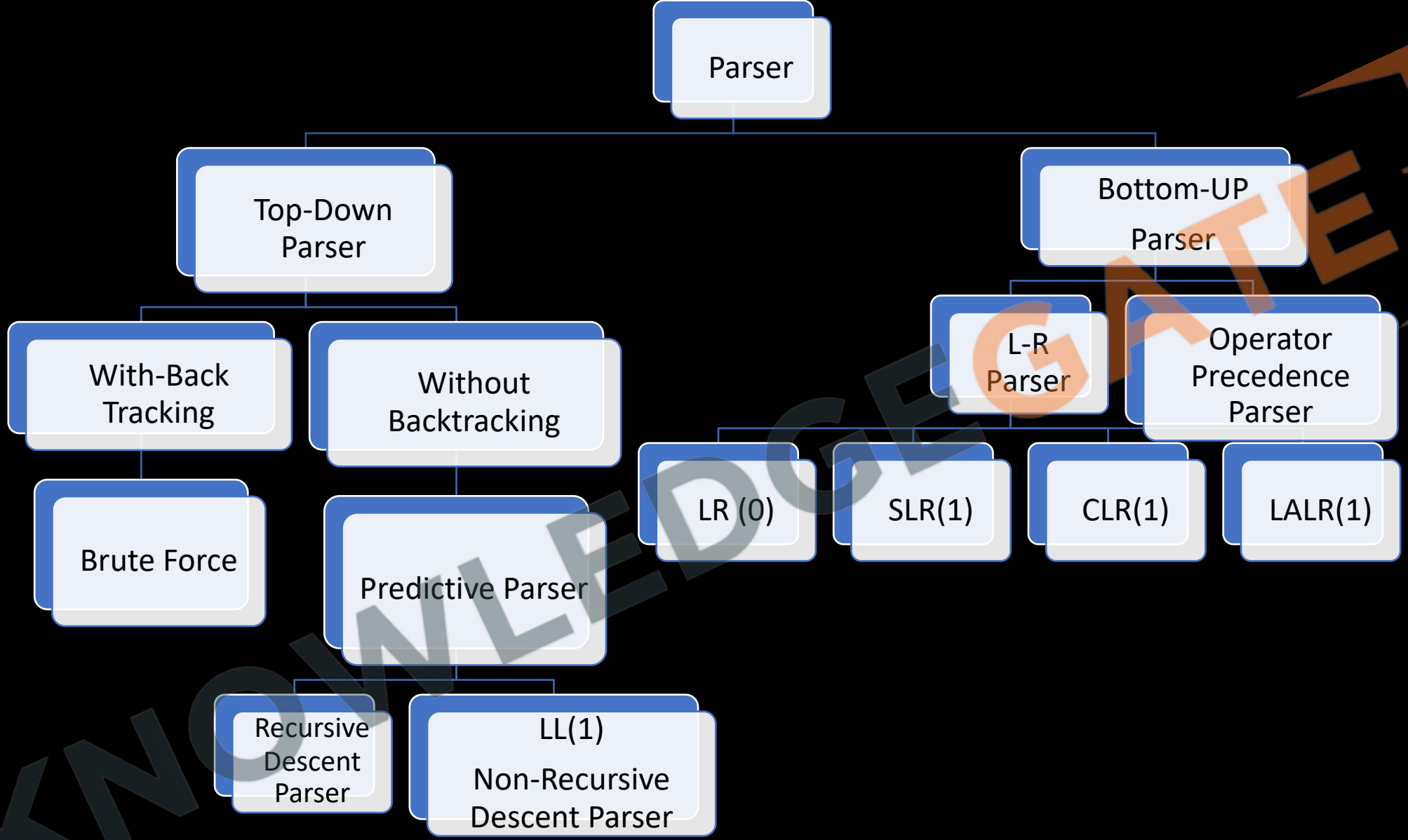
- **LL(1) parsing algorithm**
  - let x is a grammar symbol (stack symbol) and a is the look ahead symbol
  - if x == a == $, then the parsing is successful
  - if x==a! = $, then pop of and increment the i/p pointer
  - if x!= a!=$ and m[x, a] contain the production, x → abc, then replace x by abc in the reverse order and continue the process.
  - out of the production which is used for expanding the non-terminal, i.e. the production which is used for PUSH operation

- LL(1) Grammar: Grammar for which LL(1) parser can be constructed is known as LL(1) Grammar

- Or the Grammar whose LL(1) parse table does not contains multiple entries in the same cell, then the grammar is LL(1)

LL(1) or Table Driven Parser

- LL(1) is constructed for the Grammar, if
  - free from left recursion
  - free from ambiguity
  - left factored



Bottom Up parser

- The process of constructing the parse tree in the Bottom-Up manner, i.e. starting from the children & proceeding towards root.

- S → aABc
  A → b / bc
  B → d

- w = abcdc

- LR parsers were invented by Donald Knuth in 1965 as an efficient generalization of precedence parsers. Knuth proved that LR parsers were the most general-purpose parsers possible.

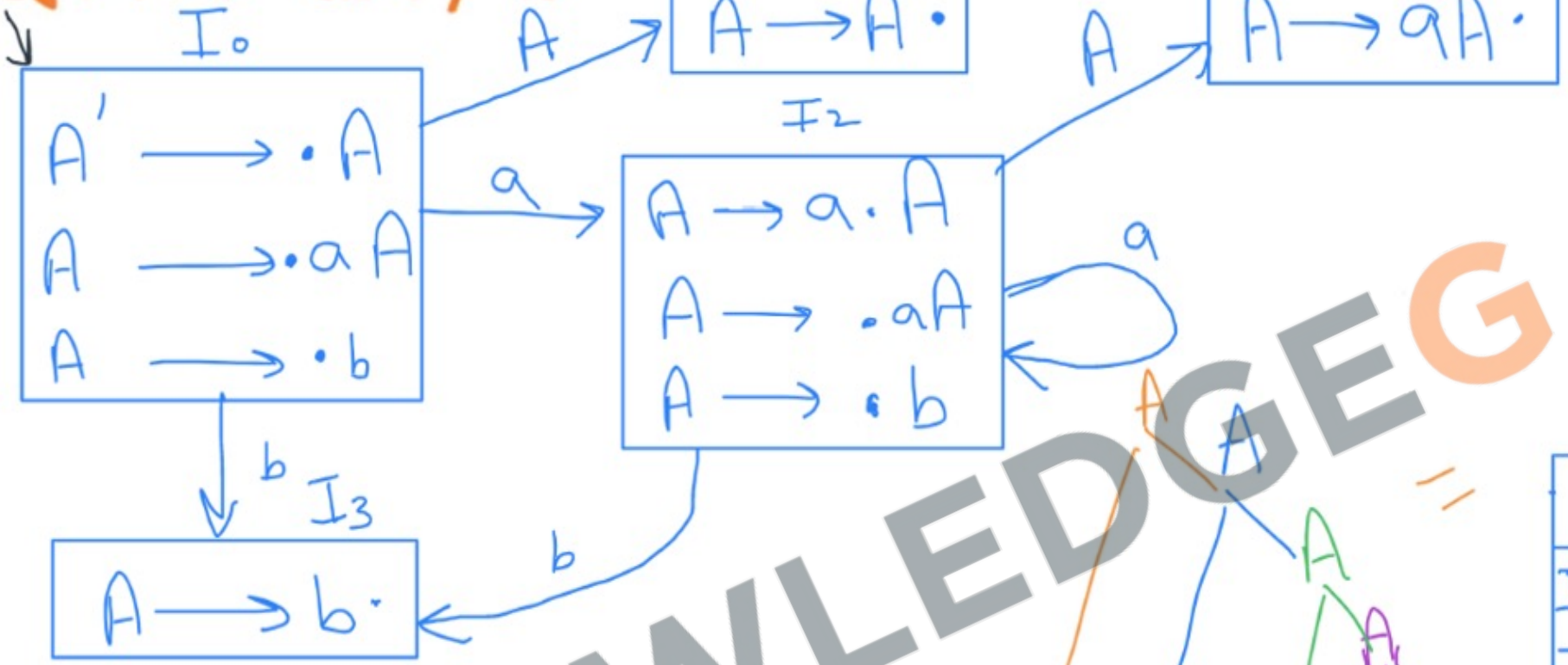
- **Donald Ervin Knuth** (born January 10, 1938) is an American computer scientist, mathematician, and professor emeritus at Stanford University. He is the 1974 recipient of the ACM Turing Award, informally considered the Nobel Prize of computer science. Knuth has been called the "father of the analysis of algorithms".

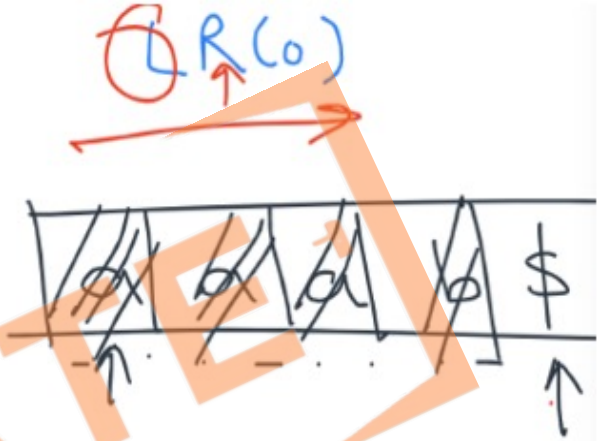
Q A → aA / b

a	a	a	b	$

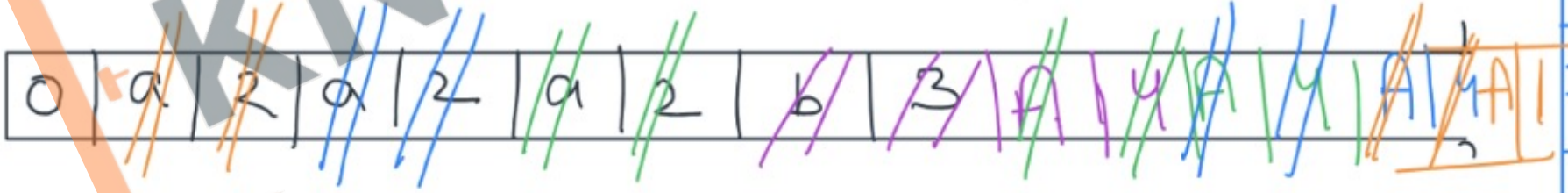
	Action			Goto

Q $A \rightarrow aA \;(r_1) \;/\; b \;(r_2)$





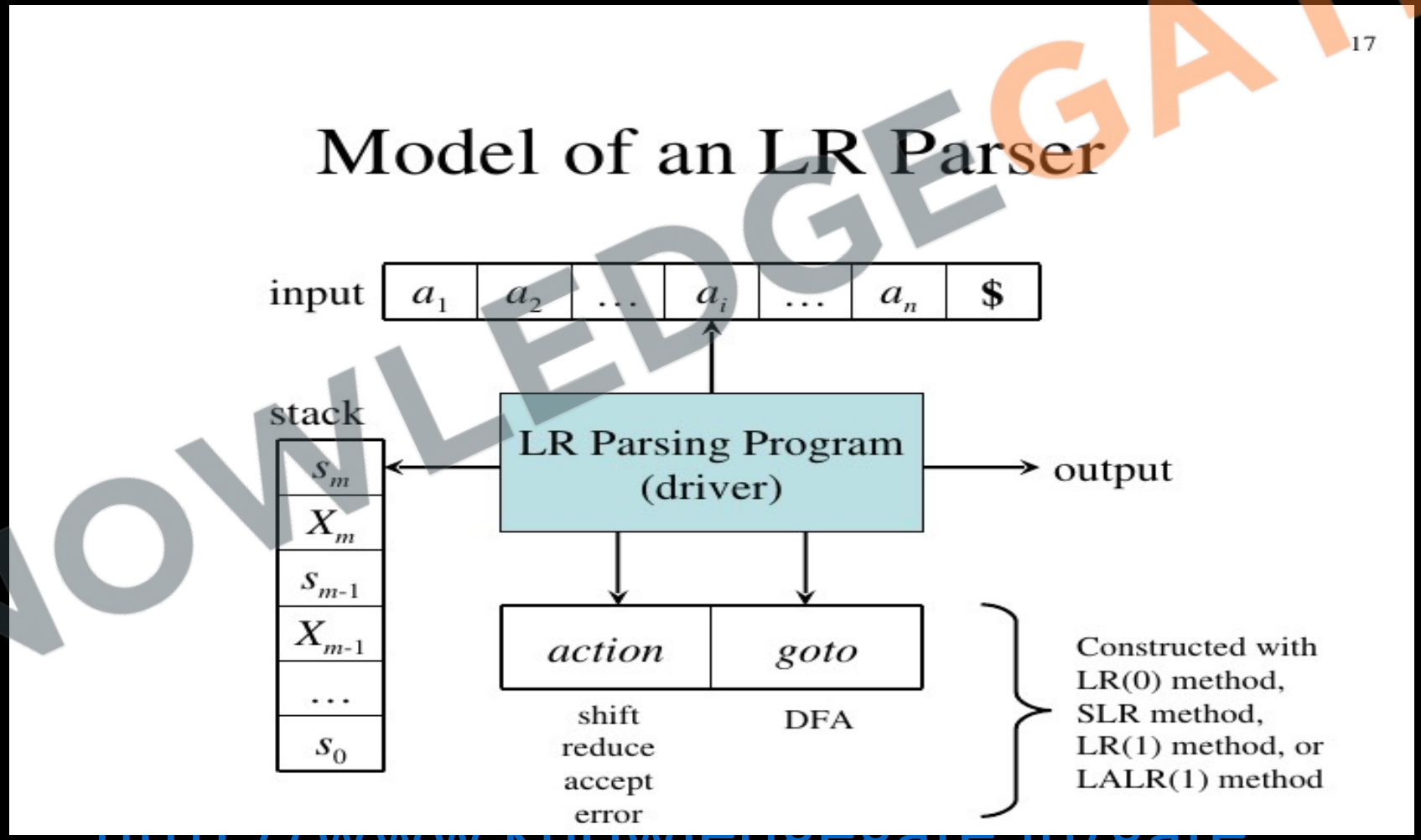
	Action			Goto
	a	b	$	A
I_0	S_2	S_3		S_1
I_1			accept	
I_2	S_2	S_3		S_4
I_3	r_2	r_2	r_2	
I_4	r_1	r_1	r_1	



- **Handle**: - Substring of the i/p string that matches with RHS of any production, is called as Handle.
- The process of finding the handle & replacing that handle by it's LHS variable is called Handle Pruning.
- Bottom-Up-Parser is also known as Shit-Reduced Parser.
- BUP can be constructed for both Ambiguous & Unambiguous grammar
  - Ambiguous →OPP
  - Unambiguous → LR(k)

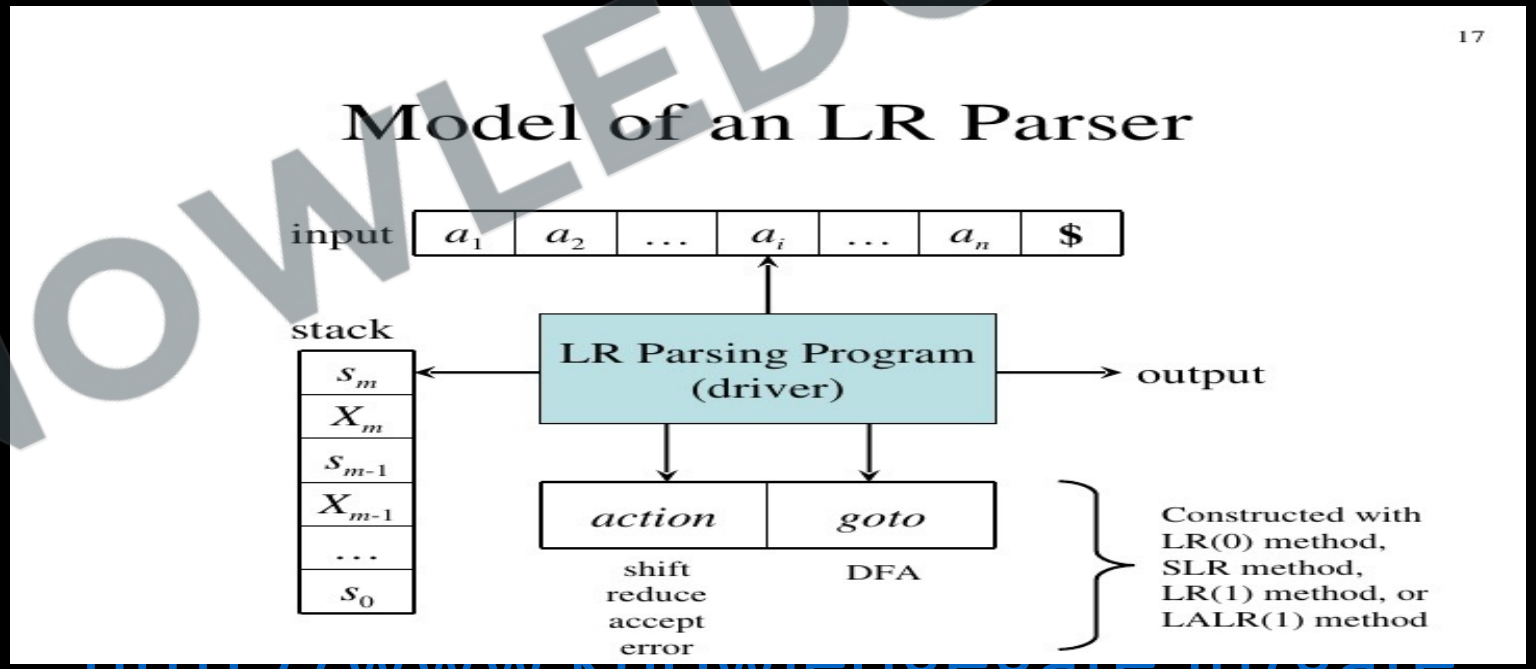
- LR(K) parser can be constructed for Unambiguous grammar.
- BUP Simulates the Reverse of Right Most Derivation.
- BUP can be constructed for the grammar which has more complexity.
- Bottom-up parsing is faster than Top-Up-Parsing, such that BUP is more efficient than TDP.

- Bottom Up Parser consist of three components
  - i/p buffer
  - Parse stack
  - Parse table

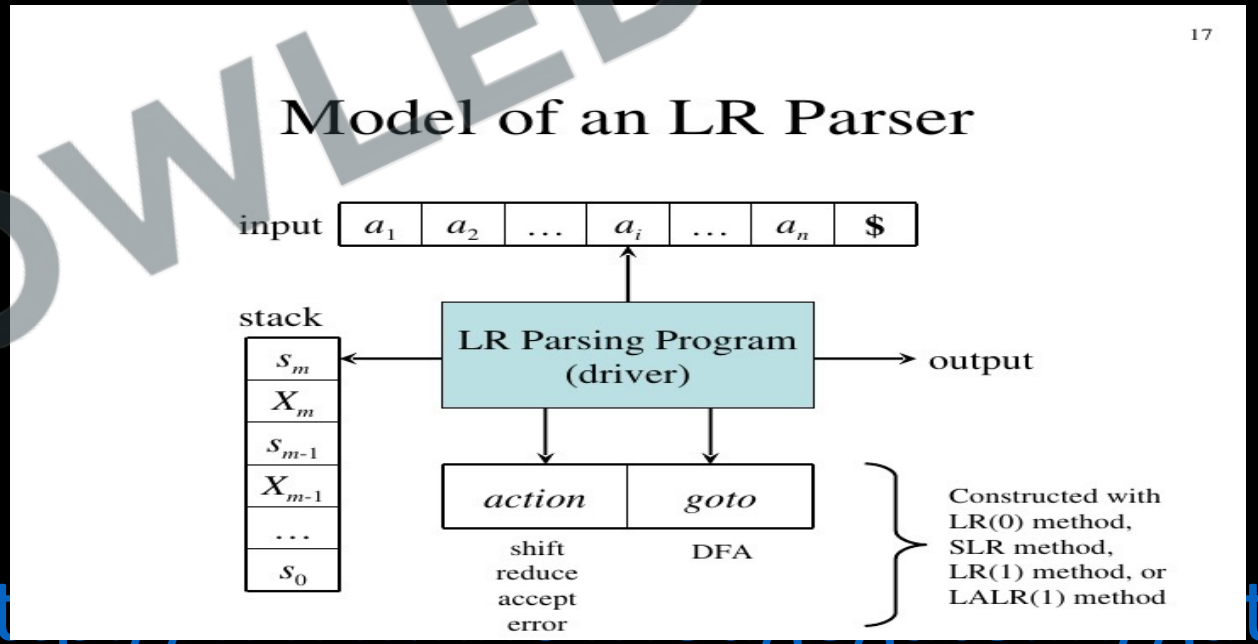


- **Input buffer**: divide into cells & each cell contains only one i/p symbol.

- **Stack**: stack contains the grammar symbol, the grammar symbol are push into stack or pop from the stack, using shift & reduced operation. if habdle occurs from the topmost symbol of the stack, then apply the reduced operation & if handle does not occurs in the topmost symbol of the stack, then apply the shift operation.



- **Parse table**: parse table is constructed using terminals, non-terminals & LR(0) items. this parse table consist of two parts:
  - Action
  - Goto
- Action part contains shift & reduced operation over the terminals
- Goto part consists of only Shift operation over the Non-terminals.



- Operation in shift/reduced passer
  - shift
  - reduced
  - accept
  - error

	Action	Goto
I_0	Terminals	Non-Terminals
I_{n-1}	Shift / Reduce	Shift

- **Shift**: shift operation can be used when handle does not occurs from the topmost symbol of the stack. using shift operation, will moving a look ahead symbol in stack.

- **Reduce**: reduce operation can be whenever handle occurs from the Topmost symbol from the stack. using reduced operation, we rename the topmost symbol of the stack that matches with look-ahead symbol.

- **Accept**: after scanning the complete i/p string from the i/p buffer, if the stack contains only the start symbol of the grammar as topmost symbol, then the i/p string is accepted and the parsing is successful.

- **Error**: after the complete i/p string, if the attack contains any symbol which is different from start symbol as a topmost symbol, then the parsing is unsuccessful and hence error.

- LR(K)
  - First L stand for left to right scanning
  - Second R stand for reverse of right most derivation
  - K is Look-Ahead symbol

- Procedure for the construction of LR Parser table:
  1. Obtain the augmented grammar for the given grammar
  2. Create the canonical collection of LR items or compiler items.
  3. Draw the DFA & prepare the table based on LR items.

- **Augmented grammar**
  - The grammar which is obtained by addition one more production that generate the start symbol of the grammar, is known as Augmented grammar.
  - S → AB A → a B→ b
  - S' → S S → AB A → a B→ b

- **LR(0) or Compiler item**
  - The production, which has dot(.) anywhere on RHS is known as LR(0) items.

  - A → abc

  - LR(0) items:
    - A → .abc
    - A → a.bc
    - A → ab.c
    - A → abc. Final / Completed items

- **Canonical Collection:**
  - The set $C = \{I_0, I_1, I_2, I_3, \ldots\ldots\ldots I_N\}$ is known as canonical collection of LR(0) items.

- Function used to generate LR(0) item's:
  - Closure: i/p set of items & o/p also set of items
  - GOTO

- Add everything from i/p to o/p
- If A → α. βB is in closure(I) & β → б is in the grammar G
  - Then add β → .б to the closure(I)
  - A → α. βB
  - β → .б
- Repeat the previous step for every newly added item

- Goto(I, X)
  - Goto(I, X)
  - Goto(A → α.Xβ, X) = A → αX.β

Procedure to construct LR parse Table

- LR parse table consist of two parts
  - Action
  - Goto

- **Action**
  - Action part consists of both shift & reduced operation that are performed on terminals.

- **Goto**
  - Goto parts consists of shift operation performed on Non-terminals

$Goto(I_i, X) = I_j$ (X is terminal)

	X
I_i	S_j

$Goto(I_i, X) = I_j$ (X is non-terminal)

	X
I_i	j

If I_i is any final item & represent the production R_i, then place R_i under all the terminal symbols in the action part of the table.

	t_1	t_2	t_3					t_n	$
I_i	r_i	r_i	r_i					r_i	r_i

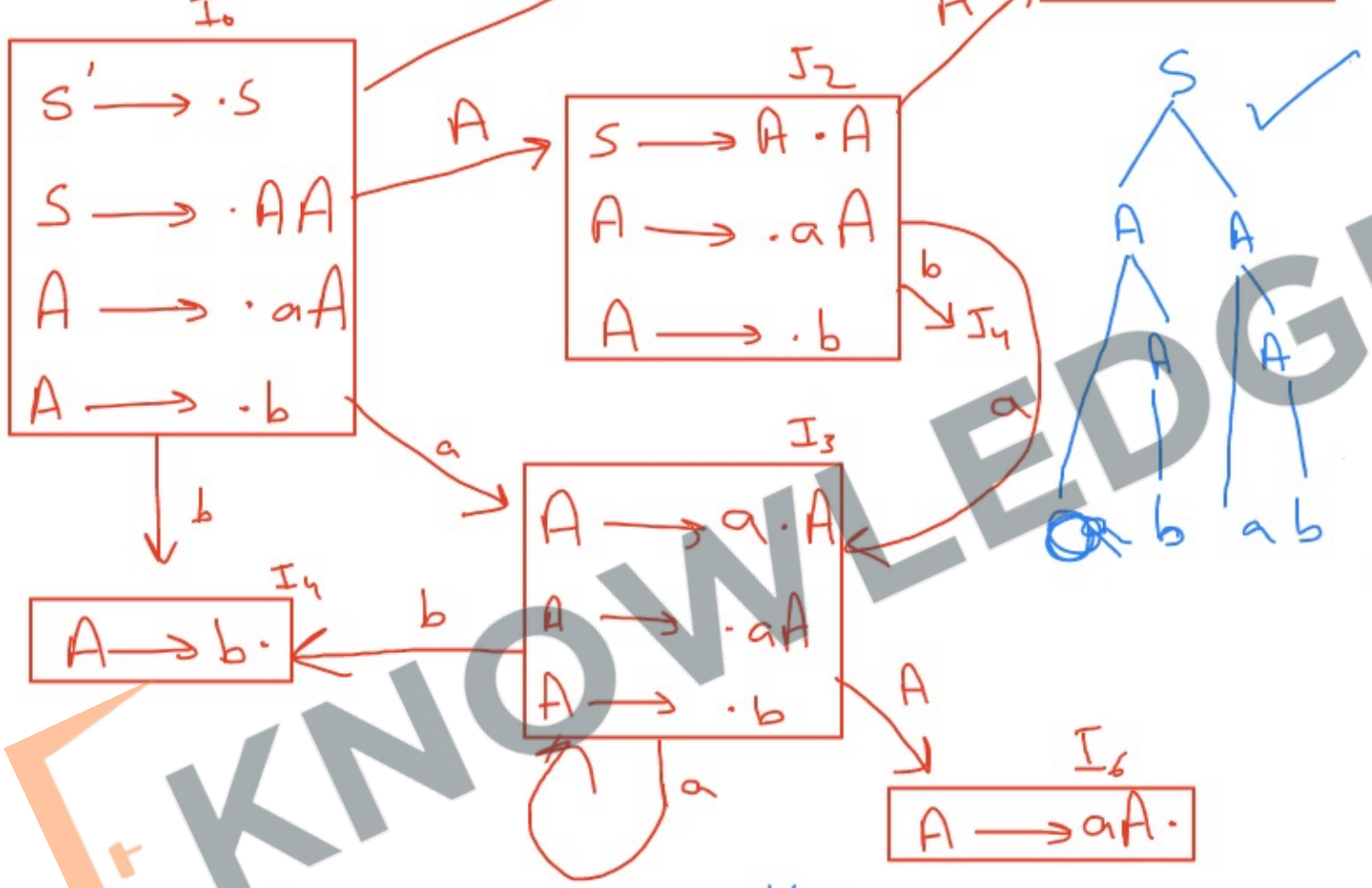
S → AA

A → aA / b

	Action			Goto	

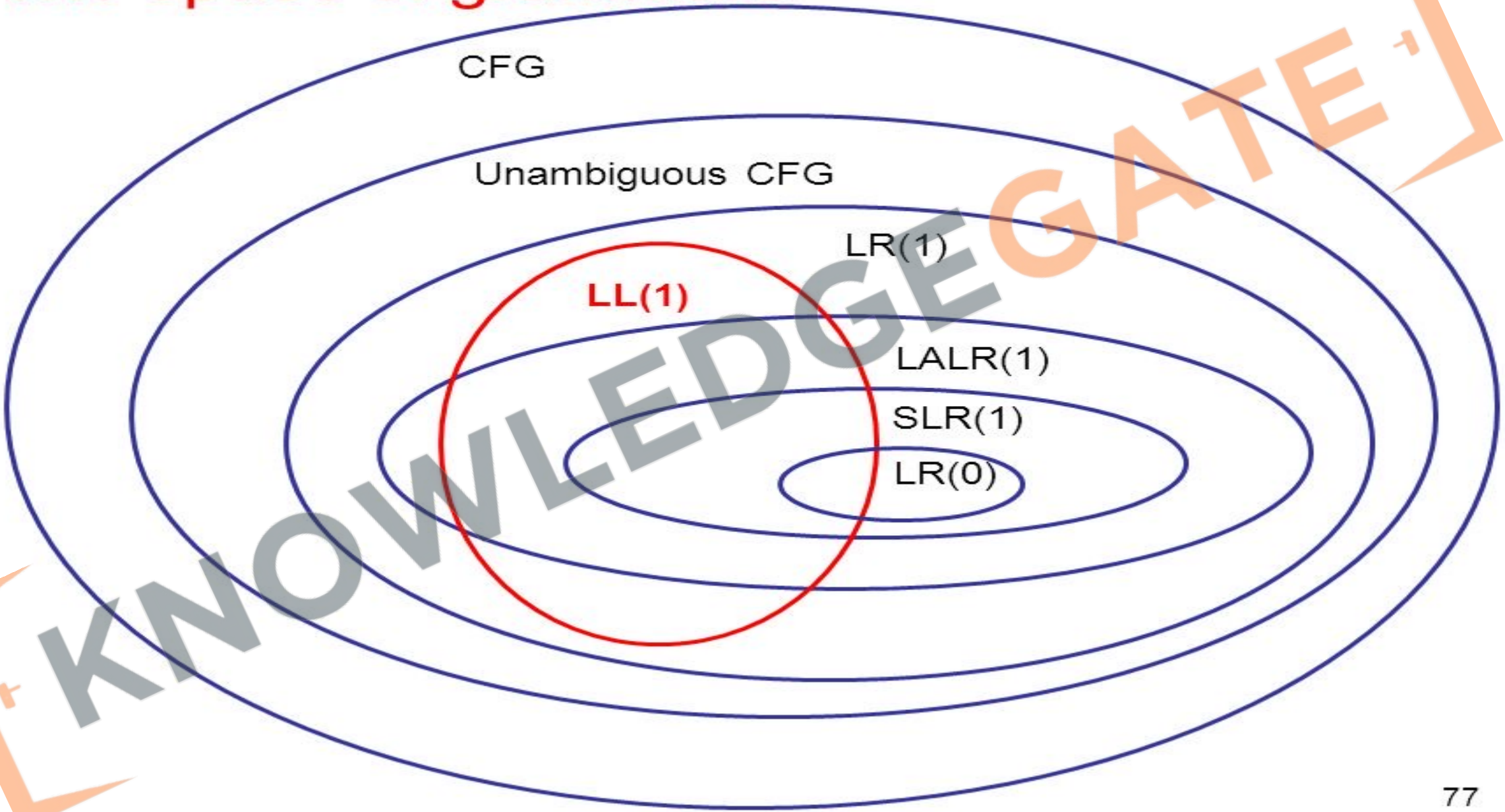
$S \rightarrow AA$ r_1

$A \rightarrow aA$ r_2 / b r_3



	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

The space of grammars



$E \rightarrow T + E \mid T$

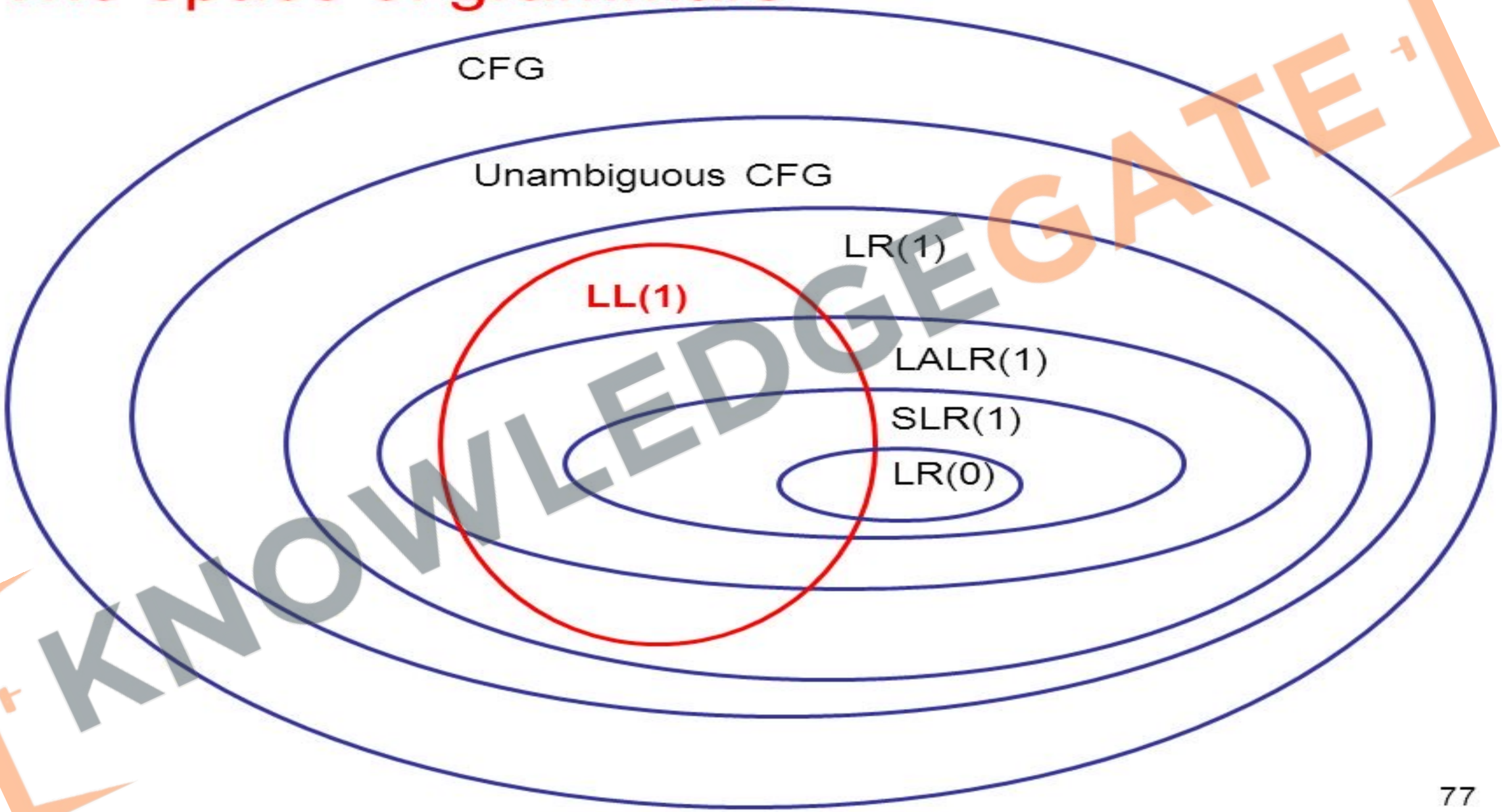
$T \rightarrow id$

	Action			Goto	

SLR(1)

- The procedure for constructing the parse table is similar to LR(0) parse, but there is a restriction in the reducing entries.
- Whenever there is a final item, then placed the reduced entries under the follow symbol of LHS symbol.
- If the SLR(1) parse table is free from multiple entries than the grammar is SLR(1) grammar.
- Every LR(0) grammar is SLR(1), but every SLR(1) grammar need not be LR(0)
- SLR(1) parser is more powerful then LR(0) parser

The space of grammars



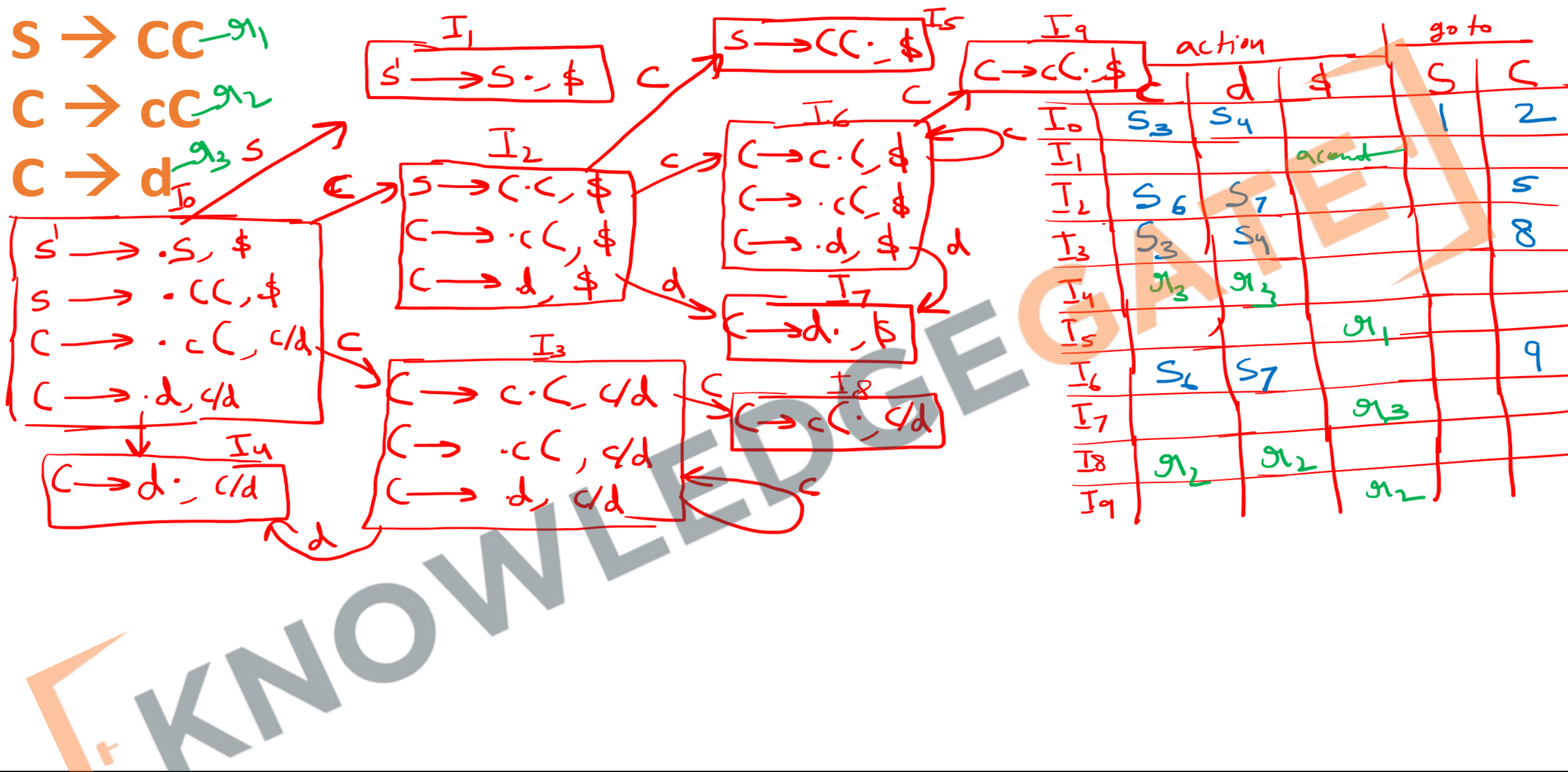
S → CC

C → cC

C → d

CLR(1)

	Action			Goto	



CLR(1)

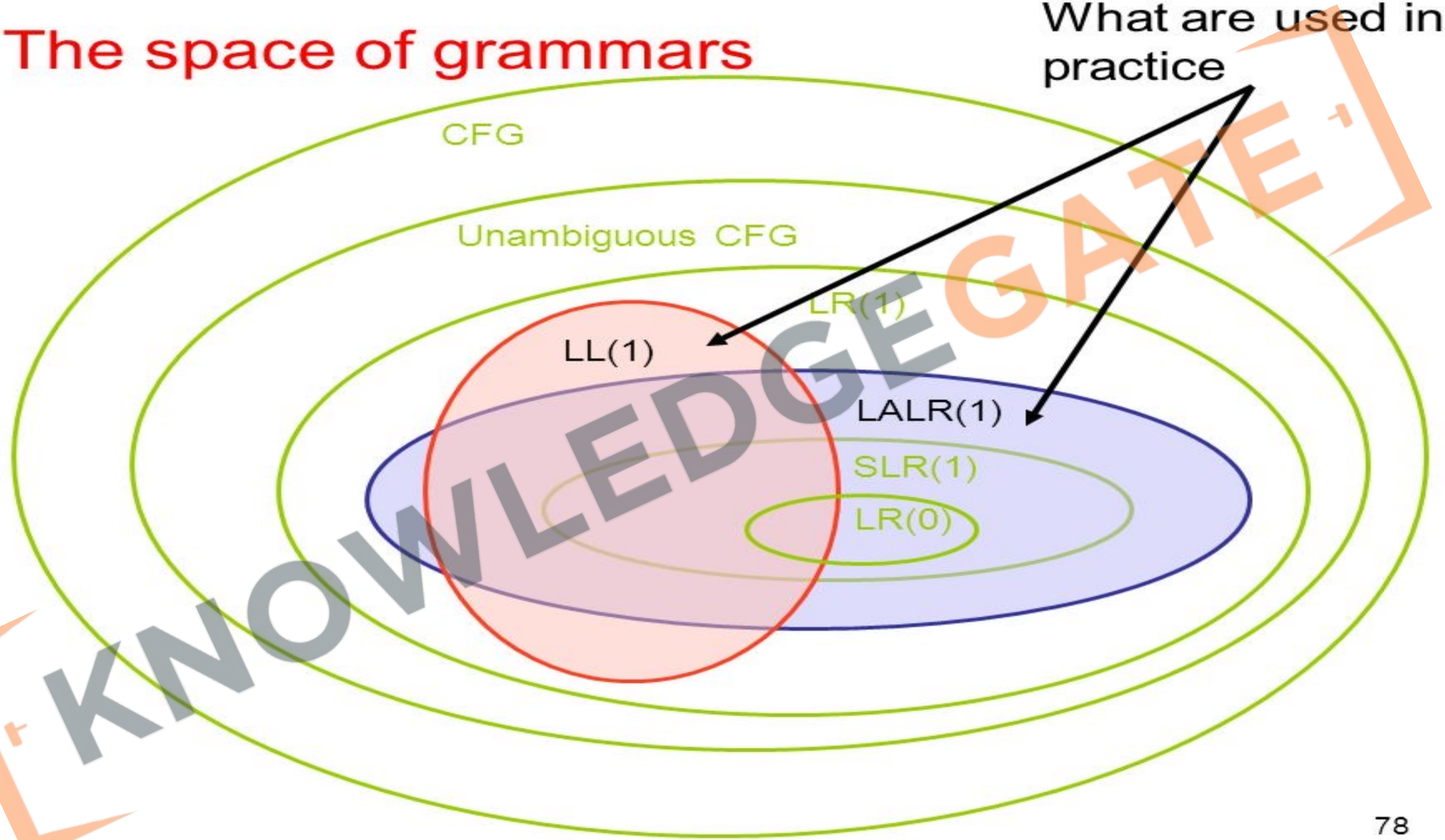
- LR(1) depends on one look Ahead symbol

- Closure(I):
  - Add everything from i/p to o/p
  - A → α.Bβ, $ is in closure I and β → б is in the grammar G, then add β → .б, first(β, $), to the closure I
  - repeat previous step for every newly added items

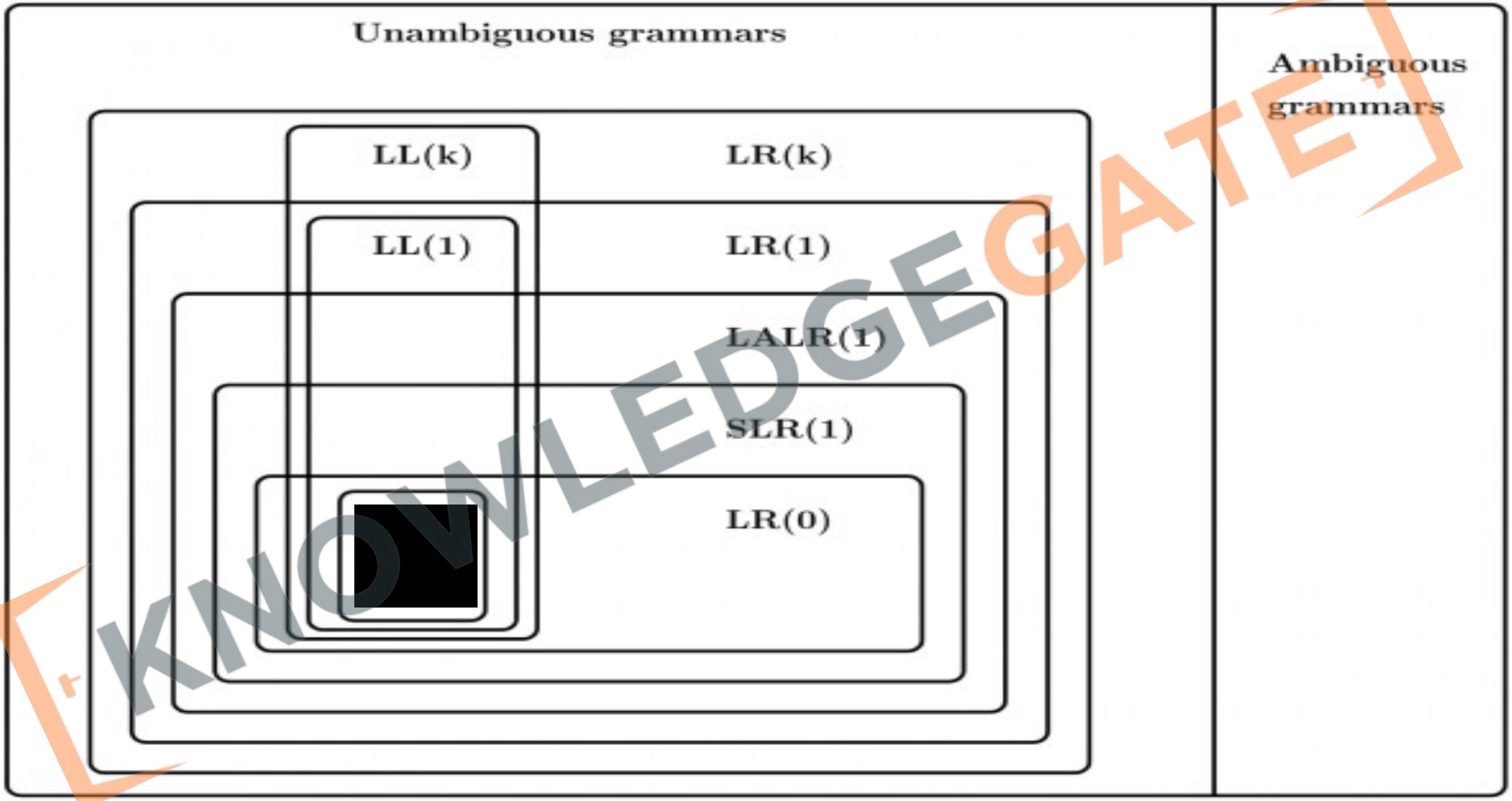
- Goto(I, x):
  - there will not be any change in the goto part while finding the transition.
  - these many be change in the follow or look Ahead part while finding the closure.

- LR(1) grammar: the grammar for, which LR(1) is constructed is known as LR(1) or CLR(1).
- the grammar whose LR(1) parse is free from multiple entries or conflicts, then it is LR(1) grammar.
- Every SLR(1) grammar is CLR(1). but every CLR(1) grammar need not be SLR(1)
- CLR(1) is more powerful than SLR(1)

The space of grammars



LL(1) versus LR(k)

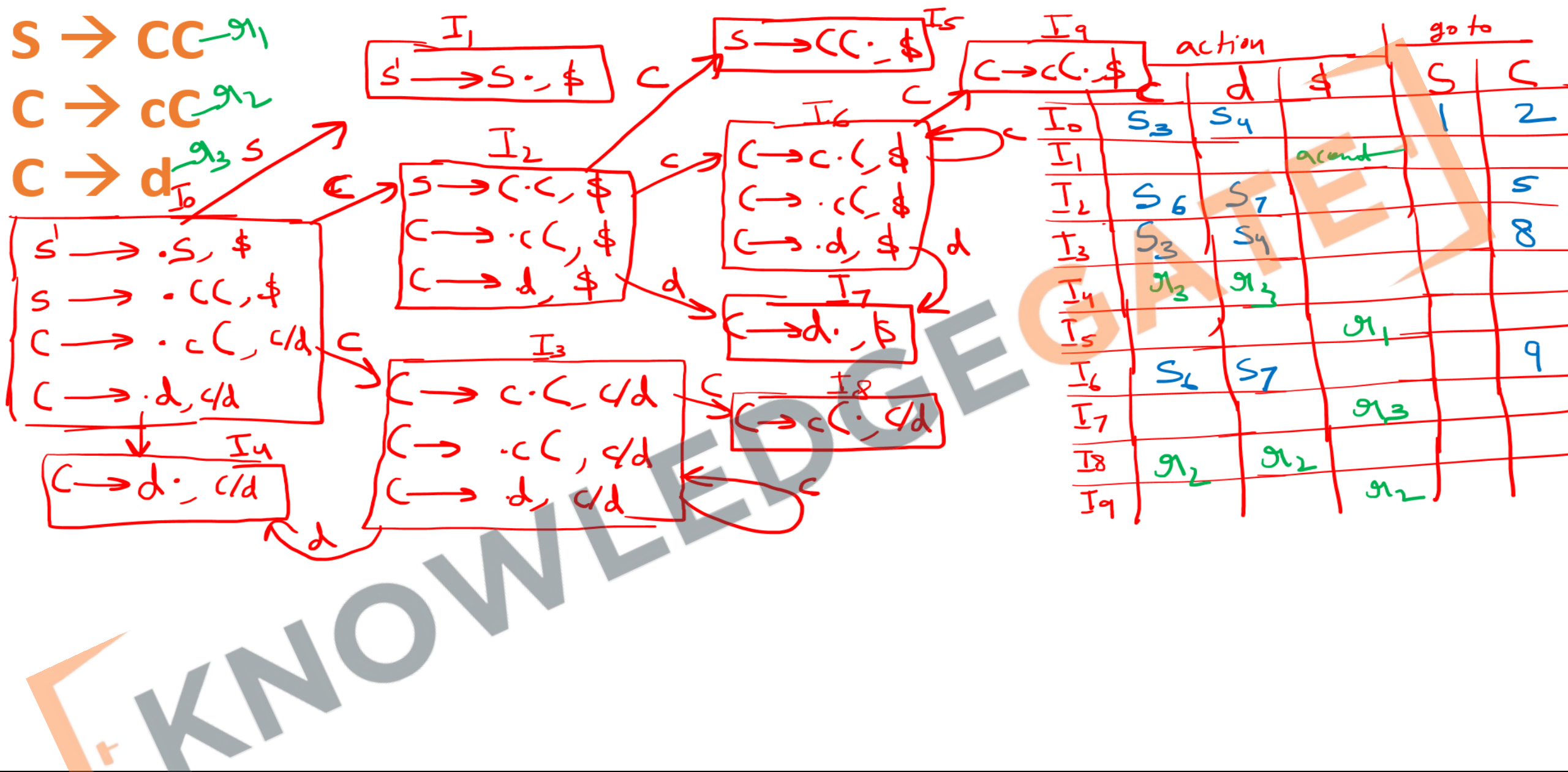


- In CLR(1) parser, we can find some of the state contains more than one production whose production part is same but follow part is different.

- because of this reason, the CLR(1) parse contains more no of entries & hence CLR(1) parser become most costly.

LALR(1)

- In CLR(1) parser, there can ne more than one state, having same production part and different follow part. Now combine those state whose production part is common and follow part is different, it is a single state and then construct the parse table, if the parse table is free from multiple entries, then the grammar is LALR(1).

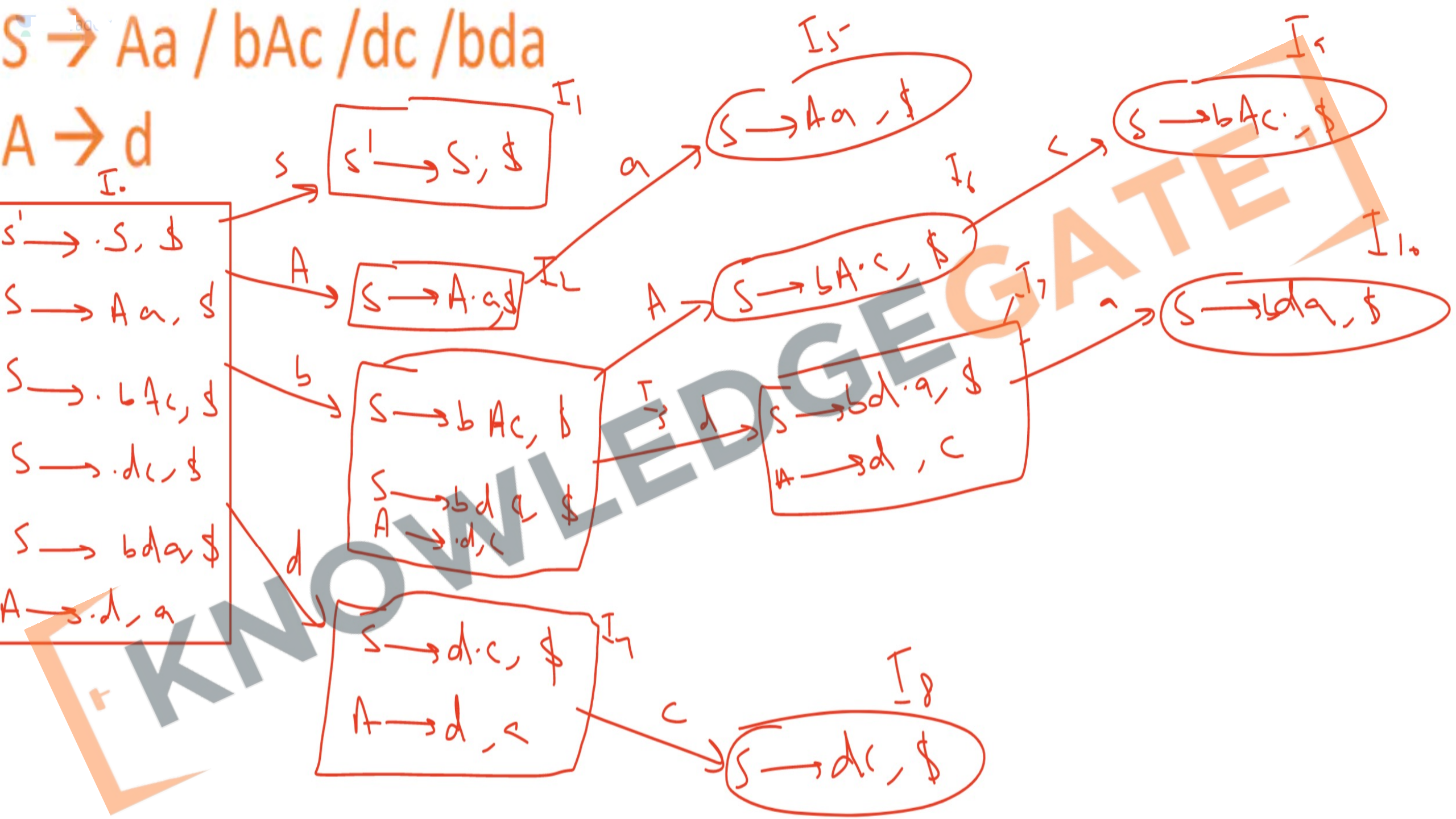


S → Aa / bAc /dc /bda

A → d

$S \rightarrow Aa \mid bAc \mid dc \mid bda$

$A \rightarrow d$



- if in CLR(1), if there are no states having some production, but different follow part, the grammar is CLR(1) and LALR(1)

S → Aa

S → bAc

S → Bc

S → bBa

A → d

B → d

S → Aa

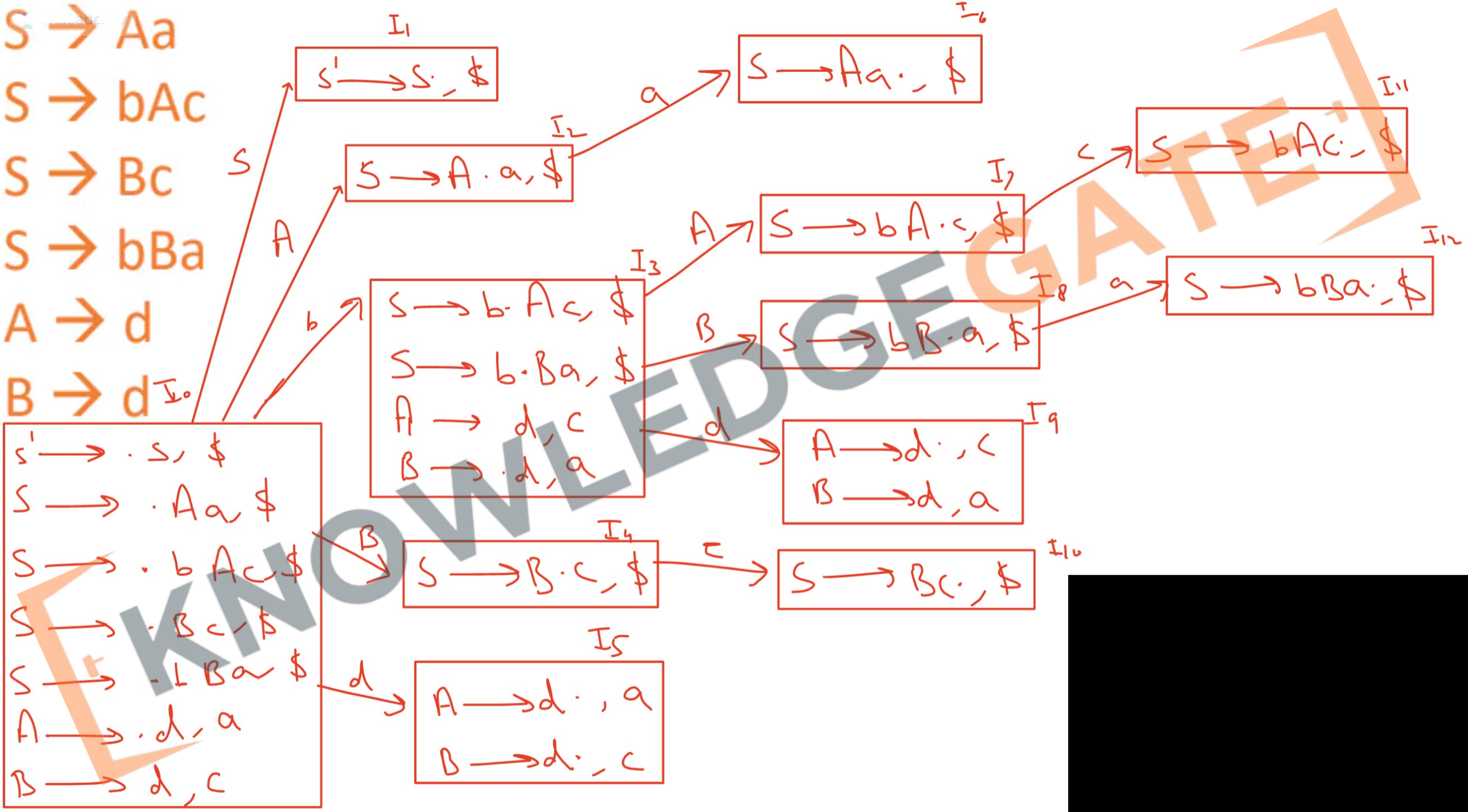
S → bAc

S → Bc

S → bBa

A → d

B → d



Operator Precedence Grammar

- Operator precedence parser can be constructed for both ambiguous and unambiguous grammar.
- In general operator precedence grammar have less complexity
- Every CFG is not operator precedence grammar
- Generally used for languages which are useful in scientific application.

- Operator Grammar is a context free grammar that has following properties
  - Does not contain ε production
  - No adjacent non-terminals on RHS of any production.

S → AB
A → a
B → b

S → AaB
B → aA / b
A → b
B → a

S → AaB
A → a / ε
B → b

S → AOB / int
O → + / * / -
A → b
B → a

Algorithm for computing precedence function

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

① $a :> b$

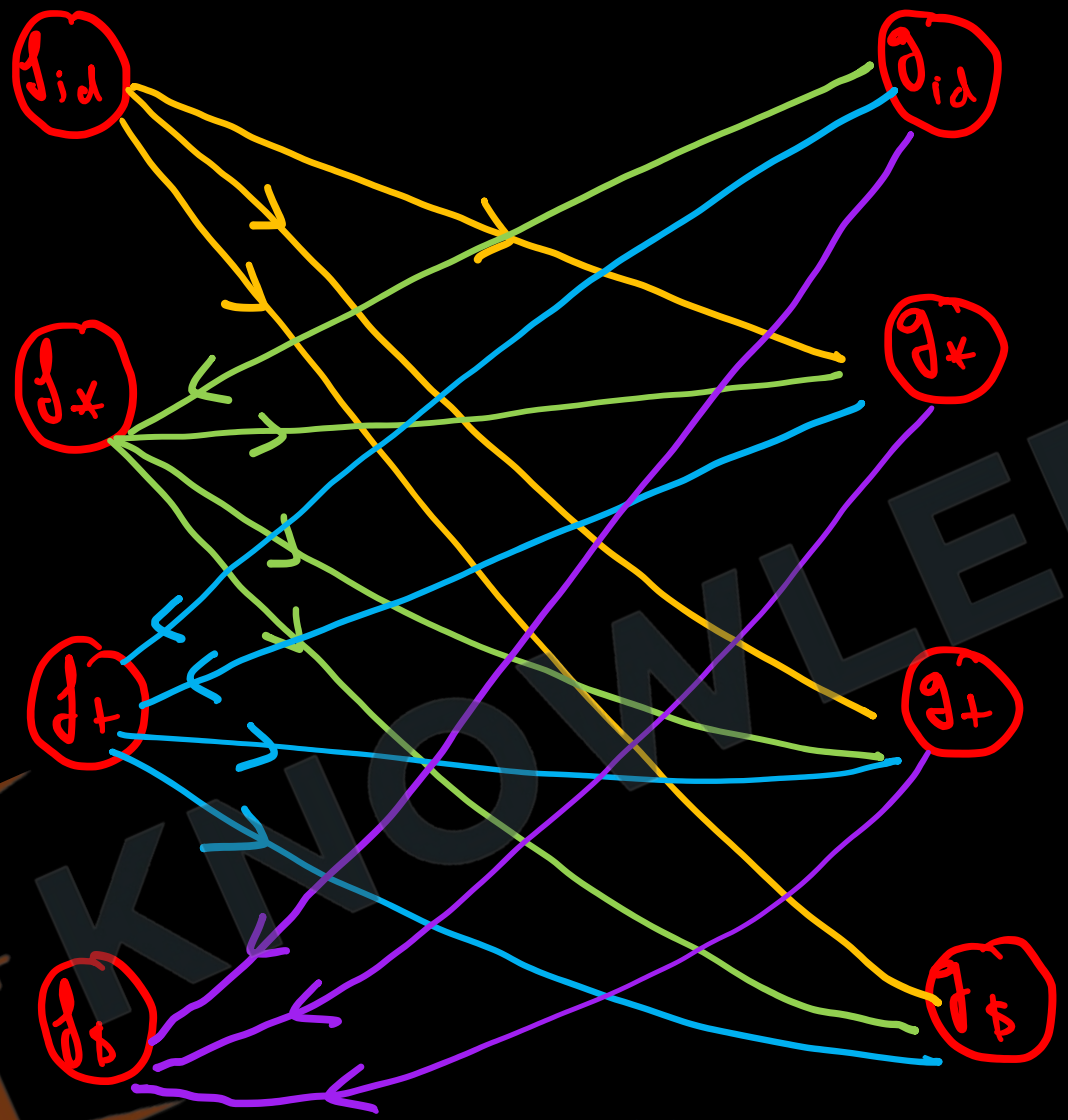
② $a <\cdot b$

③ $a \doteq b$

- Now consider the string *id + id * id*
- We will insert $ symbols at the start and end of the input string. We will also insert precedence operator by referring the precedence relation table.
- $ < · *id* · > + < · *id* · > * < *id* · > $
- We will follow following steps to parse the given string :
  - Scan the input from left to right until first · > is encountered.
  - Scan backwards over = until < · is encountered.
  - The handle is a string between < · and · >.

$ < · id · > + < · id · > * < · id · > $	Handle id is obtained between <··>. Reduce this by E → id.
E + < · id · > * < · id · > $	Handle id is obtained between <··>. Reduce this by E → id.
E + E * < · id · > $	Handle id is obtained between <··>. Reduce this by E → id.
E+E*E	Remove all the non-terminals.
+*	Insert $ at the beginning and at the end. Also insert the precedence operators.
$<·+<·*·>$	The * operator is surrounded by <··>. This indicates that * becomes handle. That means, we have to reduce E * E operation first.
$<·+·>$	Now + becomes handle. Hence, we evaluate E + E.
$$	Parsing is done.

Algorithm for computing precedence function



f_{id} → f_* → f_+ → $f_\$$

g_{id} → g_* → g_+ → $g_\$$

	id	*	+	$
id	-	>	>	>
*	<	>	>	>
+	<	<	>	>
$	<	<	<	-

	id	*	+	$
f	4	4	2	0
g	5	3	1	0

Aspect	Top-Down Parsing	Bottom-Up Parsing
Direction of Analysis	Begins from the start symbol and works towards the leaves of the syntax tree.	Starts from the leaves (input symbols) and works towards the root of the syntax tree.
Parse Tree Construction	Constructs the parse tree from the top (root) to the bottom (leaves).	Constructs the parse tree from the bottom (leaves) to the top (root).
Type of Grammar Used	Generally uses non-left-recursive grammars.	Can handle a more extensive range of grammars, including left-recursive grammars.
Example Methods	Recursive Descent Parsing, LL(k) Parsing (where k is the number of lookahead tokens).	LR(k) Parsing (including SLR, LALR, CLR), Operator-precedence parsing.
Handling Ambiguity	Less efficient in handling ambiguity and requires backtracking in some cases.	More efficient in handling ambiguous grammars and reduces the need for backtracking.

Chapter-3

(SYNTAX-DIRECTED TRANSLATION): Syntax-directed Translation schemes, Implementation of Syntax- directed Translators, Intermediate code, postfix notation, Parse trees & syntax trees, three address code, quadruple & triples, translation of assignment statements, Boolean expressions, statements that alter the flow of control, postfix translation, translation with a top down parser. More about translation: Array references in arithmetic expressions, procedures call, declarations and case statements.

Semantic Analysis

Input and output: Semantic analysis takes an Abstract Syntax Tree (AST) generated by the syntax analysis phase as its inputs.

Process: It performs type checking, scope resolution, and validates semantic consistency, ensuring that the operations and expressions in the source code are according to the language's rules and semantics.

Semantic Analysis

- Grammar + Semantic Rule + Semantic Actions = Syntax Directed Translation. SDT is the generalization of CFG.
- With grammar we give meaningful rules, and apart from semantic analysis SDT can also be used to perform things like
  - Code generation
  - Intermediate code generation
  - Value in the symbol table
  - Expression evaluation
  - Converting infix to post fix
- Things can be done in parallel to parsing…so with semantic action and rule parsers become much powerful

Q Consider the following grammar along with translation rules. Here # and % are operators and id is a token that represents an integer and $id._{val}$ represents the corresponding integer value. The set of non-terminals is {S,T, R, P} and a subscripted non-terminal indicates an instance of the non-terminal.Using this translation scheme, the computed value of $S._{val}$ for root of the parse tree for the expression 20 #10%5 #8%2%2 is _____.

20 #10%5 #8%2%2

$S \rightarrow S_1 \# T$	$\{S._{val} = S_{1}._{val} * T._{val}\}$
$S \rightarrow T$	$\{S._{val} = T._{val}\}$
$T \rightarrow T_1 \% R$	$\{T._{val} = T_{1}._{val} \div R._{val}\}$
$T \rightarrow R$	$\{T._{val} = R._{val}\}$
$R \rightarrow id$	$\{R._{val} = id._{val}\}$

Q Consider the grammar with the following translation rules and E as the start symbol.

$E \rightarrow E_1$ # T { E.value = E_1.value * T.value }

$E \rightarrow T$ { E.value = T.value }

$T \rightarrow T_1$ & F { T.value = T_1.value + F.value }

$T \rightarrow F$ { T.value = F.value }

$F \rightarrow$ num { F.value = num.value }

Compute E.value for the root of the parse tree for the expression: 2 # 3 & 5 # 6 & 4.

2 # 3 & 5 # 6 & 4

Q Consider the grammar with the following translation rules and E as the start symbol.

$E \rightarrow E_1 + T$ {print ('+');}

$E \rightarrow T$

$T \rightarrow T_1 * F$ {print ('*');}

$T \rightarrow F$

$F \rightarrow$ num {print ('num.val');}

Construct the parse tree for the string 2 + 3 * 4, and find what will be printed.

Q Consider the translation scheme shown below S → T R

R → + T {print ('+');} R / ε

T → num {print (num.val);}

Here num is a token that represents an integer and num.val represents the corresponding integer value. For an input string '9 + 5 + 2', this translation scheme will print

Q Consider the following translation scheme.

S → ER

R → *E{print(“*”);}R | ε

E → F + E {print(“+”);} | F

F → (S) | id {print(id.value);}

Here id is a token that represents an integer and id.value represents the corresponding integer value. For an input ‘2 * 3 + 4’, this translation scheme prints

Q Consider the following Syntax Directed Translation Scheme (SDTS), with non-terminals {S, A} and terminals {a, b}}. Using the above SDTS, the output printed by a bottom-up parser, for the input 'aab' is

$$S \rightarrow aA \quad \{ \text{print } 1 \}$$
$$S \rightarrow a \quad \{ \text{print } 2 \}$$
$$A \rightarrow Sb \quad \{ \text{print } 3 \}$$

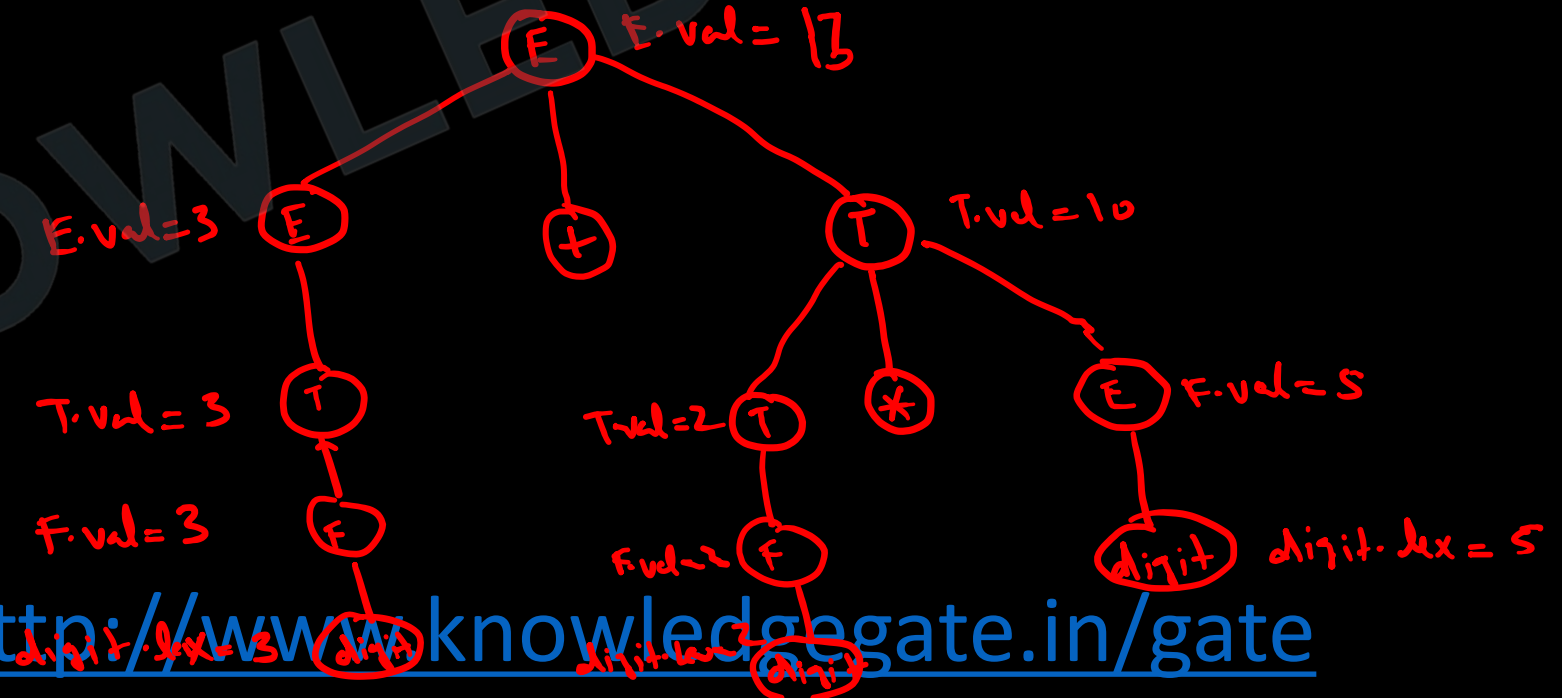
Attributes

- Attributes attach relevant information like strings, numbers, types, memory locations, or code fragments to grammar symbols of a language, which are used as labels for nodes in a parse tree.
- The value of each attribute at a parse tree node is determined by semantic rules associated with the production applied at that node, defining the context-specific information for the language construct.

Classification of Attributes

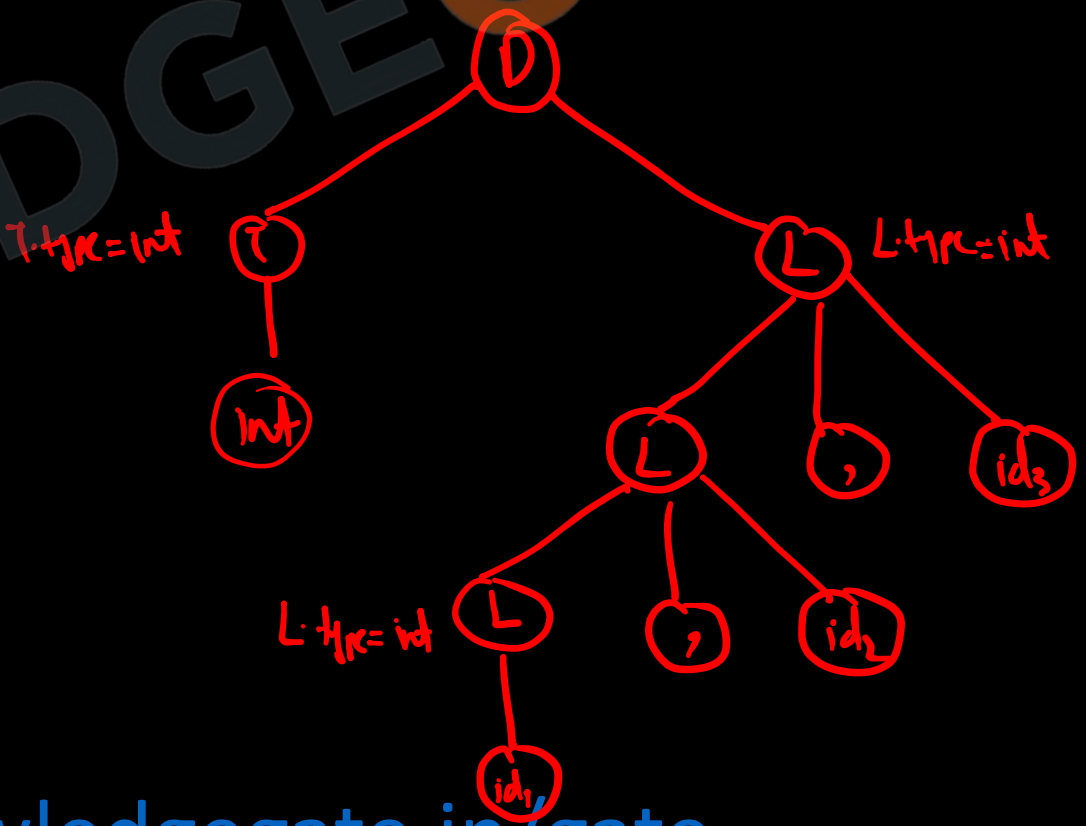
- Based on the process of Evaluation of the values, attributes are classified into two types:
  - Synthesised Attributes
  - Inherited Attributes

- Synthesized attributes are derived from a node's children within a parse tree, and a syntax-directed definition relying solely on these attributes is termed as S-attributed.
- S-attributed definitions allow parse trees to be annotated from the leaves up to the root, enabling parsers to directly evaluate semantic rules during the parsing process.
- A → XYZ {A.S = f(X.S / Y.S / Z.S)}



- **Inherited Attributes**: The attribute whose values are evaluated in terms of attribute value of parents & Left siblings is known as inherited attributes.
  - Inherited attributes are convenient for expressing dependence of a programming language construct on the context in which it appears.

$D \rightarrow T\ L$	$L.type = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.type = L.type$
$L \rightarrow id$	enter(id.pent, L.type)
	enter(id.pent, L.type)



Synthesized Attributes	Inherited Attributes
Computed from the attribute values of a node's children in the parse tree.	Computed from the attribute values of a node's siblings and parent.
Used to pass information up the parse tree.	Used to pass information down or across the parse tree.
Examples include the evaluated value of an expression or the size of a data type.	Examples include the data type expected for a child node or the environment in which a node should be evaluated.
Often associated with bottom-up parsing techniques like LALR or SLR.	Often associated with top-down parsing techniques such as LL parsers.
They do not need context from parent nodes, only from children and the node itself.	They require context from parent or surrounding nodes to be computed.

S-Attributed SDT	L-Attributes SDT
Uses only Synthesized attributes	Uses both inherited and synthesised attributes. Each inherited attribute is restricted to inherit either form parent or left sibling only.
Semantic actions are placed at extreme right on right end of production	Semantic actions are placed anywhere on right hand side of the production.
Attributes are evaluated during BUP	Attributes are evaluated by traversing parse tree depth first left to right.

Intermediate Code Generation

- Intermediate code generation in compilers creates a machine-independent, low-level representation of source code, facilitating optimization and making the compiler design more modular. This abstraction layer allows for:
  - **Portability**: Easier adaptation of the compiler to different machine architectures, as only the code generation phase needs to be machine-specific.
  - **Optimization Opportunities**: More efficient target code through optimizations performed on the intermediate form rather than on high-level source or machine code.
  - **Ease of Compiler Construction**: Simplifies the development and maintenance of the compiler by decoupling the source language from the machine code generation.

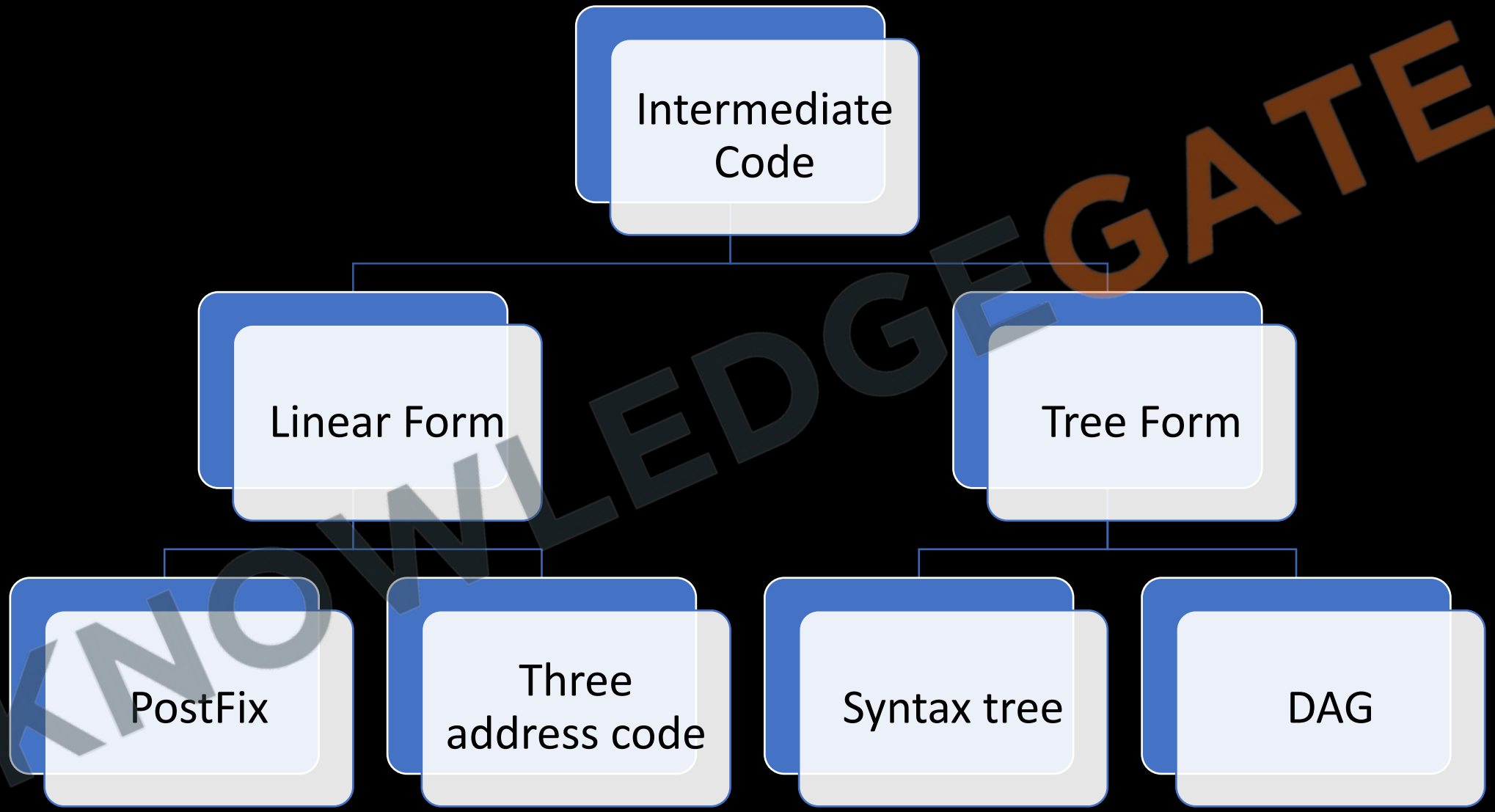
- $x = y + z * 60$

  $t_1 = z * 60$

  $t_2 = y + t_1$

  $x = t_2$

Intermediate Code Generation



Q Draw syntax tree for the arithmetic expressions : $a * (b + c) - d/2$. Also write the given expression in postfix notation **?**

① if e
then x
else
y

e ? x y → e x y ?

② if a
then if c - d
then a + c
else
a * c
else
a + b

a ? c - d ? a + c a * c a + b

a ? c d - ? a c + a c * a b +

a ? c d - a c + a c * ? a b +

a c d - a c + a c * ? a b + ?

(a+b) * (a + b + c)

Post fix

ab+ab+c+*

Three address code

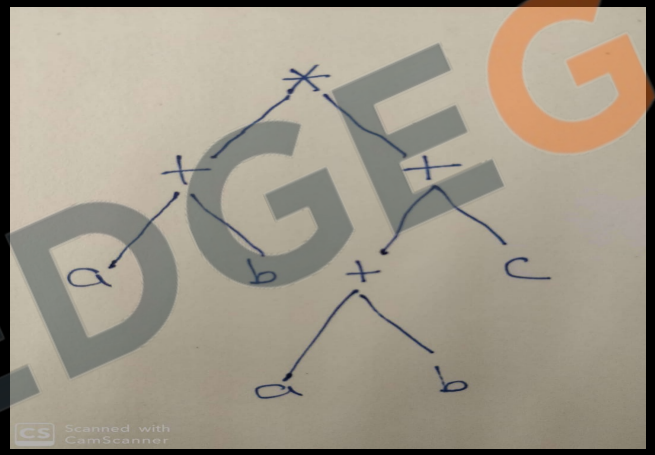
$t_1 = a+b$

$t_2 = a+b$

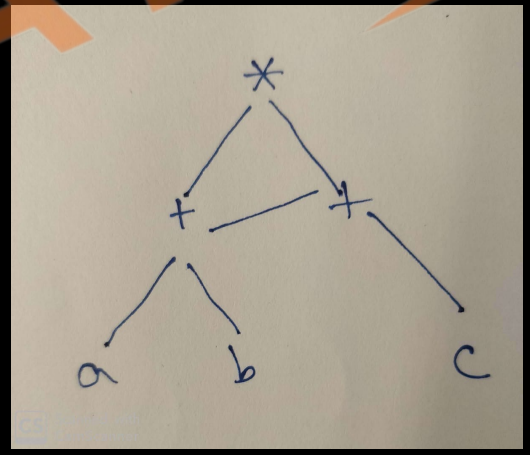
$t_3 = t_2 + c$

$t_4 = t_1 * t_3$

Syntax Tree



Direct Acyclic Graph



3 Address Code

- Three-address code is a type of intermediate code where each instruction can have at most three operands and one operator, like a := b op c. It simplifies complex operations into a sequence of simple statements, supporting various operators for arithmetic, logic, or boolean operations.

3 Address Code

Types of 3 address codes

1) x = y operator z
2) x = operator z
3) x = y
4) goto L
5) A[i] = x
   y = A[i]
6) x = *p
   y = &x

- 3 address codes can be implemented in a number of ways

-(a + b) * (c + d) + (a + b + c)

1) $t_1 = a+b$
2) $t_2 = -t_1$
3) $t_3 = c+d$
4) $t_4 = t_2 * t_3$
5) $t_5 = a+b$
6) $t_6 = t_5 + c$
7) $t_7 = t_4 + t_6$

Quadruples

	Operator	$Operand_1$	$Operand_2$	Result
1)	+	a	b	t_1
2)	-	t_1		t_2
3)	+	c	d	t_3
4)	*	t_2	t_3	t_4
5)	+	a	b	t_5
6)	+	t_5	c	t_6
7)	+	t_4	t_6	t_7

1) $t_1 = a+b$
2) $t_2 = -t_1$
3) $t_3 = c+d$
4) $t_4 = t_2 * t_3$
5) $t_5 = a+b$
6) $t_6 = t_5 + c$
7) $t_7 = t_4 + t_6$

Triplet

	Operator	$Operand_1$	$Operand_2$
1)	+	a	b
2)	-	1	
3)	+	c	d
4)	*	2	3
5)	+	a	b
6)	+	5	c
7)	+	4	6

1) $t_1 = a+b$
2) $t_2 = -t_1$
3) $t_3 = c+d$
4) $t_4 = t_2 * t_3$
5) $t_5 = a+b$
6) $t_6 = t_5 + c$
7) $t_7 = t_4 + t_6$

- **Advantage**
  - Space is not wasted
- **Disadvantage**
  - Statement cannot be moved

Indirect Triplet

Triple can be separated by order of execution and uses the pointers concepts

- **Advantage**
  - Statement can be moved
- **Disadvantage**
  - two memory access

Q Write the quadruples, triple and indirect triple for the following expression : **(*x*+*y*)* (*y*+*z*)+(*x*+*y*+*z*) ?**

$t_1 = x + y$

$t_2 = y + z$

$t_3 = t_1 + t_2$

$t_4 = t_1 + z$

$t_5 = t_3 + t_4$

Index	Operator	Arg1	Arg2	Result
1	+	x	y	t1
2	+	y	z	t2
3	*	t1	t2	t3
4	+	x	y	t4
5	+	t4	z	t5
6	+	t3	t5	t6

Q Write the quadruples, triple and indirect triple for the following expression : **(*x*+*y*)* (*y*+*z*)+(*x*+*y*+*z*) ?**

$t_1 = x + y$

$t_2 = y + z$

$t_3 = t_1 + t_2$

$t_4 = t_1 + z$

$t_5 = t_3 + t_4$

Index	Operator	Arg1	Arg2
0	+	x	y
1	+	y	z
2	*	0	1
3	+	x	y
4	+	3	z
5	+	2	4

Q Write the quadruples, triple and indirect triple for the following expression : **(*x*+*y*)* (*y*+*z*)+(*x*+*y*+*z*) ?**

$t_1 = x + y$

$t_2 = y + z$

$t_3 = t_1 + t_2$

$t_4 = t_1 + z$

$t_5 = t_3 + t_4$

Index	Operator	Arg1	Arg2
0	+	x	y
1	+	y	z
2	*	0	1
3	+	x	y
4	+	3	z
5	+	2	4

Pointer	Index
p0	0
p1	1
p2	2
p3	3
p4	4
p5	5

One Dimensional array

- Address of the element at k^{th} index
  - a[k] = B + W*k
  - a[k] = B + W*(k – Lower bound)

- B is the base address of the array
- W is the size of each element
- K is the index of the element
- Lower bound index of the first element of the array
- Upper bound index of the last element of the array

$a[k] = Addr_A + w \times k - w \times LB$

$a[k] = \underbrace{Addr_A - w}_{t_2} + \underbrace{w * k}_{t_1}$

if [LB = 1]

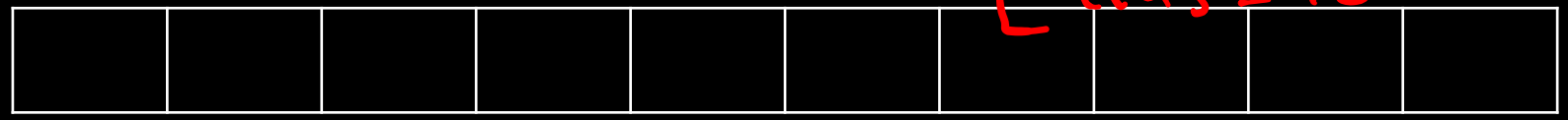
e.g. a[i] = 16 w = 4

$A[i] = t_2[t_1]$

↳ $t_1 = 4 \times i$

$t_2 = Addr_A - 4$

$t_2[t_1] = 16$



if(a, b) then t=1
else t = 0

i) if (a<b) goto (i+3)
i+1) t=0
i+2) goto (i+4)
i+3) t=1
i+4) exit

while(C) do S

i) if (E) goto i+2
i+1) goto i+4
i+2) S
i+3) goto i
i+4) exit

```
for(1=0; i<10 ; i++)
    s
```

i) i = 0
i+1) if(i<10) goto i+3
i+2) goto i+6
i+3) S
i+4) i = i + 1
i+5) goto i+1
i+6) exit

Q Generate three address code for the following code :

```
switch a + b
{
    case 1 : x = x + 1
    case 2 : y = y + 2
    case 3 : z = z + 3
    default : c = c – 1
}
```

101: $t_1 = a + b$ goto 103
102 : goto 115
103 : $t = 1$ goto 105
104 : goto 107
105: $t_2 = x + 1$
106 : $x = t_2$
107 : if $t = 2$ goto 109
108 : goto 111
109: $t_3 = y + 2$
110 : $y = t_3$
111 : if $t = 3$ goto 113
112 : goto 115 113: $t_4 = z + 3$
114 : $z = t_4$
115: $t_5 = c - 1$
116 : $c = t_5$
117 : Next statement

Q Consider the grammar with the following translation rules and E as the start symbol.

$E \rightarrow E_1 + T$ { E.nptr = mknode(E_1.nptr, +, T.ptr);}

$E \rightarrow T${ E.nptr = T.nptr }

$T \rightarrow T_1 * F$ { T.nptr = mknode(T_1.nptr, *, F.ptr);}

$T \rightarrow F${ T.nptr = F.nptr }

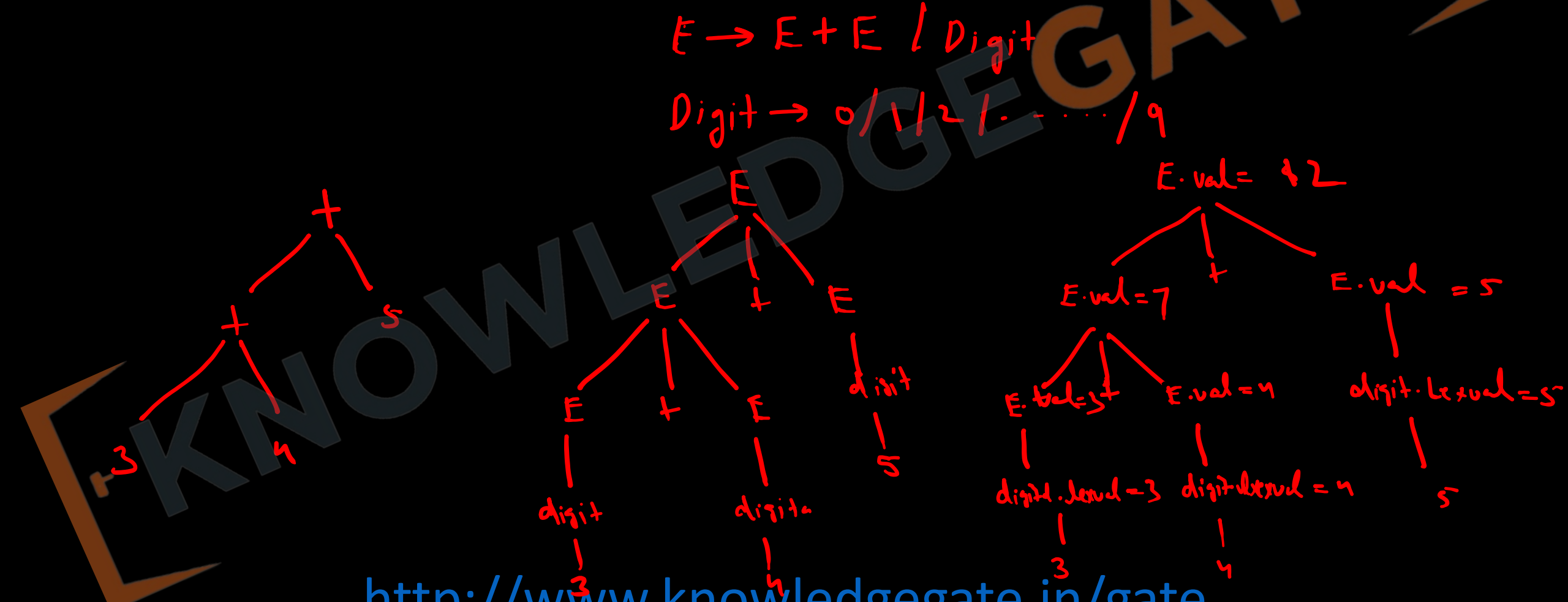
$F \rightarrow$ id { F.nptr = mknode(null, id name, null);}

Construct the parse tree for the expression: 2+3*4

Types of 3-address Statements in intermediate Code

Assignment Statement	X = y op z
Assignment Instruction	X = op y
Copy Statement	X = y
Unconditional Jump	Goto L
Conditional Jump	If x relop y goto L
Procedure Call	Parm x1 Parm x2 . Parmxn Call p,n Return y
Array Statement	X = y[i] X[i] = y
Address and Pointer Assignment	X=&y X=*y *x=y

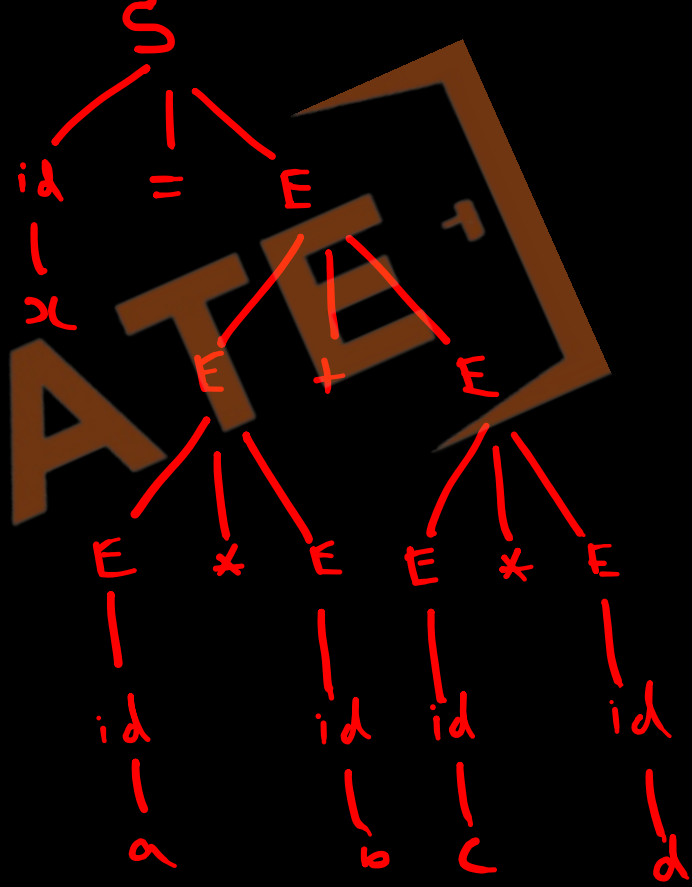
- **Syntax Tree**: An abstract representation of the syntactic structure of a program, omitting syntactical elements like brackets and punctuation.
- **Parse Tree**: A detailed tree diagram showing all the syntactical elements of a program, including brackets, punctuation, and keywords.
- **Annotated Parse Tree**: A parse tree enhanced with additional information like values, types, or variable bindings, useful for semantic analysis and code generation.



SDT of Boolean Expression

$E \rightarrow E_1$ OR E_2	E.place=newtemp(); Emit(E.place=E_1.place 'or' E_2.place);
$E \rightarrow E_1$ AND E_2	E.place=newtemp(); Emit(E.place=E_1.place 'and' E_2.place);
$E \rightarrow$ NOT E_1	E.place=newtemp(); Emit(E.place= 'not' E_1.place);
$E \rightarrow (E_1)$	E.place=E_1.place;
$E \rightarrow$ TRUE	E.place=newtemp(); Emit(E.place='1');
$E \rightarrow$ FALSE	E.place=newtemp(); Emit(E.place='0');

Production rule	Semantic actions
S → id := E	{ id_entry := look_up(id.name); if id_entry != nil then append (id_entry ':=' E.place) else error; /* id not declared*/ }
E→E1 +E2	{ E.place := newtemp(); append (E.place ':=' E1.place '+' E2.place) }
E→E1 *E2	{ E.place := newtemp(); append (E.place ':=' E1.place '*' E2.place) }
E → – E1	{ E.place := newtemp(); append (E.place ':=' 'minus' E1.place) }
E → (E1)	{ E. place:= E1.place }
E → id	{ id_entry: = look_up(id.name); if id_entry != nil then append (id_entry ':=' E.place) else error; /* id not declared*/ }



$t_1 = a + b$

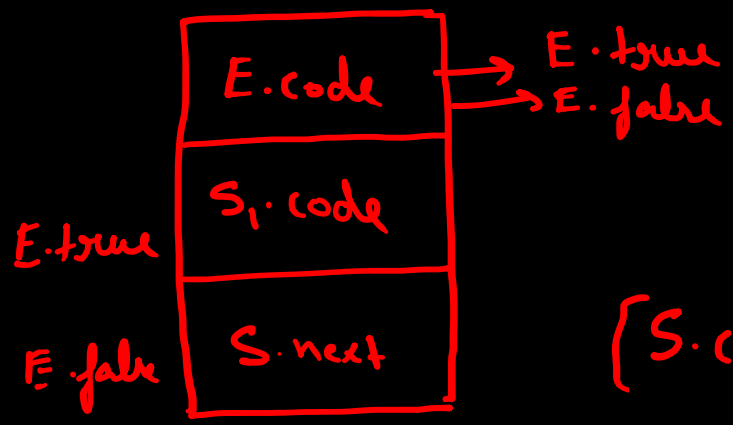
$t_2 = c + d$

$t_3 = t_1 + t_2$

$x = t_3$

SDT for Flow Control

$S \rightarrow$ if E then S_1



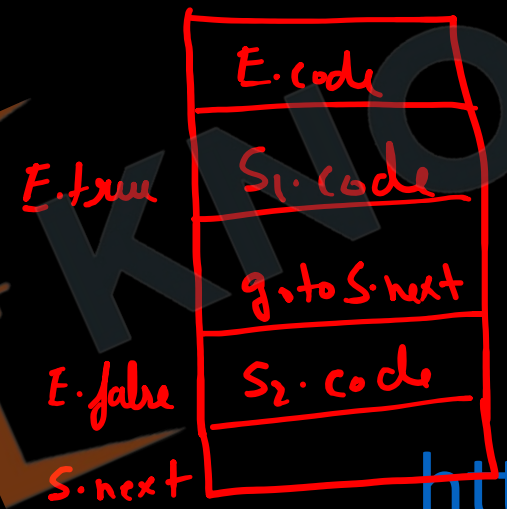
$E.true = newLabel$

$E.false = S.next$

$S_1.next = S.next$

$\{S.code = E.code \,||\, gen(E.true':') \,||\, S_1.code\}$

$S \rightarrow$ if E then S_1 else S_2



$E.true = newlabel$

$E.false = new\ label$

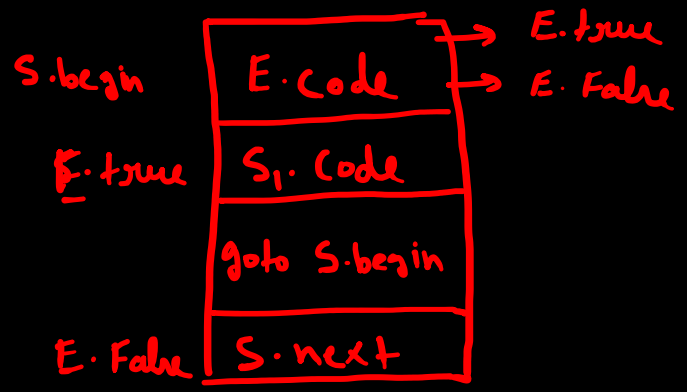
$S_1.next = S.next$

$S_2.next = S.next$

$S.code = E.code \,||\, gen(E.true':') \,||\, S_1.code \,||\, gen(S_2.next) \,||\, gen(E.false':') \,||\, S_2.code$

SDT for Flow Control

$S \rightarrow$ while E S_1



S.begin = newlabel()

E.true = newlabel

E.False = S.next

S_1.next = S.begin

S.code = gen(S.begin) || E.code || gen(E.true ':') || S_1.code || gen('goto' S.begin)

Production rule	Semantic action

Switch E
{
case v_1 : s_1
case v_2 : s_2 ...
 	
case v_{n-1} : s_{n-1}
default : s_n
}
	Evaluate E into t such that
	t = E goto check
	L_1 : code for s_1 goto last
	
	L_2 :code for s_2 goto last
	
	L_n :code for s_n goto last
	
	check:if t=v_1 goto L_1
	if t = v_2 goto L_2
	...
	if t = v_{n-1} goto L_{n-1}
	goto L_n
	last

```
switch(ch)
{
    case 1 : c = a + b;
    break;
    case 2 : c = a – b;
    break;
}
```

if *ch* = 1 goto L_1
if *ch* = 2 goto L_2
L_1: t_1:=*a*+*b*
c := t_1
goto last
L_2 : t_2 := *a* – *b*
c := t_2
goto last

Backpatching

- Backpatching is an essential technique used during the generation of three-address code in compiler design, especially in single-pass compilers.

- **Backpatching**: It's a method used in single-pass generation of three-address code where the address for jump statements is temporarily left unspecified due to unknown labels; backpatching later fills these addresses with correct values once the target labels are determined during the code generation process.

if(a, b) then t=1
else t = 0

i) if (a<b) goto (i+3)
i+1) t=0
i+2) goto (i+4)
i+3) t=1
i+4) exit

while(C) do S

i) if (E) goto i+2
i+1) goto i+4
i+2) S
i+3) goto i
i+4) exit

```
for(1=0; i<10 ; i++)
    s
```

i) i = 0
i+1) if(i<10) goto i+3
i+2) goto i+6
i+3) S
i+4) i = i + 1
i+5) goto i+1
i+6) exit

A<B OR C<D AND P<Q

1. if A<B goto ____
2. goto ____
3. if C<D goto ____
4. goto ____
5. if P<Q goto ____
6. goto ____
7. next statement

Chapter-4

(SYMBOL TABLES): Data structure for symbols tables, representing scope information. Run-Time Administration: Implementation of simple stack allocation scheme, storage allocation in block structured language. Error Detection & Recovery: Lexical Phase errors, syntactic phase errors semantic errors.

- **Definition**:- The symbol table is a data structure used by a compiler to store information about the source program's variables, functions, constants, user-defined types, and other identifiers.
- **Need**:- It helps the compiler track the scope, life, and attributes (like type, size, value) of each identifier. It is essential for semantic analysis, type checking, and code generation.

- The information is collected by the analysis phases of the compiler and used by the synthesis phases of the compiler.
  - Lexical Analysis:- Creates new table entries
  - Syntax Analysis:- add information about attributes types, scope and use in the table.
  - Semantic Analysis:- to check expression are semantically correct and type checking.
  - Intermediate Code Generation:-symbol table helps in adding temporary variable information in code.
  - Code Optimization:- use symbol table in machine dependent optimization.
  - Code Generation:- uses address information of identifier present in the table for code generation.

- **Information used by compiler from symbol table**
  - Data type and name.
  - Declaring procedure.
  - Pointer to structure table or record.
  - Parameter passing by value or by reference.
  - No and types of argument passed to function.
  - Base address.

A symbol table should possess these essential functions:

- **Lookup**: This function checks if a specific name exists in the table.
- **Insert**: This allows the addition of a new name (or a new entry) into the table.
- **Access**: It enables retrieval of information associated with a specific name.
- **Modify**: This function is used to update or add additional information about an already existing name.
- **Delete**: This capability is for removing a name or a set of names from the table.

Important symbol table requirements

- **Adaptive Structure**: Entries must be comprehensive, reflecting each identifier's specific use.
- **Quick Lookup/Search**: High-speed search functionality is essential, dependent on the symbol table's design.
- **Space-Efficient**: The table should dynamically adjust in size for optimal space use.
- **Language Feature Accommodation**: It needs to support language-specific aspects like scoping and implicit declarations.

Different data structures used in implementing a symbol table

- **Unordered List**:
  - Easy to implement using arrays or linked lists.
  - Linked lists allow dynamic growth, avoiding fixed size constraints.
  - Quick insertion $O(1)$ time, but slower lookup $O(n)$ for larger tables.

Student Name	Age
John	20
Maria	19
Alex	21
Sarah	18
Brian	22

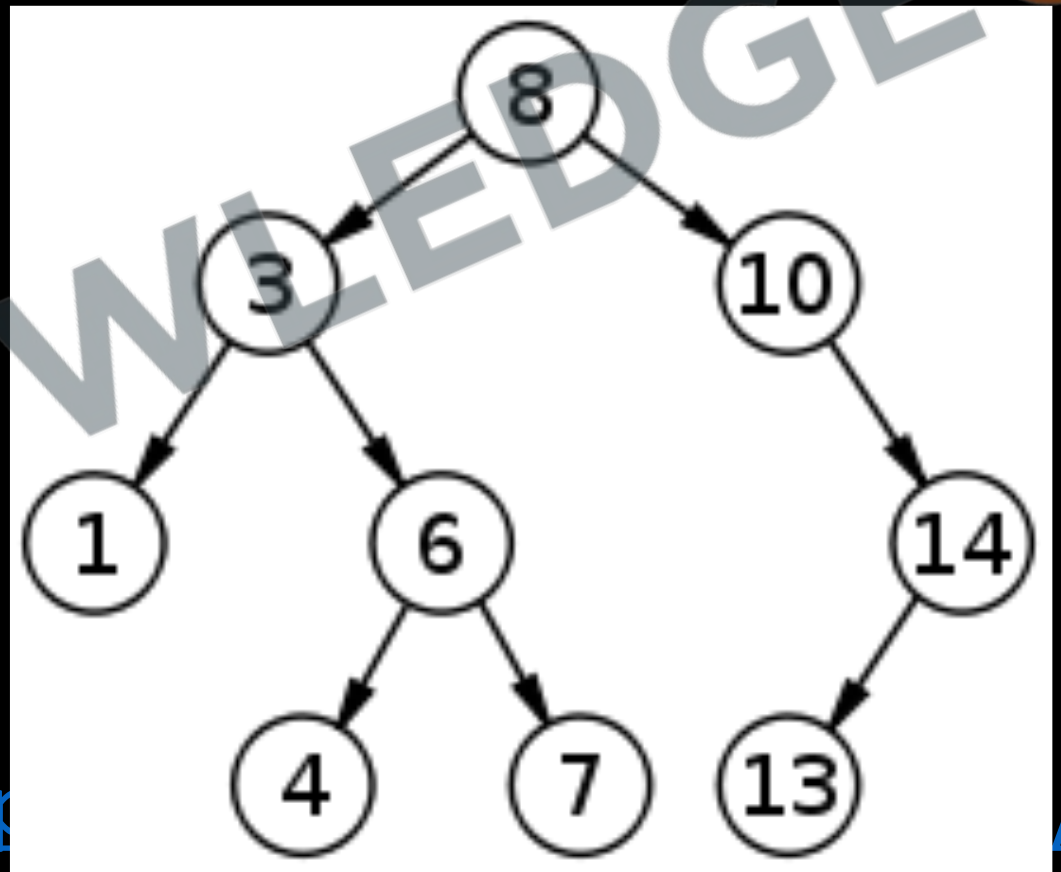
Different data structures used in implementing a symbol table

- **Ordered List**:
  - Binary search on a sorted array enables faster search $O(\log_2 n)$.
  - Insertion is slower $O(n)$ due to sorting.
  - Beneficial for fixed sets of names, like reserved words.

Variable Name	Data Type
age	int
height	float
name	string
score	double
width	float

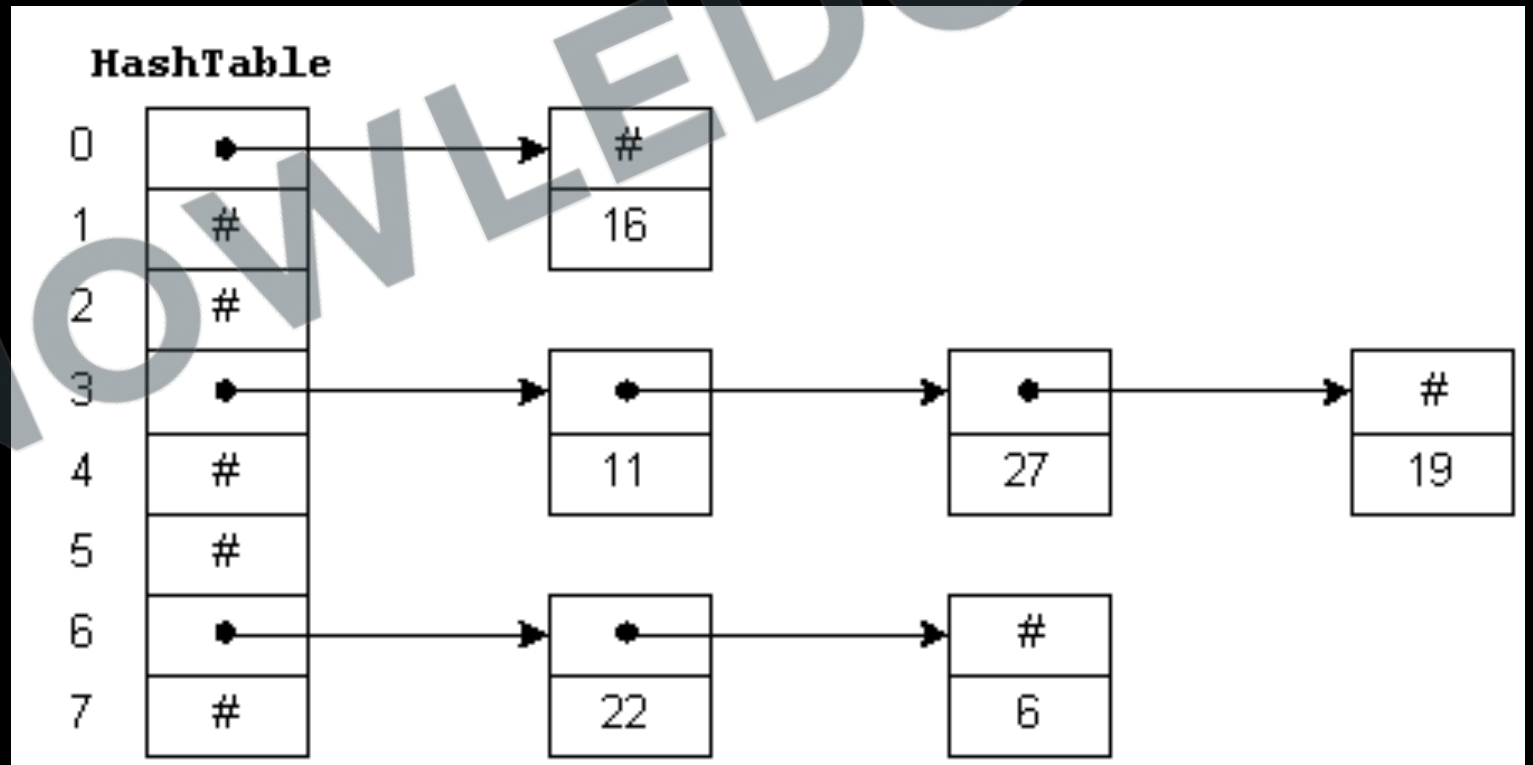
Different data structures used in implementing a symbol table

- **Search Tree**:
  - Offers logarithmic time for operations and lookup.
  - Balancing is achieved through AVL or Red-black tree algorithms.



Different data structures used in implementing a symbol table

- **Hash Tables and Hash Functions:**
  - Hash tables map elements to a fixed range of values (hash values) using hash functions.
  - They minimize element movement within the symbol table.
  - Hash functions ensure a uniform distribution of names.



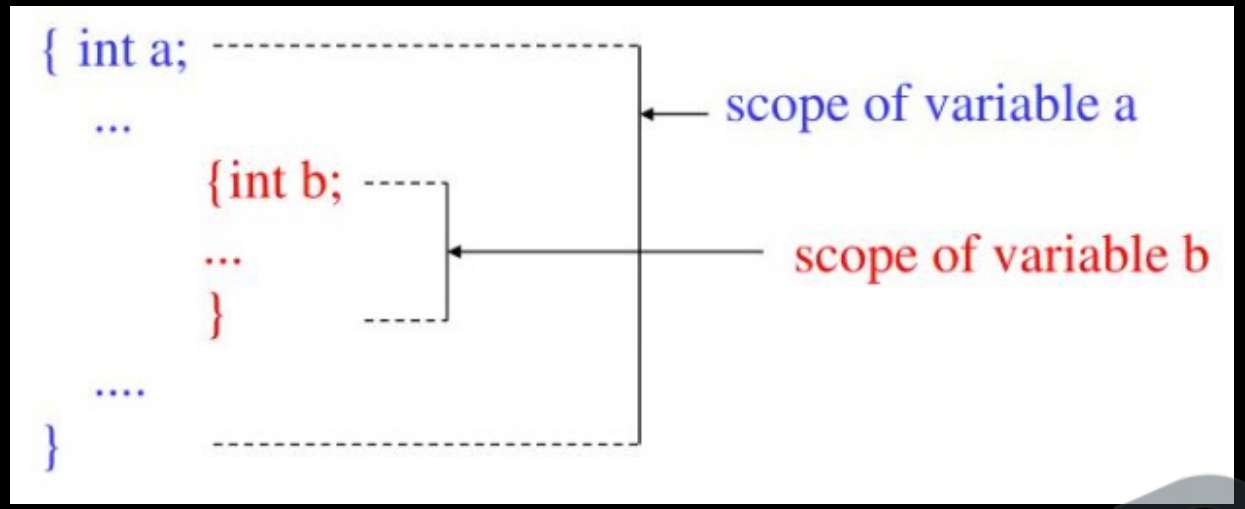
Symbol table and its entries

- **Variables**:
  - Identifiers with changeable values during and between program executions.
  - Represent memory location contents.
  - Symbol table tracks their names and runtime storage allocation.
- **Constants**:
  - Identifiers for immutable values.
  - Runtime storage allocation is unnecessary.
  - Compiler embeds them directly into the code during compilation.
- **User-Defined Types**:
  - Combinations of existing types defined by the user.
  - Accessed by name, referring to a specific type structure.
- **Classes**:
  - Abstract data types offering controlled access and language-level polymorphism.
  - Include information about constructors, destructors, and virtual function tables.
- **Records**:
  - Collections of named, possibly heterogeneous members.
  - Symbol table likely records each member within a record.

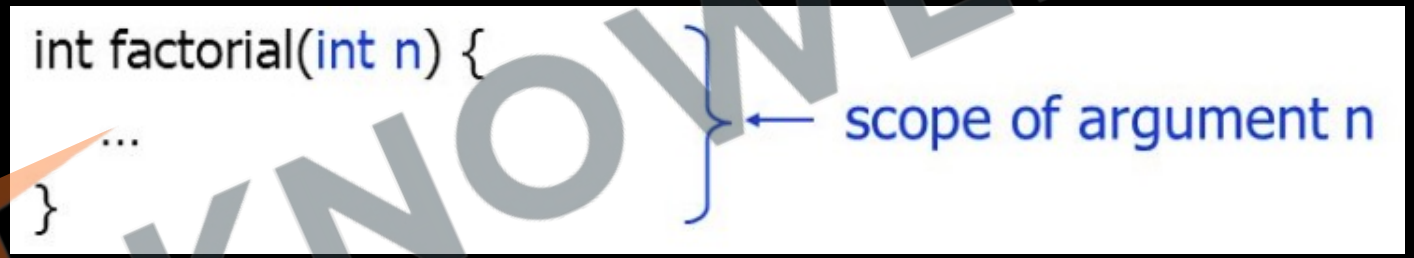
Representing Scope Information

- **Scope of Identifiers**: Scope defines where in a program an identifier is valid and accessible.
- **Language-Dependent Scope**: Different programming languages have varying scopes. For instance, in FORTRAN, a name's scope is limited to a single subroutine, while in ALGOL, it's confined to the section or procedure where it's declared.
- **Multiple Declarations**: An identifier can be declared multiple times in different scopes as distinct entities, each with unique attributes and storage locations.
- **Symbol Table's Role**: The symbol table maintains the uniqueness of each identifier's declarations.
- **Unique Identification**: Each program element is assigned a unique number, which helps in differentiating local data in different scopes.
- **Scope Determination**: Semantic rules from the program's structure are utilized to identify the active scope, especially in subprograms.
- **Scope-Related Semantic Rules**: a. An identifier's usage is confined to its defined scope. b. Identifiers with the same name and type cannot coexist within the same lexical scope.

- Scope of variables in statement blocks



- Scope of formal arguments of functions



Scoping in Programming Languages

- Scoping refers to the rules that govern the visibility and lifespan of variables and functions within different parts of a program. It defines the context in which an identifier, such as a variable or function name, is valid and accessible.

Aspect	Lexical Scoping	Dynamic Scoping
Determination Time	At compile time.	At runtime.
Scope Basis	Location in source code.	Calling sequence of functions.
Predictability	High; scope is clear from code structure.	Low; depends on program's execution path.
Variable Accessibility	Determined by code blocks and functions.	Influenced by function call order.
Common Usage	Preferred in most modern languages.	Less common; found in older languages.

Run-Time Administration: Implementation of simple stack allocation scheme

- The activation record is crucial for handling data necessary for a procedure's single execution. When a procedure is called, this record is pushed onto the stack, and it's removed once control returns to the calling function.

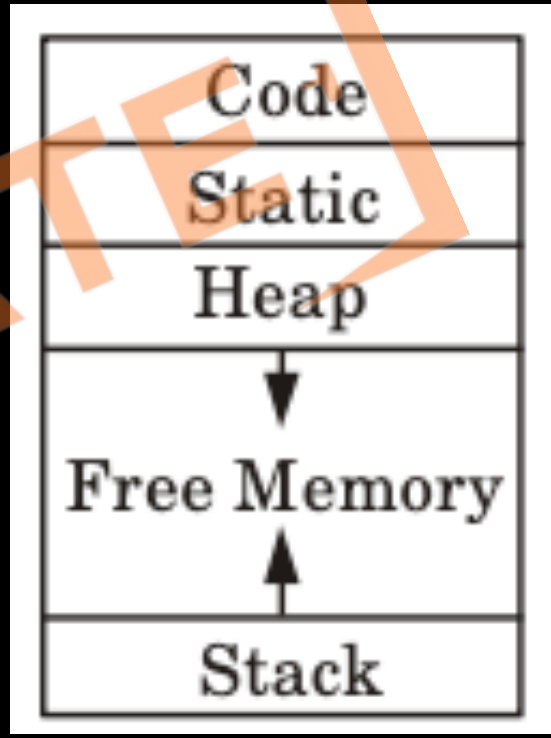
Return value
Actual parameters
Control link
Access link
Saved machine status
Local data
Temporaries

Activation record fields include:

- **Return Value**: Allows the called procedure to return a value to the caller.
- **Actual Parameters**: Used by the caller to provide parameters to the called procedure.
- **Control Link**: Connects to the caller's activation record.
- **Access Link**: References non-local data in other activation records.
- **Saved Machine Status**: Preserves machine state prior to the procedure call.
- **Local Data**: Contains data specific to the procedure's execution.
- **Temporaries**: Stores interim values during expression evaluation.

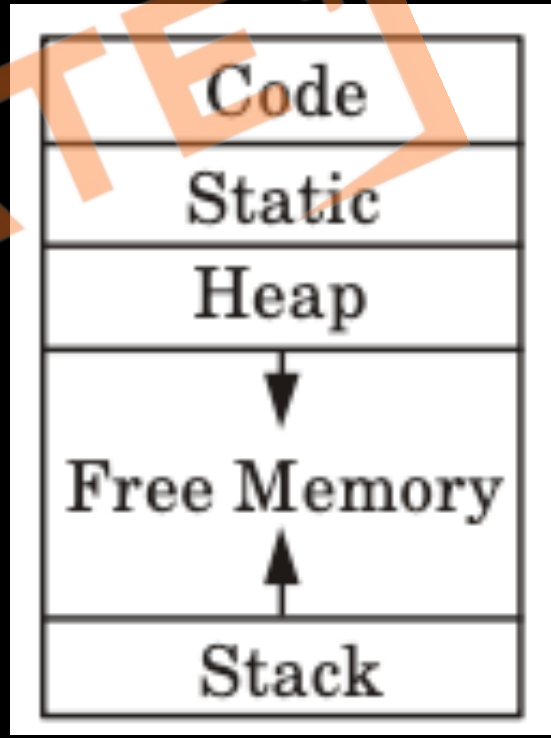
Overview of Memory Allocation Methods in Compilation

- **Code Storage**:
  - Contains fixed-size, unchanging executable target code during compilation.
- **Static Allocation**:
  - Allocates storage for all data objects at compile time.
  - Object sizes are known at compile time.
  - Object names are bound to storage locations during compilation.
  - The compiler calculates the required storage for each object, simplifying address determination in the activation record.
  - Compiler sets up addresses for the target code to access data at compile time.



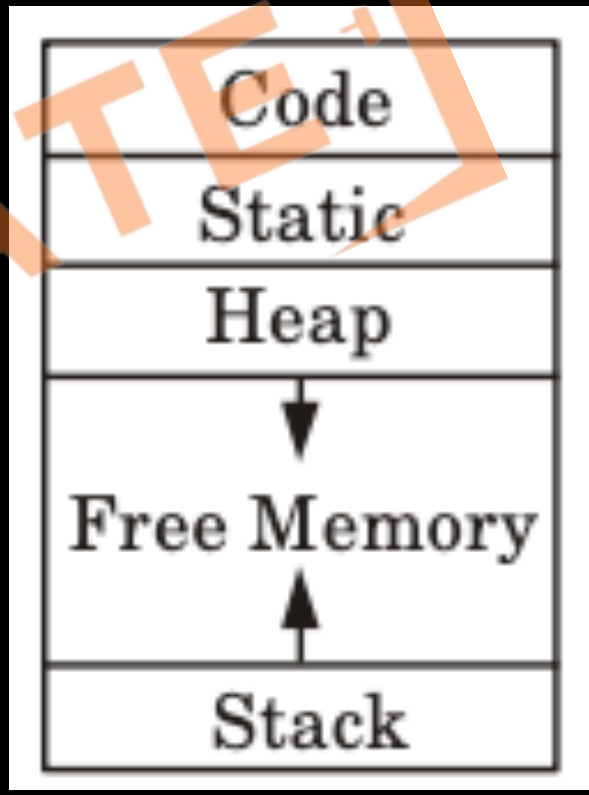
Overview of Memory Allocation Methods in Compilation

- **Heap Allocation Methods**:
  - **Garbage Collection**:
    - Handles objects that persist after losing all access paths.
    - Reclaims object space for reuse.
    - Garbage objects are identified by a 'garbage collection bit' and returned to free space.
  - **Reference Counting**:
    - Reclaims heap storage elements as soon as they become inaccessible.
    - Each heap cell has a counter tracking the number of references to it.
    - The counter is adjusted as references are made or removed.



Overview of Memory Allocation Methods in Compilation

- **Stack Allocation**:
  - Manages data structures known as activation records.
  - Activation records are pushed onto the stack at call time and popped when the call ends.
  - Local variables for each procedure call are stored in the respective activation record, ensuring fresh storage for each call.
  - Local values are removed when the procedure call completes.



Overview of Error Recovery in Compilers

Error Recovery Significance:

- Essential for compilers to process and execute programs even with errors.

Error Message Characteristics:

- Messages should relate to the original source code, not its internal representation.
- Simplicity is key in error messaging.
- Precision in pinpointing and fixing errors is crucial.
- Avoid repetition of the same error message.

Objectives of Error Handling:

- Identify errors and provide clear, useful diagnostics.
- Rapid recovery to identify further errors in the code.
- Ensuring error handling doesn't substantially delay the compilation of correct programs.

Lexical Phase Errors in Compiling

- **Definition of Lexical Phase Error**:
  - Occurs when a sequence of characters doesn't form a recognizable token during the source program scanning, preventing valid token generation.
- **Common Causes of Lexical Errors**:
  - Adding an unnecessary character.
  - Omitting a required character.
  - Substituting a character incorrectly.
  - Swapping two characters.
- **Examples of Lexical Errors**:
  - In Fortran, identifiers exceeding 7 characters are considered lexical errors.
  - The presence of characters like ~, &, and @ in a Pascal program constitutes a lexical error.

Syntactic Phase Errors (Syntax Errors)

- **Overview of Syntax Errors**:
  - Syntax errors occur due to coding mistakes made by programmers.

- **Common Sources of Syntax Errors**:
  - Omitting semicolons.
  - Imbalanced parentheses and incorrect punctuation usage.

- **Example**:
  - Consider the code snippet: int x; int y //Syntax error
  - The error arises from the missing semicolon after int y.

Semantic Phase Errors

- **Definition of Semantic Errors**:
  - Semantic errors relate to incorrect use of program statements, impacting the program's meaning or logic.
- **Typical Reasons for Semantic Errors**:
  - Using undeclared names.
  - Type mismatches.
  - Inconsistencies between actual and formal arguments in function calls.
- **Example**:
  - In the code scanf(“%f%f”, a, b);, the error lies in not using the addresses of variables a and b. The correct usage should be scanf(“%f%f”, &a, &b); to provide the address locations.

Logical Errors(Run Time Error) in Programming

- **Nature of Logical Errors**:
  - Logical errors are mistakes in a program's logic that the compiler does not catch.
  - These errors occur in programs that are syntactically correct but do not function as intended.
- **Example of a Logical Error**:
  - Consider the code snippet:

```
x = 4;
y = 5;
average = x + y / 2;
```

Basics of Panic Mode Recovery

- A straightforward and commonly used method in various parsing techniques.
- Upon detecting an error, the parser discards input symbols until it encounters a synchronizing token from a predefined set.

- **Characteristics**:
  - Panic mode may bypass large sections of input without further error checking.
  - This approach ensures that the parser does not enter an infinite loop.
  - a = b + c;
  - d = e + f;
  - In panic mode, the parser might skip over the entire line a = b + c; without identifying specific errors in it.

Phrase-Level Recovery

- This method involves making localized corrections to the input when an error is detected by the parser.
- It involves substituting a part of the input with a string that allows the parser to proceed.
- **Correction Mechanism**:
  - Common corrections include replacing a comma with a semicolon, removing an unnecessary semicolon, or inserting a missing one.
  - while(x>0)
    - y=a+b;
  - Phrase-level recovery might correct this by adding 'do' to form while(x>0) do y=a+b;, enabling the parsing process to continue smoothly.

Error Production

- **Overview of Error Production**:
  - This technique allows a parser to generate relevant error messages while continuing the parsing process.
- **Functionality**:
  - When the parser encounters an incorrect production, it issues an error message and then resumes parsing.
  - E → + E | - E | * A | / A
  - A → E
  - If the parser comes across the production '* A' and deems it incorrect, it can alert the user with a message, perhaps querying whether '*' is intended as a unary operator, before continuing with the parsing.

Global Correction in Parsing

- **Nature of Global Correction**:
  - Primarily a theoretical concept in the realm of parsing.

- **Implications on Parsing Process**:
  - Utilizing global correction significantly increases both the time and space resources required during the parsing stage.

Error Recovery in Operator Precedence Parsing

- **Detection Points in Operator Precedence Parsing:**
  - Errors are identifiable at two key stages:
    - When there's no precedence relation between the terminal at the top of the stack and the current input symbol.
    - When a handle is identified, but no corresponding production exists.

Error Recovery in Operator Precedence Parsing

- **Error Types and Diagnostics**: The parser can detect several types of errors:
  - **Errors Detected During Reduction**:
    - Missing operand.
    - Missing operator.
    - Absence of expression within parentheses.
  - **Errors Detected During Shift/Reduce Actions**:
    - Missing operand.
    - Unbalanced or missing right parenthesis.
    - Missing operators.

Chapter-5

(CODE GENERATION): Design Issues, the Target Language. Addresses in the Target Code, Basic Blocks and Flow Graphs, Optimization of Basic Blocks, Code Generator. Code optimization: Machine-Independent Optimizations, Loop optimization, DAG representation of basic blocks, value numbers and algebraic laws, Global Data-Flow analysis.

Code Generation

- Code generation is the process of converting the intermediate representation (IR) of source code into a target code(assembly level). Which is also optimized.
- It involves translating the syntax and semantics of the high-level language into assembly level code, typically after the source code has passed through lexical analysis, syntax analysis, semantic analysis, and intermediate code generation.

```
int main() {
    int a = 5;
    int b = 3;
    int sum = a + b;
}
```

```
t1 = 5
t2 = 3
t3 = t1 + t2
```

```
MOV R1, 5
MOV R2, 3
ADD R3, R1, R2;
```

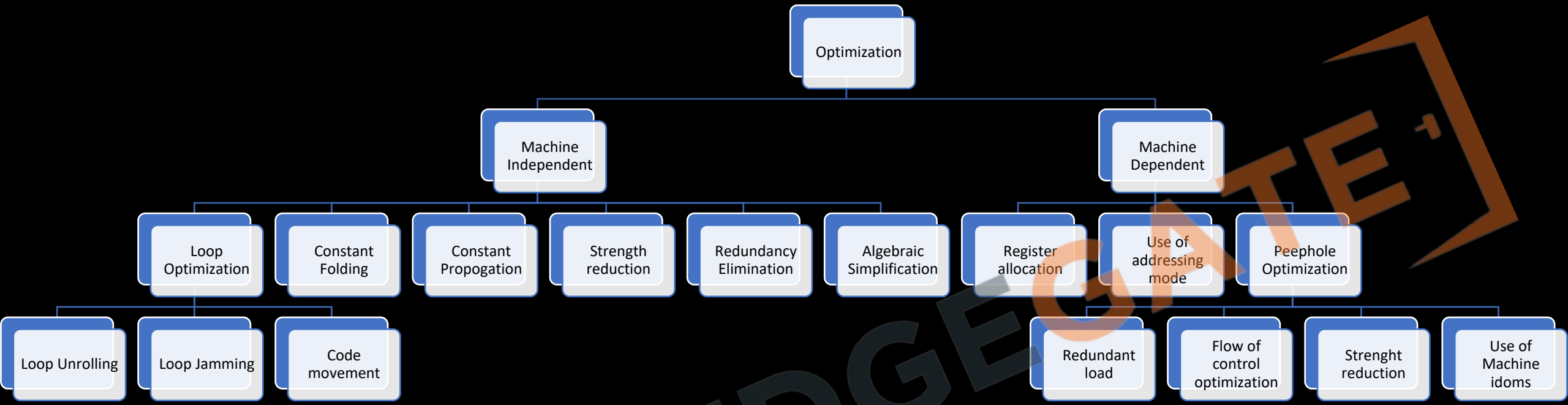
- **Code Generator Input**:
  - Uses the source program's intermediate representation (IR) and symbol table data.
- **Intermediate Representation (IR)**:
  - Consists of three-address and graphical forms.
- **Target Program**:
  - Influenced by the target machine's instruction set architecture (ISA).
  - Common ISAs: RISC, CISC, and stack-based.
- **Instruction Selection**:
  - Converts IR to executable code for the target machine.
  - High-level IR may use code templates for translation.
- **Register Allocation**:
  - Critical to decide which values to store in registers.
  - Non-registered values stay in memory.
  - Register use leads to shorter, faster instructions.
  - Involves two steps: i. Choosing variables for register storage. ii. Assigning specific registers to these variables.
- **Evaluation Order**:
  - The sequence of computations impacts code efficiency.
  - Some sequences minimize register usage for intermediate results.

- **Optimization**: During code generation, various optimization techniques are applied to improve efficiency and performance. This could include optimizing for speed, memory usage, or even power consumption.

- **Target Platform**: The generated code is specific to the target platform's architecture, such as x86, ARM, etc. This means the same high-level code will have different generated machine code for different platforms.

Optimization

- Process of reducing the execution time of a code without effecting the outcome of the source program, is called as optimization.



Aspect	Machine-Independent Optimization	Machine-Dependent Optimization
Definition	Optimizations that are not specific to any processor or machine architecture.	Optimizations tailored to the specifics of a particular machine or processor architecture.
Focus	Concentrates on improving the logic and efficiency of the code at a high level, often in the intermediate code.	Focuses on enhancing performance by taking advantage of the unique features and instructions of the target machine.
Examples	Common subexpression elimination, Code motion, Dead code elimination, Loop unrolling, Loop fusion	Instruction scheduling, Register allocation, Pipeline optimization, Use of machine-specific instructions, Cache optimization
Portability	Highly portable across different architectures as they do not rely on machine-specific features.	Not portable; optimizations must be re-applied or altered for different architectures.
Stage of Application	Applied before the code is mapped to the target machine's instruction set, often during the intermediate code generation phase.	Applied during or after the generation of the target machine code, tailoring the optimizations to the specifics of the machine's hardware.

Algorithm to partition a sequence of three address statements into basic blocks

- Loop Optimization
  - To apply loop optimization, we must first detect loops.
  - For detecting loops, we use control flow analysis (CFA) using program flow graph (PFG)
  - To find PFG, we need to find basic blocks.
  - A Basic block is a sequence of 3-adress statements where control enters at the beginning and leaves only at the end without any jumps or halts

- The block can be identified with the help of leader
  - Finding the leader
  - Finding the bocks
  - Construct PFG

- In order to find the basic blocks, we need to finds the leader in the program then a basic block will start from one leader to the next leader but not including next leader.
- identifying leaders in a basic block
  - First statement is a leader
  - Statement that is the target of conditional or unconditional statement is a leader
  - Statement that follow immediately a conditional or unconditional statement is a leader

```
Fact(x)
{
    int f=1
    for(i=2 ; i<=x ; i++)
        f = f*i;
    return f;
}
```

1) f=1;
2) i=2
3) if(i>x), goto 9
4) t1=f*i;
5) f=t1;
6) t2=i+1;
7) i=t2;
8) goto(3)
9) goto calling program

1) f=1;
2) i=2

3) if(i>x), goto 9

4) t1=f*i;
5) f=t1;
6) t2=i+1;
7) i=t2;
8) goto(3)

9) goto calling program

Q Consider the intermediate code given below: The number of nodes and edges in the control-flow-graph constructed for the above code, respectively, are

1. i = 1
2. j = 1
3. $t_1 = 5 * i$
4. $t_2 = t_1 + j$
5. $t_3 = 4 * t_2$
6. $t_4 = t_3$
7. $a[t_4] = -1$
8. j = j + 1
9. if j <= 5 goto(3)
10. i = i + 1
11. if i < 5 goto(2)

Q Consider the following sequence of three address codes :

1. Prod:=0
2. *I*:=1
3. *T*1:=4***I*
4. *T*2:=addr(*A*)–4
5. *T*3:=*T*2[*T*1]
6. *T*4:=addr(*B*)–4
7. *T*5:=*T*4[*T*1]
8. *T*6:=*T*3***T*5
9. Prod:=Prod+*T*6
10. *I*=*I*+1
11. If *I* <= 20 goto (3)

Loop Jamming: combining the bodies of two loops, whenever they share the same index and same no of variables

```
for (int i=0; i<=10; i++)
      for (int j=0; j<=10; j++)
            x[i, j]=”TOC”
for (int j=0; j<=10; j++)
            y[i]=”CD”
```

```
for (int i=0; i<=10; i++)
{
      for (int j=0; j<=10; j++)
      {
            x[i, j]=”TOC”
      }
      y[i]=”CD”
}
```

Loop Unrolling: getting the same output with less no of iteration is called loop unrolling

```
int i=1;
while(i<=100)
{
    print(i)
    i++
}
```

```
int i=1;
while(i<=100)
{
    print(i)
    i++
    print(i)
    i++
}
```

Code movement(Loop invariant computation): removing those code out from the loop which is not related to loop.

```
int i=1;
while(i<=100)
{
    a =b+c
    print(i)
    i++
}
```

Optimization of basic blocks

- Algebraic Simplification
- Redundant code Elimination / Common subexpression elimination
- Strength reduction
- Constant Propagation
- Constant Folding

Constant Folding: Replacing the value of expression before compilation is called as constant folding

x = a + b + 2 * 3 + 4

x = a + b + 10

Constant Propagation: replacing the value of constant before compile time, is called as constant propagation.

pi = 3.1415

x = 360 / pi

x = 360/3.1415

Strength reduction: replacing the costly operator by cheaper operator, this process is called strength reduction.

y = 2 * x

y = x + x

Redundant code Elimination / Common subexpression elimination:

Avoiding the evaluation of any expression more than once is redundant code elimination.

```
x = a + b
y = b + a

x = a + b
y = x
```

Algebraic Simplification: Basic laws of math's which can be solved directly.

a = b*1

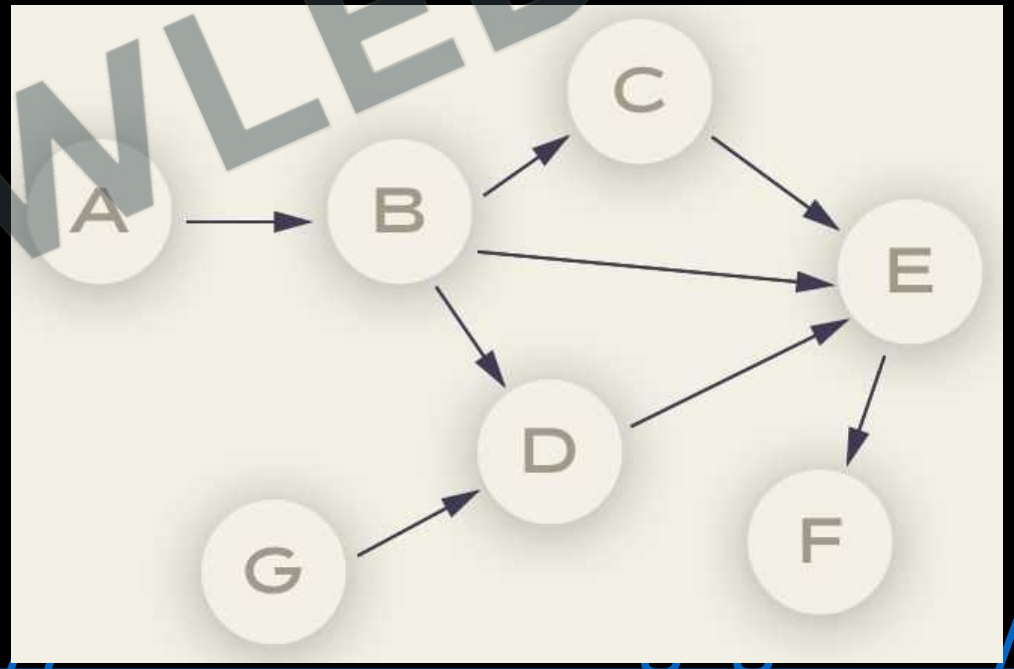
a=b

a=b+0

a=b

Direct Acyclic Graph (DAG)

- A Direct Acyclic Graph is a graph that is directed and contains no cycles. So it is impossible to start at any vertex v and follow a sequence of edges that eventually loops back to v again.
- By representing expressions and operations in a DAG, compilers can easily identify and eliminate redundant calculations, thus optimizing the code.
  - We automatically detect common sub-expressions with the help of DAG algorithm.
  - We can determine which identifiers have their values used in the block.
  - We can determine which statements compute values which could be used outside the block.



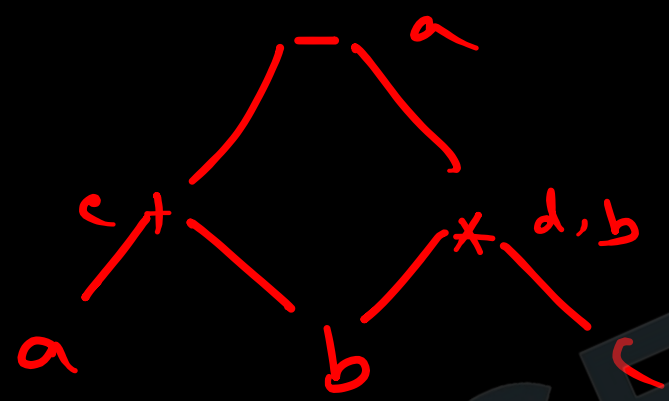
((a+a) + (a+a)) + ((a+a) + (a+a))

d = b * c

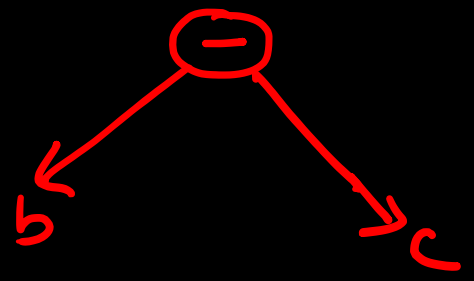
c = a + b

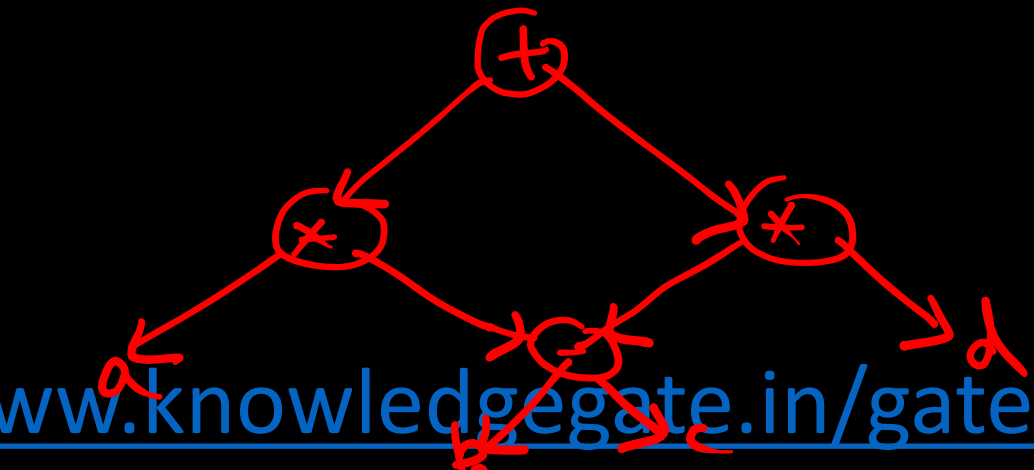
b = b * c

a = c - d



a * (b − c) + (b − c) * d



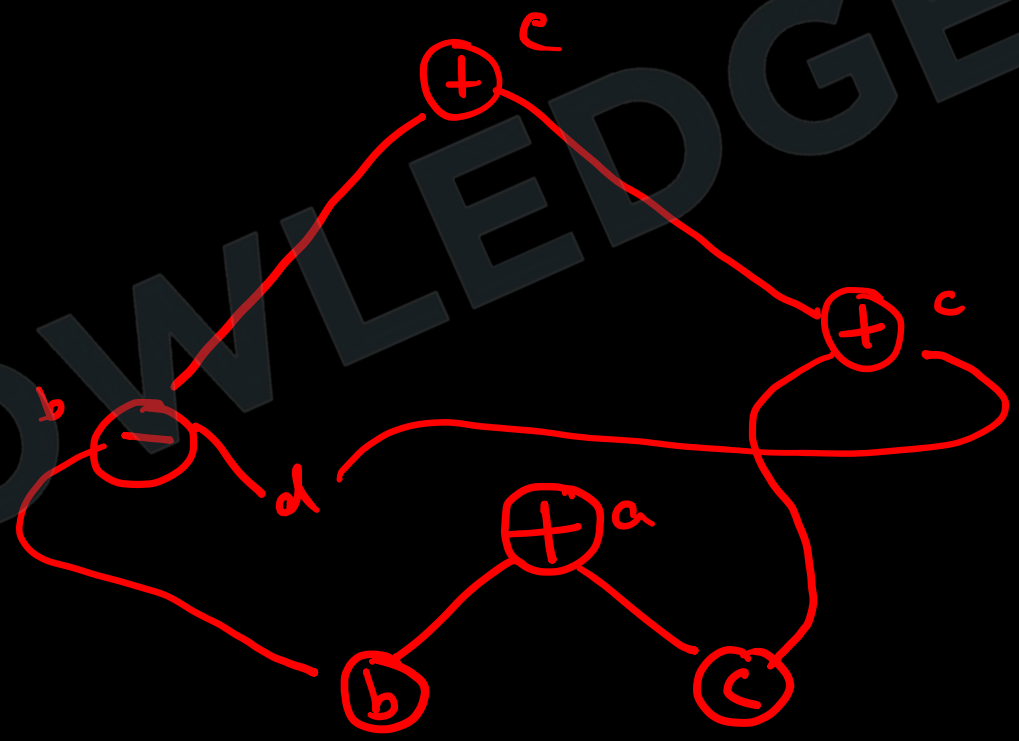


a = b + c

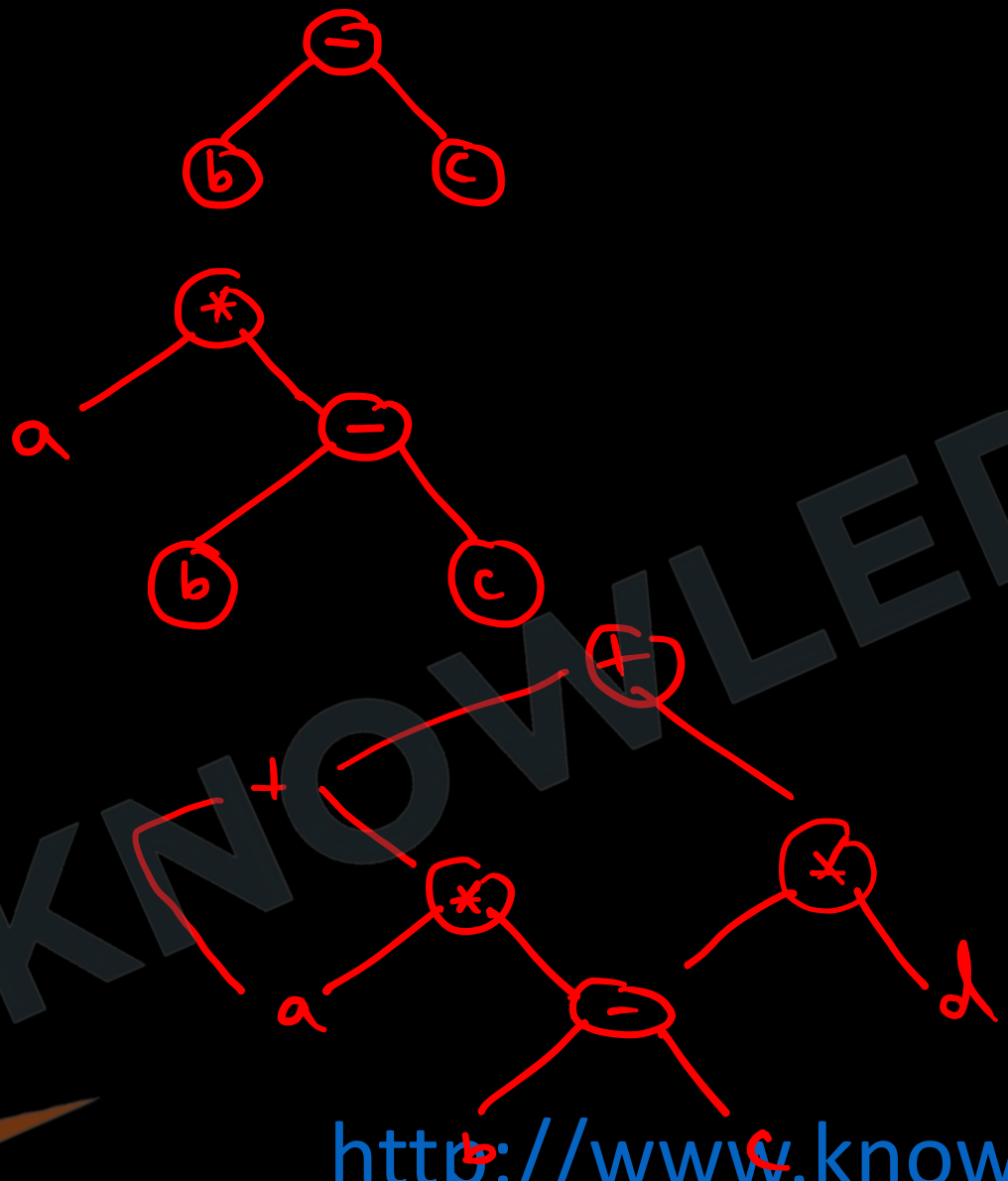
b = b - d

c = c + d

e = b + c



a + a * (b - c) + (b - c) * d



a = b * -c + b * -c