JECRC Foundation

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

Year & Sem – 3rd Year & 5th Sem
Subject – COMPILER DESIGN
Unit – 2
Presented by – (Abhishek Dixit, Assistant Prof., Dept of CSE)

VISION AND MISSION OF INSTITUTE

To become a renowned center of outcome based learning and work towards academic, professional, cultural and social enrichment of the lives of individuals and communities

M1: Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.

M2: Identify, based on informed perception of Indian, regional and global needs, the areas of focus and provide platform to gain knowledge and solutions.

M3: Offer opportunities for interaction between academia and industry.

M4: Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

VISION OF THE DEPARTMENT

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

MISION OF THE DEPARTMENT

M1: To impart outcome based education for emerging technologies in the field of computer science and engineering.

M2: To provide opportunities for interaction between academia and industry.

M3: To provide platform for lifelong learning by accepting the change in technologies

M4: To develop aptitude of fulfilling social responsibilities

PROGRAM OUTCOMES

Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
Problem analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM EDUCATIONAL OBJECTIVES

1. To provide students with the fundamentals of Engineering Sciences with more emphasis in Computer Science &Engineering by way of analyzing and exploiting engineering challenges.
2. To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.
3. To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues.
4. To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self motivated life-long learning needed for a successful professional career.
5. To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge

PROGRAM SPECIFIC OBJECTIVES

1. PSO1. Ability to interpret and analyze network specific and cyber security issues, automation in real word environment.

2. PSO2. Ability to Design and Develop Mobile and Web-based applications under realistic constraints.

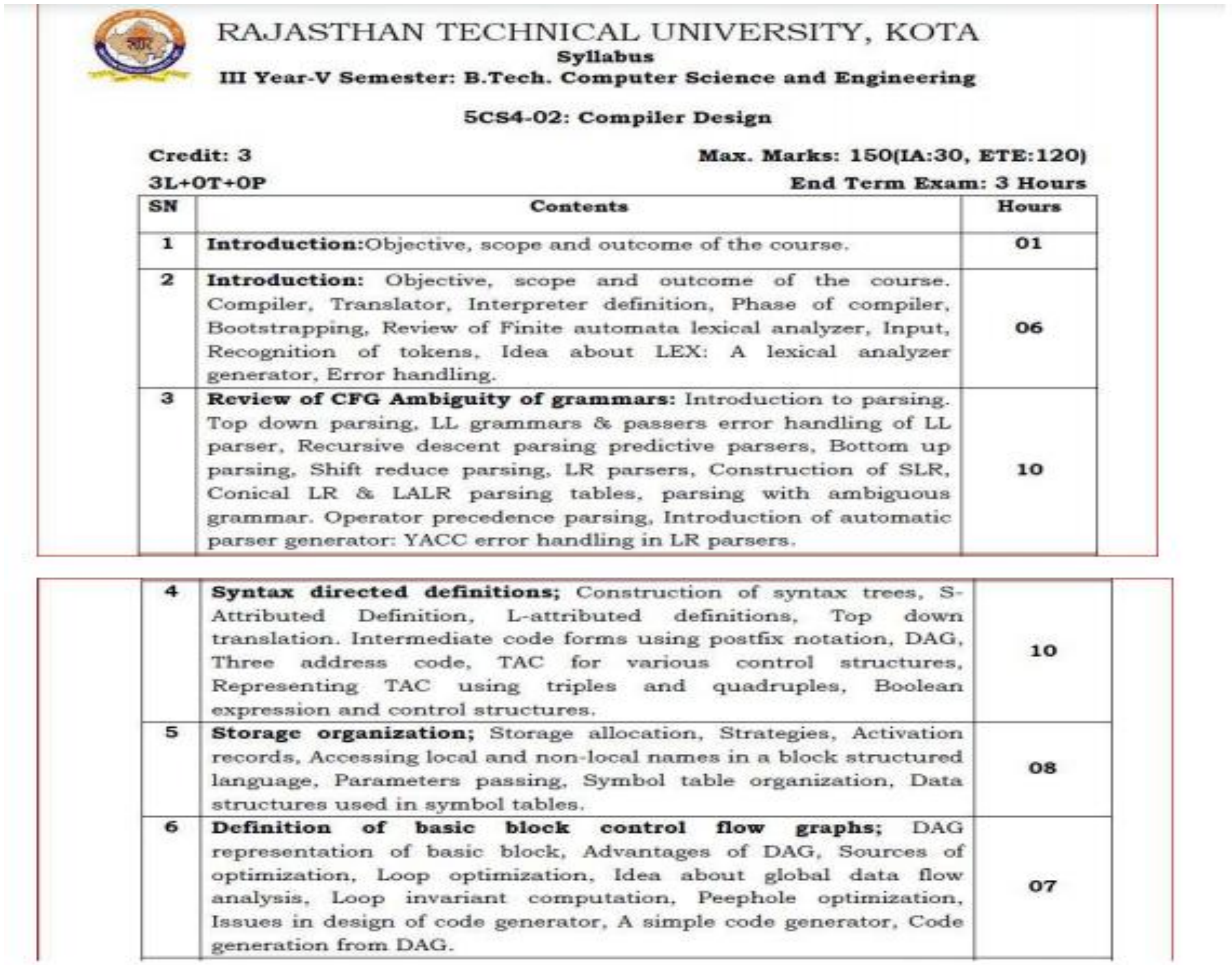
COURSE OUTCOME

CO1: Compare different phases of compiler and design lexical analyzer.
CO2: Examine syntax and semantic analyzer by understanding grammars.
CO3: Illustrate storage allocation and its organization & analyze symbol table organization.
CO4: Analyze code optimization, code generation & compare various compilers.

CO-PO MAPPING

Semester	Subject	Code	L/T/P	CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
V	COMPILER DESIGN	5CS4-02	L	1. Compare different phases of compiler and design lexical analyzer..	3	3	3	3	1	1	1	1	1	2	1	3
			L	2. Examine syntax and semantic analyzer by understanding grammars.	3	3	3	2	1	1	1	0	1	2	1	3
			L	3. Illustrate storage allocation and its organization & analyze symbol table organization.	3	3	2	2	1	1	1	1	1	2	1	3
			L	4.Analyze code optimization, code generation & compare various compilers.	3	3	3	3	2	1	1	1	1	2	1	3

SYLLABUS



RAJASTHAN TECHNICAL UNIVERSITY, KOTA

Syllabus

III Year-V Semester: B.Tech. Computer Science and Engineering

5CS4-02: Compiler Design

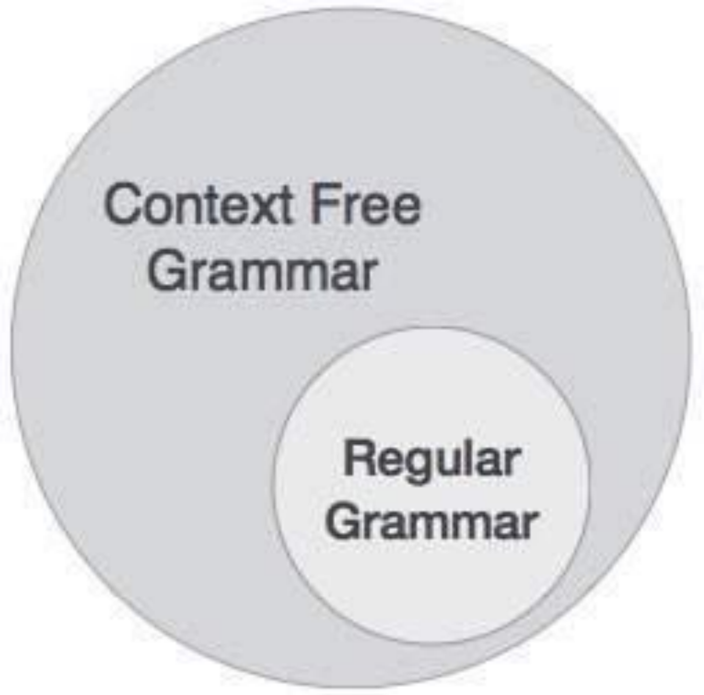
Credit: 3 **Max. Marks: 150(IA:30, ETE:120)**

3L+0T+0P **End Term Exam: 3 Hours**

SN	Contents	Hours
1	**Introduction:**Objective, scope and outcome of the course.	01
2	**Introduction:** Objective, scope and outcome of the course. Compiler, Translator, Interpreter definition, Phase of compiler, Bootstrapping, Review of Finite automata lexical analyzer, Input, Recognition of tokens, Idea about LEX: A lexical analyzer generator, Error handling.	06
3	**Review of CFG Ambiguity of grammars:** Introduction to parsing. Top down parsing, LL grammars & passers error handling of LL parser, Recursive descent parsing predictive parsers, Bottom up parsing, Shift reduce parsing, LR parsers, Construction of SLR, Conical LR & LALR parsing tables, parsing with ambiguous grammar. Operator precedence parsing, Introduction of automatic parser generator: YACC error handling in LR parsers.	10
4	**Syntax directed definitions;** Construction of syntax trees, S-Attributed Definition, L-attributed definitions, Top down translation. Intermediate code forms using postfix notation, DAG, Three address code, TAC for various control structures, Representing TAC using triples and quadruples, Boolean expression and control structures.	10
5	**Storage organization;** Storage allocation, Strategies, Activation records, Accessing local and non-local names in a block structured language, Parameters passing, Symbol table organization, Data structures used in symbol tables.	08
6	**Definition of basic block control flow graphs;** DAG representation of basic block, Advantages of DAG, Sources of optimization, Loop optimization, Idea about global data flow analysis, Loop invariant computation, Peephole optimization, Issues in design of code generator, A simple code generator, Code generation from DAG.	07

Syntax Analysis

- Syntax analysis or parsing is the second phase of a compiler.
- We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions.
- Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.
- CFG, on the other hand, is a superset of Regular Grammar.
- It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar.



Context-Free Grammar

A context-free grammar has four components:

A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.

A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.

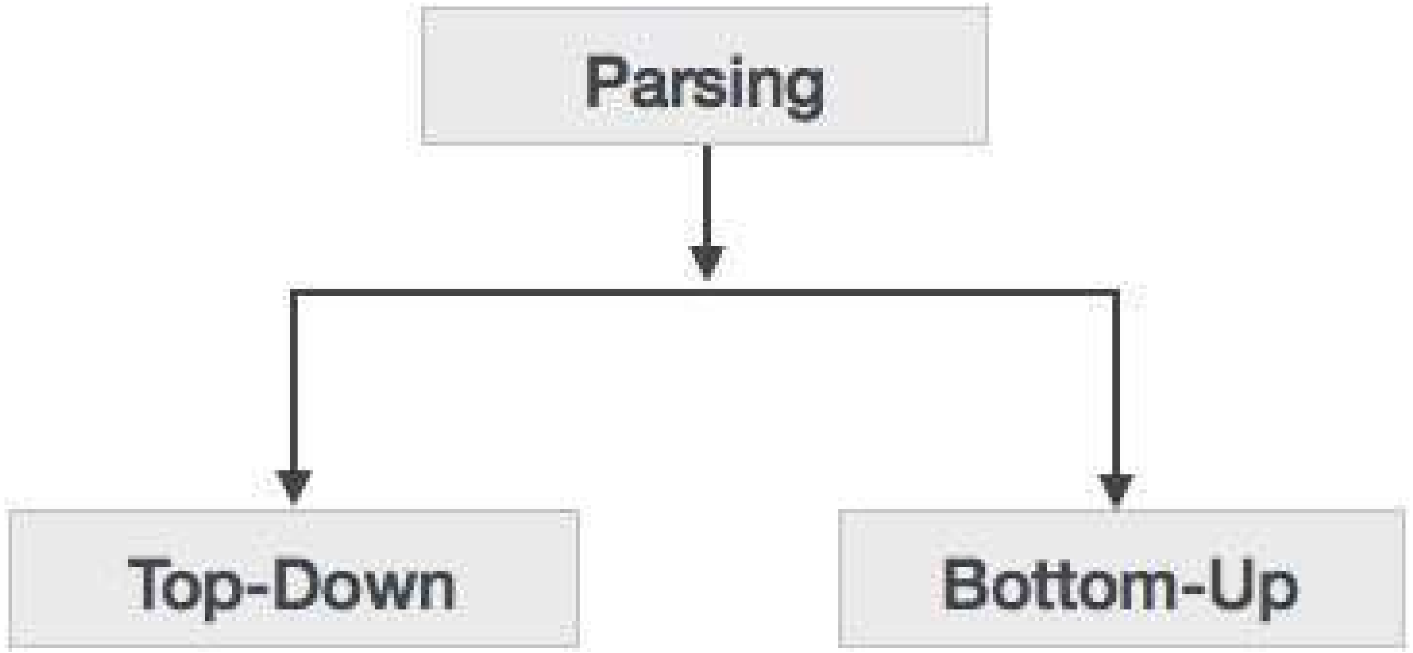
A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.

One of the non-terminals is designated as the start symbol (S); from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Compiler Design - Types of Parsing

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types : top-down parsing and bottom-up parsing.



Top-down Parsing

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

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

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

Bottom-up Parsing

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

Example:

Input string : a + b * c

Production rules:

$S \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow E * T$

$E \rightarrow T$

$T \rightarrow id$

Let us start bottom-up parsing

a + b * c

Read the input and check if any production matches with the input:

a + b * c

T + b * c

E + b * c

E + T * c

E * c

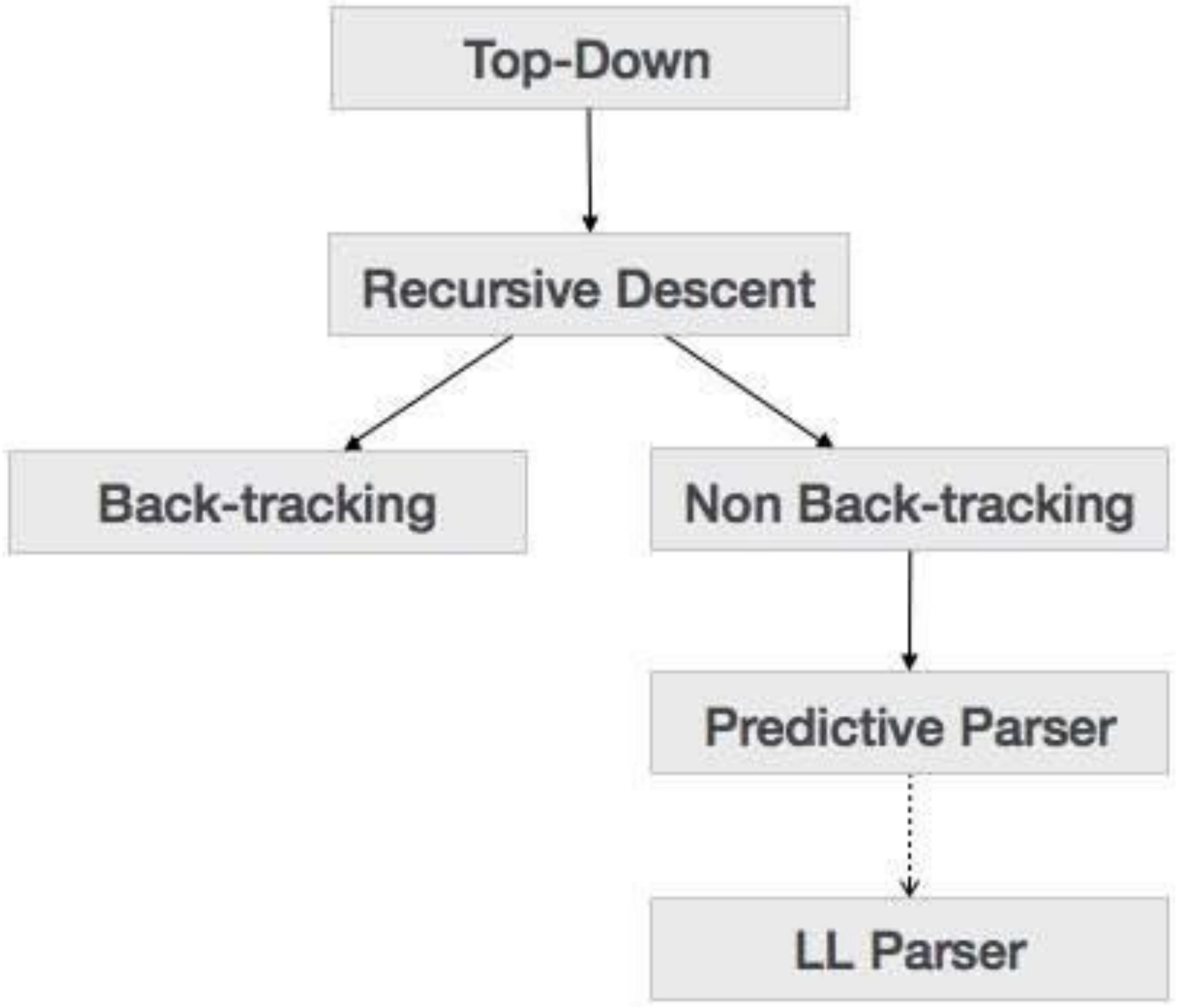
E * T

E

S

Top-Down Parser

We have learnt that the top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. The types of top-down parsing are depicted below:



Recursive Descent Parsing:

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

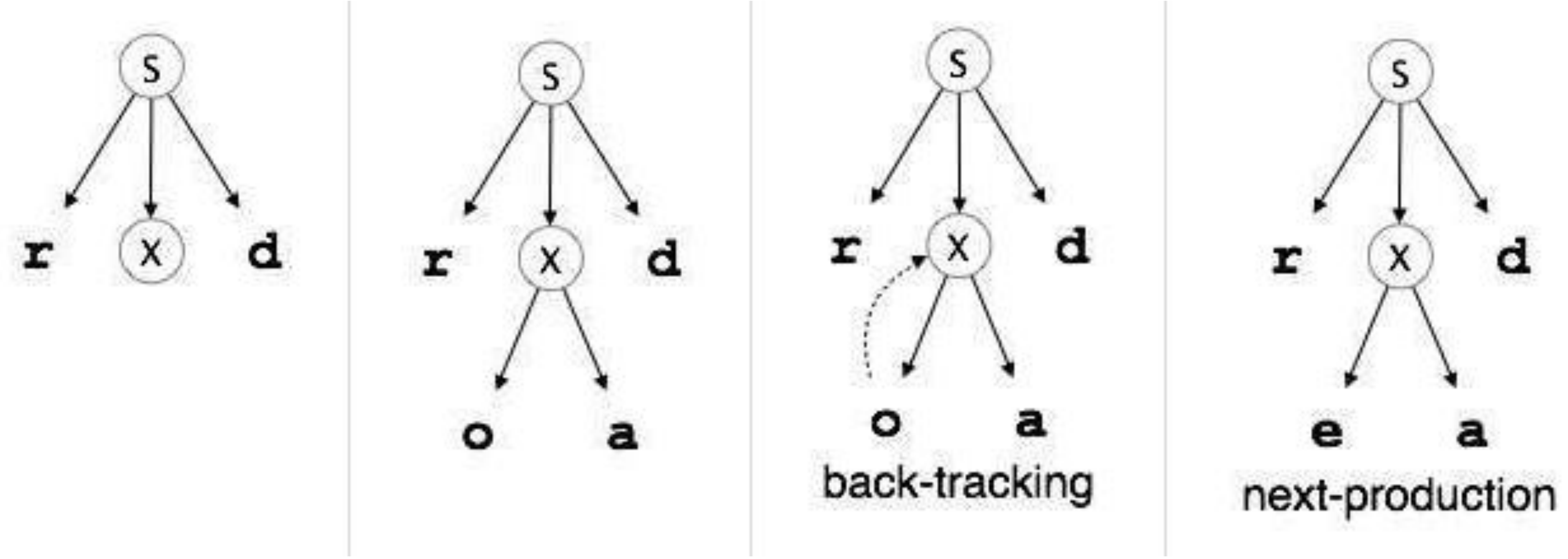
This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Back-tracking:

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

S → rXd | rZd
X → oa | ea
Z → ai

For an input string: read, a top-down parser, will behave like this:
It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. ‘e’). The parser tries to expand non-terminal ‘X’ and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).
Now the parser matches all the input letters in an ordered manner. The string is accepted.



FIRST and FOLLOW in Compiler Design

FIRST(X) for a grammar symbol X is the set of terminals that begin the strings derivable from X.

Rules to compute FIRST set:

If x is a terminal, then FIRST(x) = { ‘x’ }
If x-> Є, is a production rule, then add Є to FIRST(x).
If X->Y1 Y2 Y3….Yn is a production,
    FIRST(X) = FIRST(Y1)
    If FIRST(Y1) contains Є then FIRST(X) = { FIRST(Y1) – Є } U { FIRST(Y2) }
    If FIRST (Yi) contains Є for all i = 1 to n, then add Є to FIRST(X).

Example 1

Production Rules of Grammar

E -> TE'
E' -> +T E'|Є
T -> F T'
T' -> *F T' | Є
F -> (E) | id

FIRST sets

FIRST(E) = FIRST(T) = { (, id }
FIRST(E') = { +, Є }
FIRST(T) = FIRST(F) = { (, id }
FIRST(T') = { *, Є }
FIRST(F) = { (, id }

Example 2:
Production Rules of Grammar
S -> ACB | Cbb | Ba
A -> da | BC
B -> g | Є
C -> h | Є

FIRST sets

FIRST(S) = FIRST(A) U FIRST(B) U FIRST(C)
= { d, g, h, Є, b, a}

FIRST(A) = { d } U FIRST(B) = { d, g , h, Є }

FIRST(B) = { g , Є }

FIRST(C) = { h , Є }

FOLLOW Set in Syntax Analysis

1) FOLLOW(S) = { $ } // where S is the starting Non-Terminal
2) If A -> pBq is a production, where p, B and q are any grammar symbols, then everything in FIRST(q) except Є is in FOLLOW(B).
3) If A->pB is a production, then everything in FOLLOW(A) is in FOLLOW(B).
4) If A->pBq is a production and FIRST(q) contains Є, then FOLLOW(B) contains { FIRST(q) – Є } U FOLLOW(A)

Production Rules:

E → TE’

E’ → +T E’|Є

T → F T’

T’ → *F T’ | Є

F → (E) | id

FIRST set

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

FIRST(E’) = { +, Є }

FIRST(T’) = { *, Є }

FOLLOW Set

FOLLOW(E) = { $,) } // Note ')' is there because of 5th rule

FOLLOW(E') = FOLLOW(E) = { $,) } // See 1st production rule

FOLLOW(T) = { FIRST(E') – Є } U FOLLOW(E') U FOLLOW(E) = { + , $,) }

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

FOLLOW(F) = { FIRST(T') – Є } U FOLLOW(T') U FOLLOW(T) = { *, +, $,) }

https://www.youtube.com/watch?v=pP1-ragPlkQ

Creating a predictive parsing Table

Input: Grammar *G*.
Output: Predictive parsing table *M*.
Method:
1. for (each production $A \rightarrow \alpha$ in *G*) follow step 2 and 3
2. for (each terminal *a* in FIRST(α)) add $A \rightarrow \alpha$ to *M*[*A*, *a*];
3. if (ε is in FIRST(α)) for (each symbol *b* in FOLLOW(*A*)) add $A \rightarrow \alpha$ to *M*[*A*, *b*] for each terminal b in follow (A). If ε is in First(α) and $ is in Follow(A) add $A \rightarrow \alpha$ to *M*[*A*, *$*];
4. make each undefined entry of *M* be **error**;

https://www.youtube.com/watch?v=KbTkcSz4ulE

Grammar G

E → TE'

E' → +T E'|Є

T → F T'

T' → *F T' | Є

F → (E) | id

FIRST set

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

FIRST(E') = { +, Є }

FIRST(T') = { *, Є }

FOLLOW set

FOLLOW(E) = { $,) }

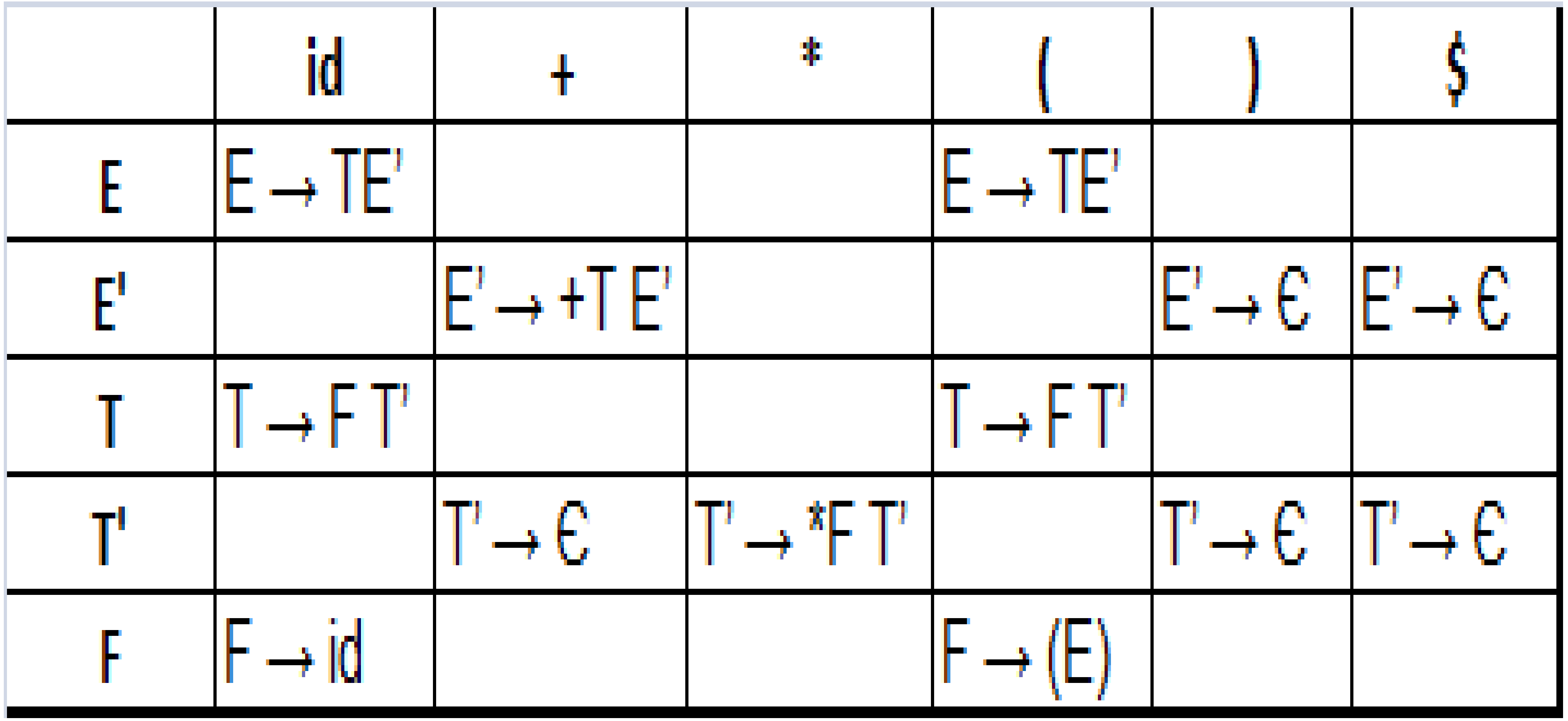
FOLLOW(E') = FOLLOW(E) = { $,) }

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

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

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

Parsing Table for Grammar G



	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

LL(1) Parsing:

Here the 1st **L** represents that the scanning of the Input will be done from Left to Right manner and second **L** shows that in this Parsing technique we are going to use Left most Derivation Tree. and finally the **1** represents the number of look ahead, means how many symbols are you going to see when you want to make a decision.

Construction of LL(1) Parsing Table:

Note: Every grammar is not feasible for LL(1) Parsing table.
It may be possible that one cell may contain more than one production.
Let’s see with an example.

Parsing Table:

Example-2:
Consider the Grammar
S --> A | a
A --> a

	a	$
S	S –> A, S –> a	
A	A –> a	

Operator Precedence Parsing:

Any Grammar G is called operator precedence grammar if it meets the following conditions

1. If there exist no production rule which contain no Є (Epsilon) on its right hand side.

2. There exist no production rule which contain two non terminal adjacent to each other on its RHS.

Operator Precedence Parsing**:**

An operator precedence parser is a bottom-up parser that interprets an operator grammar. This parser is only used for operator grammars. *Ambiguous grammars are not allowed* in any parser except operator precedence parser.
There are two methods for determining what precedence relations should hold between a pair of terminals:

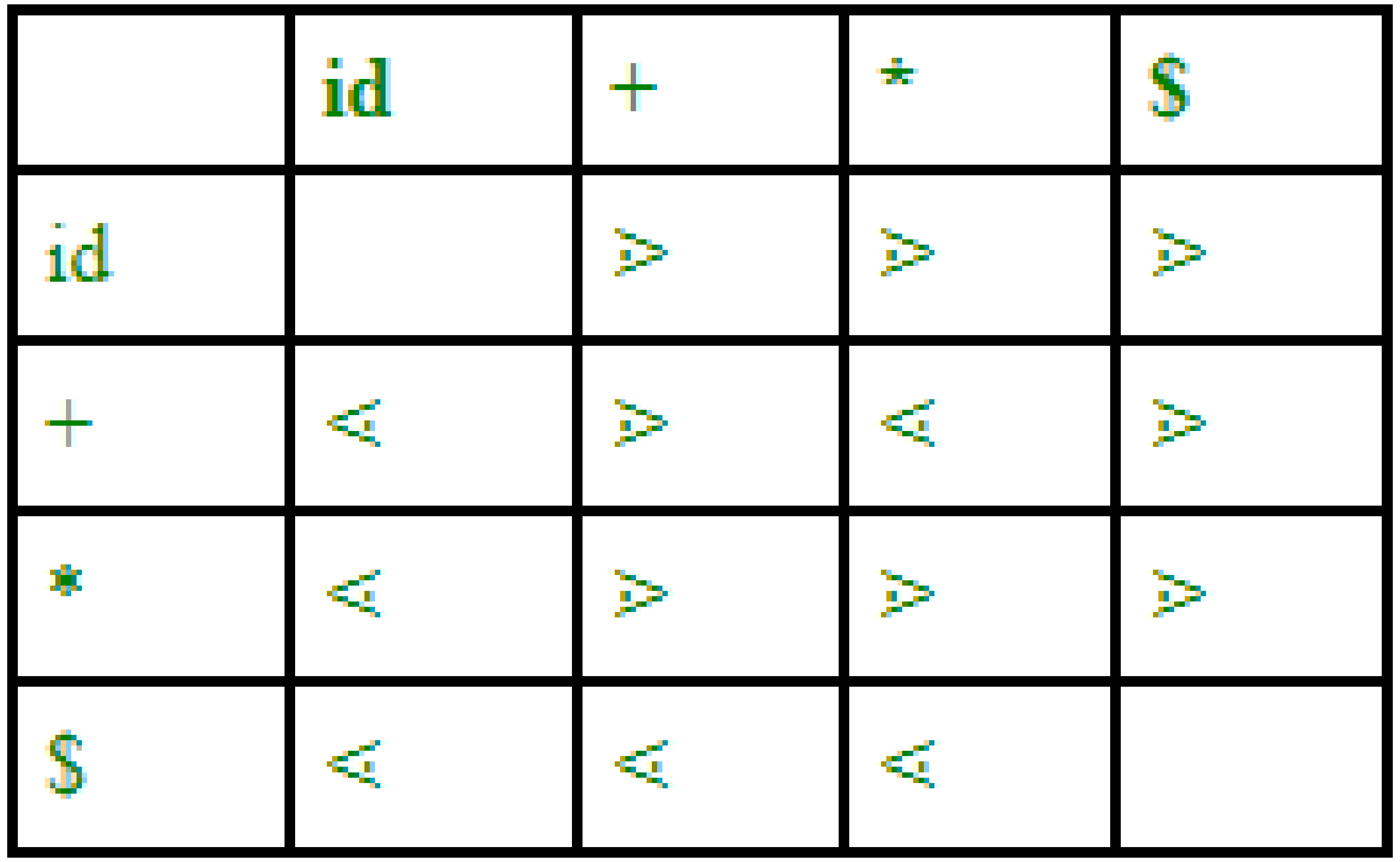
1. Use the conventional associatively and precedence of operator.
2. The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associatively and precedence in its parse trees.

This parser relies on the following three precedence relations: **⋖, ≐, ⋗**

a ⋖ b This means a “yields precedence to” b.
a ⋗ b This means a “takes precedence over” b.
a ≐ b This means a “has same precedence as” b.

Parsing Action

1. Both end of the given input string, add the \$ symbol.
2. scan the input string from left right until the $\gtrdot$ is encountered.
3. Scan towards left over all the equal precedence until the first left most $\lessdot$ is encountered.
4. Everything between left most $\lessdot$ and right most $\gtrdot$ is a handle.
5. \$ on \$ means parsing is successful.



	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

Grammar:

E → E+T/T

T → T*F/F

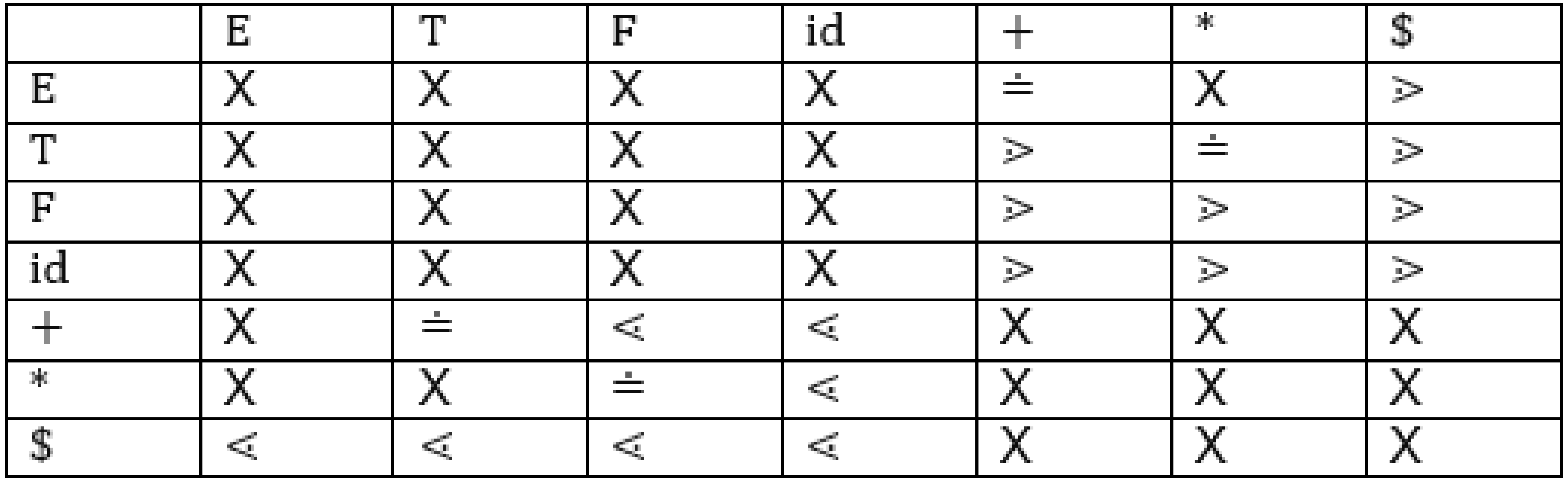
F → id

Given string:

w = id + id * id

.

On the basis of above tree, we can design following operator precedence table:



	E	T	F	id	+	*	$
E	X	X	X	X	$\doteq$	X	$\gtrdot$
T	X	X	X	X	$\gtrdot$	$\doteq$	$\gtrdot$
F	X	X	X	X	$\gtrdot$	$\gtrdot$	$\gtrdot$
id	X	X	X	X	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	X	$\doteq$	$\lessdot$	$\lessdot$	X	X	X
*	X	X	$\doteq$	$\lessdot$	X	X	X
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	X	X	X

$\$ \lessdot \text{id1} \gtrdot + \text{id2} * \text{id3} \$$

$\$ \lessdot F \gtrdot + \text{id2} * \text{id3} \$$

$\$ \lessdot T \gtrdot + \text{id2} * \text{id3} \$$

$\$ \lessdot E \doteq + \lessdot \text{id2} \gtrdot * \text{id3} \$$

$\$ \lessdot E \doteq + \lessdot F \gtrdot * \text{id3} \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \lessdot \text{id3} \gtrdot \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \doteq F \gtrdot \$$

$\$ \lessdot E \doteq + \doteq T \gtrdot \$$

$\$ \lessdot E \doteq + \doteq T \gtrdot \$$

$\$ \lessdot E \gtrdot \$$

Accept.

Introduction to YACC

YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File:

YACC input file is divided in three parts

```
/* definitions */
....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```

Input File: Definition Part:

The definition part includes information about the tokens used in the syntax definition:
%token NUMBER
%token ID
Yacc automatically assigns numbers for tokens, but it can be overridden by % token NUMBER 621
Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should not overlap ASCII codes.
The definition part can include C code external to the definition of the parser and variable declarations, within **%{** and **%}** in the first column.
It can also include the specification of the starting symbol in the grammar:%start nonterminal

Input File: Rule Part:

The rules part contains grammar definition in a modified BNF form.
Actions is C code in { } and can be embedded inside (Translation schemes).

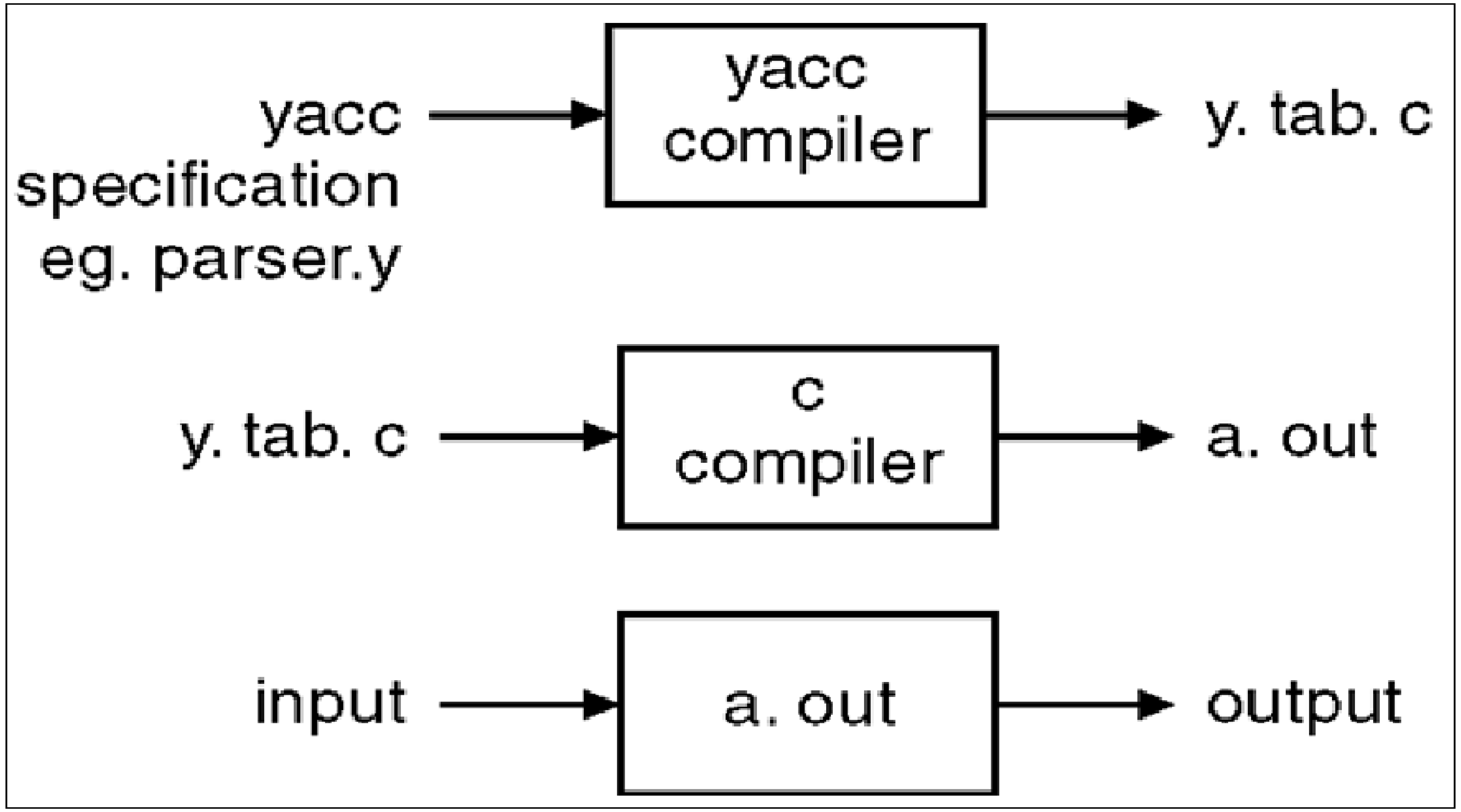
Input File: Auxiliary Routines Part:

The auxiliary routines part is only C code.
It includes function definitions for every function needed in rules part.
It can also contain the main() function definition if the parser is going to be run as a program.
The main() function must call the function yyparse().

Input File:
If yylex() is not defined in the auxiliary routines sections, then it should be included:#include "lex.yy.c"
YACC input file generally finishes with: .y

Output Files:

The output of YACC is a file named **y.tab.c**
If it contains the **main()** definition, it must be compiled to be executable.
Otherwise, the code can be an external function definition for the function **int yyparse()**
If called with the **–d** option in the command line, Yacc produces as output a header file **y.tab.h** with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
If called with the **–v** option, Yacc produces as output a file **y.output** containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.
Example:
Yacc File (.y)


yacc specification eg. parser.y
yacc compiler
y. tab. c
y. tab. c
c compiler
a. out
input
a. out
output

REFERENCES/BIBLOGRAPHY

1. https://www.tutorialspoint.com/compiler_design/
2. http://www1.cs.columbia.edu/~aho/cs4115/lectures/13-02-20.htm
3. https://www.youtube.com/watch?v=KbTkcSz4ulE
4. https://www.youtube.com/watch?v=pP1-ragPlkQ
5. https://www.javatpoint.com/operator-precedence-parsing
6. https://www.youtube.com/watch?v=hF9aIV5H7Xo

JECRC Foundation

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

Thank you!