



# NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE (NAAC Accredited)

(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)



## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

### ***COURSE MATERIALS***



### ***CS 304 COMPILER DESIGN***

#### **VISION OF THE INSTITUTION**

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

#### **MISSION OF THE INSTITUTION**

**NCERC** is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

## ABOUT DEPARTMENT

- ◆ Established in: 2002
- ◆ Course offered : B.Tech in Computer Science and Engineering  
M.Tech in Computer Science and Engineering  
M.Tech in Cyber Security
- ◆ Approved by AICTE New Delhi and Accredited by NAAC
- ◆ Affiliated to the University of A P J Abdul Kalam Technological University.

## DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

## DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.

## PROGRAMME EDUCATIONAL OBJECTIVES

- PEO 1:** Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.
- PEO 2:** Graduates will be able to Analyze, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.
- PEO 3:** Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.
- PEO 4:** Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Team work and leadership qualities.



## PROGRAM OUTCOMES (POS)

### Engineering Graduates will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **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.
4. **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.
5. **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.
6. **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.
7. **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.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **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.
11. **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.
12. **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 SPECIFIC OUTCOMES (PSO)

**PSO1:** Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

**PSO2:** Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance

optimization.

**PSO3:** Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

## COURSE OUTCOMES

C311.1	To acquire the knowledge on concepts and different phases of compilation with compile time error handling.
C311.2	To design lexical analyzer for a language and can represent language tokens using regular expressions, context free grammar and finite automata
C311.3	To acquire the knowledge on top down and bottom up parsers, and can develop appropriate parser to produce parse tree representation of the input.
C311.4	To generate intermediate code for statements in high level language.
C311.5	To design syntax directed translation schemes for a given context free grammar
C311.6	To apply optimization techniques to intermediate code and generate machine code for high level language program.

## MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

CO'S	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C311.1	3	3										
C311.2	2	3	3	2	3							
C311.3	3	3	3									
C311.4	2	3	3	3	3							
C311.5	3	3	3	3								
C311.6	3	3		3	3							
C311	3	3	3		3							

CO'S	PSO1	PSO2	PSO3
C311.1	3		
C311.2	3	3	
C311.3	3	3	
C311.4			3
C311.5		3	
C311.6		3	3
C311	3	3	3

**Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1**

## SYLLABUS

Course code	Course Name	L-T-P Credits	Year of Introduction
CS304	COMPILER DESIGN	3-0-0-3	2016
<b>Prerequisite: Nil</b>			
<b>Course Objectives</b>			
<ul style="list-style-type: none"> <li>To provide a thorough understanding of the internals of Compiler Design.</li> </ul>			
<b>Syllabus</b>			
Phases of compilation, Lexical analysis, Token Recognition, Syntax analysis, Bottom Up and Top Down Parsers, Syntax directed translation schemes, Intermediate Code Generation, Triples and Quadruples, Code Optimization, Code Generation.			
<b>Expected Outcome</b>			
The students will be able to			
<ol style="list-style-type: none"> <li>Explain the concepts and different phases of compilation with compile time error handling.</li> <li>Represent language tokens using regular expressions, context free grammar and finite automata and design lexical analyzer for a language.</li> <li>Compare top down with bottom up parsers, and develop appropriate parser to produce parse tree representation of the input.</li> <li>Generate intermediate code for statements in high level language.</li> <li>Design syntax directed translation schemes for a given context free grammar.</li> <li>Apply optimization techniques to intermediate code and generate machine code for high level language program.</li> </ol>			
<b>Text Books</b>			
<ol style="list-style-type: none"> <li>Aho A. Ravi Sethi and D Ullman. Compilers – Principles Techniques and Tools, Addison Wesley, 2006.</li> <li>D. M.Dhamdhare, System Programming and Operating Systems, Tata McGraw Hill &amp; Company, 1996.</li> </ol>			
<b>References</b>			
<ol style="list-style-type: none"> <li>Kenneth C. Loudon, Compiler Construction – Principles and Practice, Cengage Learning Indian Edition, 2006.</li> <li>Tremblay and Sorenson, The Theory and Practice of Compiler Writing, Tata McGraw Hill &amp; Company, 1984.</li> </ol>			

Course Plan			
Module	Contents	Hours	End Sem. Exam Marks
I	Introduction to compilers – Analysis of the source program, Phases of a compiler, Grouping of phases, compiler writing tools – bootstrapping <b>Lexical Analysis:</b> The role of Lexical Analyzer, Input Buffering, Specification of Tokens using Regular Expressions, Review of Finite Automata, Recognition of Tokens.	07	15%
II	<b>Syntax Analysis:</b> Review of Context-Free Grammars – Derivation trees and Parse Trees, Ambiguity. <b>Top-Down Parsing:</b> Recursive Descent parsing, Predictive parsing, LL(1) Grammars.	06	15%

FIRST INTERNAL EXAM			
III	<b>Bottom-Up Parsing:</b> Shift Reduce parsing – Operator precedence parsing (Concepts only) LR parsing – Constructing SLR parsing tables, Constructing, Canonical LR parsing tables and Constructing LALR parsing tables.	07	15%
IV	<b>Syntax directed translation:</b> Syntax directed definitions, Bottom- up evaluation of S-attributed definitions, L- attributed definitions, Top-down translation, Bottom-up evaluation of inherited attributes. <b>Type Checking :</b> Type systems, Specification of a simple type checker.	08	15%
SECOND INTERNAL EXAM			
V	<b>Run-Time Environments:</b> Source Language issues, Storage organization, Storage-allocation strategies. <b>Intermediate Code Generation (ICG):</b> Intermediate languages – Graphical representations, Three-Address code, Quadruples, Triples. Assignment statements, Boolean expressions.	07	20%
VI	<b>Code Optimization:</b> Principal sources of optimization, Optimization of Basic blocks <b>Code generation:</b> Issues in the design of a code generator. The target machine, A simple code generator.	07	20%
END SEMESTER EXAM			

#### Question Paper Pattern

- There will be *five* parts in the question paper – A, B, C, D, E
- Part A
  - Total marks : 12      b.. Four questions each having 3 marks, uniformly covering modules I and II; All four questions have to be answered.
- Part B
  - Total marks : 18      b. Three questionseach having 9 marks, uniformly covering modules I and II; Two questions have to be answered. Each question can have a maximum of three subparts.
- Part C
  - Total marks : 12      b. Four questions each having 3 marks, uniformly covering modules III and IV; All four questions have to be answered.
- Part D
  - Total marks : 18      b. Three questions each having 9 marks, uniformly covering modules III and IV; Two questions have to be answered. Each question can have a maximum of three subparts
- Part E
  - Total Marks: 40      b. Six questions each carrying 10 marks, uniformly covering modules V and VI; four questions have to be answered.
  - A question can have a maximum of three sub-parts.
- There should be at least 60% analytical/numerical questions.



## QUESTION BANK

MODULE I				
Q:N O:	QUESTIONS	CO	K L	PAG E NO:
1	Construct a parse tree for position:=initial+rate *100	CO 1	K5	13
2	With neat sketch explain the phases of a compiler.	CO 1	K3	21
3	Discuss in detail about the necessity of compiler writing tools.	CO 1	K2	32
4	With neat sketch explain the working of bootstrapping	CO 1	K3	34
5	Explain the role of lexical analyzer.	CO 1	K2	37
6	Discuss about the functions of lexical analyzer.	CO 1	K2	38
7	Define the terms Tokens, Patterns and Lexemes.	CO 1	K3	39
8	Identify the tokens and lexemes of the given expression. Printf(“Total=%d\n”, score);	CO 1	K2	40
9	Explain input buffering.	CO 1	K2	44
10	Explain about regular expression in detail.	CO 1	K4	50
11	Explain Regular Definition with notational short hands.	CO 1	K2	52
MODULE II				
1	Define syntax analysis with example	CO 2	K2	62
2	Elucidate in detail about the context free grammar.	CO 2	K4	68

3	Derive the string $-(id+id)$ from the grammar $E \rightarrow E+E   E * E   (E)   -E   id$	CO 2	K2	71
4	Derive the string "aabbabba" for the LMD and RMD using context free grammar.	CO 2	K5	73
5	Construct a parse tree for the string $id+id*id$ and grammar G is $E \rightarrow E * E   E + E   id$ .	CO 2	K5	75
6	Elucidate in detail about Ambiguity.	CO 2	K3	77
7	<p><b>EXAMPLE</b></p> <p>consider the following grammar for arithmetic expression.</p> $E \rightarrow E + T   T$ $T \rightarrow T * F   F$ $F \rightarrow (E)   id$ <p>Eliminating the immediate left recursion to the prod for E and then for T, we obtain</p> $E \rightarrow TE'$ $E' \rightarrow +TE'   \epsilon$ $T \rightarrow FT'$ $T' \rightarrow *FT'   \epsilon$ $F \rightarrow (E)   id$	CO 2	K5	80
8	<p>Eliminate nondeterministic (ND) from the below grammar.</p> $S \rightarrow a s s b s$ $  a s a s b$ $  a b b$ $  b$	CO 2	K2	85
9	Discuss in detail about the Recursive Descent Parsing.	CO 2	K2	88
10	Narrate the properties of predictive parser.	CO 2	K3	90



11	Construct the transition diagram for the predictive parser of the grammar.  <div style="border: 1px solid black; padding: 10px; width: fit-content; margin: 10px auto;"> <math display="block">\begin{aligned} E &amp;\rightarrow TE' \\ E' &amp;\rightarrow +TE' \mid \epsilon \\ T &amp;\rightarrow FT' \\ T' &amp;\rightarrow *FT' \mid \epsilon \\ F &amp;\rightarrow (E) \mid id \end{aligned}</math> </div>	CO 2	K2	92
12	Explain in detail about Non Recursive Predictive Parser.	CO 2	K2	94
13	Explain the FIRST and FOLLOW with example	CO 2	K2	97
14	construct the parsing Table for the following grammar.  $\begin{aligned} S &\rightarrow AB \\ A &\rightarrow a/\epsilon \\ B &\rightarrow b/\epsilon \end{aligned}$	CO 2	K2	106
<b>MODULE III</b>				
1	Discuss in detail about bottom up parsing.	CO 3	K3	109
2	Discuss the basic properties of shift reduce parsing.	CO 3	K3	112
3	Consider the following grammar  $\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ E &\rightarrow id \end{aligned}$ Perform the action of shift-reduce parser to the input string $id_1 * (id_2 + id_3)$ .	CO 3	K2	116

4	Discuss in detail about operator precedence parser.	CO 3	K3	120
5	Construct operator precedence parsing table for the following given grammar $P \rightarrow SR   S$ $R \rightarrow bSR   bs$ $S \rightarrow Wbs   W$ $W \rightarrow L * W   L$ $L \rightarrow id$	CO 3	K5	129
6	Discuss in detail about LR Parser.	CO 3	K3	130
7	Construct the SLR Parsing table for grammar below. $E \rightarrow E + T$ $E \rightarrow T$ $T \rightarrow T * F$ $T \rightarrow F$ $F \rightarrow (E)$ $F \rightarrow id$	CO 3	K5	142
8	Explain canonical LR parser.	CO 3	K2	151
9	Explain canonical LALR parser.	CO 3	K2	159
<b>MODULE IV</b>				
1	Briefly explain syntax directed translation.	CO 4	K2	171
2	Explain the concept of synthesized attribute.	CO 4	K1	175
3	Briefly explain about bottom up evaluation of S-attribute	CO 4	K2	187

	definitions.			
4	Describe L-attributed definition.	CO 4	K3	191
5	Explain the working of Top Down Translation.	CO 4	K1	194
6	Explain the basic bottom up evaluation of inherited attributes.	CO 4	K2	197
7	Define type checking.	CO 4	K3	202
<b>MODULE V</b>				
1	Describe Runtime Environment in detail.	CO 5	K4	212
2	Explain about Activation Trees.	CO 5	K2	214
3	Write a short note on storage organization.	CO 5	K3	218
4	State storage allocation strategies.	CO 5	K2	221
5	Write about intermediate code generation.	CO 5	K3	231
6	Briefly explain about intermediate language.	CO 5	K2	231
7	What are the main categories of three address code? Explain in detail.	CO 5	K2	235
8	Briefly explain about Assignment Statements.	CO 5	K3	244
<b>MODULE VI</b>				
1	Describe Code Optimization in detail.	CO 5	K4	251
2	Explain about basic blocks and aviation graphs.	CO 5	K2	256
3	Write a short note on function preserving transformation.	CO 5	K3	260

4	State common sub expression elimination.	CO 5	K2	260
5	Write about copy propagation	CO 5	K3	261

## APPENDIX 1

### CONTENT BEYOND THE SYLLABUS

S:NO;	TOPIC	PAGE NO:
1	ANTLR	263

## MODULE 1

Introduction to compilers — Analysis of the source program, Phases of a compiler, Grouping of Phases, Compiler writing tools — bootstrapping.

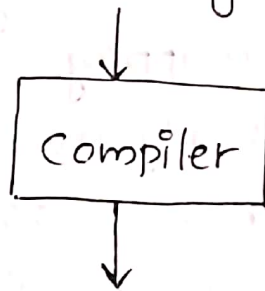
### Lexical Analysis:

The role of Lexical Analyzer, Input Buffering, Specification of Tokens using Regular Expressions, Review of Finite Automata, Recognition of Tokens.

### ① COMPILER

A compiler is a program that can read a program in one language — the source language — and translate it into an equivalent program in another language — the target language.

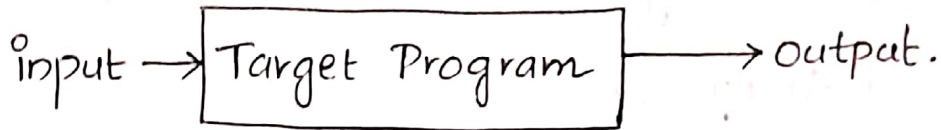
Source Program



target Program.

An important role of the compiler is to report any errors in the source program that it detects during the translation process.

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.



An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

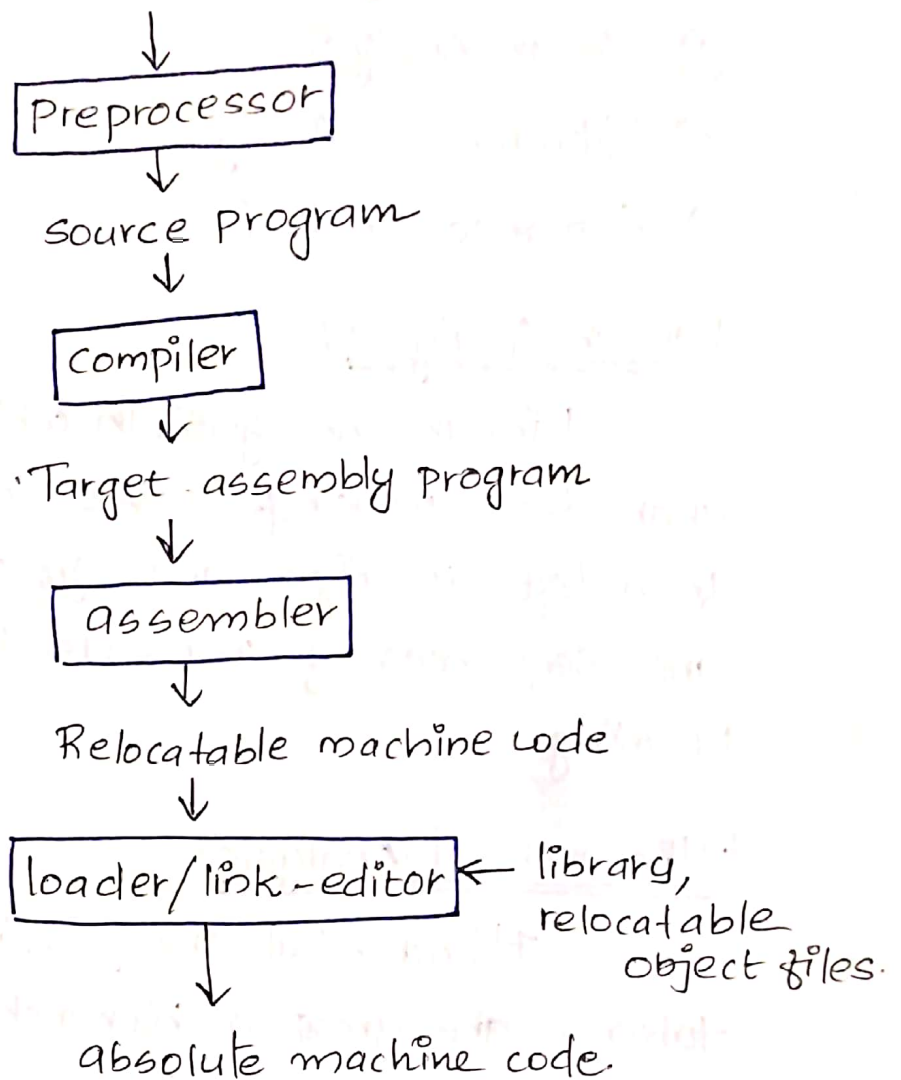


The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.



# A Language Processing System:

Skeletal source Program



A Source Program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a distinct program, called a Preprocessor.

Figure shows a typical "compilation." The target program created by the compiler may require further processing before it can be run. The compiler creates assembly code that is translated by an assembler into machine code and then linked together with some library routines into the code that actually runs on that machine.

# ANALYSIS OF THE SOURCE PROGRAM

The analysis consist of 3 Phases:

- ① Linear Analysis
- ② Hierarchical Analysis
- ③ Semantic Analysis.

## Linear Analysis:

Linear Analysis, in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

## Hierarchical Analysis:

Hierarchical analysis, in which characters or tokens are grouped hierarchically into nested collections with collective meaning.

## Semantic Analysis:

Semantic Analysis, in which certain checks are performed to ensure that the components of a program fit together meaningfully.

## ■ LEXICAL ANALYSIS (LINEAR ANALYSIS)

In a compiler, Linear analysis is called Lexical analysis or Scanning.

For example; in lexical analysis the characters in the assignment statement

$\text{Position} := \text{initial} + \text{rate} * 60$

would be grouped into following tokens:

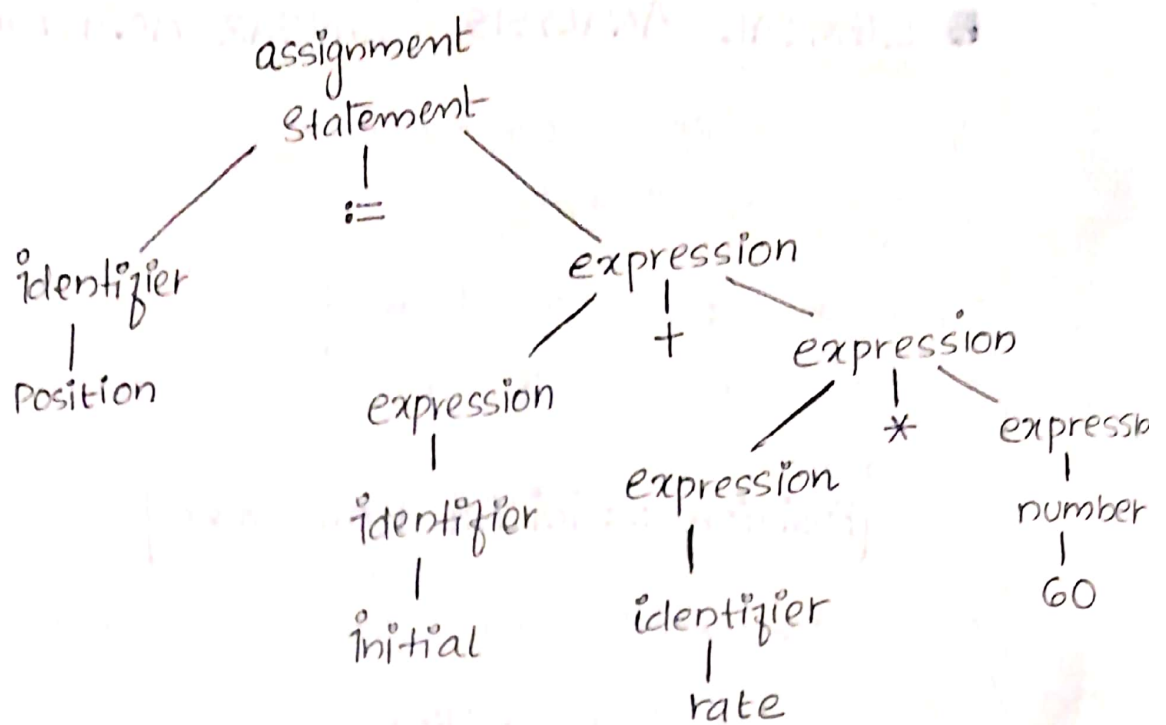
- ① The identifier Position
- ② The assignment symbol  $:=$
- ③ The identifier initial
- ④ The Plus sign.
- ⑤ The identifier rate
- ⑥ The multiplication sign.
- ⑦ The number 60.

The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

## ■ SYNTAX ANALYSIS (HIERARCHICAL ANALYSIS)

Hierarchical analysis is also called Parsing or syntax analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, the grammatical phrases of the source program are represented by a parse tree.





**Figure:** Parse tree for Position := initial + rate \* 60

The hierarchical structure of a program is usually expressed by recursive rules.

For example, we might have the following rules as part of the definition of expressions:

- ① Any identifier is an expression.
- ② Any number is an expression.
- ③ If expression<sub>1</sub> and expression<sub>2</sub> are expressions, then so are

expression<sub>1</sub> + expression<sub>2</sub>

expression<sub>1</sub> \* expression<sub>2</sub>

(expression<sub>1</sub>)

↳ Rules ① and ② are basic rules

③ defines expressions in terms of operators applied to other expressions.

Thus, by rule (1), initial and rate are expressions.  
by rule (2), GO is an expression,  
by rule (3), we can 1<sup>st</sup> infer that rate \* GO is  
an expression and finally that  
initial + rate \* GO is an expression

A parse tree describes the syntactic structure of the input. A syntax tree is a compressed representation of the parse tree in which the operators appear as the interior nodes, and the operands of an operator are the children of the node for that operator.

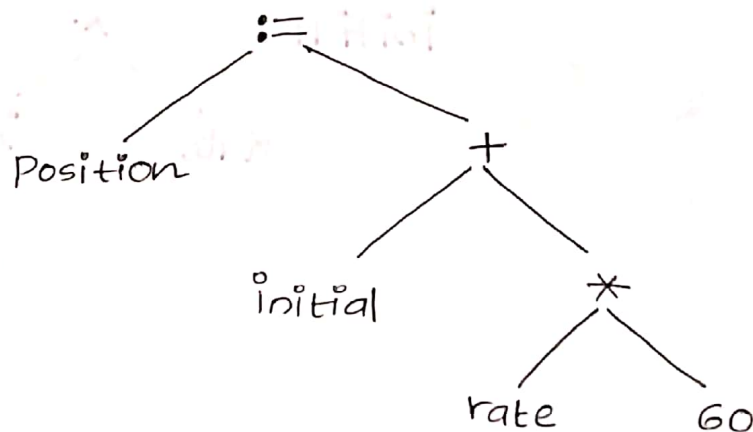


Figure : Syntax Tree.

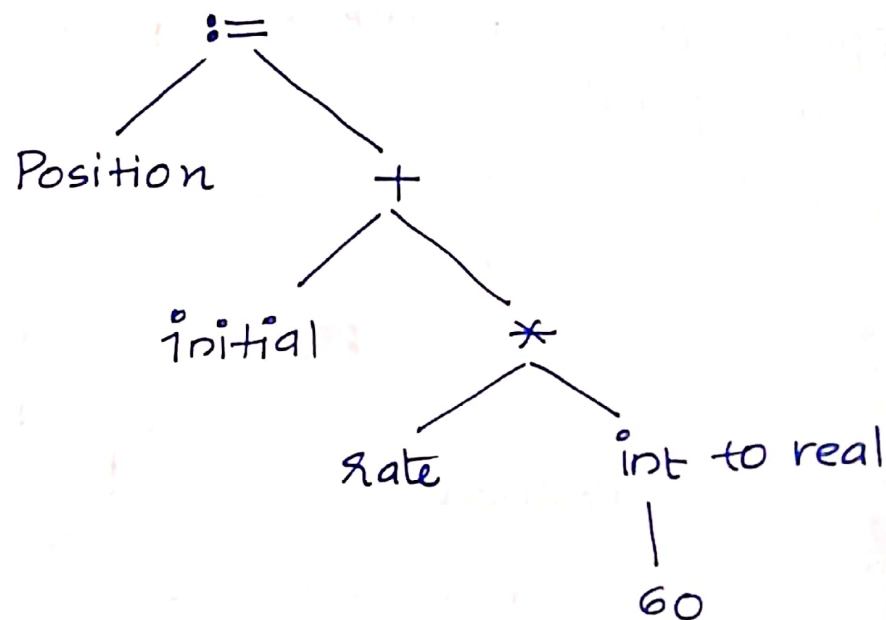
## ■ SEMANTIC ANALYSIS

The semantic analysis phase checks the source Program for semantic errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.

An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by the source language specification.

The output of the semantic analysis for the statement

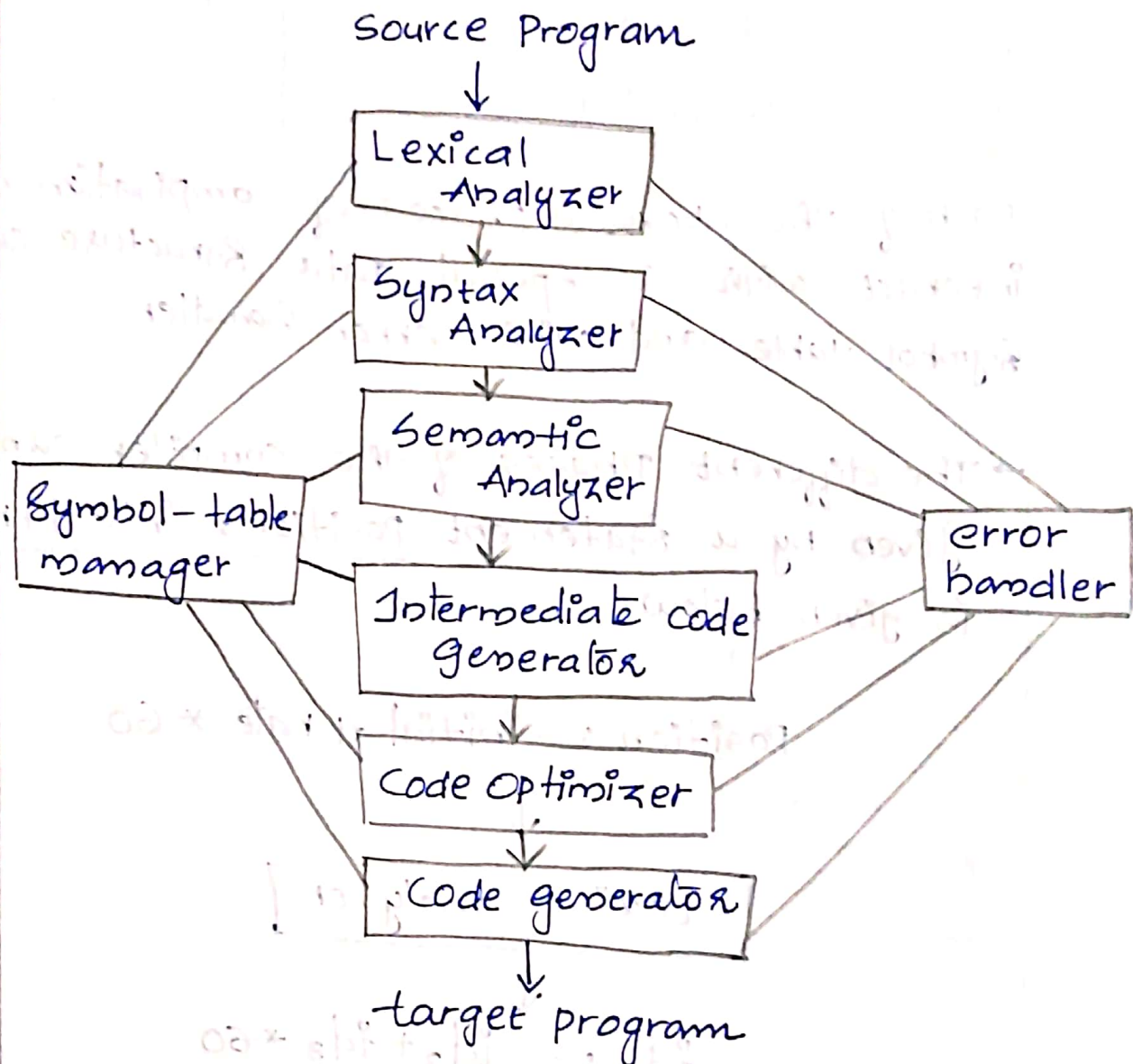
Position := initial + rate \* 60





# PHASES OF A COMPILER

A compiler operates in phases, each of which transforms the source program from one representation to another.



**Figure:** Phases of a compiler.

The complete compilation procedure is divided into six phases and is as shown in given below.

- ① Lexical Analyser
  - ② Syntax Analyser
  - ③ Semantic Analyser
  - ④ Intermediate code generator
  - ⑤ code Optimizer
  - ⑥ code generator
- } Analysis Portion
- } Synthesis Portion

Each of the above 6 phases of compilation can interact with a special data structure called symbol table and with error handler.

→ The different Phases of the compiler can be given by a statement Position := initial + rate \* 60 is given below

Position := initial + rate \* 60



Lexical analyzer

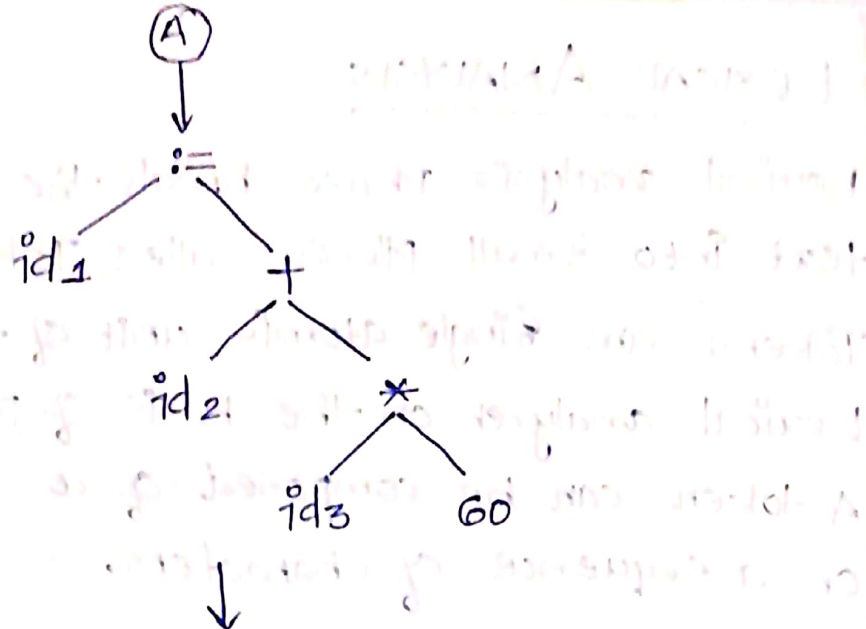


$id_1 := id_2 + id_3 * 60$

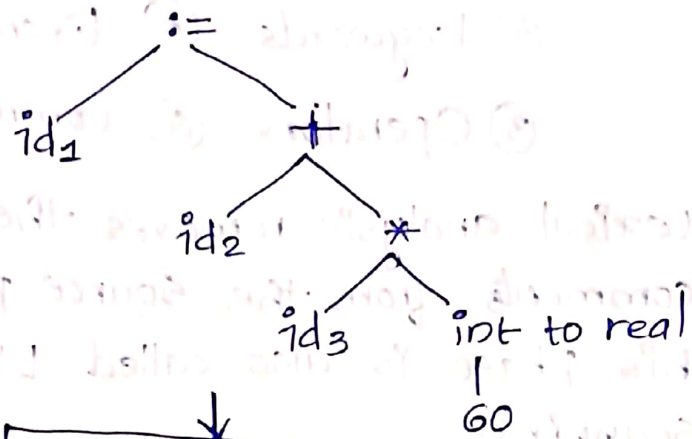


Syntax Analyzer

Ⓐ



Syntax  
semantic analyzer



Intermediate code generator

temp1 := int to real (60)  
temp2 := id3 \* temp1  
temp3 := id2 + temp2  
id1 := temp

Code Optimizer

temp1 := id3 \* 60.0  
id1 := id2 + temp1

Code generator

↓  
MOV F id3, R2  
MOV F #60.0, R2  
MOV F id2, R1  
ADDF R2, R1  
MOV F R1, id1



# LEXICAL ANALYSIS

Lexical analysis phase break the source code text into small pieces called Tokens.

Tokens are single atomic unit of the language.

Lexical analyzer on the basis of patterns.

A token can be composed of a single character or a sequence of characters.

→ The tokens can be classified as:

- ① Identifier      ④ Separators
- ② keywords      ⑤ literals
- ③ Operators      ⑥ comments.

→ Lexical analysis removes the white space and comments from the source program.

→ This phase is also called Linear Analysis or Scanning.

## Example

Position := Initial + rate \* 60

After the lexical analysis phase the identified tokens are as follows.

Position → Identifier

:= → assignment operator

Initial → Identifier

→ addition operator

rate → Identifier

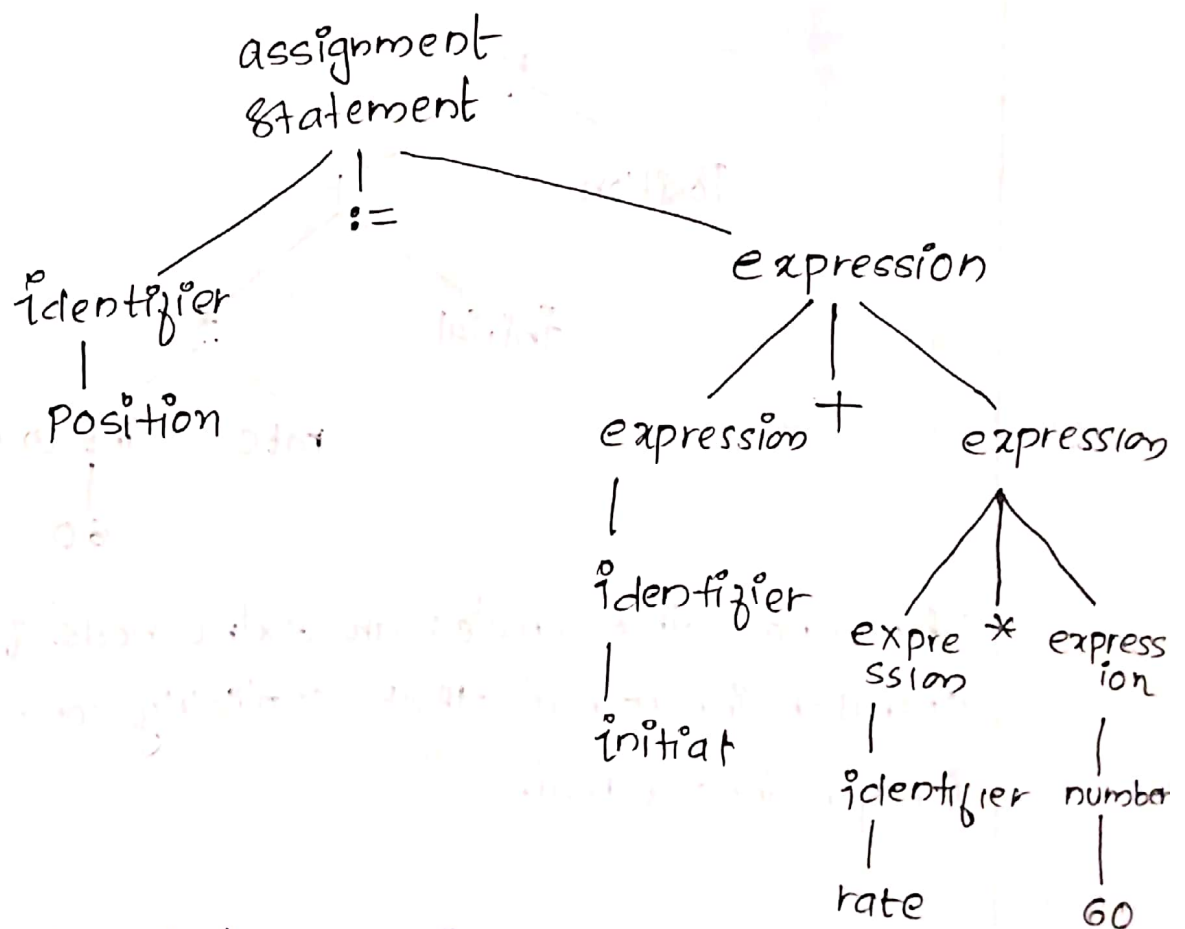
\* → multiplication Operator

60 → integer constant

# SYNTAX ANALYSIS (Parsing)

Syntax analysis or parsing involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.

- The grammatical phrases of a source program is represented by Parse Tree.
- Syntax errors are detected by the Syntax analyzer.
- The Parse Tree of the statement Position := initial rate \* 60 is as given below.

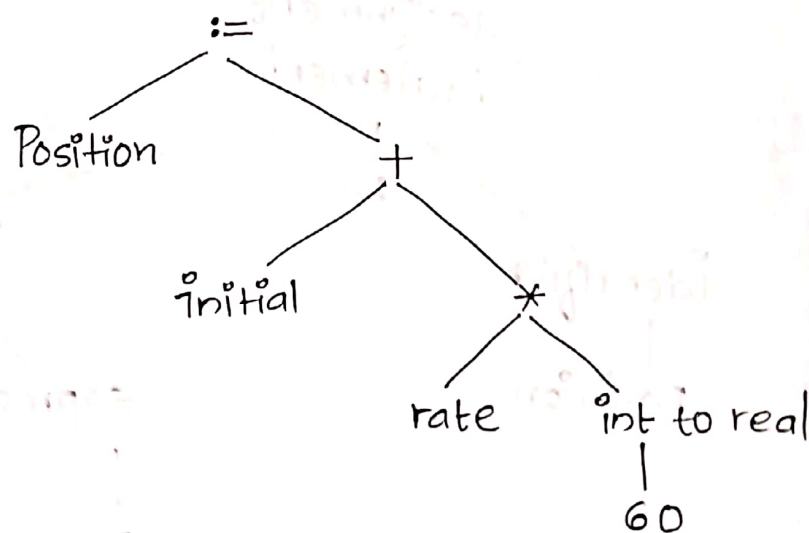


- If the syntax ~~tree~~ is correct, then we got the given input as we traverse through the generated parse tree.

# SEMANTIC ANALYZER

The Semantic analyzer gathers type information and checks the tree produced by the syntax analyzer for semantic errors. Semantic analysis performs type checking and reports compiler errors. In type checking, compiler checks that each operator has operands that are permitted by the source language.

**Example** The output of the semantic analysis for the Statement Position := initial + rate \* 60 is as given below by considering all identifiers as real.



The above tree creates an extra node for the Operator int to real that explicitly converts an integer into a real.



## INTERMEDIATE CODE GENERATION

Intermediate code generation phase generates an intermediate representation of the source program. There are different forms for the representation of intermediate code. one such form is called three address code. Three address code contain three operands per instruction. Three-address code generator has to create temporary location to hold the intermediate results.

### Example

The ICG of the statement Position := initial + rate \* 60 is given below:

temp<sub>1</sub> := int to real (60)

temp<sub>2</sub> := id<sub>3</sub> \* temp<sub>1</sub>

temp<sub>3</sub> := id<sub>2</sub> + temp<sub>2</sub>

id<sub>1</sub> := temp<sub>3</sub>

temp<sub>1</sub>, temp<sub>2</sub> and temp<sub>3</sub> are the names of ~~these~~ 3 temporary locations.

## CODE OPTIMIZATION

The code optimization phase attempts to improve the intermediate code, so that the machine code will run faster. code optimization includes loop optimization which removes unwanted statement out of the looping statement.

### Example

The optimized code for the statement Position := initial + rate \* 60 is given as follows.

$temp_1 := id_3 * 60.0$   
 $id_1 := id_2 + temp_1$

### CODE GENERATION

The final Phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected per each of the variables used by the program.

- In the code generation step the compiler has to map address names from the three-address intermediate code onto the very finite amount of registers that the machine had.
- Depending on the type of assembler code generator generates different type of code.

### Example

The code generation for the statement Position := initial + rate \* 60 is given as below.



MOVF  $id_3, R_2$

MOVF  $\#60.0, R_2$

MOVF  $id_2, R_1$

ADDF  $R_2, R_1$

MOVF  $R_1, id_1$

## SYMBOL TABLE

A Symbol Table is a data structure containing a record for each identifier with fields for the attributes of the identifier. These attributes may provide information about the storage allocated for an identifier its type and its scope.

Symbol Table allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. When an identifier in the source program is specified by the lexical analyzer, the identifier is entered into the Symbol table. The code generator enters and uses the detailed information about the storage assigned to identifiers.

## ERROR HANDLER

The main function of compiler is to detect and report of errors in source program. Each phase in the compiler encounters errors. Large fraction of errors are detected at the Syntax and Semantic analysis phase.

- Syntax analysis detect the tokens stream which violates the Syntax (Structure) rules of the language.
- During semantic analysis the compiler tries to direct constructs that have the right syntactic structure but no meaning to the operation involved.

## GROUPING OF PHASES

During the implementation of compiler, the activities of more than one phases are grouped together.

### ■ Front end and Back end:

- The front end consist of those phases that depend primarily on the source language and are legally independent of the target machine.
- The front end includes:
  - ① Lexical analysis
  - ② Syntactic analysis
  - ③ creation of the symbol table
  - ④ semantic analysis
  - ⑤ Generation of intermediate code.
  - ⑥ certain amount of code optimization
  - ⑦ error handling that goes along with each of these phases.



→ The Backend includes those portions of the compiler that depend on the target machine and do not depend on the source language.

→ The Backend includes:

① code Optimization

② code Generation along with the error handler and symbol table operations.

→ To produce the compiler for different machine, the front end will be same and the back-end should be redesign.

### ■ Passes:

Several Passes are grouped together into one Pass.

→ Lexical analysis, syntax analysis, semantic analysis and intermediate code generation are grouped into one pass. Then token stream generated after lexical analysis may be translated directly into intermediate code.

→ ~~Some~~ Syntax analyzer attempt to discover the grammatical structure on the tokens which is obtained by calling lexical analyzer.

→ As the grammatical structure is discovered, the Parser calls the intermediate code generator to perform semantic analysis and generate a portion of the code.

- Grouping the phases reduces number of passes and creates some problems. The entire program is forced to keep in memory b/c one pass phase may need information in different order than the previous phases produced it.
- It is difficult to perform code generation until the intermediate representation has been completely generated.
- Intermediate code generation and code generation phase can be merged into one pass using a technique called backpatching.
- Backpatch method is easy to implement if the instruction is kept in memory until all the target addresses are determined.

---

## COMPILER WRITING TOOLS

Some general tools have been created for the automatic design of specific compiler components. These tools use specialized languages for specifying and implementing the component and may use sophisticated algorithms.

The most successful tools are those that hide the details of the generation algorithms and produce components that can be easily integrated into the remainder of the compiler.



## ■ Parser Generator:

This tool produces syntax analyzer, normally from input that is based on a context-free grammar. In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler. But a large fraction of the intellectual effort of writing a compiler. Due to the creation of compiler construction tools, this phase is now considered as one of the easiest to implement.

## ■ Scanner Generator:

Scanner Generator automatically generate lexical analyzer normally from a specification based on regular expression.

## ■ Syntax Directed Translation Engines:

This tool produce collections of routines that walk through the parse tree. The basic idea is that one or more translation are associated with each node of the parse tree and each translation is defined in terms of translation at its neighbour nodes of the tree.

## ■ Automatic Code Generator:

This tool takes a collection of rules that defines the translation of each operation of the intermediate language into the machine language for the target machine.

The basic technique used is 'Template Matching'.

The intermediate code statements are replaced by templates that represent sequences of machine instructions, in such a way that the assumptions about storage of variables match from template to template.

### ■ Data Flow Engines:

code Optimization involves data-flow engines to gather information about how values are transmitted from one part of the program to other part. Different task of this nature can be performed with the user supplying details of the intermediate code statements and the gathered information.

---

## BOOTSTRAPPING

A compiler is a complex program to write it in the high level language. Usually, compilers are written in C language in the UNIX programming environment.

Bootstrapping is a technique that is widely used in compiler development.

It has 4 main uses:

- ① It enables new programming languages and compilers to be developed starting from existing one.



② It enables new features to be added to a programming language and its compiler.

③ It also allow new optimizations to be added to compilers.

④ It allows languages and compilers to be transferred b/w processors with different instruction set.

→ The main use of bootstrapping is to create compilers and to move them from one machine to another by modifying the back end.

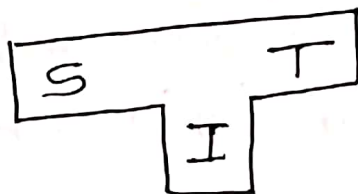
→ For bootstrapping Purpose the compiler is characterized by 3 languages:

① The source language 'S'

② The target language 'T'

③ The implementation language I.

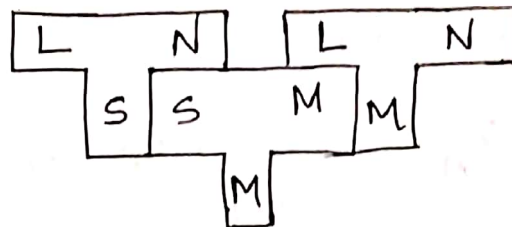
The Three languages can be represented by a T-diagram.



A compiler run on one machine and produce the target code for another machine is called cross compiler.

Suppose a cross compiler for a new language  $L$  in implementation language  $S$  to generate code for machine  $N$ . Thus we create  $L_S N$ .

If an existing compiler for  $S$  runs on machine  $M$  and generate code for  $M$  as  $S_M M$  and if  $L_S N$  runs through  $S_M M$  we get a new compiler  $L_M N$  that is a compiler from  $L$  to  $N$  that runs on  $M$ .



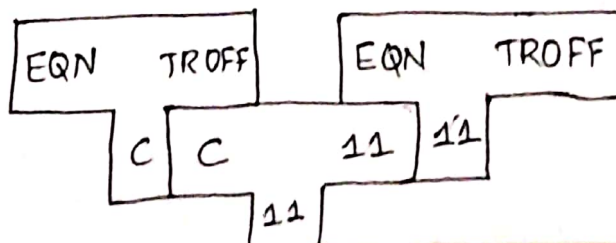
Also written as:

$$L_S N + S_M M = L_M N$$

### Example

The 1<sup>st</sup> version of EQN compiler had  $C$  as the implementation language and generated commands for the text formatter TROFF.

A cross-compiler for EQN, running on a PDP-11, was obtained by running  $EQN_C$  TROFF through the  $C$ -compiler  $C_{11}$  on the PDP-11.



One form of bootstrapping builds the computer for larger and larger subsets of language. The 1<sup>st</sup> step to implement a new language  $L$  on machine  $M$  is to write a small compiler that translates a subset  $S$  of  $L$  into the target code for  $M$ , i.e., a compiler  $S_M^M$ . The subset  $S$  can be used to write a compiler  $L_S^M$  for  $L$ . When  $L_S^M$  runs through  $S_M^M$  we get the implementation of  $L$  as  $L_M^M$ .

## LEXICAL ANALYSIS

### ■ The Role Of Lexical Analyzer

The lexical analyzer is the 1<sup>st</sup> phase of a compiler.

Main Task → Read the input and produce the output as the sequence of tokens that the parser uses for syntax analysis.

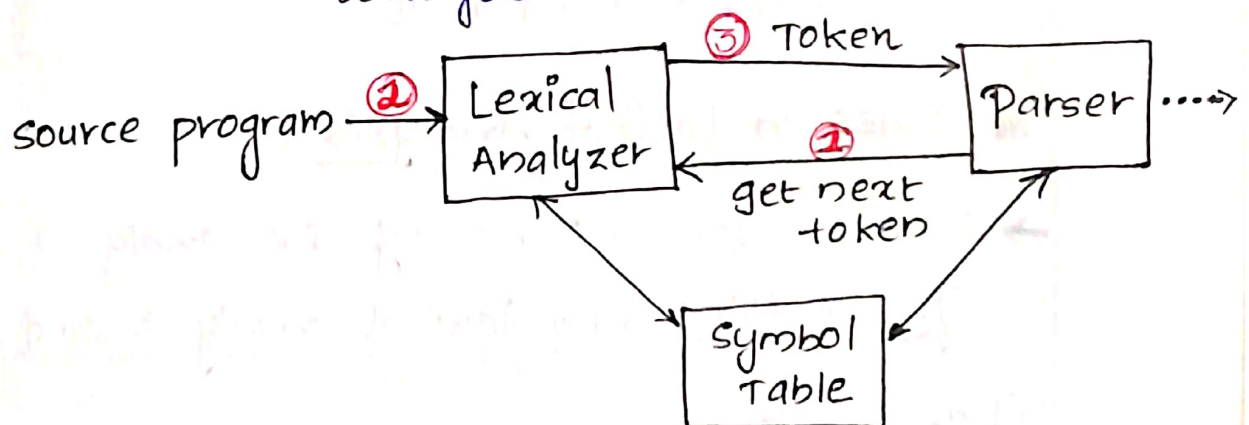


Figure: Interaction of lexical analyzer with Parser.

When the lexical analyzer receives a "get next token" command from the parser, it reads the input character until it identifies the next token.



## ■ Functions of Lexical Analyzer

- (i) Reads the input program character by character.
- (ii) Produces a stream of tokens which is used by the parser.
- (iii) Lexical analyzer removes comments and white space in the form of blanks, tab and newline character.
- (iv) Correlating error messages with source Program.
- (v) Lexical analyzer may keep track of the number of newline characters, so that line number can be associated with the error message.
- (vi) In some compiler, lexical analyzer make a copy of the source program with the error messages marked in it.

## ■ Issues in Lexical Analysis

→ Reason for separating the analysis phase of compiling into lexical analysis and parsing

- (i) To simplify one or other of these phases:

Example: The parse tree generated for a source program with comments & whitespace are more complex that have already be removed by lexical analyser. analyzer.



(ii) The efficiency of compiler can be improved

- A separate lexical analysis allows us to construct a specialized and potentially more efficient processor for the task.
- Specialized buffering techniques for reading input character and processing tokens can speed up the performance of the compiler.

(iii) Compiler Portability:

- Tokens are independent of language, i.e., lexical analysis is not portable but all other phases are portable.

### ■ Tokens, Patterns, Lexemes

- Tokens are the individual atomic unit in the source program.
- Tokens are the terminal symbols in the grammar for the source language.
- Tokens include:
  - \* Keywords      \* Identifiers
  - \* Operators      \* Constants
  - \* Literal strings
  - \* Punctuation Symbols
    - Parentheses
    - Commas
    - Semicolons.
- Patterns are the production rules for generating tokens. For generating the token as an identifier the pattern used is  $\text{letter}(\text{letter}|\text{digit})^*$

- Lexemes is a sequence of characters in the Source Program that is matched by the Pattern for Program token.
- Specific instance of a token.

### Examples

① `Count = Count + temp;` → `id1 = id2 + id3;` 6 Tokens

Lexemes  
Tokens: Identifier → count, temp  
 Operator → =, +  
 Punctuation → ;

② `31 + 28 - 59` → `id1 + id2 - id3` 5 Tokens

Rule      Lexemes  
Tokens: Number →  $[0-9]^+$       31, 28, 59  
 Operator → +, -

③ `Printf ("Total = %d\n", score);` 7 Tokens

Tokens: Identifier → Printf, score  
 Literal → "Total = %d\n"  
 Operator → ;

④ `const pi = 3.1416;`

The substring `pi` is a lexeme for the token 'Identifier'.

Token	Sample Lexemes	Description of Pattern
const	Const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or > or >=
id	Pi, count, D2	letter followed by letter or digit.
num	3.1416, 0, 6.02E23	any numeric constant
literal	" cse Department"	Any character between " and " except "

Figure: Examples of Tokens, lexemes and pattern.

### ■ Attributes Of Tokens

When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler.

→ Whenever a lexeme is encountered in a source program, it is necessary to keep a track of other occurrences of the same lexeme i.e., if this lexeme has been seen before or not.

↳ To implement tracking, Symbol Table is used.

↳ In symbol table, lexemes are stored.

↳ The pointer to this Symbol Table entry becomes an attributes of that particular token.



## Example

$e = m * c + 2$   
↓ ↓ ↓ ↓  
Lexemes

→ Pointer for these tokens will be ~~present~~ pointing to the entry in the symbol table.

→ Tokens and the associated attribute-values for the statement is given below.

<id, pointer to symbol table-entry for E>

<assign-op>

<id, pointer to symbol table-entry for M>

<multi-op>

<id, pointer to symbol table-entry for C>

<add-op>

<num, integer value 2>



## ■ Lexical Errors

Lexical analyzer has a very localized view of a source program. So few errors are not detected at the lexical level alone.

Example: fi(a == f(x))...

↳ In the above statement the keyword 'if' is misspelled as 'fi' or it is an undeclared function identifier. The lexical analyzer returns fi as the valid identifier.

↳ Lexical analyzer fails if none of the pattern matches for the token. The simplest recovery strategy is 'Panic Mode' Recovery. In this recovery technique the lexical analyzer delete successive character from the remaining input until the lexical analyzer finds a well formed token.

↳ Some possible error-recovery actions are:

- ① Deleting an extraneous character.
- ② Inserting a missing character.
- ③ Replacing an incorrect character by a correct character.
- ④ Transposing two adjacent characters.

# INPUT BUFFERING

There are mainly 2 methods used for the buffering of input. They are:

① Two-Buffer input scheme (Buffer Pair)

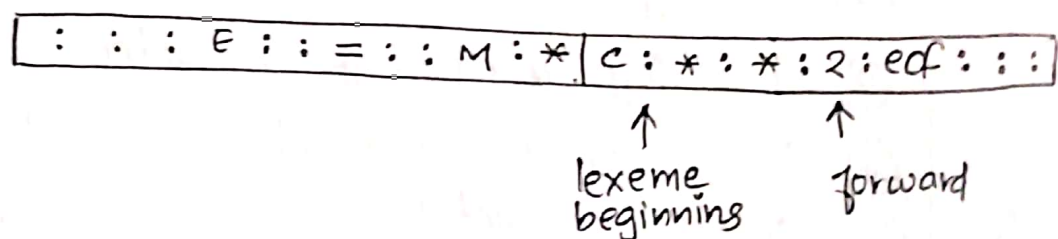
② Sentinels.

→ Two-Buffer input scheme is used to identify tokens and the Sentinel Method is used to speed up the lexical analyzer.

## ■ Buffer Pairs

→ This method of buffering is developed to reduce the amount of overhead required to process an input character.

→ In this scheme, the buffer is divided into two N-character halves as shown in the figure below.



Where N is the no. of characters on one disk block. eg: 1024 or 4096.

↳ N - input characters are read from each half of the buffer. If fewer than N-characters remain in the input then a special character eof



is read into the buffer after the input character.

→ eof marks the end of the source file and is different from any other input character.

→ Two pointers to the input buffer are maintained.

The string of characters b/w the 2 pointers is the current lexeme.

→ Initially, both pointers point to the 1<sup>st</sup> character of the next lexeme to be found.

→ The forward pointer scans the input character ahead until it find a pattern match.

→ Once the lexeme is determined, the forward pointer is set to the character at its right end. After the lexeme is processed, both pointers are set to the character immediately

Past the lexeme. With this scheme comments and whitespace can be treated as pattern with no token.

→ If the forward pointer is at the end of the half, then the right half is filled with new  $N$ -characters.

→ If the forward pointer is at the end of the right half of the buffer then left half is filled with  $N$ -new characters and the forward pointer wraps around to the beginning of the buffer.

→ The code to advance forward pointer is given below.

```
if forward at end of first half then begin
    reload second half;
    forward := forward + 1;
end
else if forward at end of second half then begin
    reload first half;
    move forward to beginning of first half
end
else forward := forward + 1;
```

→ The main limitation of this scheme is that the above code requires two tests for each move of the forward pointer.

## ■ Sentinels

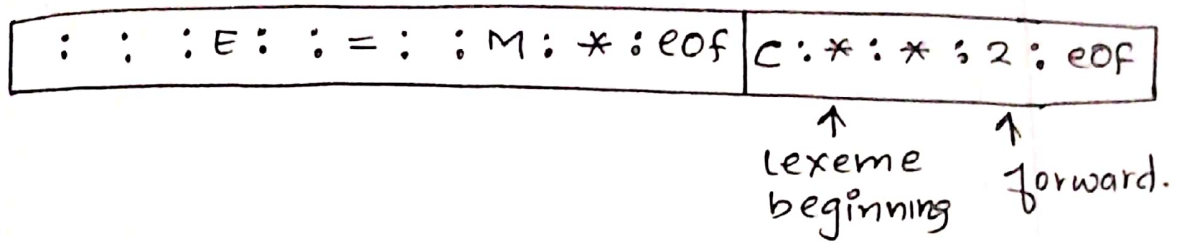
→ This method of buffering reduces the two tests of the above scheme to one if we extend each buffer half to hold a sentinel character at the end.

→ The sentinel is a special character that cannot be the part of the source program.

→ A general sentinel used is eof.



→ The buffer arrangement with the sentinel added is given below:



→ The code to advance the forward pointer with the sentinels added is given below.

```
forward := forward + 1;  
if forward := eof then begin  
  if forward at end of first half then begin  
    reload second half;  
    forward := forward + 1  
  end  
  else if forward at end of second half then begin  
    reload first half;  
    move forward to beginning of first half  
  end  
  else  
    terminate lexical analysis  
end
```

# SPECIFICATION OF TOKENS USING REGULAR EXPRESSIONS

Regular Expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expression will serve as names for set of strings.

## ■ Strings and Languages

- Alphabet or character class denotes any finite set of symbols. Examples of symbols are letters and characters. The set  $\{0,1\}$  is the binary alphabet. ASCII and EBCDIC are two examples of computer alphabets.
- A string over some alphabet is a finite sequence of symbols drawn from that alphabet. eg:- sentence and words are often used as synonym for the term 'string'.

The length of a string 's', usually written  $|s|$ , is the number of occurrences of symbols in s.

eg:- banana → String of length 6.

The empty string denoted ' $\epsilon$ ' → special string of length 0.

- Language denotes any set of strings over some fixed alphabet. empty set →  $\phi$  or  $\{\epsilon\}$

## ■ Operations On Languages

→ The important operations used in lexical analysis are:

- ① Union
- ② Concatenation
- ③ Closure.

Operation	Definition
→ LUM (union)	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
→ LM (concatenation)	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
→ $L^*$ (Kleene closure of $L$ )	$L^* = \bigcup_{i=0}^{\infty} L^i$ $L^*$ denotes " <u>zero or more concatenation of</u> " $L$
→ $L^+$ (Positive closure of $L$ )	$L^+ = \bigcup_{i=1}^{\infty} L^i$ $L^+$ denotes " <u>one or more concatenation of</u> " $L$

Figure: Definition of Operations on Languages.

### Example

Let  $L$  be the set  $\{A, B, \dots, Z, a, b, \dots, z\}$  and the set  $\{0, 1, \dots, 9\}$ .

→ New languages created from  $L$  and  $D$ :

- ①  $L \cup D$  is the set of letters and digits.



- ② LD is the set of strings consisting of a letter followed by a digit.
- ③ L<sup>4</sup> is the set of all 4-letter strings.
- ④ L<sup>\*</sup> is the set of all strings of letters, including  $\epsilon$ , the empty string.
- ⑤ L(LUD)<sup>\*</sup> is the set of all strings of letters and digits beginning with a letter.
- ⑥ D<sup>+</sup> is the set of all strings of one or more digits.

### ■ Regular Expressions

A regular expression is a sequence of symbols and characters expressing a string or pattern.

→ In Pascal language, the identifiers are defined by the following regular expression.

$\text{letter}(\text{letter}|\text{digit})^*$  or  $L(L|D)^*$

- A regular expression is built up out of simpler regular expressions using a set of defining rules.
- Each regular expression 'r' denotes a language L(r).

→ The rules that define the RE over alphabet  $\Sigma$  is given below:

- ①  $\epsilon$  is a regular expression that denotes  $\{\epsilon\}$ , i.e., empty string.



② If 'a' is a symbol in  $\Sigma$ , then 'a' is a regular expression that denotes  $\{a\}$ , i.e., the set containing the string 'a'.

③ Suppose 'r' and 's' are RE denoting the languages  $L(r)$  and  $L(s)$ . Then,

(i)  $(r)|(s)$  is a regular expression denoting  $L(r) \cup L(s)$ .

(ii)  $(r)(s)$  is a regular expression denoting  $L(r)L(s)$ .

(iii)  $(r)^*$  is a R.E denoting  $(L(r))^*$ .

(iv)  $(r)$  is a RE denoting  $L(r)$ .

→ A language denoted by a RE → Regular Set.

→ Unnecessary Parentheses can be avoided in RE if we adopt the conventions that:

① the unary operator  $*$  has the highest Precedence and is left associative.

② Concatenation has the 2<sup>nd</sup> highest precedence and is left associative

③ | has the lowest precedence and is left associative.

→  $(a) | ((b) * (c)) \xrightarrow{\text{equal.}} a | b * c$

Example

① The regular expression  $a|b$  denotes the set  $\{a, b\}$ .

- ② The RE  $(a|b)(a|b)$  denotes  $\{aa, ab, ba, bb\}$  or  $\{aa|ab|ba|bb\}$ .
- ③ The RE  $a^*$  denotes the set of all strings of zero or more a's  $\rightarrow \{\epsilon, a, aa, aaa, \dots\}$
- ④ The RE  $(a|b)^*$  denotes the set of all strings containing zero or more instances of an 'a' or 'b'. [Equivalent  $\rightarrow (a^*b^*)^*$ ]
- ⑤ The RE  $a|a^*b$  denotes the set containing the string 'a' and all strings consisting of zero or more ~~as~~ a's followed by a b.

## ■ Regular Definition

- $\rightarrow$  For notational convenience, we give names to regular expressions and to define RE using these names as if they were symbols.
- $\rightarrow$  If  $\Sigma$  is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

$\vdots$

$$d_n \rightarrow r_n$$

where each  $d_i$  is a distinct name, and each  $r_i$  is a RE over symbols in  $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$

i.e., the basic symbols and the previously defined names.

### Example

The set of Pascal identifiers is the set of strings of letters and digits beginning with a letter.

Regular Definition for the set is:

letter  $\rightarrow A|B|\dots|Z|a|b|\dots|z$

digit  $\rightarrow 0|1|\dots|9$

id  $\rightarrow \text{letter}(\text{letter}|\text{digit})^*$

### ■ Notational Shorthands

certain constructs occur so frequently in RE that it is convenient to introduce notational short-hands for them.

#### ① One or more instances:

→ The unary postfix operator  $+$  means "one or more instances of."

→ If 'r' is a RE that denotes the language  $L(r)$ ,  
 $r^+$  is a RE that denotes  $(L(r))^+$ .

→  $\therefore$  a RE  $a^+$  denotes the set of all strings of one or more a's.

→  $+$  has same precedence & associativity as the Operator  $*$ .

→  $r^* = r^+ | \epsilon$  and  $r^+ = rr^*$ .



## ② zero or more instances:

→ The unary postfix operator ? means "zero or one instance of"

→ The notation  $r?$  is a shorthand for  $r/\epsilon$ .

→ If  $r$  is a regular expression, then  $(r)?$  is a regular expression that denotes the language  $L(r) \cup \{\epsilon\}$ .

**Example** Regular Definition for num.

digit  $\rightarrow 0|1|\dots|9$

digits  $\rightarrow \text{digit}^+$

Optional-fraction  $\rightarrow (\cdot \text{digits})?$

Optional-exponent  $\rightarrow (\epsilon(+|-)? \text{digits})?$

num  $\rightarrow \text{digits optional-fraction optional-exponent}$

Strings: 5280, 39.37, 6.336E4 or 6.894E-4

## ③ character classes:

→ The notation  $[abc]$  where  $a, b$ , and  $c$  are alphabet symbols denotes the regular expression  $a|b|c$ .

→ An abbreviated character class such as  $[a-z]$  denotes the RE  $a|b|c \dots |z$ .

→ Using character classes, we can describe identifiers as being strings generated by the RE



$[A-Za-z][A-Za-z0-9]^*$

## ■ Nonregular Sets

→ Some languages cannot be described by regular expression. Regular expressions ~~cannot~~ cannot be used to describe balanced or nested constructs.

For example, the set of all strings of balanced parentheses cannot be described by a RE.

This set can be described by a context-free Grammar.

→ Repeating strings cannot be described by ~~RE~~ RE.

The set  $\{wcw|w \text{ is a string of a's and b's}\}$  cannot be denoted by any regular expression, nor can it be described by a context-free Grammar.

→ RE can be used to denote only a fixed no. of repetitions or an unspecified no. of repetition of a given construct.

## RECOGNITION OF TOKENS

Consider the following Grammar fragment:

stmt	→	if expr then stmt	
		if expr then stmt else stmt	
		ε	
expr	→	term relop term	//relop → Relational Operator
		term	
term	→	id	
		num	

where the terminals if, then, else, relop, id and num generate sets of strings given by the following regular definitions:

if  $\rightarrow$  if

then  $\rightarrow$  then

else  $\rightarrow$  else

relop  $\rightarrow$  < | <= | = | <> | > | >=

id  $\rightarrow$  letter (letter | digit)\*

num  $\rightarrow$  digit<sup>+</sup> (.digit<sup>+</sup>)? (E(+|-)? digit<sup>+</sup>)?

where letter and digit are as defined previously.

→ For this language, the lexical analyzer will recognize the keywords if, then, else as well as the lexemes denoted by relop, id and num.

→ Lexemes are separated by white space, consisting of nonnull sequences of blanks, tabs and newlines.

→ Lexical analyzer will strip out white space.

→ The Regular Definition of whitespace:

delim  $\rightarrow$  blank | tab | newline

ws.  $\rightarrow$  delim<sup>+</sup>

// delimiter  $\rightarrow$  1 or more chara that separate text string

If a match for ws is found, the lexical analyzer does not return a token to the parser. Rather, it proceeds to find a token following the white space & returns that to the parser.



Regular Expression	Token	Attribute - Value
ws	—	—
if	if	
then	then	
else	else	
id	id	Pointer to table entry
num	num	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Figure: Regular Expression patterns for Tokens.

## ■ Transition Diagrams

- As an intermediate steps in the construction of a lexical analyzer, we first produce a stylized flowchart, called a transition diagram.
- Transition diagrams depicts the actions that take place when a lexical analyzer is called by the Parser to get the next token.
- We use transition diagrams to keep track of information about characters that are seen as the forward pointer scans the input. We do so by



moving from position to position in the diagram as characters are read.

- Positions in a transition diagram are drawn as circles and are called states.
- The states are connected by arrows, called edges.
- Edges leaving state 's' have labels indicating the input characters that can next appear after the transition diagram has reached state 's'.
- The label Other refers to any character that is not indicated by any of the other edges leaving 's'.
- One state is labeled the start state, → initial state of the transition diagram where control resides when we begin to recognize a token.

### Example

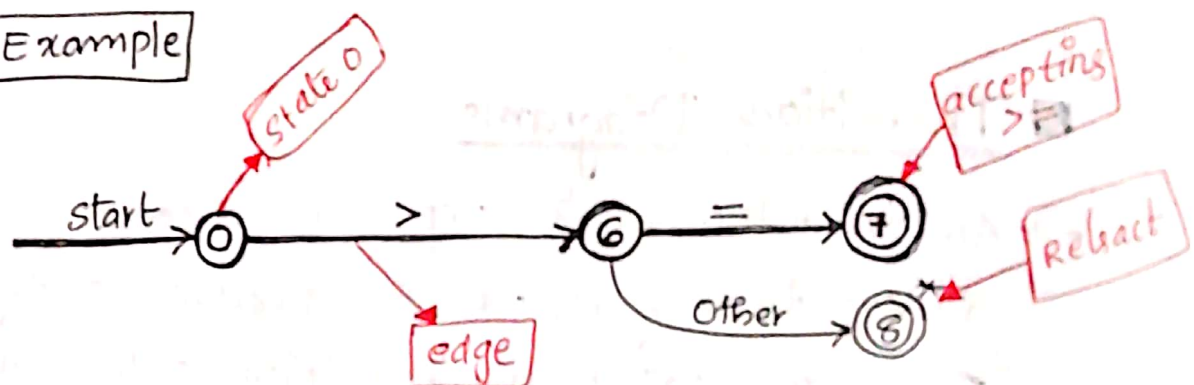


Figure: Transition diagram for `>=`.

### Working

- Its start state is 0.
- In state 0, we read the next input character.
- The edge labelled `>` from state 0 is to be

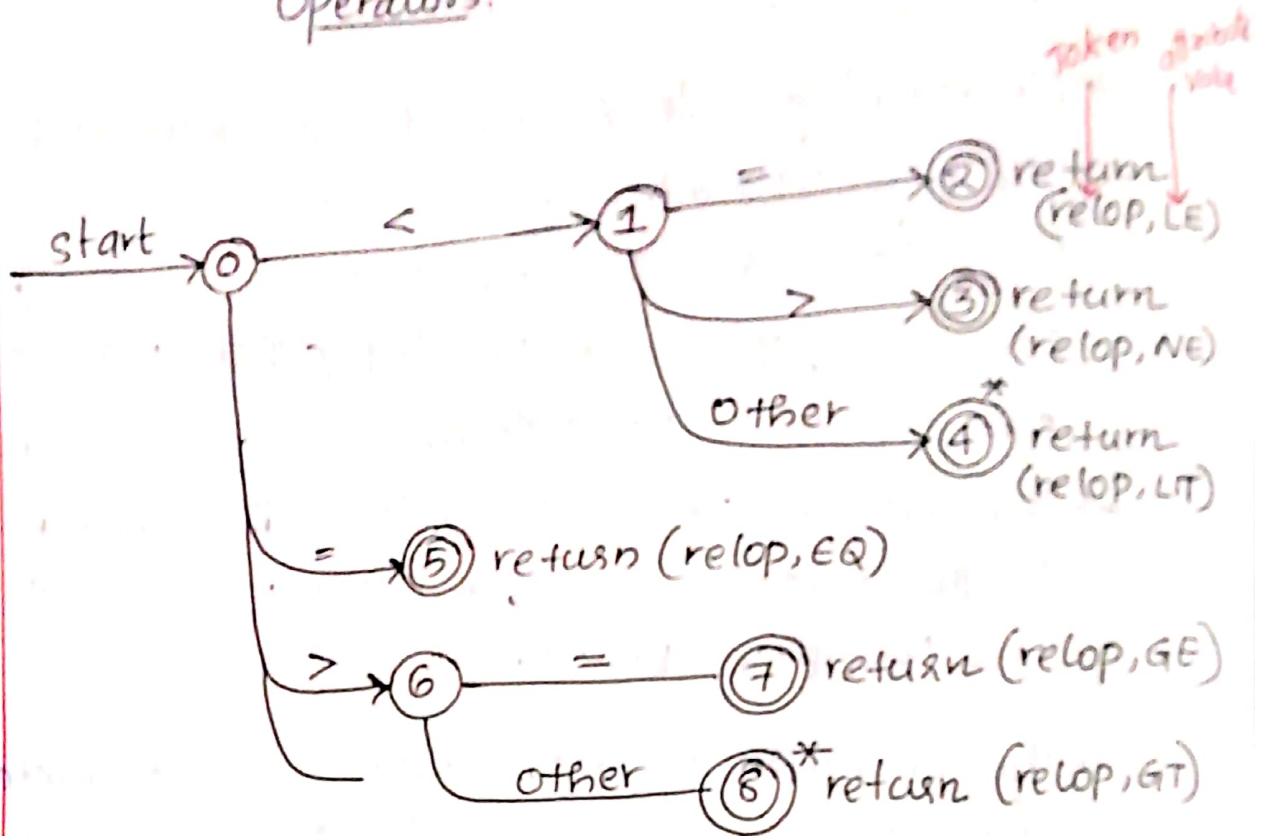
followed to state 6 if this input character is  $>$ . Otherwise we have failed to recognize either  $>$  or  $>=$ .

- On reaching state 6 we read the next input character.
- The edge labelled  $=$  from state 6 is to be followed to state 7 if this input character is an  $=$ . Otherwise, the edge labelled other indicates that we are to go to state 8.
- The double circle on state 7 indicates that it is an accepting state, a state in which the token  $>=$  has been found.
- The character  $>$  and another extra character are read as we follow the sequence of edges from the start state to the accepting state 8. Since the extra character is not a part of the relational operation  $>$ , we must retract the ~~for~~ forward pointer one character. We use a \* to indicate state on which this input retraction must take place.

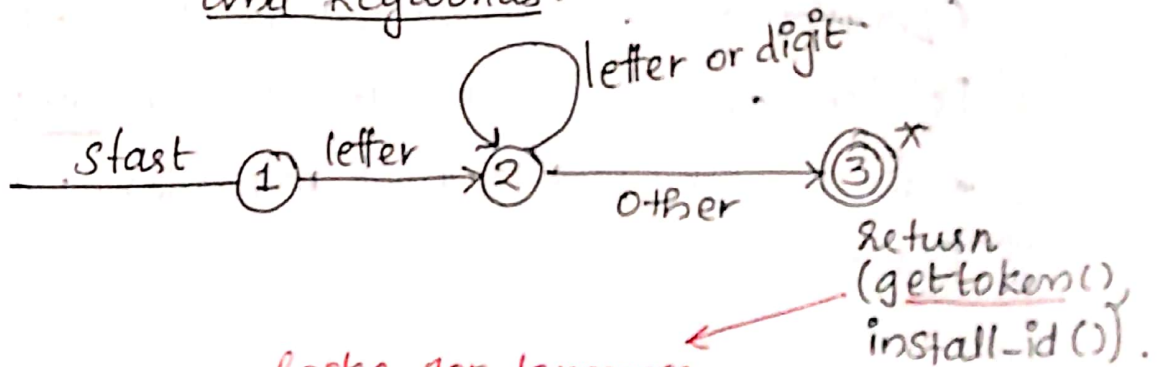


**Figure:** Transition diagram for count > 40

**Example** Transition diagram for relational operators:



**Example** Transition diagram for identifiers and keywords.



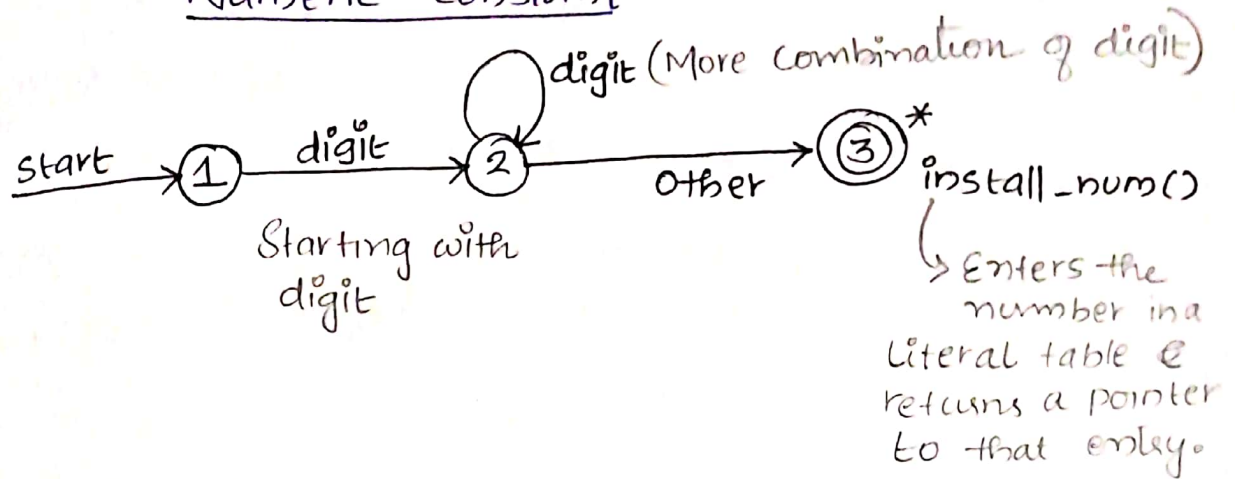
looks for lexemes

in the symbol table

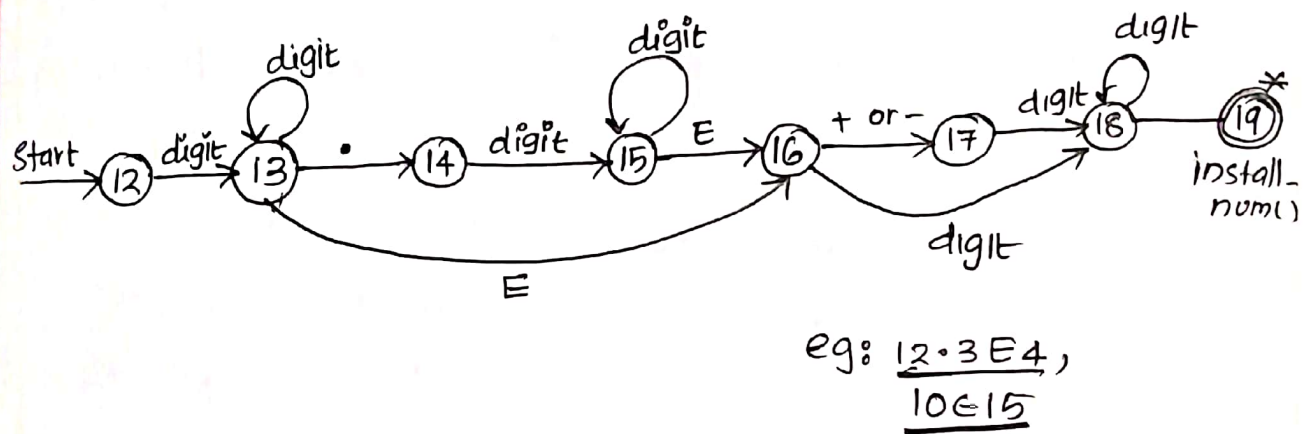
Obtain the token & attribute value



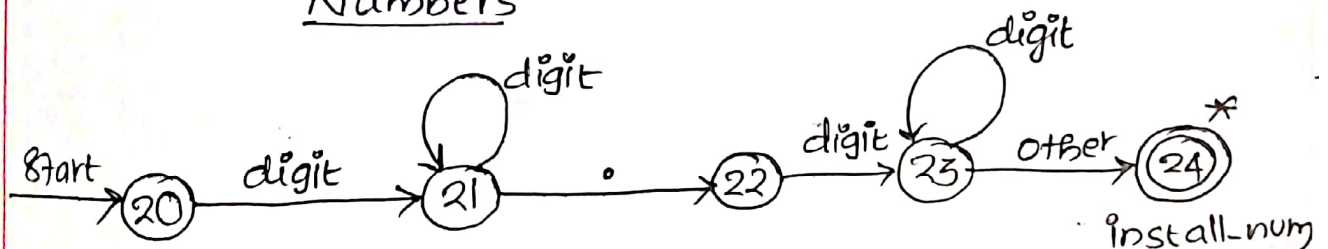
### Example Transition Diagram for Unsigned Integer Numeric Constant



### Example Transition Diagram for Unsigned Numbers

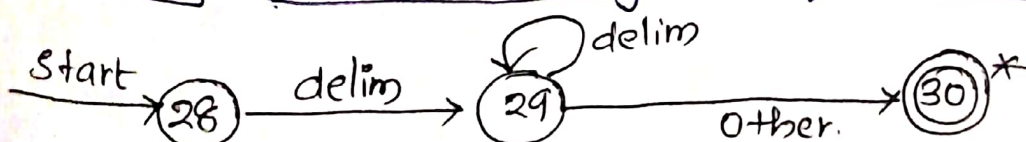


### Example Transition Diagram for fractional Numbers



eg: 12.3, 10.15

### Example Transition Diagram for white space





## MODULE II

### Syntax Analysis:

Review of Context-Free Grammars — Derivation Trees and Parse Trees, Ambiguity.

### Top-Down Parsing:

Recursive Descent Parsing, Predictive Parsing, LL(1) Grammars.

## SYNTAX ANALYSIS

- Every Programming language has rules that prescribe the syntactic structure of well formed programs.
- The Syntax of Programming language constructs can be described by context-free grammars or BNF.

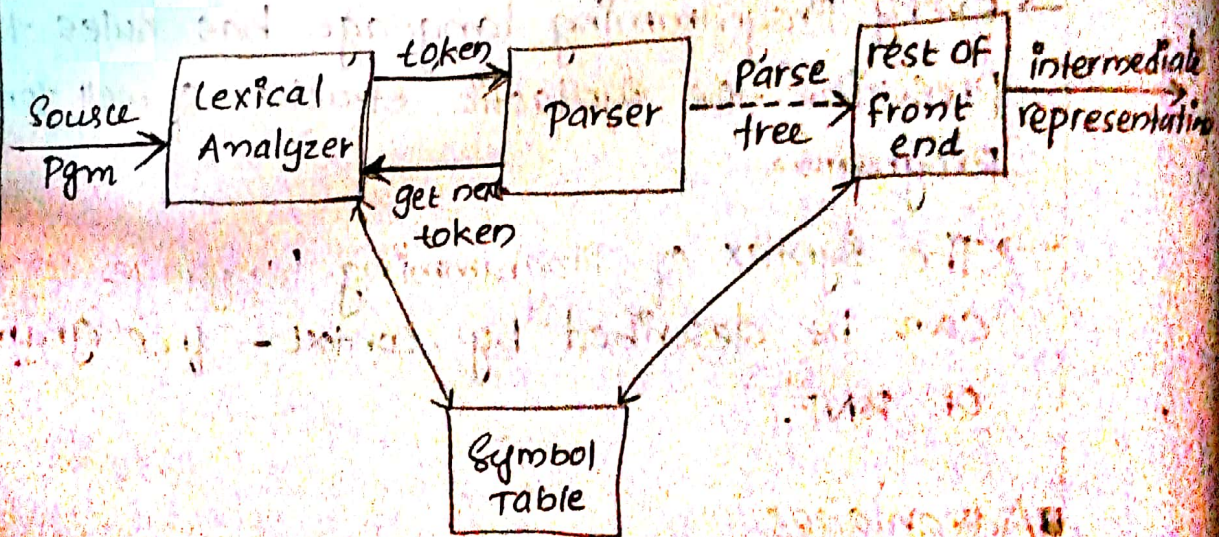
### ■ Advantages

- ① A Grammar gives a precise, easy to understand specification of a programming language.
- ② We can automatically construct an efficient Parser from certain classes of grammars that determines if a source program is syntactically well formed.
- ③ The parser construction process can reveal the syntactic ambiguities and other difficult-to-parse constructs.



## ■ The Role of the Parser

- The Parser obtains a strings of tokens from the lexical analyzer, and verifies that the string can be generated by the grammar for the source language.
- The Parser report any syntax errors in an intelligible fashion.
- It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.



**Figure:** Position of parser in compiler model.

The methods commonly used in compilers are classified as being either top-down or bottom-up.

- Top-down parser build parse tree from the top (root) to the bottom (leaves), while bottom-up parsers start from the leaves and work up to the root.



- In both cases, the input to the parser is scanned from left to right, one symbol at a time.
- The output of the parser is some representation of the parse tree for the stream of tokens produced by the lexical analyzer.
- There are no. of tasks that might be conducted during parsing, such as collecting information about various tokens into the symbol table, performing type checking and other kinds of semantic analysis and generating intermediate code.

### ■ Syntax Error Handling

Programs can contain errors at many different levels.

Errors are :

- ① Lexical, such as misspelling an identifier, keyword, or operator.
- ② Syntactic, such as an arithmetic expression with unbalanced parentheses.
- ③ Semantic, such as an operator applied to an incompatible operand.
- ④ Logical, such as an infinitely recursive call.

→ Most of the error detection and recovery in a compiler is centered around the syntax analysis phase.

→ One reason for this is that many errors are syntactic in nature or are exposed when the stream of tokens coming from the lexical analyzer disobeys the grammatical rules defining the programming language.

The main Goals of error-handler in a parser:

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct program.

### ■ Error-Recovery Strategies

There are many different general strategies that a Parser can employ to recover from a syntactic error.

- ① Panic Mode
- ② Phrase Level
- ③ Error Productions
- ④ Global Correction



## Panic Mode Recovery

- This is the simplest method to implement and can be used by most parsing methods.
- On discovering an error, the Parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found.
- The synchronizing tokens are usually delimiters, such as semicolon or end.
- While Panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has the advantage of simplicity and guaranteed not to go into an infinite loop.

## Phrase - Level Recovery

- On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the Parser to continue.
- A typical local correction would be to replace a comma by a semicolon; delete an extraneous semicolon, or insert a missing semicolon.

## Error Productions

- If we have a good idea of the common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs.



- We then use the grammar augmented by these error productions to construct a parser.
- If an error production is used by the parser, we can generate appropriate error diagnostics to indicate the erroneous construct that has been recognized in the input.

### Global Correction

- compiler make a few changes as possible in processing an incorrect input string.
- There are algorithms for choosing a minimal sequence of changes to obtain a globally least cost correction.
- Give an incorrect input string  $x$  and grammar  $G$ , these algorithms will find a parse tree for a related string  $y$ , such that the number of insertions, deletions, and changes of tokens required to transform  $x$  into  $y$  is as small as possible.
- Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.

# CONTEXT-FREE GRAMMARS

A context free grammar is a set of recursive rewriting rules (or productions) used to generate patterns of strings.

CFG can define with 4 tuples:

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

$S \rightarrow$  Start symbol, which is a special nonterminal symbol that appear in the initial string generated by the grammar.

$P \rightarrow$  Set of Productions, which are rules for replacing nonterminal symbols in a string with other nonterminal or terminal symbol.

$V \rightarrow$  Set of variables, or nonterminals, which are place holders for patterns of terminal symbol that can be generated by the nonterminal symbols.

$T \rightarrow$  Set of terminal symbols, which are the characters of the alphabet that appear in the strings generated by the grammar.

**Example** The grammar with the following productions defines simple arithmetic expressions.



$\text{expr} \longrightarrow \text{expr op expr}$

$\text{expr} \longrightarrow (\text{expr})$

$\text{expr} \longrightarrow -\text{expr}$

$\text{expr} \longrightarrow \text{id}$

$\text{op} \longrightarrow +$

$\text{op} \longrightarrow -$

$\text{op} \longrightarrow *$

$\text{op} \longrightarrow /$

$\text{op} \longrightarrow \uparrow$

→ The terminal symbols →  $\text{id} + - * / \uparrow ( )$

→ The nonterminal symbols →  $\text{expr}$  and  $\text{op}$ .

→  $\text{expr}$  is the start symbol.

### ■ Notational Conventions

#### Terminal Symbols:

- ① Lower case letters →  $a, b, c, \dots$
- ② Operator symbols →  $+, -, *, /, \dots$
- ③ Punctuation symbols → Parentheses, comma, ...
- ④ Digits →  $0, 1, \dots, 9, \dots$
- ⑤ Boldface strings →  $\text{id}$ , or  $\text{if}$ .

#### Nonterminal Symbols:

- ① Upper case letters →  $A, B, C, \dots$
- ② The letter  $S$  → start symbol
- ③ lower case italic names →  $\text{expr}$  or  $\text{stmt}$ .



Using shorthands, we could write the grammar as

$$E \rightarrow EAE \mid (E) \mid -E \mid \text{id}$$

$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

→ E and A are non-terminals, with the start symbol E. The remaining symbols are terminals.

### Examples Grammars:

①  $S \rightarrow aB \mid bA$       → Nonterminal symbols.  
     $A \rightarrow a \mid aS \mid bAA$       → Terminal symbols.  
     $B \rightarrow b \mid bS \mid aBB$

②  $S \rightarrow AB \mid \epsilon$   
     $A \rightarrow aB$   
     $B \rightarrow Sb$

③  $E \rightarrow E+T \mid T$   
     $T \rightarrow T * F \mid F$   
     $F \rightarrow (E) \mid a$

④  $S \rightarrow S+S \mid S * S \mid a$

# DERIVATIONS

The central idea here is that a production is treated as a rewriting rule in which the nonterminal on the left is replaced by the string on the right side of the production.

→ For example, consider the following grammar for arithmetic expressions, with the non-terminal  $E$  representing an expression.

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

→ The Production  $E \rightarrow -E$  signifies that an expression preceded by a minus sign is also an expression.

→ Let  $E \rightarrow -E$  which reads " $E$  derives  $-E$ ".

→ We can take a single  $E$  and repeatedly apply productions in any order to obtain a sequence of replacements. For example,

$$E \rightarrow -E \rightarrow -(E) \rightarrow -(id)$$

Example

Derive the string  $-(id+id)$  from the grammar

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

$$\begin{aligned} \rightarrow E &\rightarrow -E \\ &\rightarrow -(E) \\ &\rightarrow -(E + E) \\ &\rightarrow -(id + E) \\ &\rightarrow -(id + id) \end{aligned}$$

The derivation is broadly classified into 2 types

① Leftmost Derivation

② Rightmost Derivation

→ The Derivation in which leftmost nonterminal is any sentential form to be replaced in each step is called leftmost derivation.

ie.,  $\alpha \xrightarrow{lm} \beta$

**Example:**  $E \xrightarrow{lm} -E$   
 $\xrightarrow{lm} -(E+E)$   
 $\xrightarrow{lm} -(id+E)$   
 $\xrightarrow{lm} -(id+id)$

→ The Derivation in which rightmost nonterminal in any sentential form is replaced in each step is called rightmost derivation.

ie.,  $\alpha \xrightarrow{rm} \beta$

**Example:**  $E \xrightarrow{rm} -E$   
 $\xrightarrow{rm} -(E)$   
 $\xrightarrow{rm} -(E+E)$   
 $\xrightarrow{rm} -(E+id)$   
 $\xrightarrow{rm} -(id+id)$



## EXAMPLES

- ① Derive the string "aabbabba" for the left most derivation (LMD) and right most derivation (RMD), using the context-free Grammar

$$\begin{aligned} S &\rightarrow aB \mid bA \\ AB &\rightarrow a \mid as \mid bAA \\ B &\rightarrow b \mid bs \mid aBB \end{aligned}$$

→ LMD

$$\begin{aligned} S & \\ a \underline{B} & \quad S \rightarrow aB \\ a a \underline{BB} & \quad B \rightarrow aBB \\ a a b \underline{B} & \quad B \rightarrow b \\ a a b b \underline{s} & \quad B \rightarrow bs \\ a a b b a \underline{B} & \quad S \rightarrow aB \\ a a b b a b \underline{s} & \quad B \rightarrow bs \\ a a b b a b b \underline{A} & \quad S \rightarrow bA \\ a a b b a b b a & \quad A \rightarrow a \\ & \rightarrow \underline{\text{String Derived}} \end{aligned}$$

RMD

$$\begin{aligned} S & \\ a \underline{B} & \quad S \rightarrow aB \\ a a \underline{BB} & \quad B \rightarrow aBB \\ a a B b \underline{s} & \quad B \rightarrow bs \\ a a B b b \underline{A} & \quad S \rightarrow bA \\ a a \underline{B} b b a & \quad A \rightarrow a \\ a a b \underline{s} b b a & \quad B \rightarrow bs \\ a a b b \underline{A} b b a & \quad S \rightarrow bA \\ a a b b a b b \underline{a} & \quad A \rightarrow a \\ & \rightarrow \underline{\text{String Derived}} \end{aligned}$$

②

$$\begin{aligned} S &\rightarrow AB \mid \epsilon \\ A &\rightarrow aB \\ B &\rightarrow Sb \\ \underline{\text{String}} &\rightarrow abb \end{aligned}$$

LMD

S  
AB       $S \rightarrow AB$   
aBB       $A \rightarrow aB$   
asbB       $B \rightarrow sb$   
abb       $S \rightarrow \epsilon$   
absb       $B \rightarrow sb$   
abb       $S \rightarrow \epsilon$

$\hookrightarrow$  String Derived

RMD

S  
AB       $S \rightarrow AB$   
Asb       $B \rightarrow sb$   
Ab       $S \rightarrow \epsilon$   
aBb       $A \rightarrow aB$   
asbb       $B \rightarrow sb$   
abb       $S \rightarrow \epsilon$

$\hookrightarrow$  String Derived

③

$E \rightarrow E+T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid a$   
String  $\rightarrow a+a*a$

LMD

$E \rightarrow E+T$   
 $\rightarrow T+T$   
 $\rightarrow F+T$   
 $\rightarrow a+T * F$   
 $\rightarrow a+F * F$   
 $\rightarrow a+a * F$   
 $\rightarrow a+a * a$

$\hookrightarrow$  String Derived

RMD

$E \rightarrow E+T$   
 $\rightarrow E+T * F$   
 $\rightarrow E+T * a$   
 $\rightarrow E+F * a$   
 $\rightarrow E+a * a$   
 $\rightarrow T+a * a$   
 $\rightarrow F+a * a$   
 $\rightarrow a+a * a$

$\hookrightarrow$  String Derived



# PARSE TREES

A parse tree is a graphical representation for a derivation that filters out the choice ~~two~~ regarding replacement order.

→ Parse tree is also called as Derivation Tree.

→ A parse tree is an ordered tree in which nodes are labelled with the left side of production and in which the children of a node represent its corresponding right sides.

→ Let  $G = (V_n, V_t, P, S)$  be CFG. An ordered tree for this CFG,  $T$  is a derivation tree if it has the following properties.

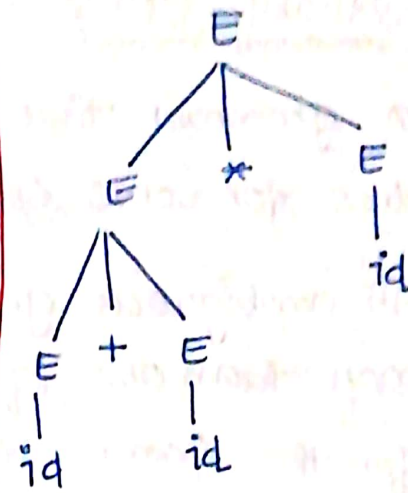
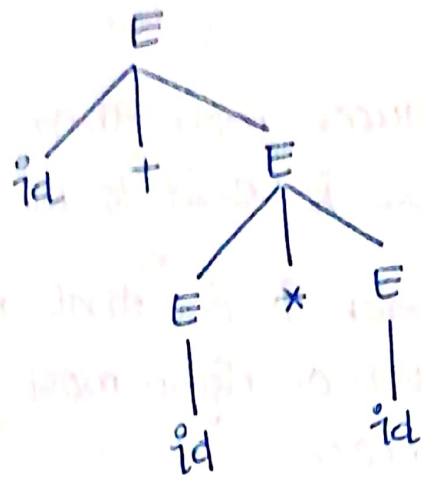
(a) The root is labelled by the starting non-terminals or terminals and read from left to right.

(b) Every leaf of the parse tree ~~is~~ is labelled by non terminals or terminals and read from left to right.

(c) Interior node of parse tree has labeled from  $V_n$ .

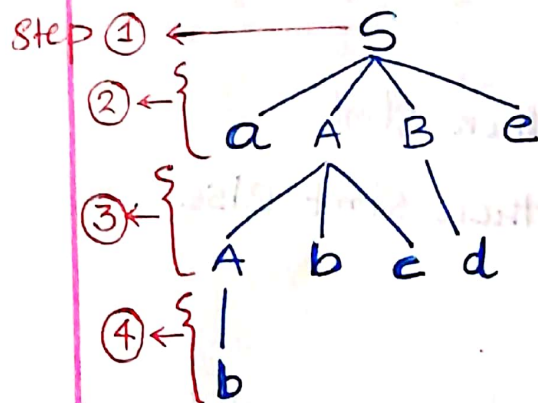
① Example construct a parse tree for the string  $id + id * id$  and grammar  $G$  is  $E \rightarrow E * E$   
 $E \rightarrow E + id$





**Figure:** Two parse trees for the string  $id + id * id$ .

- ②  $S \rightarrow aABe$   
 $A \rightarrow Abc|b$   
 $B \rightarrow d$   
 String  $\rightarrow abbcde$
- construct Parse Tree.



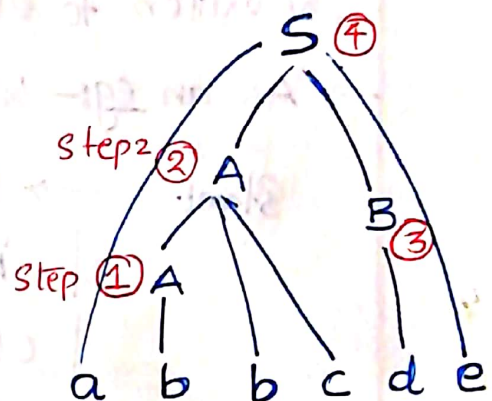
Top-down Parsing

Task

→ What to Reduce.

$S \Rightarrow aABe$   
 $\Rightarrow aAbcBe$   
 $\Rightarrow abbcBe$   
 $\Rightarrow abbcde$

Use LMD



Bottom-Up Parsing

Task

→ When to Reduce

$S \Rightarrow aABe$   
 $\Rightarrow aAbcBe$   
 $\Rightarrow aAbcde$   
 $\Rightarrow abbcde$

Use RMD

## AMBIGUITY

A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

An ambiguous grammar is one that produces more than one left most or right most derivation for the same sentences.

→ Some parsers only accept the unambiguous grammar so we can use certain ambiguous grammar together with disambiguating rules that throw away undesirable parse trees, leaving us with only one tree for each sentence.

### ■ Eliminating Ambiguity

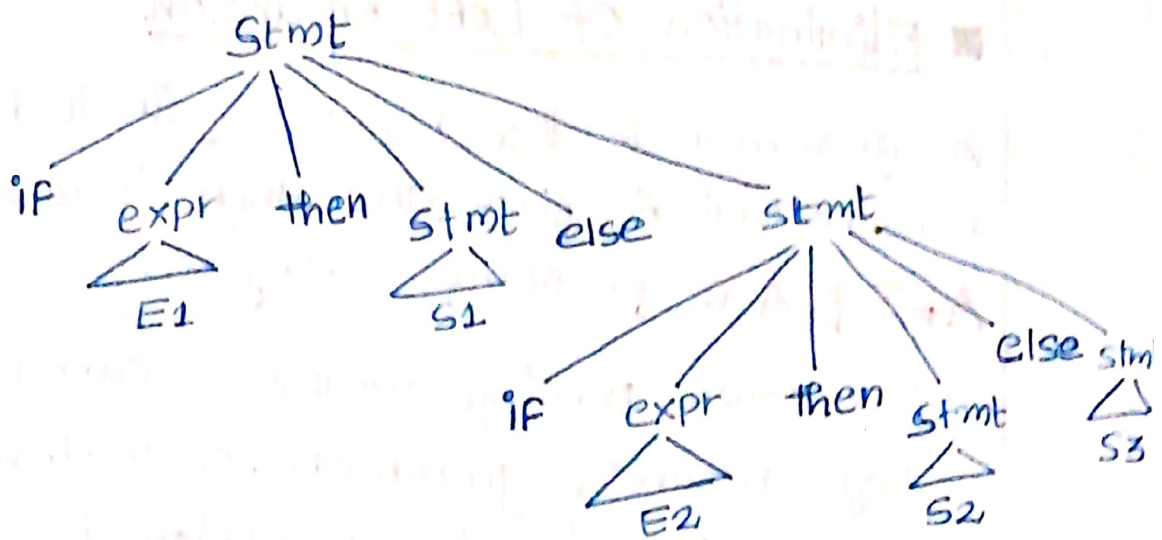
→ Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.

→ As an eg:-

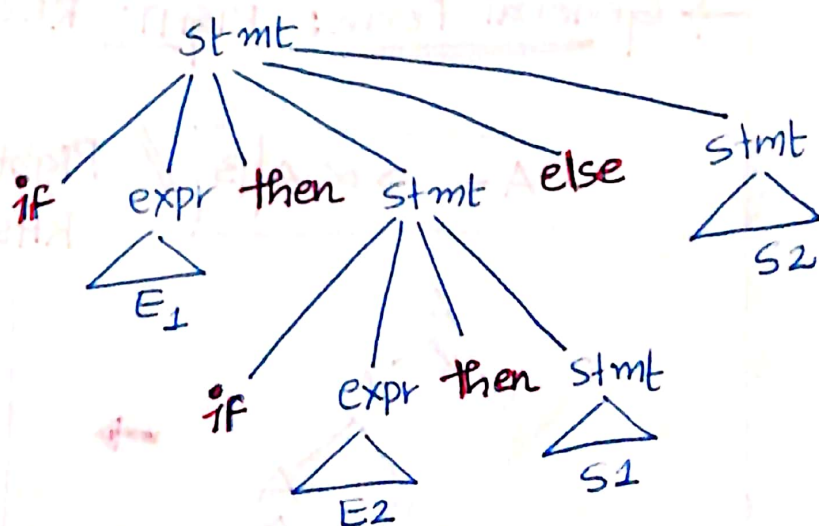
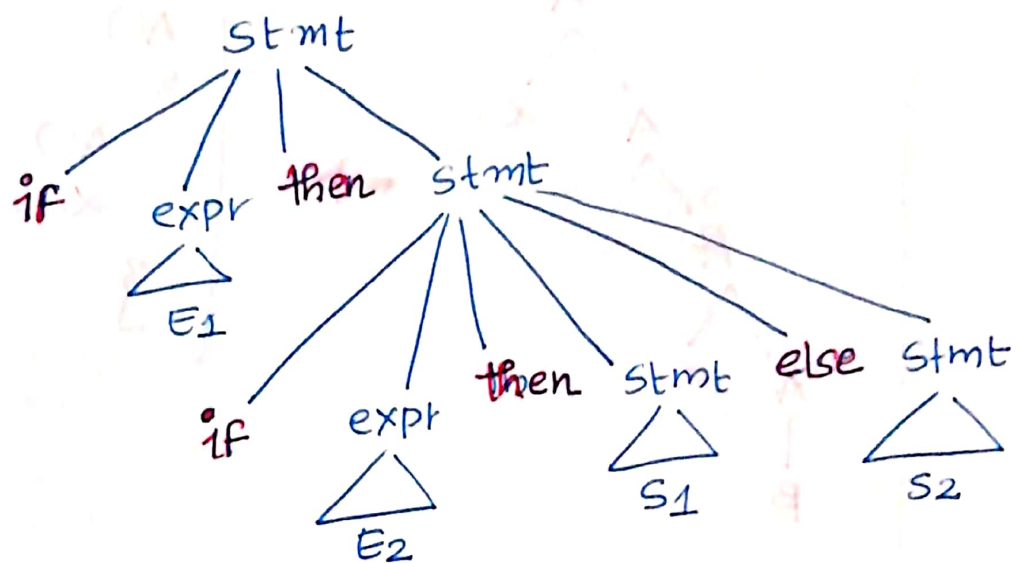
stmt  $\longrightarrow$  if expr then stmt  
                  | if expr then stmt else stmt  
                  | other.

Here other stands for any other statement.  
According to this grammar, ~~then the compound~~  
the compound conditional statement

if  $E_1$  then  $S_1$  else if  $E_2$  then  $S_2$  else  $S_3$   
has the Parse tree.



→ Grammar is ambiguous since the string  
 if  $E_1$  then if  $E_2$  then  $S_1$  else  $S_2$   
 has the two parse trees.





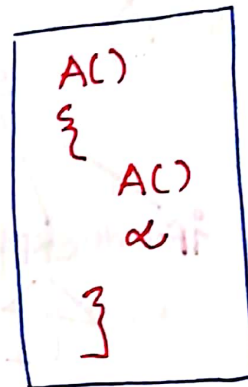
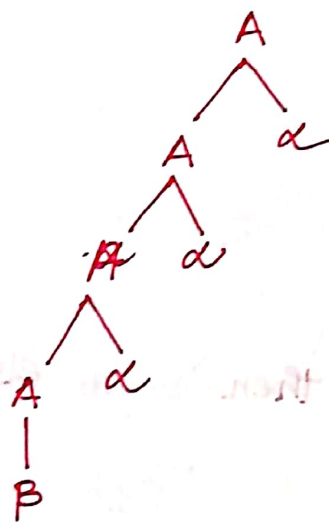
## ■ Elimination of Left Recursion

A grammar is left recursive if it has a nonterminal  $A$  such that there is a derivation  $A \xRightarrow{+} A\alpha$  for some string  $\alpha$ .

→ Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed.

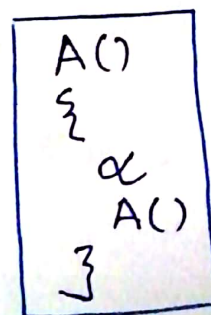
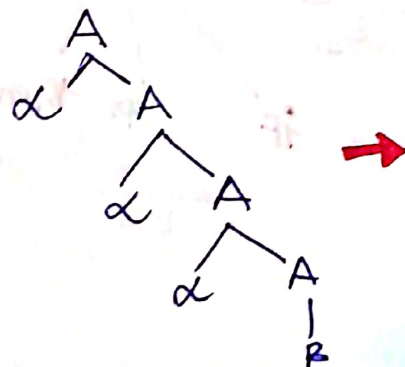
→ General Form: LEFT RECURSIVE

$$A \rightarrow A\alpha \mid B \quad // \text{Left-most symbol of RHS=LHS}$$



→ General Form: RIGHT RECURSIVE

$$A \rightarrow \alpha A \mid B \quad // \text{Right-most symbol of RHS = LHS.}$$



→ The left-recursive pair of productions  
 $A \rightarrow A\alpha | \beta$  could be replaced by the non-  
left-recursive production.

$$\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' | \epsilon \end{array}$$

### EXAMPLE

① Consider the following grammar for arithmetic expression.

$$E \rightarrow E + T | T$$

$$T \rightarrow T * F | F$$

$$F \rightarrow (E) | id$$

Eliminating the immediate left recursion to the production for E and then for T, we obtain

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' | \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' | \epsilon$$

$$F \rightarrow (E) | id$$

②  $S \rightarrow SOS' | 01$



$$S \rightarrow 01S'$$

$$S' \rightarrow \epsilon | OS'S'$$

$$(3) \quad L \rightarrow L, s \mid S$$



$$L \rightarrow SL'$$

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

No matter how many A-productions there are, we can eliminate immediate left recursion from them by the following technique.

→ 1st we group the A-productions as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

where no  $\beta_i$  begins with an A. Then, we replace the A-productions by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

The nonterminal A generates the same string as before but is no longer left recursive. This procedure eliminates all immediate left recursion from the A and A' productions, but it does not eliminate left recursion involving derivations of two or more steps.

Example

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid e$$

→ The nonterminal S is left recursive because

$S \Rightarrow Aa \Rightarrow Sda$ , but it is not immediately left recursive.



## ALGORITHM Eliminating left Recursion.

Input: Grammar  $G$  with no cycles or  $\epsilon$ -productions.

Output: An equivalent grammar with no left recursion.

Method:

① Arrange the nonterminals in some order

$A_1, A_2, \dots, A_n$ .

② For  $i := 1$  to  $n$  do begin

for  $j := 1$  to  $i-1$  do begin.

replace each production of the form  $A_i \rightarrow A_j \gamma$

by the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ .

Where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all the

current  $A_j$ -productions.

end

eliminate the immediate left recursion among

the  $A_i$ -productions.

end.

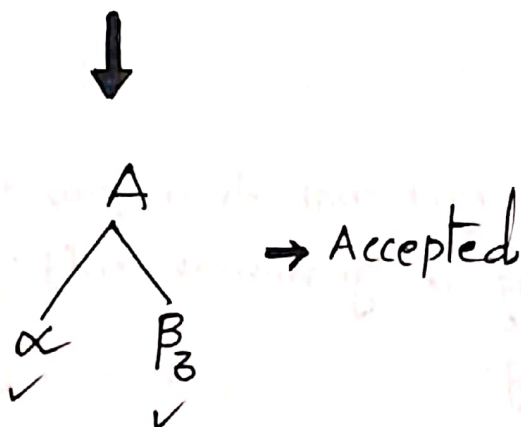
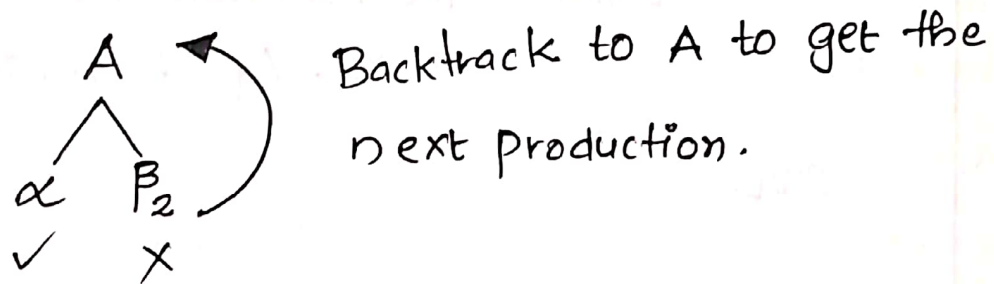
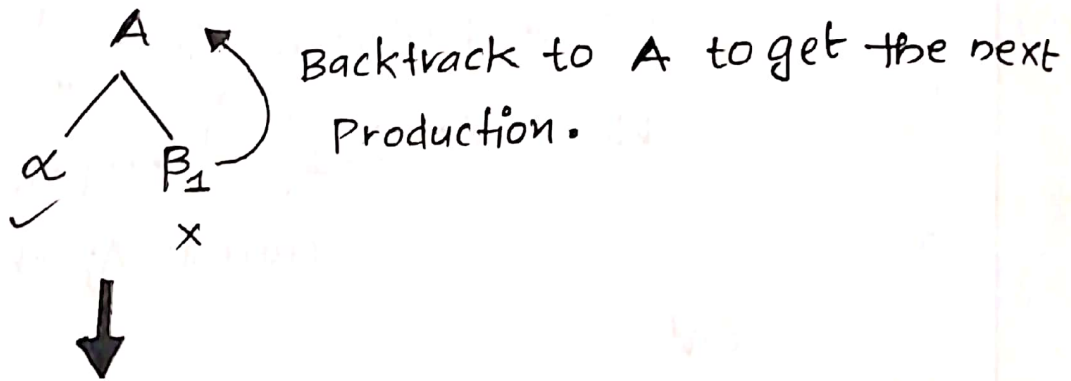
## ■ Left Factoring

Left Factoring is a grammar transformation that is useful for producing a grammar suitable for Predictive Parsing.

If we have one or more alternative productions to use to expand a nonterminal 'A'. We may be able to

rewrite the A-productions with different decision until we have enough of the input to make the right choice.

- Let the grammar be  $A \rightarrow \alpha/\beta_1/\alpha/\beta_2/\alpha/\beta_3$  be the nondeterministic grammar because we are having several options on a single nonterminal.
- Let us assume we want to derive  $\alpha/\beta_3$ . The parser choose the first production  $\alpha/\beta_1$ .



→ If a single nonterminal is having one or more productions then they arise a problem called Backtracking.

→ Backtracking happens due to the common prefixes. This problem is also called as common prefix problem.

→ The solution to backtracking is that we have to postpone the decision making until we reach  $P_3$ . To have a common solution a method called left factoring is introduced.

→ In this method, whatever common prefixes we are having we take them out.

### Example

Let the grammar be:  $A \rightarrow \alpha | \beta_1 | \alpha | \beta_2 | \alpha | \beta_3 \dots$

The above grammar can be rewritten as

$$\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 | \beta_2 | \beta_3 \dots \end{array}$$

① Convert Nondeterministic grammar to Deterministic Grammar.

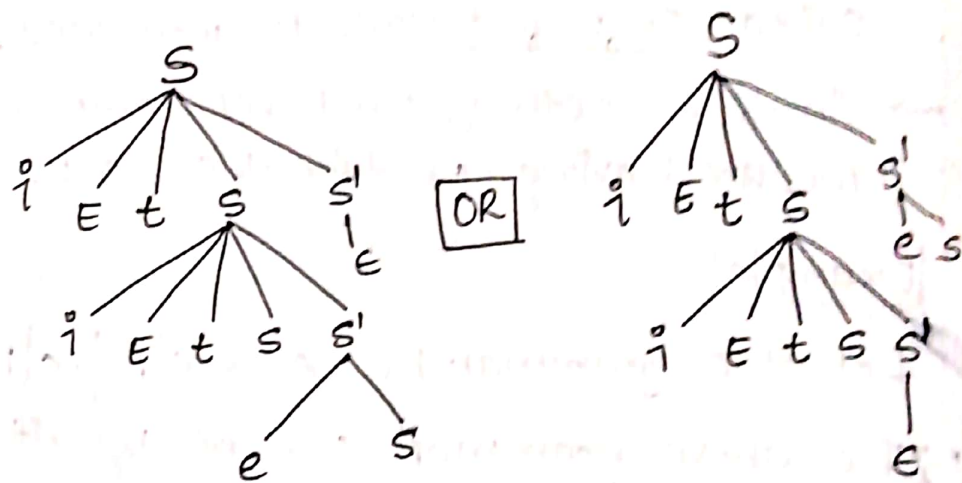
$$\begin{array}{l} S \rightarrow i E t s \\ \quad | i E t S e S \\ \quad | a \\ E \rightarrow b \end{array}$$



$$\begin{aligned} \rightarrow S &\rightarrow iEtss'|a \\ S' &\rightarrow \epsilon|es \\ E &\rightarrow b \end{aligned}$$

Eliminating Non-determinism or left factoring do not eliminate ambiguity.

example: consider the string  $iEtisEs$ .



From the above parse tree, we understand the left factoring or deterministic grammar do not eliminate ambiguity.

- ② Eliminate nondeterministic (ND) from the below grammar.

$$\begin{aligned} S &\rightarrow assbs \\ &\quad | asasb \\ &\quad | abb \\ &\quad | b \end{aligned}$$

→ we are pulling out 'a' first

$$S \rightarrow as' | b$$

$$s' \rightarrow ssbs | sasb | bb$$



$$s' \rightarrow ss'' | bb$$

$$s'' \rightarrow sbs | asb$$

Final Grammar is,

$$S \rightarrow as' | b$$

$$s' \rightarrow ss'' | bb$$

$$s'' \rightarrow sbs | asb$$

③ Eliminate ND from the below grammar.

$$\begin{array}{l} S \rightarrow bss\underline{a}as \\ \quad | bs\underline{s}asb \\ \quad | bs\underline{b} \\ \quad | a \end{array}$$

↓ Left Factoring

→

$$\begin{array}{l} S \rightarrow bss/a \\ s' \rightarrow saas | sasb | b \end{array}$$

↓

$$\begin{array}{l} s' \rightarrow sas'' | b \\ s'' \rightarrow as | sb \end{array}$$

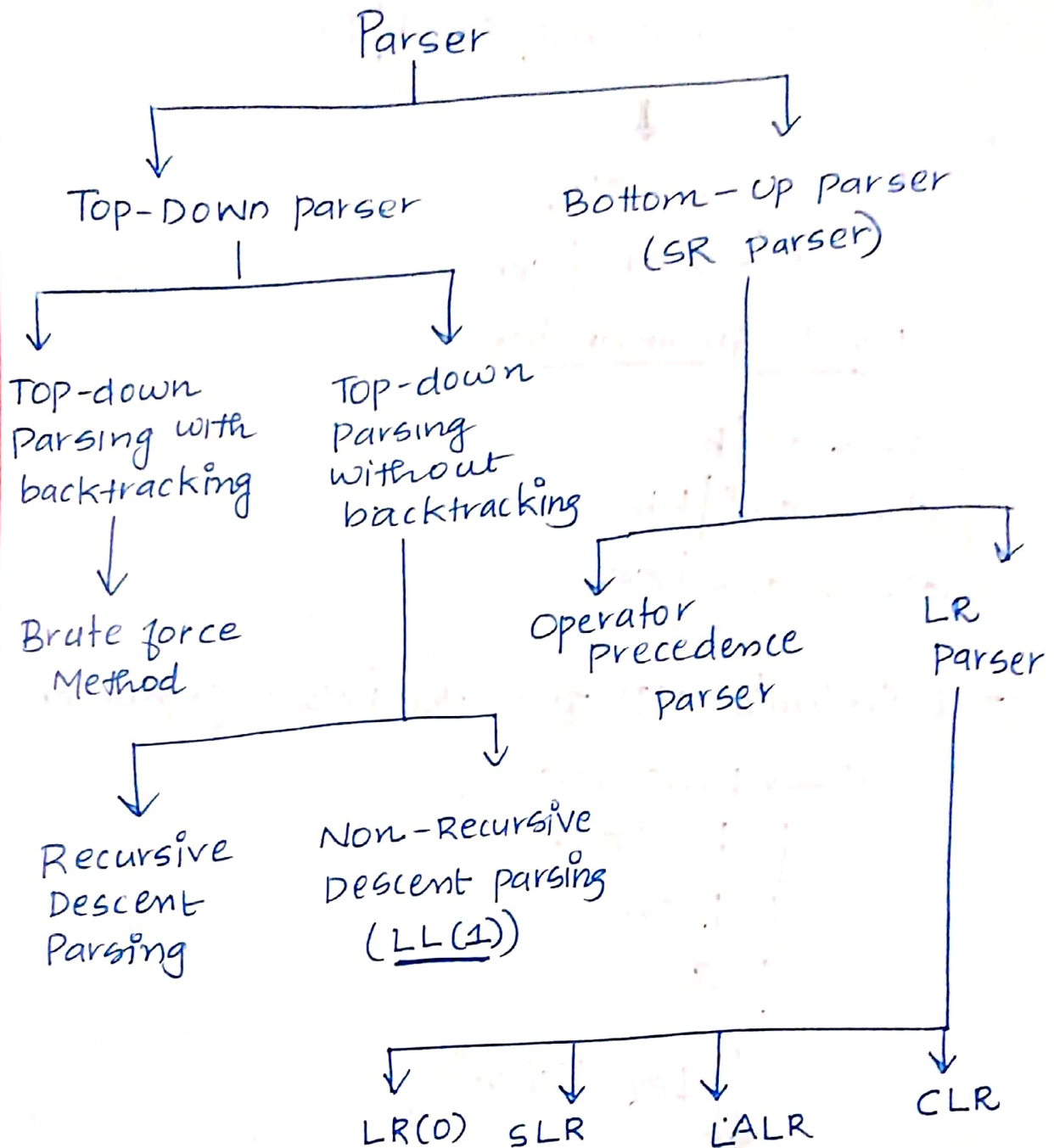
Final Grammar

$$s' \rightarrow bss'/a$$

$$s' \rightarrow sas'' | b$$

$$s'' \rightarrow as | sb$$

# TOP-DOWN PARSING





## ■ Recursive - Descent Parsing

Top-down Parsing is an attempt to find a left most derivation for an input string. Top-down parsing constructs a parse tree for the given input string from the root and creating the nodes of the parse tree in the preorder.

- General form of top-down parsing is called Recursive descent that may involve backtracking that is making repeated scans of the input.
- Backtracking parsers are not used frequently due to its time consumption for constructing the given input.

### Example

Consider the grammar:-

$$S \rightarrow cAd$$

$$A \rightarrow ab|a$$

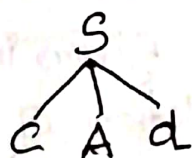
$$\text{input string} \rightarrow w \rightarrow \underline{cad}.$$

To construct parse tree for this string by top-down parser we initially create a tree consisting of a single non-terminal node  $S$ .

$$S \quad w \rightarrow c a d$$

↑

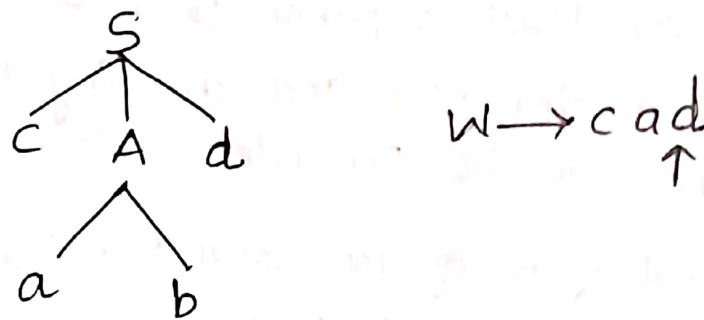
Then we use the first expansion of  $S$ .



$$w \rightarrow c a d$$

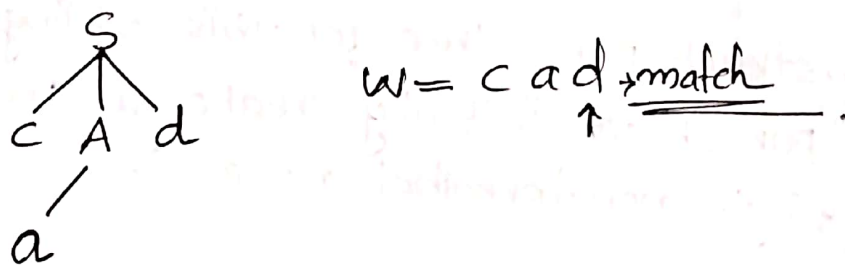
↑

The leftmost symbol of parse tree matches with the first character of the input, so we advance pointers to the next character 'a'.



We expand A using the first alternative for A. We now have a match with the 2nd input symbol. So we advance pointer to 'd', third symbol and compared 'd' against the next leaf labelled 'b'. Since 'b' does not match with 'd', we report failure and go back to A to see whether there is any other alternative.

While doing back to A, we must reset the Input pointer to position-2, i.e., to 'a'. we find the alternative of A as 'a'.



The leaf 'a' matches with the second symbol of the input, and the input pointer advance to 3rd symbol 'd', it matches with the leaf 'd'. Since we produced parse tree for the given input, we halt and announce successful completion of Parsing.



## ■ Predictive Parsers

Recursive descent Parsing without backtracking is called Predictive Parsers.

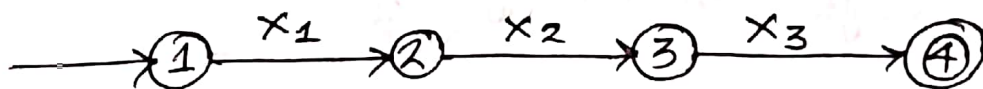
### Transition Diagram for Predictive Parsers:

In case of the parser there is no one diagram for each non-terminal. The labels of edges are tokens and nonterminals. A transition on a terminal (token) means we should take that transition if that token is the next input symbol. A transition on a nonterminal  $A$  is call for the procedure for  $A$ .

To construct the transition diagram of the predictive parser from a grammar, first eliminate left recursion from the grammar, and then left factor the grammar. Then for each non-terminal  $A$  do the following:

- ① create an initial and final state.
- ② For each production  $A \rightarrow X_1 X_2 \dots X_n$  create a path from the initial to the final state, with edges labelled  $X_1 X_2 \dots X_n$ .

Let  $A \rightarrow X_1 X_2 X_3$



- ⊗ A predictive Parser working off the transition diagrams behaves as follows:



→ The Parser begins in the Start State for the Start Symbol.

→ If after some actions, it is in state 's' with an edge labelled by terminal 'a' to state 't', and if the next input symbol is 'a', then the parser moves the input cursor one position right and goes to state 't'.

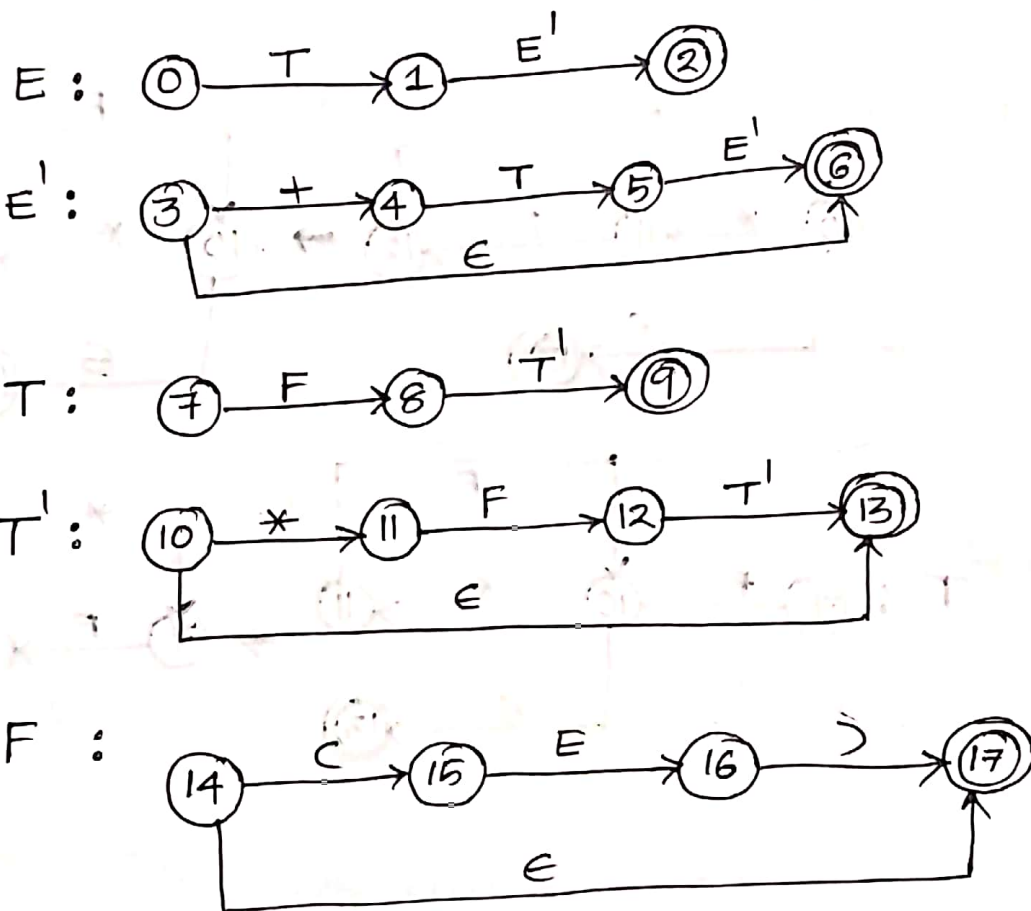
→ If, on the other hand, the edge is labeled by a nonterminal A, the parser instead goes to the start state for A, without moving the input cursor.

→ If it ever reaches the final state for A, immediately goes to state 't' in effect having "read" A from the input during the time it moved from state 's' to 't'. Finally, if there is an edge from 's' to 't' labeled  $\epsilon$ , then from state 's' the parser immediately goes to state 't', without advancing the input.

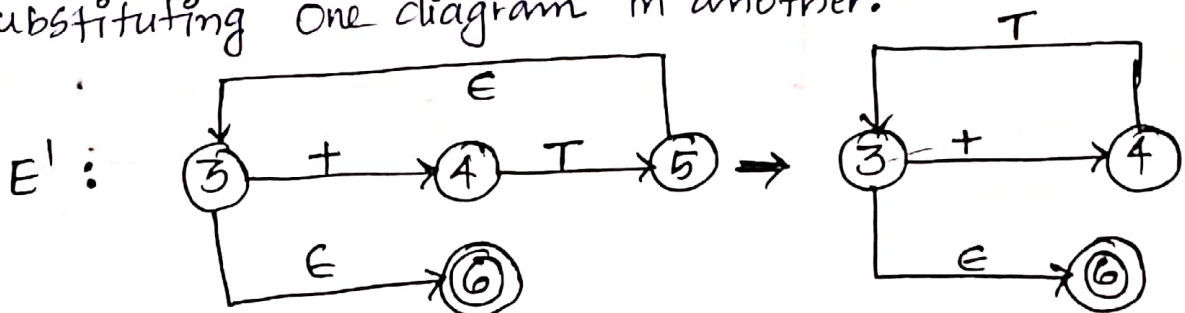
→ The predictive parsing Program based on a transition diagram attempt to match terminal symbols against the input, and make recursive procedure call whenever it has to follow any edge labeled by any nonterminal.

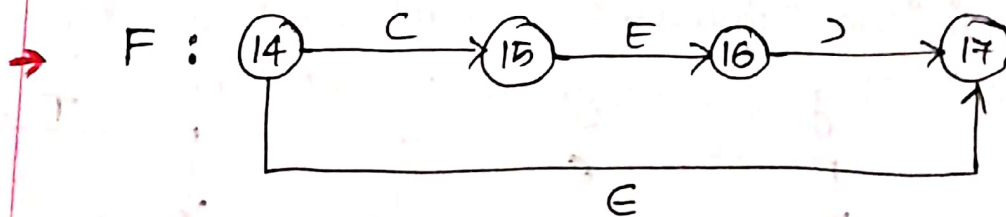
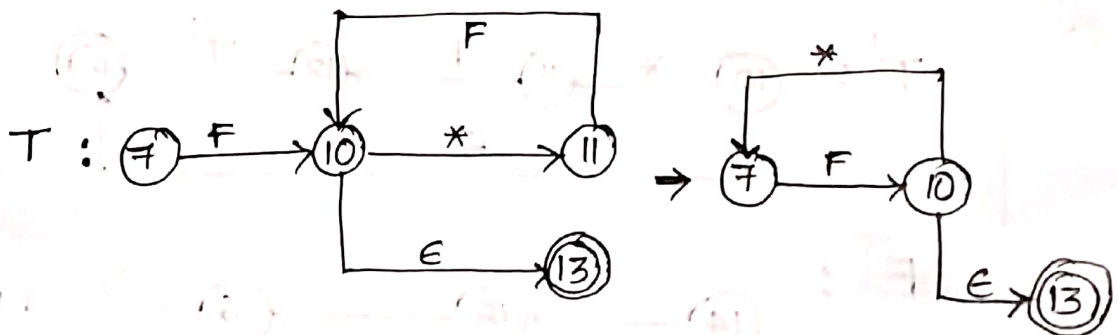
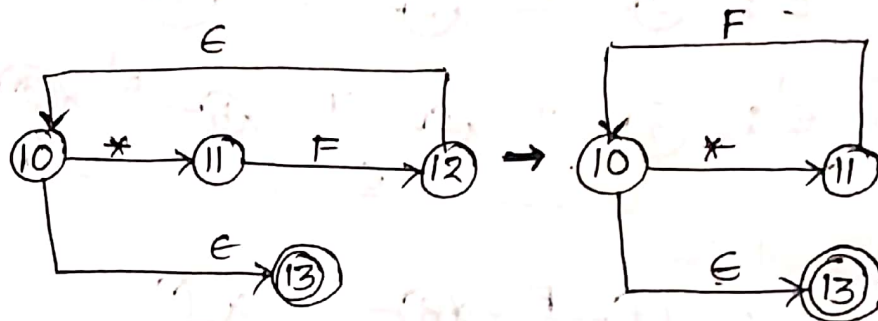
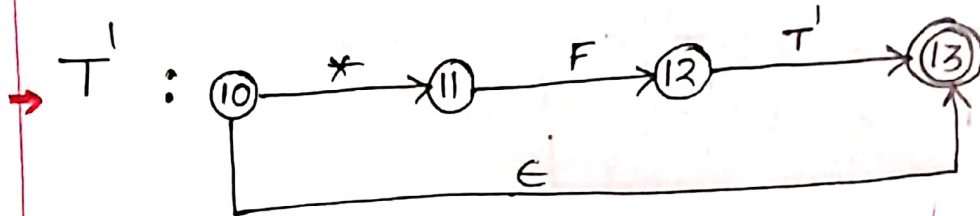
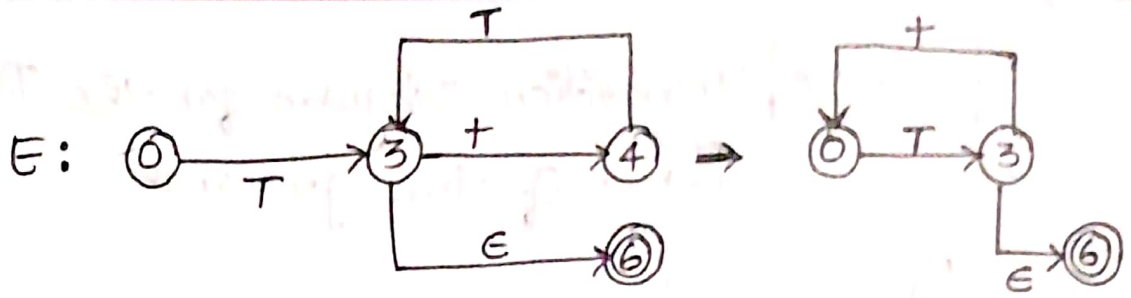
# Example Transition Diagram for the Predictive Parser of the grammar

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



Transition Diagrams can be simplified by substituting one diagram in another.







## ■ Nonrecursive Predictive Parsing (LL(1))

It is possible to build a nonrecursive Predictive Parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of determining the production to be applied for a nonterminal.

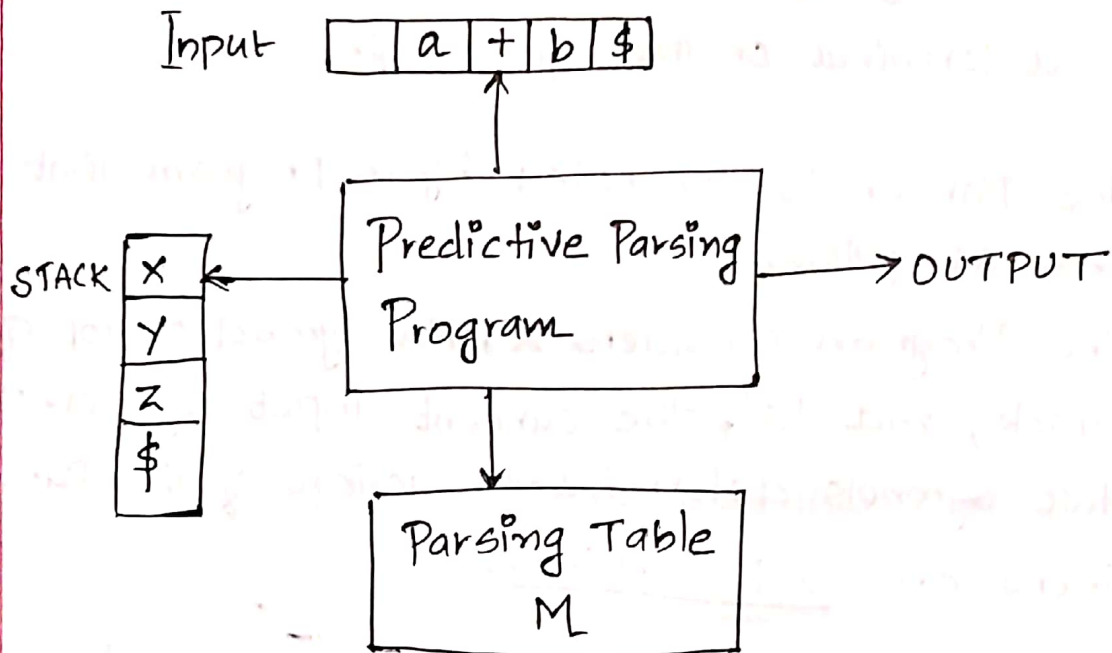


Fig: Model of a nonrecursive Predictive Parser.

The nonrecursive parser looks up the production to be applied in a parsing table.

→ A table driven Predictive Parser has an Input buffer, a stack, a Parsing table and an output stream.

→ The input buffer contains the string to be parsed, followed by \$, a symbol used as a right endmarker to indicate the end of the input buffer.

→ The Stack contains a sequence of grammar symbols with  $\$$  on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on the top of  $\$$ .

→ The Parsing Table is a two-dimensional array  $M[A, a]$ , where 'A' is a nonterminal, and 'a' is a terminal or the symbol  $\$$ .

The Parser is controlled by a Program that behaves as follows.

The Program considers  $X$ , the symbol on top of the stack, and 'a', the current input symbol. These two symbols determine the action of the Parser.

There are 3 possibilities:

① If  $X = a = \$$ , the Parser halts and announces successful completion of parsing.

② If  $X = a \neq \$$ , the Parser pops  $X$  off the stack and advances the input pointer to the next input symbol.

③ If  $X$  is a non-terminal, the program consults entry  $M[X, a]$  of the Parsing table  $M$ . This entry will be either an  $X$ -production of the grammar or an error entry.

If for example,  $M[X, a] = \{X \rightarrow UVW\}$ , the Parser replaces  $X$  on top of the stack by

WVU (with U on top). As output, we shall assume that the parser just prints the production used; any other code could be executed here.

If  $M[x, a] = \text{error}$ , the parser calls an error recovery routine.

### ALGORITHM

Input: A string  $w$  and a parsing table  $M$  for grammar  $G$ .

Output: If  $w$  is in  $L(G)$ , a leftmost derivation of  $w$ ; otherwise, an error indication.

#### Method:

Set a pointer  $ip$  to point to the first symbol of  $w\$$ .  
repeat

let  $x$  be the top stack symbol and ' $a$ ' the symbol

Pointed to by  $ip$

if  $x$  is a terminal or  $\$$  then

if  $x = a$  then

Pop  $x$  from the stack and advance  $ip$

else error()

else /\*  $x$  is a nonterminal \*/

if  $M[x, a] = x \rightarrow \gamma_1 \gamma_2 \dots \gamma_k$  then begin

Pop  $x$  from the stack;

Push  $\gamma_k, \gamma_{k-1}, \dots, \gamma_1$  onto the stack,  
with  $\gamma_1$  on top;

Output the production  $x \rightarrow \gamma_1 \gamma_2 \dots \gamma_k$

end

else error()

until  $x = \$$  /\* stack is empty \*/



## FIRST AND FOLLOW

The construction of Predictive Parser is aided by two functions associated with a grammar  $G$ . These functions are:

① FIRST

② FOLLOW

These functions allow us to fill in the entries of a Predictive Parsing table for  $G$ , whenever possible.

If  $\alpha$  is any string of grammar symbols, let FIRST( $\alpha$ ) be the set of terminals that begins the strings derived from  $\alpha$ .

If  $\alpha \xrightarrow{*} \epsilon$ , then  $\epsilon$  is also in FIRST( $\alpha$ ).

Define FOLLOW( $A$ ), for nonterminal  $A$ , to be the set of terminals 'a' that can appear immediately to the right of  $A$  in some sentential form, that is, the set of terminals 'a' such that there exists a derivation of the form  $S \xrightarrow{*} \alpha A a \beta$  for some  $\alpha$  and  $\beta$ .

Note that there may, at some time during the derivation, have been symbols b/w  $A$  and  $a$ , but if so, they derived  $\epsilon$  and disappeared. If  $A$  can be the rightmost symbol in some sentential form, then  $\$$  is in FOLLOW( $A$ ).

### Algorithm Used to compute FIRST(X) for Grammar Symbol X.

1. If  $X$  is terminal, then  $\text{FIRST}(X)$  is  $\{X\}$ .
2. If  $X \rightarrow \epsilon$  is a production, then add  $\epsilon$  to  $\text{FIRST}(X)$ .
3. If  $X$  is non-terminal and  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production, then place 'a' in  $\text{FIRST}(X)$  if for some  $i$ , 'a' is in  $\text{FIRST}(Y_i)$ , and  $\epsilon$  is in all of  $\text{FIRST}(Y_1) \dots \text{FIRST}(Y_{i-1})$ .

If  $\epsilon$  is in  $\text{FIRST}(Y_j)$  for all  $j=1, 2, \dots, k$ , then add  $\epsilon$  to  $\text{FIRST}(X)$ .

For example, everything in  $\text{FIRST}(Y_1)$  is surely in  $\text{FIRST}(X)$ . If  $Y_1$  does not derive  $\epsilon$ , then we add nothing more to  $\text{FIRST}(X)$ , but if  $Y_1 \xrightarrow{*} \epsilon$ , then we add  $\text{FIRST}(Y_2)$  and so on.

### Algorithm Used to compute FOLLOW(X)

1. Place  $\$$  in  $\text{FOLLOW}(S)$ , where  $S$  is the start symbol and  $\$$  is the input right endmarker.
2. If there is a production  $A \rightarrow \alpha B \beta$ , then everything in  $\text{FIRST}(\beta)$  except for  $\epsilon$  is placed in  $\text{FOLLOW}(B)$ .
3. If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B \beta$  where  $\text{FIRST}(\beta)$  contains  $\epsilon$  (ie.,  $\beta \xrightarrow{*} \epsilon$ ), then everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ .

## Examples

①

$S \rightarrow ABCDE$

$A \rightarrow a|e$

$B \rightarrow b|e$

$C \rightarrow c$

$D \rightarrow d|e$

$E \rightarrow e|e$

FIRST

$\{a, b, c\}$

$\{a, e\}$

$\{b, e\}$

$\{c\}$

$\{d, e\}$

$\{e, e\}$

FOLLOW

$\{\$ \}$

$\{b, c\}$

$\{c\}$

$\{d, e, \$ \}$

$\{e, \$ \}$

$\{\$ \}$

②

$S \rightarrow Bb|Cd$

$B \rightarrow aB|e$

$C \rightarrow cC|e$

$\{a, b, c, d\}$

$\{a, e\}$

$\{c, e\}$

$\{\$ \}$

$\{b\}$

$\{d\}$

③

$E \rightarrow TE'$

$E' \rightarrow +TE'|e$

$T \rightarrow FT'$

$T' \rightarrow *FT'|e$

$F \rightarrow id|(e)$

$\{id, c\}$

$\{+, e\}$

$\{id, c\}$

$\{*, e\}$

$\{id, c\}$

$\{\$, \}$

$\{\$, \}$

$\{+, \$, \}$

$\{+, \$, \}$

$\{*, +, \$, \}$

④

$S \rightarrow ACB|cbB|Ba$

$A \rightarrow da|BC$

$B \rightarrow g|e$

$C \rightarrow h|e$

$\{d, g, b, e, b, a\}$

$\{d, g, b, e\}$

$\{g, e\}$

$\{h, e\}$

$\{\$ \}$

$\{b, g, \$ \}$

$\{\$, a, b, g\}$

$\{g, \$, b, h\}$



	FIRST	FOLLOW
⑤ $S \rightarrow aABb$ $A \rightarrow c e$ $B \rightarrow d e$	$\{a\}$ $\{c, e\}$ $\{d, e\}$	$\{\$ \}$ $\{d, b\}$ $\{b\}$ $\{\$ \}$
⑥ $S \rightarrow aBDh$ $B \rightarrow cC$ $C \rightarrow bC e$ $D \rightarrow EF$ $E \rightarrow g e$ $F \rightarrow f e$	$\{a\}$ $\{c\}$ $\{b, e\}$ $\{g, f, e\}$ $\{g, e\}$ $\{f, e\}$	$\{\$ \}$ $\{g, f, b\}$ $\{g, f, b\}$ $\{b\}$ $\{f, b\}$ $\{b\}$
⑦ $S \rightarrow ABCD$ $A \rightarrow b e$ $B \rightarrow c$ $C \rightarrow d$ $D \rightarrow e$	$\{b, c\}$ $\{b, e\}$ $\{c\}$ $\{d\}$ $\{e\}$	$\{\$ \}$ $\{e\}$ $\{d\}$ $\{e\}$ $\{\$ \}$
⑧ $S \rightarrow AB$ $A \rightarrow a e$ $B \rightarrow b e$	$\{a, b, e\}$ $\{a, e\}$ $\{b, e\}$	$\{\$ \}$ $\{b, \$ \}$ $\{\$ \}$

# Construction of Predictive Parsing Tables

Consider the grammar,

	<u>FIRST</u>	<u>FOLLOW</u>
$E \rightarrow TE'$	$\{id, c\}$	$\{\$, \}$
$E' \rightarrow +TE'   \epsilon$	$\{+, \epsilon\}$	$\{\$, \}$
$T \rightarrow FT'$	$\{id, c\}$	$\{+, \}, \{\$, \}$
$T' \rightarrow *FT'   \epsilon$	$\{*, \epsilon\}$	$\{+, \}, \{\$, \}$
$F \rightarrow id   (E)$	$\{id, c\}$	$\{*, +, \}, \{\$, \}$

- To construct the LL(1) Parsing table we need to find out the FIRST() and FOLLOW() of the given grammar first.
- The Parsing table consist of rows and columns as 2D-array.
- Rows are the LHS of the productions of the given ~~array~~ grammar.
- COLUMNS are the terminals in the given grammar which is the FIRST(RHS) of the grammar.
- Instead of  $\epsilon$ , we include  $\$$  in the Parsing table b/c  $\epsilon$  is not present in the input string.
- The main purpose of LL(1) parser is to construct the parsing table using the FIRST() and FOLLOW() of the given grammar.

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

→ The First production of the grammar is  $E \rightarrow TE'$ .

→ To enter this production into the parsing table, we first consider the  $FIRST(TE') = \{id, ( \}$ . So we need to enter the production  $E \rightarrow TE'$  across 'c' and 'id' terminals.

→ The Second Production Grammar is having 2 Productions:

$$(1) E' \rightarrow +TE'$$

$$(2) E' \rightarrow E$$

→ In the 1<sup>st</sup> production  $E' \rightarrow +TE'$ , the  $FIRST(+TE') = \{+\}$ . So we need to enter production  $E' \rightarrow +TE'$  across '+' terminal.

→ In the 2<sup>nd</sup> production, we are having  $E$ , in this case, we have to consider the  $FOLLOW(E') = \{ \$, ) \}$ . So we need to enter  $E' \rightarrow E$  production across the column \$ and ).



→ The third grammar is  $T \rightarrow FT'$ , we need to enter this grammar  $T \rightarrow FT'$  in the  $FIRST(FT')$ .  $\{id, c\}$ . So  $T \rightarrow FT'$  is entered across 'id' and 'c'.

→ The 4<sup>th</sup> grammar is  $T' \rightarrow *FT' | \epsilon$ , having 2 Productions

$$T' \rightarrow *FT'$$

$$T' \rightarrow \epsilon$$

The 1<sup>st</sup> production need to enter across  $FIRST(*FT') \rightarrow *$ . The 2<sup>nd</sup> production  $T' \rightarrow \epsilon$  need to be enter across  $FOLLOW(T')$  i.e., across '(', ')', '\$'.

→ The last grammar  $F \rightarrow id | (E)$  need to be enter across  $FIRST(id)$  and  $FIRST(E)$  across 'id' and 'c'.

### EXAMPLE

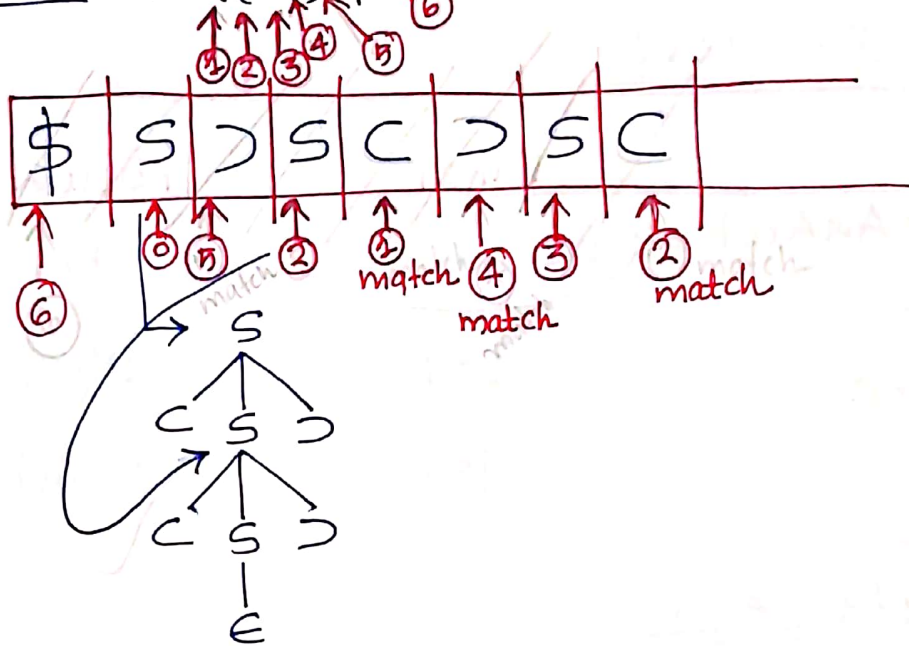
consider the grammar  $S \rightarrow (S) | \epsilon$ . This grammar is used to generate parenthesis.

	FIRST	FOLLOW
$S \rightarrow (S)   \epsilon$	$\{c, \epsilon\}$	$\{\$, \epsilon\}$

# Predictive Parsing Table

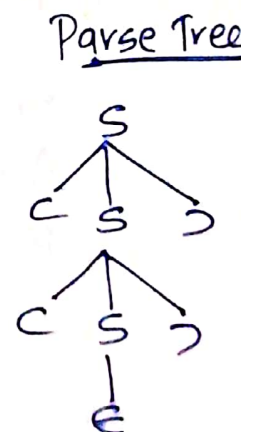
	$($	$)$	$\$$
$S$	$S \rightarrow (S)$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

Pattern  $\rightarrow ((S))\$ \rightarrow$  Accepted.



Let the input we want to generate be  $(( ))$ .

Stack	Input	Output	Parse Tree
$\$$	$(( ))\$$		
$\$S$	$(( ))\$$		
$\$ ) S ($	$(( ))\$$	$S \rightarrow (S)$	
$\$ ) S$	$(( ))\$$		
$\$ ) S ($	$(( ))\$$	$S \rightarrow (S)$	
$\$ ) S$	$(( ))\$$		
$\$ ) \epsilon$	$(( ))\$$	$S \rightarrow \epsilon$	
$\$ )$	$(( ))\$$		
$\$$	$(( ))\$$	accepted.	



## EXAMPLES

- ① Construct Parsing Table for the given grammar.

$$S \rightarrow AaAb | BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

→ FIRST & FOLLOW

	FIRST	FOLLOW
$S \rightarrow AaAb   BbBa$	$\{a, b\}$	$\{\$, \}$
$A \rightarrow \epsilon$	$\{\epsilon\}$	$\{a, b\}$
$B \rightarrow \epsilon$	$\{\epsilon\}$	$\{b, a\}$

## PARSING TABLE

	a	b	\$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$	

- ② Construct the Parsing Table for the following grammar.

$$S \rightarrow aABb$$

$$A \rightarrow c | \epsilon$$

$$B \rightarrow d | \epsilon$$



## ⇒ FIRST & FOLLOW

$S \rightarrow aABb$	<b>FIRST</b> $\{a\}$	<b>FOLLOW</b> $\{\$ \}$
$A \rightarrow c e$	$\{c, e\}$	$\{d, b\}$
$B \rightarrow d e$	$\{d, e\}$	$\{b\}$

## PARSING TABLE

	a	b	c	d	\$
S	$S \rightarrow aABb$				
A		$A \rightarrow e$	$A \rightarrow c$	$A \rightarrow e$	
B		$B \rightarrow e$		$B \rightarrow d$	

- ③ construct the parsing Table for the following grammar.

$S \rightarrow AB$   
 $A \rightarrow a|e$   
 $B \rightarrow b|e$

## ⇒ FIRST AND FOLLOW

	<b>FIRST</b>	<b>FOLLOW</b>
$S \rightarrow AB$	$\{a, b, e\}$	$\{\$ \}$
$A \rightarrow a e$	$\{a, e\}$	$\{b, e\}$
$B \rightarrow b e$	$\{b, e\}$	$\{\$ \}$

## PARSING TABLE

	a	b	\$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
A	$A \rightarrow a$	$A \rightarrow e$	$A \rightarrow e$
B		$B \rightarrow b$	$B \rightarrow e$

④ consider the following grammar

$$S \rightarrow A$$

$$A \rightarrow Bb | cd$$

$$B \rightarrow aB | \epsilon$$

$$C \rightarrow cC | \epsilon$$

⇒ FIRST AND FOLLOW

	FIRST	FOLLOW
$S \rightarrow A$	$\{a, b, c, d\}$	$\{\$ \}$
$A \rightarrow Bb   cd$	$\{a, b, c, d\}$	$\{\$ \}$
$B \rightarrow aB   \epsilon$	$\{a, \epsilon\}$	$\{b\}$
$C \rightarrow cC   \epsilon$	$\{c, \epsilon\}$	$\{d\}$

PARSING TABLE

	a	b	c	d	\$
S	$S \rightarrow A$	$S \rightarrow A$	$S \rightarrow A$	$S \rightarrow A$	
A	$S \rightarrow Bb$	$S \rightarrow Bb$	$S \rightarrow cd$	$S \rightarrow cd$	
B	$B \rightarrow aB$	$B \rightarrow \epsilon$			
C			$C \rightarrow cC$	$C \rightarrow \epsilon$	

# EXERCISES

6) construct parsing table for the following grammar.

$$\begin{aligned} \text{(a)} \quad E &\rightarrow TA \\ A &\rightarrow +TA \mid \epsilon \\ T &\rightarrow FB \\ B &\rightarrow *FB \mid \epsilon \\ F &\rightarrow (E) \mid id \end{aligned}$$

$$\begin{aligned} \text{(b)} \quad S &\rightarrow 1AB \mid \epsilon \\ A &\rightarrow 1AC \mid OC \\ B &\rightarrow OS \\ C &\rightarrow 1 \end{aligned}$$

$$\begin{aligned} \text{(c)} \quad S &\rightarrow A \\ A &\rightarrow aB \mid Ad \\ B &\rightarrow bBd \mid f \\ d &\rightarrow g \end{aligned}$$

$$\begin{aligned} \text{(d)} \quad S &\rightarrow iEtSS_1 \mid a \\ S_1 &\rightarrow eS \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

$$\begin{aligned} \text{(e)} \quad S &\rightarrow aBDh \\ B &\rightarrow cC \\ C &\rightarrow bC \mid \epsilon \\ D &\rightarrow EF \\ E &\rightarrow g \mid \epsilon \\ F &\rightarrow f \mid \epsilon \end{aligned}$$

$$\begin{aligned} \text{(f)} \quad E &\rightarrow TE' \\ E' &\rightarrow +E \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow T \mid \epsilon \\ F &\rightarrow PF' \\ F' &\rightarrow *F' \mid \epsilon \\ P &\rightarrow (E) \mid a \mid b \mid \epsilon \end{aligned}$$



# MODULE III

## Bottom Up Parsing:

Shift - Reduce Parsing — Operator precedence Parsing.

LR Parsing — Constructing SLR Parsing tables, Constructing Canonical LR Parsing tables and Constructing LALR Parsing Tables.

## BOTTOM UP PARSING

- Bottom up parsing is also known as Shift - Reduce Parser.
- An easy form of shift reduce implementation is the Operator Precedence Parser.
- A general Method of Shift reduce parsing is called LR Parsing. It is used in a no. of automatic parser generator.
- Shift Reduce Parser attempt to construct a parse tree for an input string beginning at the leaves (bottom) and moves towards the root (top).  
This process reduces a string 'w' to the start symbol of a grammar.
- At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left side of that production.

### EXAMPLE

Consider the grammar

$$S \rightarrow aT Ue$$

$$T \rightarrow Tbc|b$$

$$U \rightarrow d$$

The rightmost derivation of the sentence for the string "abbcd e" is as given below.

$$\begin{aligned} S &\xrightarrow{rm} aT Ue \\ &\xrightarrow{rm} aT d e \\ &\xrightarrow{rm} aT b c d e \\ &\xrightarrow{rm} \underline{\underline{a b b c d e}} \end{aligned}$$

Let us read the string abbcde from left to right and perform the right most derivation in the reverse order, it can be performed by the following 4 steps:

- ① choose the 1<sup>st</sup> 'b' and reduce it to the left side of the  $T \rightarrow b$  production to produce the sentential form aTbcde.
- ② Now reduce Tbc by the left hand side of the production  $T \rightarrow Tbc$  to produce the sentential form aTde.
- ③ Now reduce d by left hand side of the production  $U \rightarrow d$  to produce aT Ue.
- ④ Finally aT Ue can be replaced by the left hand side of  $S \rightarrow aT Ue$  production to get S.

## EXAMPLE

$$S \rightarrow aABe$$

$$A \rightarrow Abc|b$$

$$B \rightarrow d$$

The sentence "abbcede" can be reduced to  $S$  by the following steps.

abbcede

aAbcde

aAde

aABe

S ✓

The sentence "abbcede" can be derived from the above grammar through RMS as follows:

$$S \xrightarrow{rm} dABe$$

$$S \xrightarrow{rm} aAde$$

$$\xrightarrow{rm} aAbcde$$

$$\xrightarrow{rm} \underline{\underline{abbcede}}$$



# SHIFT - REDUCE PARSING

Bottom-up parsing is also known as Shift-Reduce Parsing, because its two main actions are  
① Shift ② Reduce.

→ At each action, the current symbol in the input string is pushed to a stack.

→ At each reduction step, the symbols at the top of the stack will be replaced by the non-terminal at the left side of that production.

## ■ Handles

→ Handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a RMD.

## Example:

Grammar

String

$S \rightarrow E$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow \text{num}$

$E \rightarrow \text{id}$

$\text{id}_1 + \text{num} * \text{id}_2$

Stack	Action
\$	Shift
\$ <u>id</u> 1	Reduce
\$ E	Shift
\$ E +	Shift
\$ E + <u>num</u>	Reduce
\$ E + E	Shift
\$ E + E *	Shift
\$ E + E * <u>id</u> 2	Reduce
\$ E + E * E	Reduce
\$ <u>E + E</u>	Reduce
\$ <u>E</u>	Reduce
\$ S	Accept

Handles  $\longrightarrow$  Underlined

### ■ Handle Pruning

$\rightarrow$  A Right most derivation in reverse can be obtained by handle pruning or reducing the handle with left production.

#### EXAMPLE

Consider the grammar

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

Let the input string  $id_1 + id_2 * id_3$ . The sequence of reduction of string to start symbol as given below.

Right sequential Form	Handle	Reducing Production
<u><math>id_1</math></u> + $id_2 * id_3$	$id_1$	$E \rightarrow id$
$E + $ <u><math>id_2</math></u> $ * id_3$	$id_2$	$E \rightarrow id$
$E + E * $ <u><math>id_3</math></u>	$id_3$	$E \rightarrow id$
$E + $ <u><math>E * E</math></u>	$E * E$	$E \rightarrow E * E$
<u><math>E + E</math></u>	$E + E$	$E \rightarrow E + E$
$E$		

## ■ Stack Implementation of Shift-Reduce Parsing

Problems during parsing by handle Pruning:

- The first problem is to locate the substring to be reduced in a right sequential form (identification of correct handle).
- The second is to determine what production to choose in case there is more than one production with that substring on the right side.



- A convenient way to implement a shift reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string  $w$  to be parsed.
- A parser goes on shifting the output symbols on the top of the stack, until a handle comes on top of the stack. When the handle appears on top of the stack, perform reduction.
- Shift-reduce parser have the following 4 possible actions.

- ① Shift Action: Move the next input symbol on to the top of the stack.
- ② Reduce Action: Reduce the handle on right end of the stack by popping it off the stack and pushing the left side of the production of the right end of the stack.
- ③ Accept Action: Announces successful completion of parsing.
- ④ Error Action: The parser discovers syntax error has occurred and calls an error recovery routine.

### EXAMPLE

Consider the following grammar

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow id$$

Perform the action of shift-reduce parser to parse the input string  $id_1 * (id_2 + id_3)$ .

Stack content	Input String	Actions.
\$	$id_1 * (id_2 + id_3) \$$	shift $id_1$
\$ $id_1$	$* (id_2 + id_3) \$$	Reduce $E \rightarrow id$
\$ $E$	$* (id_2 + id_3) \$$	shift $*$
\$ $E *$	$(id_2 + id_3) \$$	shift $($
\$ $E * ($	$id_2 + id_3) \$$	shift $id_2$
\$ $E * (id_2$	$+ id_3) \$$	Reduce $E \rightarrow id$
\$ $E * (E$	$+ id_3) \$$	shift $+$
\$ $E * (E +$	$id_3) \$$	shift $id_3$
\$ $E * (E + id_3$	$) \$$	Reduce $E \rightarrow id$
\$ $E * (E + E$	$) \$$	shift $)$
\$ $E * (E + E)$	\$	Reduce $E \rightarrow E + E$
\$ $E * (E)$	\$	Reduce $E \rightarrow (E)$
\$ $E * E$	\$	Reduce $E \rightarrow E * E$
\$ $E$	\$	Accept.

## EXAMPLE 2

Consider the following grammar.

$$E \rightarrow E + E$$

$$E \rightarrow T$$

$$E \rightarrow E * E$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow y$$

$$\text{String} \rightarrow y + y + y * y$$

Stack Content	Input String	Action.
\$	y + y + y * y \$	Shift y.
\$y	+ y + y * y \$	reduce $F \rightarrow y$
\$F	+ y + y * y \$	reduce $T \rightarrow F$
\$T	+ y + y * y \$	reduce $E \rightarrow T$
\$E	+ y + y * y \$	Shift +
\$E +	y + y * y \$	Shift y
\$E + y	+ y * y \$	reduce $F \rightarrow y$
\$E + F	+ y * y \$	reduce $T \rightarrow F$
\$E + T	+ y * y \$	reduce $E \rightarrow T$
\$E + E	+ y * y \$	<del>Shift +</del> reduce $E \rightarrow E + E$
\$E <del>reduced</del>	+ y * y \$	Shift +
\$E +	y * y \$	Shift y
\$E + y	* y \$	reduce $F \rightarrow y$
\$E + F	* y \$	reduce $T \rightarrow F$
\$E + T	* y \$	reduce $E \rightarrow T$
\$E + E	* y \$	reduce $E \rightarrow E + E$



\$E	*y\$	shift *
\$E*	y\$	shift y
\$E*y	\$	reduce $F \rightarrow y$
\$E*F	\$	reduce $T \rightarrow F$
\$E*T	\$	reduce $E \rightarrow T$
\$E*E	\$	reduce $E \rightarrow E*E$
\$E	\$	Accept

### EXAMPLE 3

consider the following Grammar

$S \rightarrow S+S$   
 $S \rightarrow S-S$   
 $S \rightarrow (S)$   
 $S \rightarrow a$

$\text{String} \rightarrow a_1 - (a_2 + a_3)$

Stack content	Input String	Actions.
\$	$a_1 - (a_2 + a_3) \$$	shift $a_1$
$\$a_1$	$- (a_2 + a_3) \$$	reduce $S \rightarrow a$
$\$S$	$- (a_2 + a_3) \$$	shift -
$\$S-$	$(a_2 + a_3) \$$	shift (
$\$S-($	$a_2 + a_3) \$$	shift $a_2$
$\$S-(a_2$	$+ a_3) \$$	reduce $S \rightarrow a$
$\$S-(S$	$+ a_3) \$$	shift +
$\$S-(S+$	$a_3) \$$	shift $a_3$
$\$S-(S+a_3$	$) \$$	reduce $S \rightarrow a$
$\$S-(S+S$	$) \$$	shift )
$\$S-(S+S)$	$\$$	reduce $S \rightarrow S+S$

$\$S - (S)$  $\$$ reduce  $S \rightarrow (S)$  $\$S - S$  $\$$ reduce  $S \rightarrow S - S$  $\$S$  $\$$ 

Accept.

**EXAMPLE 4**

Consider the grammar

$$E \rightarrow 2E2$$

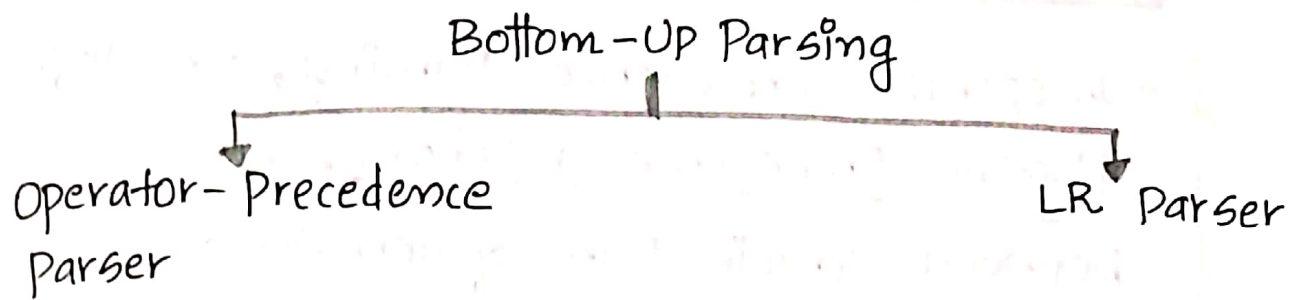
$$E \rightarrow 3E3$$

$$E \rightarrow 4$$

Perform the shift reduce parsing for input string  
~~"23432"~~ "32423".

Stack	Input string	Parsing Action.
$\$$	32423 $\$$	shift 3.
$\$3$	2423 $\$$	shift 2
$\$32$	423 $\$$	shift 4
$\$324$	23 $\$$	reduce $4 \rightarrow E$
$\$32E$	23 $\$$	shift 2
$\$32E2$	3 $\$$	reduce $E \rightarrow 2E2$
$\$3E$	3 $\$$	shift 3
$\$3E3$	$\$$	reduce $E \rightarrow 3E3$
$\$E$	$\$$	Accept.

# OPERATOR - PRECEDENCE PARSING



- Small class of grammar uses operator precedence parser to build the bottom up parser.
- It is used to define mathematical operator.

## Operator Grammar:

- An operator grammar is having following operator
  - ① NO RHS of any operator has  $\epsilon$
  - ② No two non-terminals are adjacent.

## EXAMPLE:

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

consider the grammar  $E \rightarrow EAE \mid id$   
 $A \rightarrow + \mid *$  } This grammar

is same as the above given example grammar but this grammar is not an operator grammar since  $EAE$  has two consecutive nonterminals. This can be converted to operator grammar as

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



An easy-to-implement parsing technique called Operator-Precedence Parsing.

→ In operator Precedence Parsing, we define 3 disjoint Precedence relations,  $<\cdot$ ,  $\doteq$ , and  $\cdot>$ , between certain pairs of terminals.

→ These Precedence relations guide the selection of handles and have the following meanings:

RELATION	MEANING
$a <\cdot b$	a "yields precedence to" b
$a \doteq b$	a "has the same precedence as" b
$a \cdot> b$	a "takes precedence over" b

→ There are 2 common ways of determining what precedence relations should hold b/w a pair of terminals.

① Based on the traditional notion of associativity and ② Precedence of operators.

eg:- If  $*$  is having higher precedence than  $+$ , we make  $+ <\cdot *$  and  $* \cdot> +$ .

## ■ Rules to Find the Precedence Relation

- ① 'id' is having highest Precedence than any other operator.

eg:  $\text{id} \cdot \rightarrow +$ ,  $\text{id} \cdot \rightarrow *$ ,  $\text{id} \cdot \rightarrow \&$ ,  $\text{id} \cdot \rightarrow ($ ,  $\text{id} \cdot \rightarrow )$

- ② '\$' is having least Precedence.

eg:  $\& \leftarrow \cdot \theta_1$  or  $\theta_1 \cdot \rightarrow \&$

- ③ If operator  $\theta_1$  has higher precedence than  $\theta_2$ , make  $\theta_1 \cdot \rightarrow \theta_2$  and  $\theta_2 \leftarrow \cdot \theta_1$ .

eg: If  $*$  has higher precedence than  $+$ , make

$* \cdot \rightarrow +$  and  $+ \leftarrow \cdot *$ . These relations ensure that, in an expression of the form  $E + E * E + E$ , the central  $E * E$  is the handle that will be reduced first.

- ④ If  $\theta_1$  and  $\theta_2$  are operators of equal Precedence, then make  $\theta_1 \cdot \rightarrow \theta_2$  and  $\theta_2 \cdot \rightarrow \theta_1$  if the operators are left-associative, or make  $\theta_1 \leftarrow \cdot \theta_2$  and  $\theta_2 \leftarrow \cdot \theta_1$  if they are right-associative.

eg: If  $+$  and  $-$  are left-associative, then make  $+ \cdot \rightarrow +$ ,  $+ \cdot \rightarrow -$ ,  $- \cdot \rightarrow -$ , and  $- \cdot \rightarrow +$ .

If  $\uparrow$  is right-associative, then make  $\uparrow \leftarrow \cdot \uparrow$ .

These relations ensure that  $E - E + E$  will have handle  $E - E$  selected and  $E \uparrow E \uparrow E$  will have the last  $E \uparrow E$  selected.

⑦ Make  $\theta < \cdot id, id \cdot > \theta, \theta < \cdot (, ( < \cdot \theta, ) \cdot > \theta, \theta \cdot > \$$ ,  
and  $\$ < \cdot \theta$  for all operators  $\theta$ .

Also, let

( $\equiv$ )       $\$ < \cdot ($        $\$ < \cdot id$   
 $( < \cdot ($        $id \cdot > \$$        $) \cdot > \$$   
 $( < \cdot id$        $id \cdot > )$        $) \cdot > )$

### ■ Operator Precedence Relation

Consider the string  $id + id * id$  for the grammar

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

Operator Precedence relations of the above string can be as given below.

	id	+	*	\$
id	—	$\cdot >$	$\cdot >$	$\cdot >$
+	$< \cdot$	$\cdot >$	$< \cdot$	$\cdot >$
*	$< \cdot$	$\cdot >$	$\cdot >$	$\cdot >$
\$	$< \cdot$	$< \cdot$	$< \cdot$	—

→  $id$  and  $id$  is not comparing.

→  $id$  is having highest precedence than any other operator so in the  $id$  row,  $id \cdot >$  any other.

→ In the 2nd row,  $+$  &  $+$  comparison  $+\cdot > +$  since  $+$  is left associative.



→ In the 3<sup>rd</sup> row \* & \* comparison \* > \* Since \* is left associative.

→ Since \$ is least precedence over any other operator \$ < any other operator &, \$-\$ is not comparing.

### EXAMPLE

Construct the operator precedence relation table for String  $id * (id \uparrow id) - id / id$  for the Grammar.

$$E \rightarrow E + E$$

$$E \rightarrow E - E$$

$$E \rightarrow E / E$$

$$E \rightarrow E * E$$

$$E \rightarrow E \uparrow E$$

$$E \rightarrow (E)$$

$$E \rightarrow -E$$

$$E \rightarrow id$$

↳  $\uparrow$  is of higher precedence and right associative.

↳ \* and / are next highest Precedence and left associative

↳ + and - are of lowest precedence and is left associative

	+	-	*	/	$\uparrow$	id	(	)	\$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	<	<	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
$\uparrow$	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>	-	-	>	>
(	<	<	<	<	<	<	<	=	-
)	>	>	>	>	>	-	-	>	>
\$	<	<	<	<	<	<	<	-	-

## ■ Operator Precedence Parsing Algorithm.

Input : An operator Precedence matrix.

Output : Parse tree.

Initially stack contain  $\$$  and the input buffer contains the string  $w\$$ .

- ① Set up  $ip$  to point to the first symbol of  $w\$$ .
- ② Repeat following steps.
- ③ If  $\$$  is in top of the stack and  $ip$  points to  $\$$  then
- ④ return  
else begin
- ⑤ let 'a' be the top most terminal symbol on the stack and let 'b' be the symbol pointed to by  $ip$ .
- ⑥ if  $a < b$  or  $a = b$  then begin
- ⑦ Push 'b' onto the stack.
- ⑧ advance  $ip$  to the next input symbol.  
end
- ⑨ else if  $a > b$  then // Reduce
- ⑩ repeat
- ⑪ pop the stack
- ⑫ Until the top stack terminal is ~~reper~~ related by  $<$  to the terminal most recently popped
- ⑬ else error()

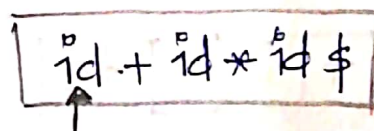
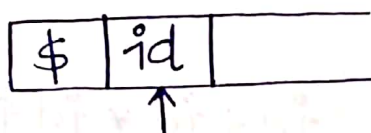
### EXAMPLE

Construct the Parse tree for the string  $id + id * id \$$  using Stack. The stack contain  $\$$  at the bottom as the relation table is as shown below.

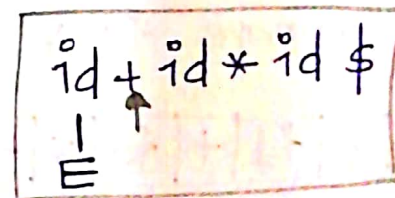
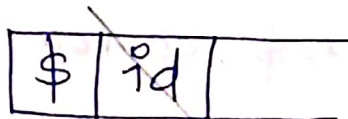
	id	+	*	\$
id	—	$\cdot >$	$\cdot >$	$\cdot >$
+	$< \cdot$	$\cdot >$	$< \cdot$	$\cdot >$
*	$< \cdot$	$\cdot >$	$\cdot >$	$\cdot >$
\$	$< \cdot$	$< \cdot$	$< \cdot$	—

→ compare  $\$$  Vs id. If the input  $(id) + id * id$  in the buffer is having higher Precedence, then push on to the stack otherwise pop it off.

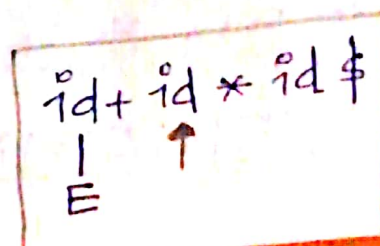
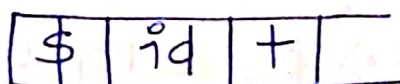
Since  $id \cdot > \$$ , push id to stack and reduce it to LHS if it is popped.



→ compare id Vs +       $id \cdot > +$ , so popped and reduce



→ compare \$ Vs +       $+ \cdot > \$$  so push +





→ compare + Vs id ,  $+ < \cdot id$  , so push id

\$	id	+	id
----	----	---	----

id	+	id	*	id	\$
			↑		
E					

→ compare id Vs \* ,  $id \cdot > *$  , so Pop and reduce to E

\$	id	+	id
----	----	---	----

id	+	id	*	id	\$
			↑		
E		E			

→ compare + Vs \* ,  $+ < \cdot *$  , so push

\$	id	+	id	*
----	----	---	----	---

id	+	id	*	id	\$
			↑		
E		E			

→ compare \* Vs <sup>id</sup> ,  $* < \cdot id$  , so push id

\$	id	+	id	*	id
----	----	---	----	---	----

id	+	id	*	id	\$
			↑		
E		E			

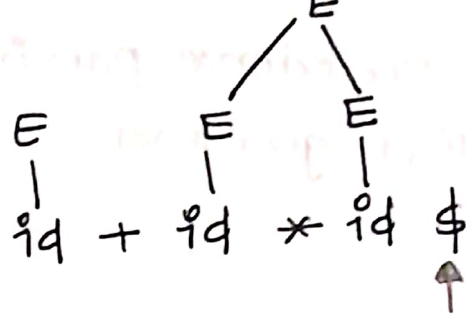
→ compare id Vs \$ ,  $id \cdot > \$$  , so Pop and reduce

\$	id	+	id	*	id
----	----	---	----	---	----

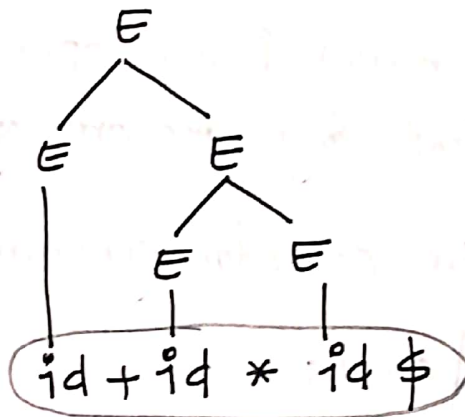
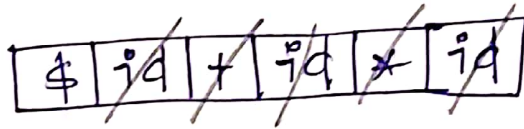
id	+	id	*	id	\$
				↑	
E		E	E		

→ compare \* Vs \$ ,  $* \cdot > \$$  , so Pop & reduce

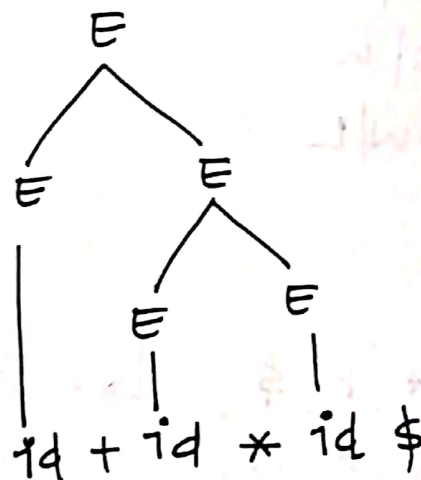
\$	id	+	id	*	id
----	----	---	----	---	----



→ compare + Vs \$ ,  $+ \rightarrow \$$ , so pop & reduce.



→ Compare \$ Vs \$, Accepted.  
So final Parse tree is



### EXERCISE

Consider the example for an input string  $id + id + id \$$  for the given grammar  $E \rightarrow E + E \mid E * E \mid id$ . Construct the parse tree.

### EXAMPLE

Construct operator precedence parsing table for the following given grammar

$$P \rightarrow SR | s$$

$$R \rightarrow bSR | bs$$

$$S \rightarrow Wbs | w$$

$$W \rightarrow L * W | L$$

$$L \rightarrow id$$

The given grammar is not operator precedence since nonterminal symbols are adjacent.

Following are the operator grammar:

$$P \rightarrow sbSR | sbs | s.$$

Again this is not operator grammar

$$P \rightarrow sbp | sbs | s$$

$$S \rightarrow wbs | w$$

$$W \rightarrow L * W | L$$

$$L \rightarrow id.$$

	id	*	b	\$
id	-	>	>	>
*	<	<	>	>
b	<	<	<	>
\$	<	<	<	-

↳ \* is right associative  
since \* is right recursive.

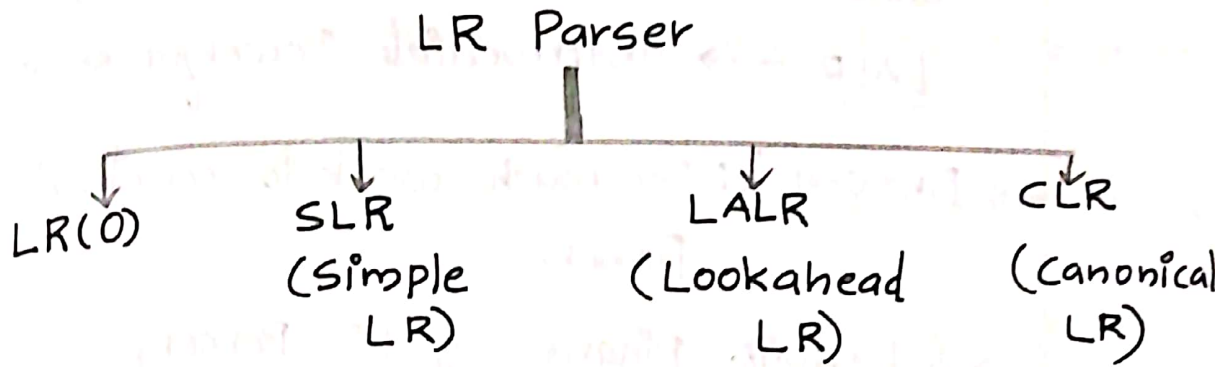
$$W \rightarrow L * W$$

↳ b is right associative  
since b is right recursive

$$\underline{S} \rightarrow wbs \quad \underline{P} \rightarrow sb\underline{P}$$



# LR PARSER



→ LR Parser is an efficient, bottom up syntax analysis technique that can be used to parse a large class of CFG.

→ LR(k)

- Left to Right Input scanning.
- constructing Right most derivation in reverse.
- No. of input symbols of lookahead use for parsing.

→ When (k) is omitted, k is assumed to be 1.

→ LR Parser is the most general form of nonbacktracking shift reduce parser.

→ Grammar parsed with LR method is the superset of the class of grammar which is being parsed with Predictive ~~Parse~~ Parsers.

→ LR Parser can detect more syntax errors.

→ LR Parser is more powerful than LL Parser.

→ Out of 4 LR Parser, CLR is highly powerful and expensive.

→ SLR → easy to implement, Least Powerful  
CLR → Most Powerful, Most expensive.  
LALR → Intermediate Powerful & cost.

→ Drawback: Too much work to construct LR Parser.

→ Schematic Diagram of LR Parser:

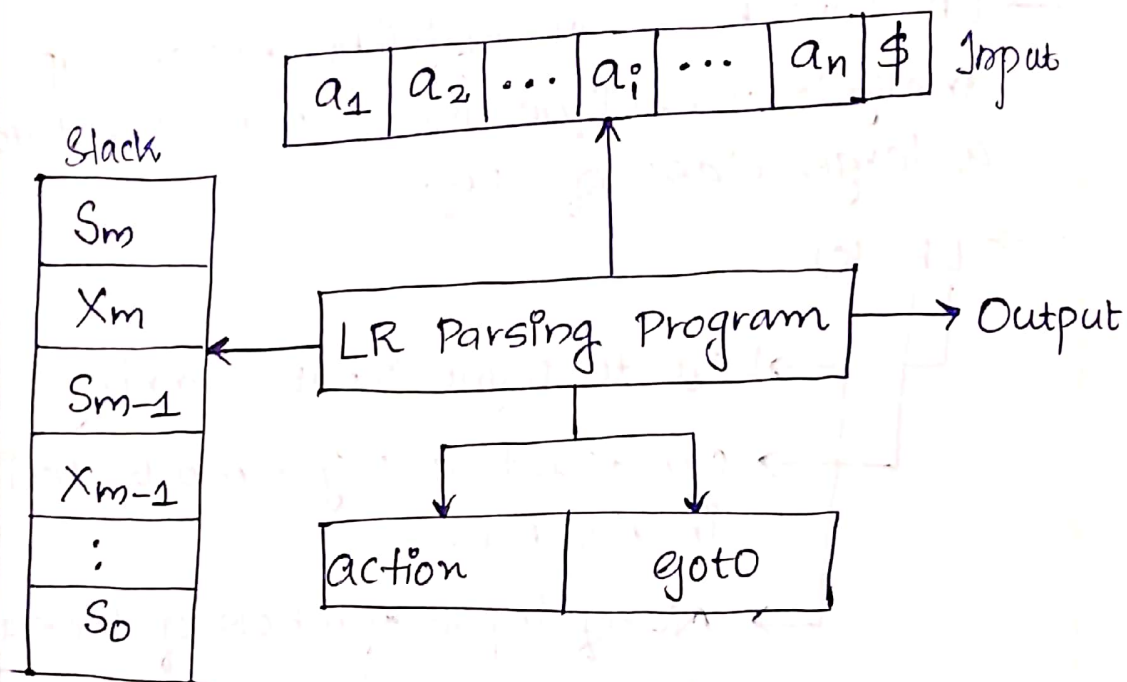


Figure: Model of LR Parser

→ The LR parser consists of

- ① an input
- ② an output
- ③ a stack
- ④ a program
- ⑤ a parsing table consist of 2 parts
  - (a) action
  - (b) goto

- The Program is the same for all LR Parsers, only the parsing table changes from one parser to another.
- The Parsing Program reads character from an input buffer once at a time.
- The program uses a stack to store a string of the form  $S_0 X_1 S_1 X_2 S_2 \dots X_m S_m$ , where  $S_m$  is on top of the stack.
- Each  $X_i$  is a grammar symbol and each  $S_i$  is a symbol called a state.
- Each <sup>state</sup> ~~Symbol~~ table
- Each state symbol summarizes the information contained in the stack below it, and the combination of the ~~stack~~ state symbol on top of the stack and the current input symbols are used to index the parsing table and determine the shift reduce parsing decision.
- The parsing table consists of 2 parts, a parsing action function action and a goto function goto.
- The Program driving the LR Parser behaves as follows:
  - It determines  $S_m$ , the state currently on top of the stack, and
  - $a_i$  the current input symbol.
- It then consults  $\text{action}[S_m, a_i]$ , the parsing action table entry for state ' $S_m$ ' and input ' $a_i$ ', which can have one of 4 values:



- ① shift  $s$ , where  $s$  is a state
- ② reduce by a grammar production  $A \rightarrow B$
- ③ accept, and
- ④ error.

### ■ LR Parsing algorithm

Input : An input string ' $w$ ' and an LR parsing table with functions action and goto for a grammar  $G$ .

Output : If ' $w$ ' is in  $L(G)$ , a bottom-up parse for  $w$ ; otherwise, an error indication.

Method : Initially, the Parser has ' $s_0$ ' on its stack, where ' $s_0$ ' is the initial state, and ' $w\$$ ' in the i/p buffer.

Set  $ip$  to point to the first symbol of  $w\$$ .

repeat forever begin

let ' $s$ ' be the state on top of the stack and ' $a$ ' the symbol pointed to by  $ip$ ;

if  $\text{action}[s, a] = \text{shift } s'$  then begin

Push ' $a$ ' then ' $s'$ ' on top of the stack;

advance  $ip$  to the next input symbol

end

else if  $\text{action}[s, a] = \text{reduce } A \rightarrow B$  then begin

Pop  $2 \times |B|$  symbols off the stack;

let ' $s$ ' be the state now on top of the stack;

Push  $A$  then goto  $[s', A]$  on top of the stack;

Output the production  $A \rightarrow B$

end

else if  $\text{action}[s, a] = \text{accept}$  then

return  
else error()  
end

## ■ Construction of SLR Parsing Tables

### LR(0) Item:

→ An LR(0) item of a grammar  $G$  is a production of  $G$  with a dot at some position of the right side.

→ Thus, production  $A \rightarrow XYZ$  yields the 4 item.

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$

→ The production  $A \rightarrow \epsilon$  generates only one item,  $A \rightarrow \cdot$ .

### Augmented Grammar:

→ If  $G$  is a grammar with start symbol  $S$ , then  $G'$  is the augmented grammar for  $G$ , is  $G$  with a new start symbol  $S'$  and production  $S' \rightarrow S$ .

→ The purpose of this new starting production is to indicate the parser when it should stop parsing and announce acceptance of the input. i.e., acceptance occurs when and only when the parser is about to reduce by  $S' \rightarrow S$ .

## The closure Operation:

→ If  $I$  is a set of items for a grammar  $G$ , then  $\text{closure}(I)$  is the set of items constructed from  $I$  by the two rules:

- ① Initially, every item in  $I$  is added to  $\text{closure}(I)$ .
- ② If  $A \rightarrow \alpha \cdot B \beta$  is in  $\text{closure}(I)$  and  $B \rightarrow \gamma$  is a production, then add the item  $B \rightarrow \cdot \gamma$  to  $I$ , if it is not already there.

We apply this rule until no more new items can be added to  $\text{closure}(I)$ .

closure computation:

function  $\text{closure}(I)$ ;

begin

$J := I$ ;

    repeat

        for each item  $A \rightarrow \alpha \cdot B \beta$  in  $J$  and each

        production  $B \rightarrow \gamma$  of  $G$  such that

$B \rightarrow \cdot \gamma$  is not in  $J$  do

            add  $B \rightarrow \cdot \gamma$  to  $J$

    until no more items can be added to  $J$ ;

    return  $J$

end



### EXAMPLE

consider the augmented expression grammar:

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

→ If  $I$  is the set of one item  $\{[E' \rightarrow \cdot E]\}$ , then closure( $I$ ) contains the items

$$E' \rightarrow \cdot E$$

$$E \rightarrow \cdot E + T$$

$$E \rightarrow \cdot T$$

$$T \rightarrow \cdot T * F$$

$$T \rightarrow \cdot F$$

$$F \rightarrow \cdot (E)$$

$$F \rightarrow \cdot id$$

### The Goto Operation:

→ The 2nd useful function is  $goto(I, X)$  where  $I$  is the set of items and  $X$  is the grammar symbol.

→  $goto(I, X)$  is defined to be the closure of the set of all items  $[A \rightarrow \alpha X \cdot \beta]$  such that  $[A \rightarrow \alpha \cdot X \beta]$  is in  $I$ .

### EXAMPLE

If  $I$  is the set of two items  $\{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$ , then  $goto(I, +)$  consist of

$$\rightarrow E \rightarrow E + \cdot T$$

$$T \rightarrow \cdot T * F$$

$$T \rightarrow \cdot F$$

$$F \rightarrow \cdot (E)$$

$$F \rightarrow \cdot id$$

→ We computed  $\text{goto}(I, +)$  by examining  $I$  for items with  $+$  immediately to the right of the dot.

→  $E' \rightarrow E \cdot$  is not such an item, but  $E \rightarrow E \cdot + T$  is. We moved the dot over the  $+$  to get

$\{E \rightarrow E + \cdot T\}$  and then took the closure of this set.

### EXAMPLE 2

Let  $I = \{E' \rightarrow \cdot E, E \rightarrow \cdot E + T, E \rightarrow \cdot T, T \rightarrow \cdot T * F, T \rightarrow \cdot F, F \rightarrow \cdot (E), F \rightarrow \cdot id\}$

→  $\text{goto}(I, E) = \{E' \rightarrow E \cdot, E \rightarrow E \cdot + T\}$

$\text{goto}(I, T) = \{E \rightarrow T \cdot, T \rightarrow T \cdot * F\}$

$\text{goto}(I, F) = \{T \rightarrow F \cdot\}$

$\text{goto}(I, id) = \{F \rightarrow id \cdot\}$

$\text{goto}(I, ( ) = \{F \rightarrow ( \cdot E)\}$

### The Sets-of-Items Construction:

→ The algorithm to construct the Canonical Collection of Sets of LR(0) items for an augmented grammar  $G'$ :

Procedure  $\text{items}(G')$ ;

begin

$C := \{ \text{closure}(\{[S' \rightarrow \cdot S]\}) \};$

repeat

for each set of items  $I$  in  $C$  and each grammar symbol  $X$  such that  $\text{goto}(I, X)$  is not empty

and not in C do

add goto(I, X) to C

until no more sets of items can be added to C  
end

→ The 1<sup>st</sup> set in C is the closure of  $\{[s' \rightarrow \cdot s]\}$  where s is the start symbol of original grammar and s' is the starting non-terminal of augmented grammar.

→ For each set I in C and each grammar symbol X where goto(I, X) is nonempty and not in C add the set goto(I, X) to C.

### EXAMPLE

Consider the augmented grammar

$E' \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

→  $I_0: E' \rightarrow \cdot E$   
 $E \rightarrow \cdot E + T$   
 $E \rightarrow \cdot T$   
 $T \rightarrow \cdot T * F$   
 $T \rightarrow \cdot F$   
 $F \rightarrow \cdot (E)$   
 $F \rightarrow \cdot id$

$I_1: (I_0, E) : E' \rightarrow E.$   
 $E \rightarrow E \cdot + T$

$I_2: (I_0, T) : E \rightarrow T.$   
 $T \rightarrow T \cdot * F$

$I_3: (I_0, F) : T \rightarrow F.$



$$I_4: (\underline{I_0}, C) : F \rightarrow ( \cdot E)$$

$$E \rightarrow \cdot E + T$$

$$E \rightarrow \cdot T$$

$$T \rightarrow \cdot T * F$$

$$T \rightarrow \cdot F$$

$$F \rightarrow \cdot (E)$$

$$F \rightarrow \cdot id$$

$$I_5: (\underline{I_0}, id) :$$

$$F \rightarrow id \cdot$$

$$I_6: (\underline{I_1}, +) : E \rightarrow E + \cdot T$$

$$T \rightarrow \cdot T * F$$

$$T \rightarrow \cdot F$$

$$F \rightarrow \cdot (E)$$

$$F \rightarrow \cdot id$$

$$I_7: (\underline{I_2}, *) :$$

$$T \rightarrow T * \cdot F$$

$$F \rightarrow \cdot (E)$$

$$F \rightarrow \cdot id$$

$$I_8: (\underline{I_4}, E) : F \rightarrow (E \cdot)$$

$$E \rightarrow \cdot E + T$$

$$I_{10}: (\underline{I_7}, F) :$$

$$T \rightarrow T * F \cdot$$

$$I_9: (\underline{I_6}, T) : E \rightarrow E + T \cdot$$

$$T \rightarrow T \cdot * F$$

$$I_{11}: (\underline{I_8}, ) :$$

$$F \rightarrow (E) \cdot$$

Transition Diagram of DFA for the set of LR(0)  
Items Canonical Collections:

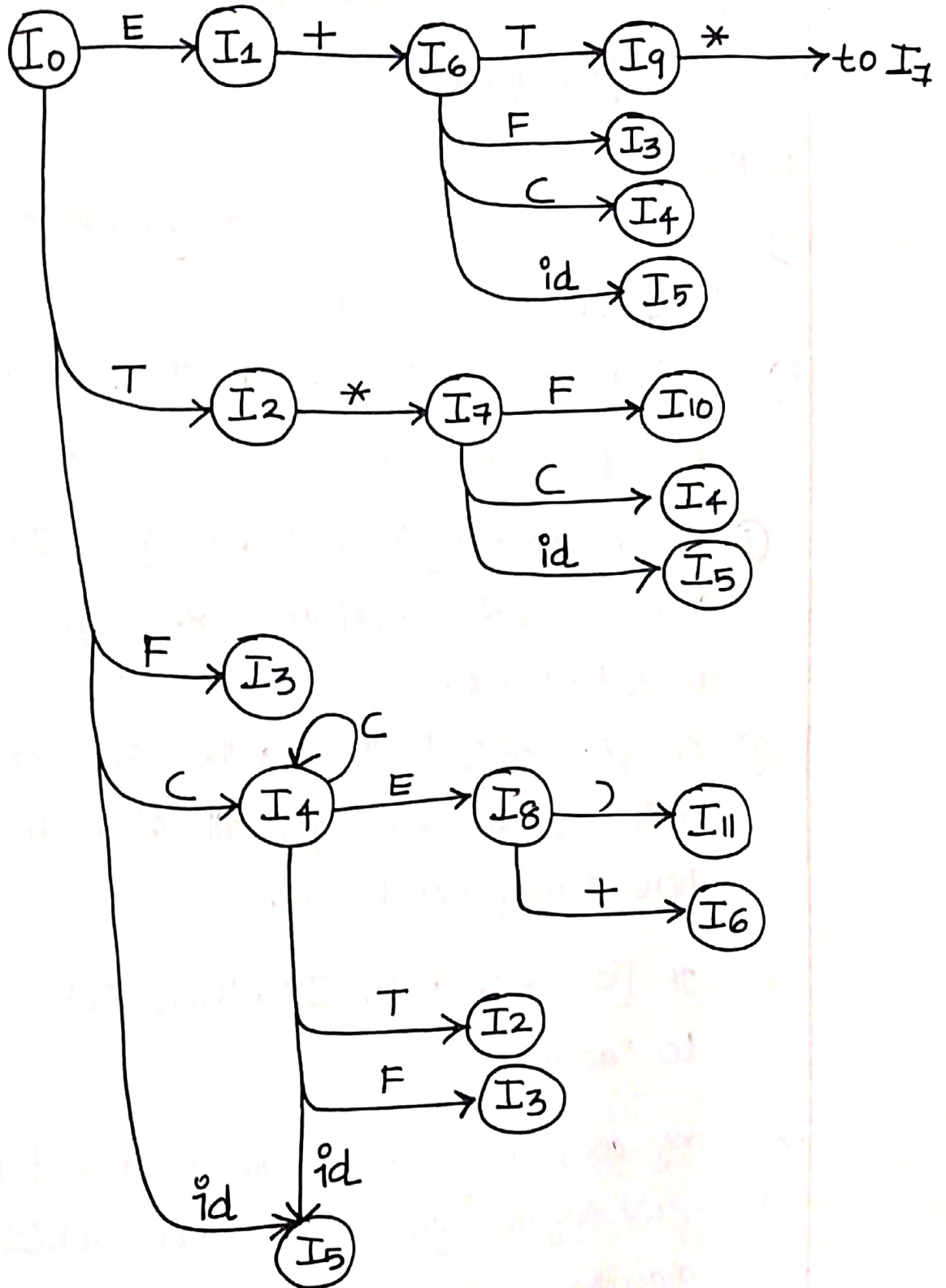


Fig:—Transition Diagram of DFA for the set of LR(0)

## Construction of SLR Parsing Table:

Input : An augmented grammar  $G'$

Output : The SLR parsing table functions action and goto for  $G'$

Method:

- ① Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of sets of LR(0) items for  $G'$
- ② State  $i$  is constructed from  $I_i$ . The parsing actions for state  $i$  are determined as follows:
  - Ⓐ If  $[A \rightarrow \alpha \cdot a \beta]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set  $\text{action}[i, a]$  to "shift  $j$ ." Here  $a$  must be a terminal.
  - Ⓑ If  $[A \rightarrow \alpha \cdot]$  is in  $I_i$ , then set  $\text{action}[i, a]$  to "reduce  $A \rightarrow \alpha$ " for all ' $a$ ' in  $\text{Follow}(A)$ ; here  $A$  may not be  $S'$ .
  - Ⓒ If  $[S' \rightarrow S \cdot]$  is in  $I_i$ , then set  $\text{action}[i, \$]$  to "accept".

→ If any conflicting actions are generated by the above rules, we say the grammar is not SLR(1).

→ The algorithm fails to produce a parser in this case.

- ③ The goto transition for state  $i$  are constructed for all nonterminals  $A$  using the rule:  
if  $\text{goto}(I_i, A) = I_j$ , then  $\text{goto}[i, A] = j$ .



- ④ All entries not defined by rules (2) and (3) are made "error".
- ⑤ The initial state of the parser is the one constructed from the set of items containing  $[s' \rightarrow \cdot s]$ .
- The Parsing table consisting of the parsing action and goto functions determined by this algorithm is called the SLR(1) table for  $G$ .

### EXAMPLE

Construct the SLR Parsing table for grammar given below.

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

→ Augmented Grammar:

$$E' \rightarrow E$$

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

The canonical collection of LR(0) items are:

$$\begin{aligned} I_0: & E' \rightarrow \cdot E \\ & E \rightarrow \cdot E + T \\ & E \rightarrow \cdot T \\ & T \rightarrow \cdot T * F \\ & T \rightarrow \cdot F \\ & F \rightarrow \cdot (E) \\ & F \rightarrow \cdot id \end{aligned}$$

$$I_1: (I_0, E) : E' \rightarrow E \cdot \\ E \rightarrow E \cdot + T$$

$$I_2: (I_0, T) : E \rightarrow T \cdot \\ T \rightarrow T \cdot * F$$

$$I_3: (I_0, F) : T \rightarrow F \cdot$$

$$\begin{aligned} I_4: (I_0, C) : & F \rightarrow ( \cdot E ) \\ & E \rightarrow \cdot E + T \\ & E \rightarrow \cdot T \\ & T \rightarrow \cdot T * F \\ & T \rightarrow \cdot F \\ & F \rightarrow \cdot (E) \\ & F \rightarrow \cdot id \end{aligned}$$

$$I_5: (I_0, id) : F \rightarrow id \cdot$$

$$\begin{aligned} I_6: (I_1, +) : & E \rightarrow E + \cdot T \\ & T \rightarrow \cdot T * F \\ & F \rightarrow \cdot F \\ & F \rightarrow \cdot (E) \\ & F \rightarrow \cdot id \end{aligned}$$

$$\begin{aligned} I_7: (I_2, *) : & T \rightarrow T * \cdot F \\ & F \rightarrow \cdot (E) \\ & F \rightarrow \cdot id \end{aligned}$$

$$I_9: (I_6, T) : E \rightarrow E + T \cdot \\ T \rightarrow T \cdot * F$$

$$\begin{aligned} I_8: (I_4, E) : & F \rightarrow (E \cdot) \\ & E \rightarrow E \cdot + T \end{aligned}$$

$$(I_6, F) = I_3 \quad (I_6, C) = I_4$$

$$(I_4, T) = I_2$$

$$(I_4, C) = I_4 \quad (I_6, id) = I_5$$

$$(I_4, F) = I_3$$

$$(I_4, id) = I_5$$

$$\begin{aligned} I_{10}: (I_7, F) : & T \rightarrow T * F \cdot \end{aligned}$$

$$(I_7, c) = I_4$$

$$(I_7, id) = I_5$$

$$(I_{11}): (I_8, >) : F \rightarrow (E). \quad (I_8, +) = I_6 \quad (I_9, *) = I_7$$

Parsing Table for expression grammar:

state	action						goto		
	id	+	*	(	)	\$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

$$\text{Follow}(E) = \{+, >, \$\}$$

$$\text{Follow}(T) = \{+, >, \$, *\}$$

$$\text{Follow}(F) = \{+, >, \$, *\}$$



- $S_i$  means shift and stack state  $i$ ,
- $r_j$  means reduce by production numbered  $j$ ,
- accept means accept.
- blank means error.

Moves of LR Parser on  $id * id + id$ :

Stack	Input	Action
(1) 0	$id * id + id \$$	shift $id$
(2) 0 $id$ 5	$* id + id \$$	reduce by $F \rightarrow id$
(3) 0 $F$ 3	$* id + id \$$	reduce by $T \rightarrow F$
(4) 0 $T$ 2	$* id + id \$$	shift $*$
(5) 0 $T$ 2 $*$ 7	$id + id \$$	shift $id$
(6) 0 $T$ 2 $*$ 7 $id$ 5	<del><math>*</math></del> $+ id \$$	reduce by $F \rightarrow id$
(7) 0 $T$ 2 $*$ 7 $F$ 10	$+ id \$$	reduce by $T \rightarrow F$
(8) 0 $T$ 2 <del><math>*</math></del> <del><math>*</math></del>	$+ id \$$	reduce by $E \rightarrow T$
(9) 0 $E$ 1	$+ id \$$	shift $+$
(10) 0 $E$ 1 $+$ 6	$* id \$$	shift $id$
(11) 0 $E$ 1 $+$ 6 $id$ 5	$\$$	reduce by $F \rightarrow id$
(12) 0 $E$ 1 $+$ 6 $F$ 3	$\$$	reduce by $T \rightarrow F$
(13) 0 $E$ 1 $+$ 6 $T$ 9	$\$$	$E \rightarrow E + T$
(14) 0 $E$ 1	$\$$	accept.

### EXAMPLE

construct the SLR Parsing table for the given grammar

$$S' \rightarrow AA$$

$$A \rightarrow aA/b$$

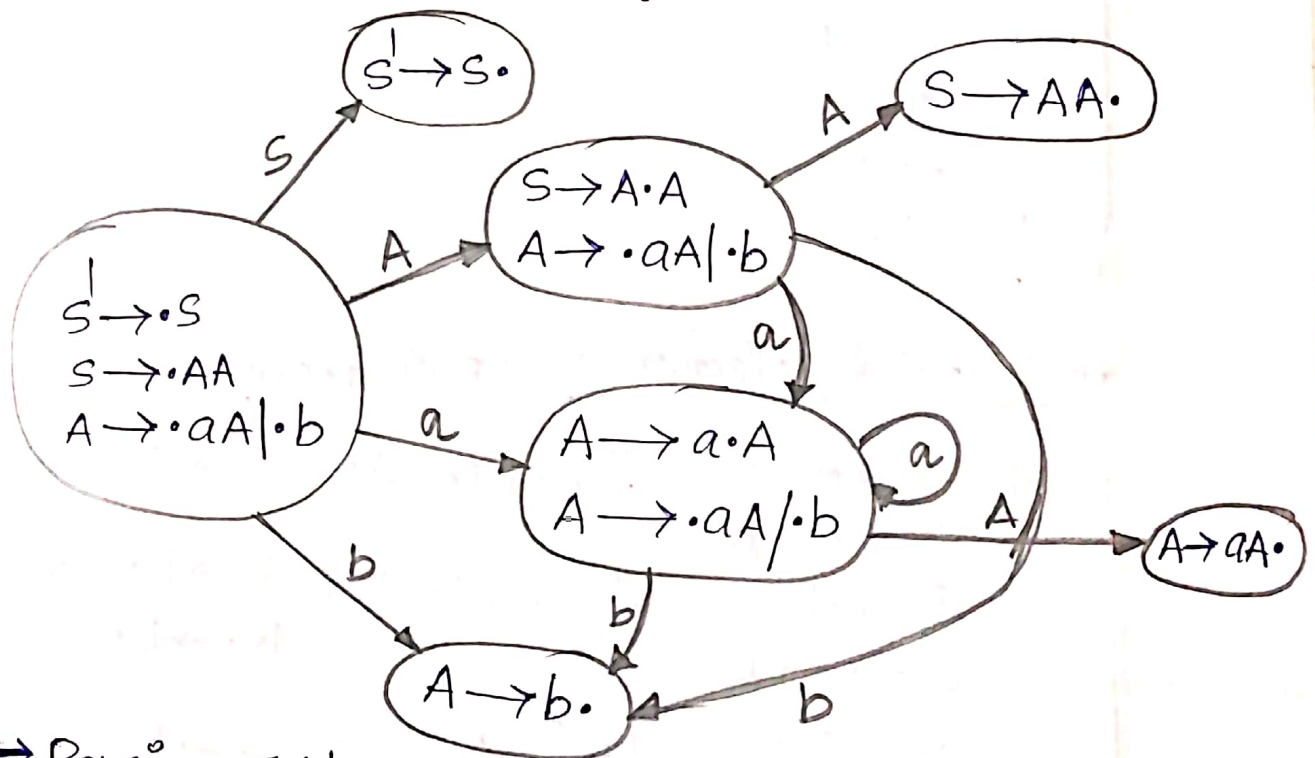
→ Augmented Grammar

$$S' \rightarrow S$$

$$S \rightarrow AA$$

$$A \rightarrow aA/b.$$

→ The canonical collection of LR(0) items



→ Parsing Table

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

## EXAMPLE

Construct SLR parsing table for the following grammar.

$$S \rightarrow L = R$$

$$S \rightarrow R$$

$$L \rightarrow *R$$

$$L \rightarrow id$$

$$R \rightarrow L$$

→ Augmented Grammar

$$S' \rightarrow S$$

$$S \rightarrow L = R$$

$$S \rightarrow R$$

$$L \rightarrow *R$$

$$L \rightarrow id$$

$$R \rightarrow L$$

→ Canonical collection of LR(0) items

$$\begin{aligned} I_0: & S' \rightarrow \cdot S \\ & S \rightarrow \cdot L = R \\ & S \rightarrow \cdot R \\ & L \rightarrow \cdot *R \\ & L \rightarrow \cdot id \\ & R \rightarrow \cdot L \end{aligned}$$

$$I_1: (I_0, S): S' \rightarrow S \cdot$$

$$I_2: (I_0, L): S \rightarrow L \cdot = R \\ R \rightarrow L \cdot$$

$$I_3: (I_0, R): S \rightarrow R \cdot$$

$$\begin{aligned} I_4: (I_0, *) &: L \rightarrow * \cdot R \\ & R \rightarrow \cdot L \\ & L \rightarrow \cdot *R \\ & L \rightarrow \cdot id \end{aligned}$$

$$\begin{aligned} I_5: (I_0, id) &: \\ & L \rightarrow id \cdot \end{aligned}$$

$$I_7: (I_4, R): L \rightarrow *R \cdot$$

$$I_6: (I_2, =):$$

$$\begin{aligned} & S \rightarrow L = \cdot R \\ & R \rightarrow \cdot L \\ & L \rightarrow \cdot *R \\ & L \rightarrow id \end{aligned}$$

$$I_8 = (I_6, L)$$

$$I_5 = (I_4, id)$$



$I_8: (I_4, L) : R \rightarrow L.$

$(I_4, *) = I_4$

$I_9: (I_6, R) : S \rightarrow L=R.$

$I_4 = (I_6, *)$

$I_5 = (I_6, id)$

→ Parsing Table

state	Action				Goto		
	=	*	id	\$	S	L	R
0		S4	S5		1	2	3
1				accept			
2	S6/r5						
3							
4		S4	S5			8	7
5							
6		S4	S5			8	9
7							
8							
9				r1			

→ The state 2 is having both shift and reduce. So shift reduce conflict arises.

→ In some cases there can be other conflicts also.

- ① Shift/Shift conflict.
- ② Shift/reduce conflict.
- ③ Reduce/Reduce conflict.

If any of the above conflict occur, the grammar cannot be parsed.

## EXAMPLE

check whether the following given grammar is in SLR(1) or not.

$S \rightarrow A$

$S \rightarrow a$

$A \rightarrow a$

→ Augmented Grammar

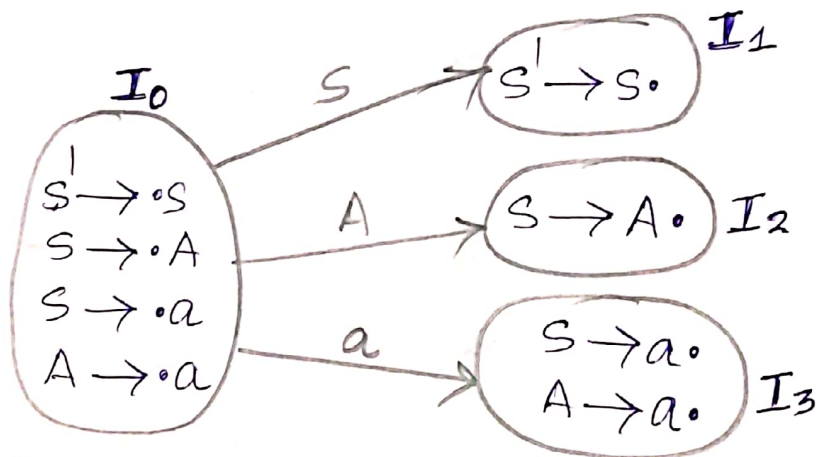
$S' \rightarrow S$

$S \rightarrow A$

$S \rightarrow a$

$A \rightarrow a$

→ Canonical collection of LR(0) items



→ Parsing Table

state	Action		Goto	
	a	\$	S	A
0	SB		1	2
1		accept		
2		$\gamma_1$		
3		$\gamma_2/\gamma_3$		

→ In state-3, we are having reduce-reduce conflict.  
so the given grammar is not in SLR(1).

**EXAMPLE** check whether the given grammar is SLR(1) or not?

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow i$

→ Augmented Grammar

$E' \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow i$

→ Canonical collection of LR(0) items

$I_0: E' \rightarrow \cdot E$   
 $E \rightarrow \cdot E+T$   
 $E \rightarrow \cdot T$   
 $T \rightarrow \cdot i$

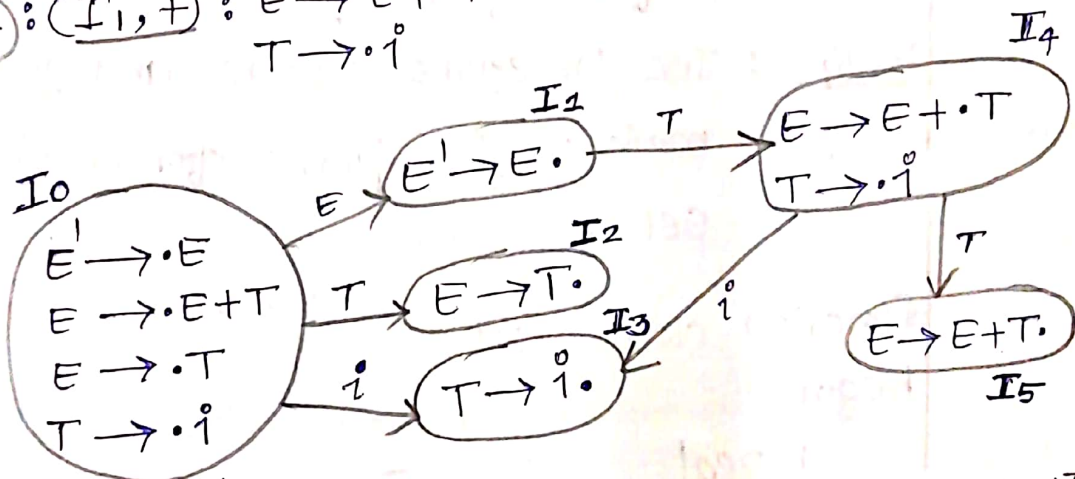
$I_1: (I_0, E): E' \rightarrow E \cdot$   
 $E \rightarrow E \cdot +T$

$I_2: (I_0, T): E \rightarrow T \cdot$

$I_3: (I_0, i): T \rightarrow i \cdot$

$I_4: (I_4, T): E \rightarrow E+T \cdot$

$I_4: (I_1, +): E \rightarrow E+ \cdot T$   
 $T \rightarrow \cdot i$   
 $I_3 = (I_4, i)$



→ Parsing Table

state	action			goto	
	+	i	\$	E	T
0		s3		1	2
1	s4		accept		
2	r2		r2		
3	r3		r3		5
4		s3			
5	r4		r1		

$\text{Follow}(E) = \{+, \$\}$   
 $\text{Follow}(T) = \{+, \$\}$

The given grammar is in SLR(1).



# ■ Canonical LR Parser

For the construction of CLR and LALR Parser, the items considered is LR(1) items.

LR(0) items are used for the construction of LR(0) and SLR Parser.

→ LR(0) items → items with dot at the RHS

→ LR(1) items → LR(0) items + Lookahead symbols.

## Construction of LR(1) items:

Input : An augmented grammar  $G'$ .

Output : The sets of LR(1) items that are the set of items valid for one or more viable prefixes of  $G'$ .

Method : The Procedure closure and goto and the main routine items for constructing the set of items

function closure( $I$ );

begin

repeat

for each item  $[A \rightarrow \alpha \cdot B \beta, a]$  in  $I$ ,

each production  $B \rightarrow \gamma'$  in  $G'$ ,

and each terminal  $b$  in  $\text{FIRST}(\beta a)$

Such that  $[B \rightarrow \cdot \gamma', b]$  is not in  $I$  do

add  $[B \rightarrow \cdot \gamma', b]$  to  $I$ ;

until no more items can be added to  $I$ ;

return I

end;

function goto(I, X);

begin

let J be the set of items  $[A \rightarrow \alpha X \cdot \beta, a]$  such that  
 $[A \rightarrow \alpha \cdot X \beta, a]$  is in I;

return closure(J)

end;

procedure items( $G'$ );

begin

$C := \{ \text{closure}(\{ [S' \rightarrow \cdot S, \$] \}) \};$

repeat

for each set of items I in C and each grammar  
symbol X such that goto(I, X) is not  
empty and not in C do

add goto(I, X) to C

until no more sets of items can be added to C.

end

### Construction of Canonical LR Parsing Table:

Input: An augmented grammar  $G'$ .

Output: Canonical LR Parsing table.

Method:

- ① Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of sets of LR(1) items for  $G'$ .

② State  $i$  of the Parser is constructed from  $I_i$ .

The Parsing actions for state  $i$  are determined as follows:

(a) IF  $[A \rightarrow \alpha \cdot a\beta, b]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set  $\text{action}[i, a]$  to "shift  $j$ ".

Here, 'a' is a terminal.

(b) IF  $[A \rightarrow \alpha \cdot, a]$  is in  $I_i$ ,  $A \neq S'$ , then set  $\text{action}[i, a]$  to "reduce  $A \rightarrow \alpha$ ".

(c) IF  $[S' \rightarrow S \cdot, \$]$  is in  $I_i$ , then  $\text{action}[i, \$]$  to "accept".

→ IF a conflict results from the above rules, the grammar is said not to be LR(1), and the algorithm is said to fail.

③ The goto transaction for state  $i$  are determined as follows: IF  $\text{goto}(I_i, A) = I_j$ , then  $\text{goto}[i, A] = j$ .

④ All entries are not defined by rule (2) and (3) are made as "error".

⑤ The initial state of the Parser is the one constructed from the set containing item  $[S' \rightarrow \cdot S, \$]$ .



## EXAMPLE

Construct CLR Parsing table for the given grammar

$$S \rightarrow CC$$

$$C \rightarrow cC$$

$$C \rightarrow d$$

→ Augmented Grammar

$$S' \rightarrow S$$

$$S \rightarrow CC$$

$$C \rightarrow cC$$

$$C \rightarrow d$$

→

$$\begin{aligned} I_0: S' &\rightarrow \cdot S, \$ \\ S &\rightarrow \cdot CC, \$ \\ C &\rightarrow \cdot cC, c|d \\ C &\rightarrow \cdot d, c|d \end{aligned}$$

$$I_1: (I_0, S) : \{ S' \rightarrow S \cdot, \$ \}$$

$$I_2: (I_0, C) : \left\{ \begin{aligned} S &\rightarrow C \cdot C, \$ \\ C &\rightarrow \cdot cC, \$ \\ C &\rightarrow \cdot d, \$ \end{aligned} \right\}$$

$$I_3: (I_0, c) : \left\{ \begin{aligned} C &\rightarrow c \cdot C, c|d \\ C &\rightarrow \cdot cC, c|d \\ C &\rightarrow \cdot d, c|d \end{aligned} \right\}$$

$$I_4: (I_0, d) : \{ C \rightarrow d \cdot, c|d \}$$

$$I_5: (I_2, C) : \{ S \rightarrow CC \cdot, \$ \}$$

$$I_6: (I_2, c) : \left\{ \begin{aligned} C &\rightarrow c \cdot C, \$ \\ C &\rightarrow \cdot cC, \$ \\ C &\rightarrow \cdot d, \$ \end{aligned} \right\}$$

$I_7: (I_2, d): \{C \rightarrow d\cdot, \$\}$

$I_6: (I_6, c)$

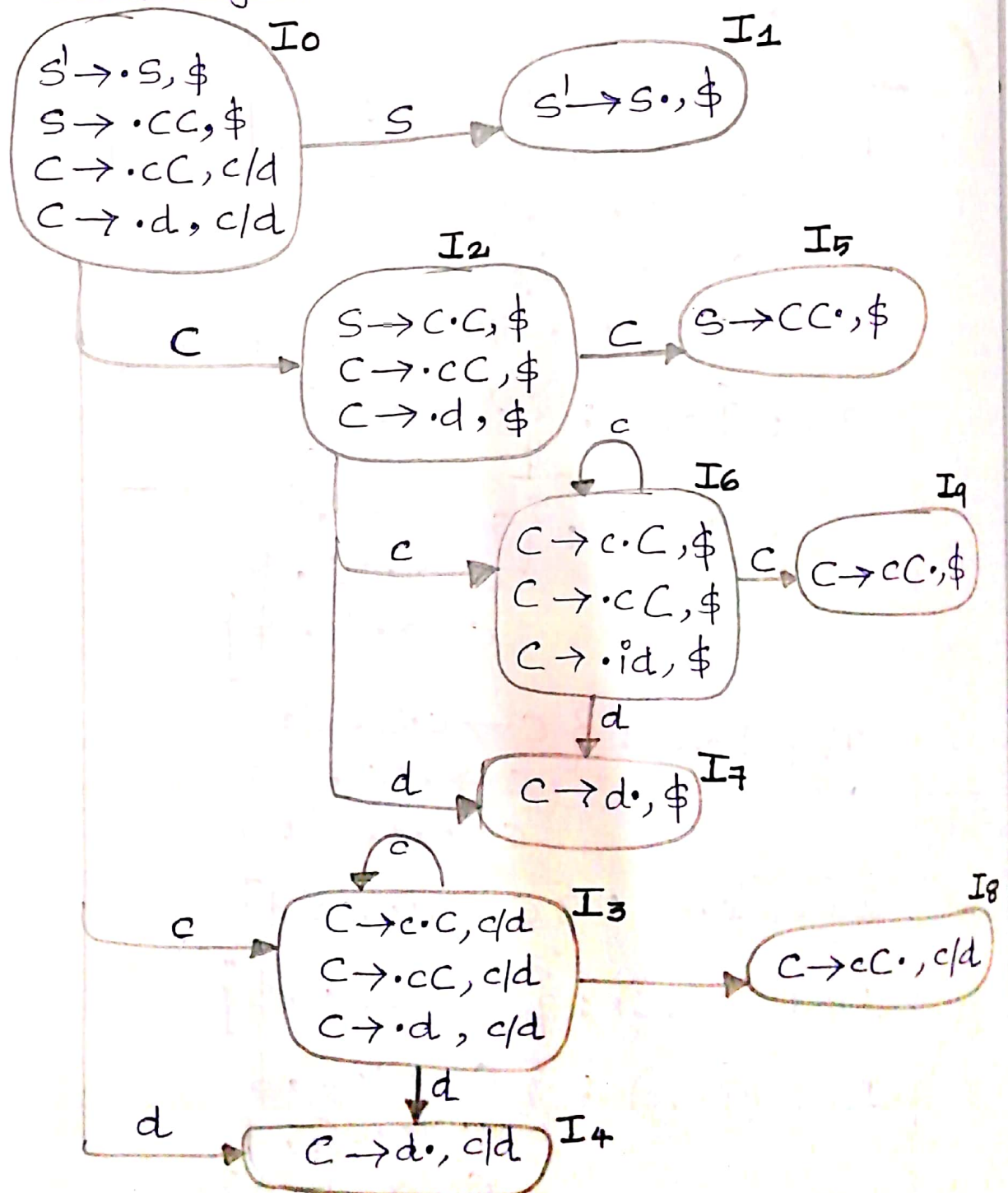
$I_8: (I_3, c): \{C \rightarrow cC\cdot, c/d\}$

$I_3: (I_3, c) \quad I_4: (I_3, d)$

$I_9: (I_6, c): \{C \rightarrow cC\cdot, \$\}$

$I_7: (I_6, d)$

→ The goto graph



## Parsing Table

state	Action			Goto	
	c	d	\$	S	C
0	S3	S4		1	2
1			accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

The 3rd Production  $c \rightarrow d$  is reduced with the lookahead symbol \$ and c/d.

So  $r_2$  is labelled across c, d & \$

### EXAMPLE

Construct CLR parsing table for the given grammar

$$S \rightarrow AA$$

$$A \rightarrow aA|b$$

→ Augmented Grammar

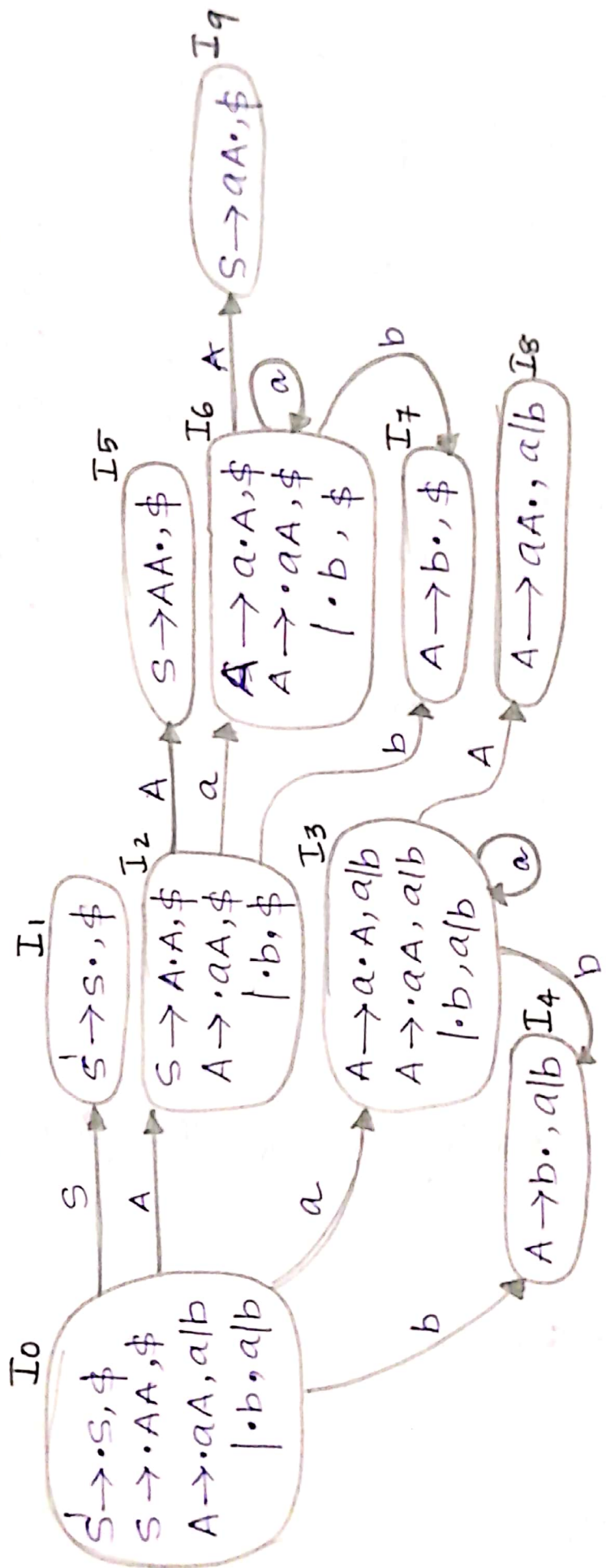
$$S' \rightarrow S$$

$$S \rightarrow AA$$

$$A \rightarrow aA|b$$



# goto graph



- In I3 and I6, both are having same LR(0) item but different lookahead.
- In I4 and I7, both are having same LR(0) items but different lookahead.
- In I8 and I9, " " " "

# CLR(1) Parsing Table

state	action			Goto	
	a	b	\$	S	A
0	S3	S4			2
1					
2	S6	S7			5
3	S3	S4			8
4	$\delta_3$	$\delta_3$			
5			$\delta_1$		
6	S6	S7			9
7		$\delta_3$			
8	$\delta_2$	$\delta_2$			
9		$\delta_2$			

$S \rightarrow A A$   $\delta_1$

$A \rightarrow a A$   $\delta_2$

$A \rightarrow b$   $\delta_3$

## ■ LALR Parsing

→ LALR → Lookahead - LR

→ This method is often used in practice because the tables obtained by it are considerably smaller than the canonical LR tables.

→ The SLR and LALR tables for a grammar always have the same no. of states.

### Construction of LALR Parsing Table

Input: An augmented Grammar.

Output: LALR Parsing Table.

Method:

- ① First obtain LR(1) items.
- ② calculate canonical collection of LR(1) items.
- ③ combine those canonicals which have same LR(0) but having different lookaheads.

### Construction of Action Table

- Ⓐ If  $(A \rightarrow \alpha \cdot a \beta, b)$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$  set action  $[I_i, a]$  to shift  $j(S_j)$ .
- Ⓑ If  $(A \rightarrow \alpha \cdot, b)$  is in  $I_i$  and  $\text{goto}(I_i, b) =$  reduce  $A \rightarrow \alpha$  ( $A \rightarrow \alpha$  in nonaugmented grammar).
- Ⓒ If  $[S' \rightarrow S \cdot, \$]$  is in  $I_i$  then set action  $[I_i, \$] =$  "accept".

### Construction of Goto Table

for every  $I_i$  in  $C$  do



For every non-terminal  $A$  do  
 if  $\text{goto}(I_i, A) = I_j$  then  
 set  $\text{goto}(I_i, A) = j$ .

### EXAMPLE

construct LALR Parsing table for the given grammar

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

→ Augmented Grammar

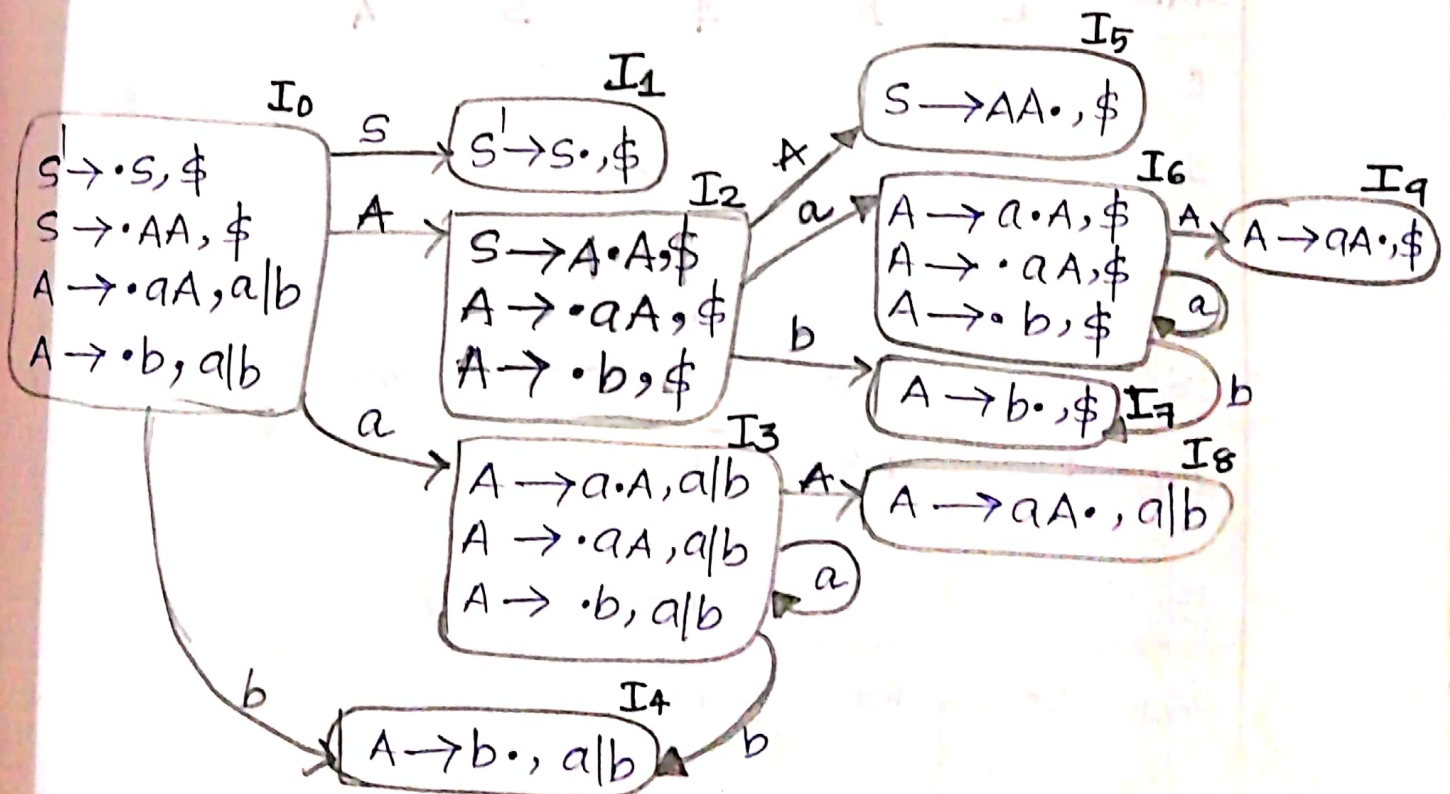
$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

→ Canonical collection of LR(1) items



→  $I_3$  and  $I_6$  are same in their LR(0) items but differ in their lookahead symbols.  
So, we can combine them and are called as  $I_{36}$ .

$$I_{36}: \left\{ \begin{array}{l} A \rightarrow a \cdot A, a|b|\$ \\ A \rightarrow \cdot aA, a|b|\$ \\ A \rightarrow \cdot b, a|b|\$ \end{array} \right\}$$

→  $I_4$  and  $I_7$  are same in their LR(0) items but differ in lookahead symbols.

$$I_{47}: \left\{ A \rightarrow b \cdot, a|b|\$ \right\}$$

→  $I_8$  and  $I_9$  becomes  $I_{89}$

$$I_{89}: \left\{ A \rightarrow aA \cdot, a|b|\$ \right\}$$

state	Action			Goto	
	a	b	\$	S	A
0	S36	S47		1	2
1			accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

## EXAMPLE

construct LALR Parsing table for the given grammar

$S \rightarrow AA$

$A \rightarrow aA \mid b$

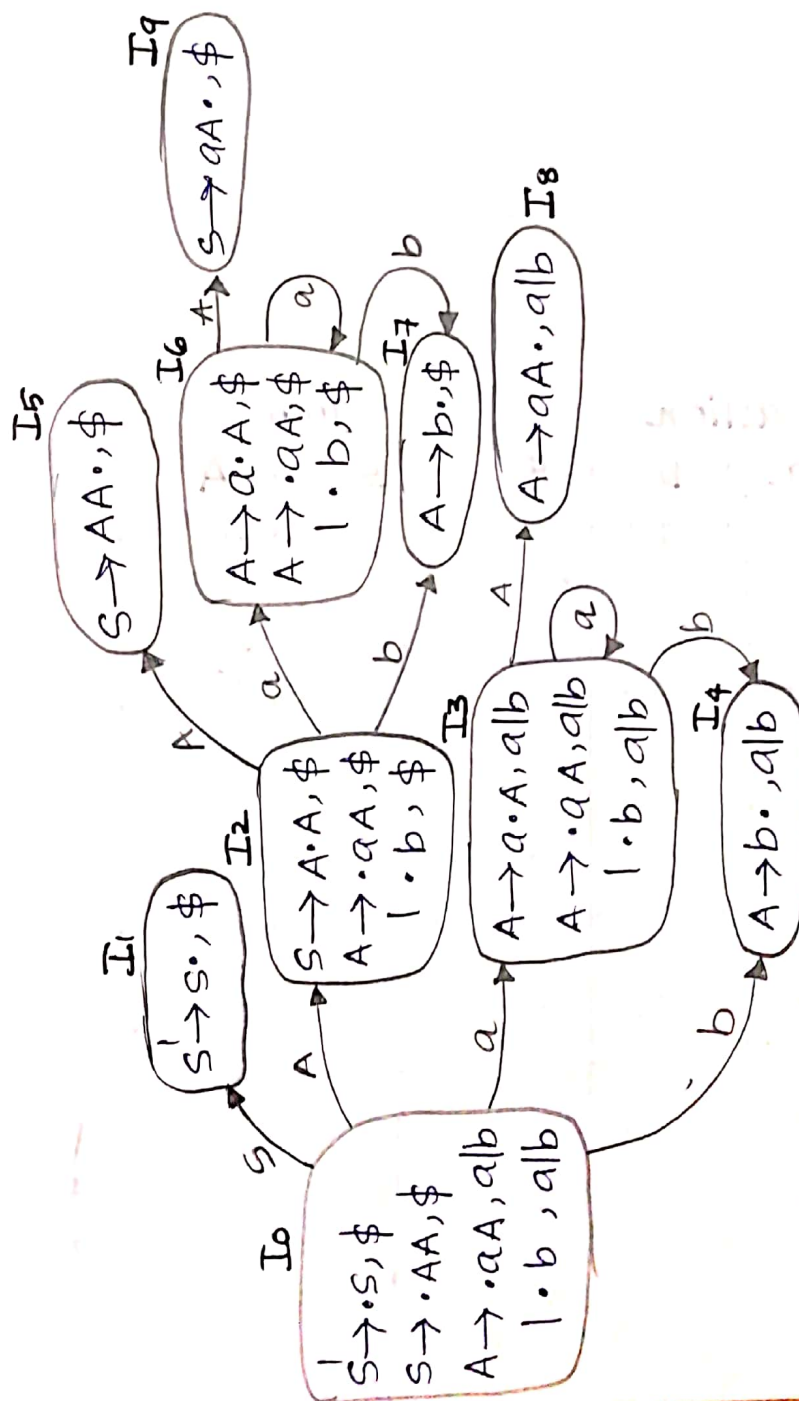
→ Augmented Grammar

$S' \rightarrow AS$

$S \rightarrow AA$

$A \rightarrow aA \mid b$

→ Canonical Collection of LR(1) Items



$I_3, I_6 \rightarrow I_{36}$   
 $I_4, I_7 \rightarrow I_{47}$   
 $I_8, I_9 \rightarrow I_{89}$

Same LR(0) items but different lookahead symbols.



# LALR Parsing Table

state	action			goto	
	a	b	\$	s	A
0	S36	S47			21
1					
2	S36	S47			5
36	S36	S47			89
47	r3	r3			
5			r1		
36	S36	S47			89
47			r3		
89	r2	r2			
89			r2		

Same

Rewrite the Table:

state	action			goto	
	a	b	\$	s	A
0	S36	S47			21
1					
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

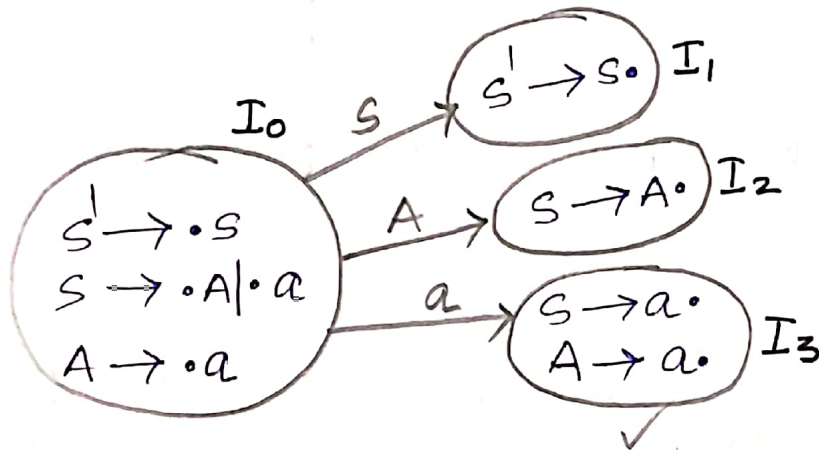
## PROBLEMS

- ① Check whether the grammar is suitable for LR(0) and SLR(1) parser.

$S \rightarrow A|a$

$A \rightarrow a$

→ Canonical collection of LR(0) items



→ Whenever we got 2 reduce states nodes in the same state, then the grammar is not in LR(0).

→  $S \rightarrow a \cdot$  place under  $\text{follow}(S) \rightarrow \$ \cdot$   
 $A \rightarrow a \cdot$  place under  $\text{follow}(S) \rightarrow \$ \cdot$  } In parsing table.

$\therefore$  the given grammar is not in SLR(1).

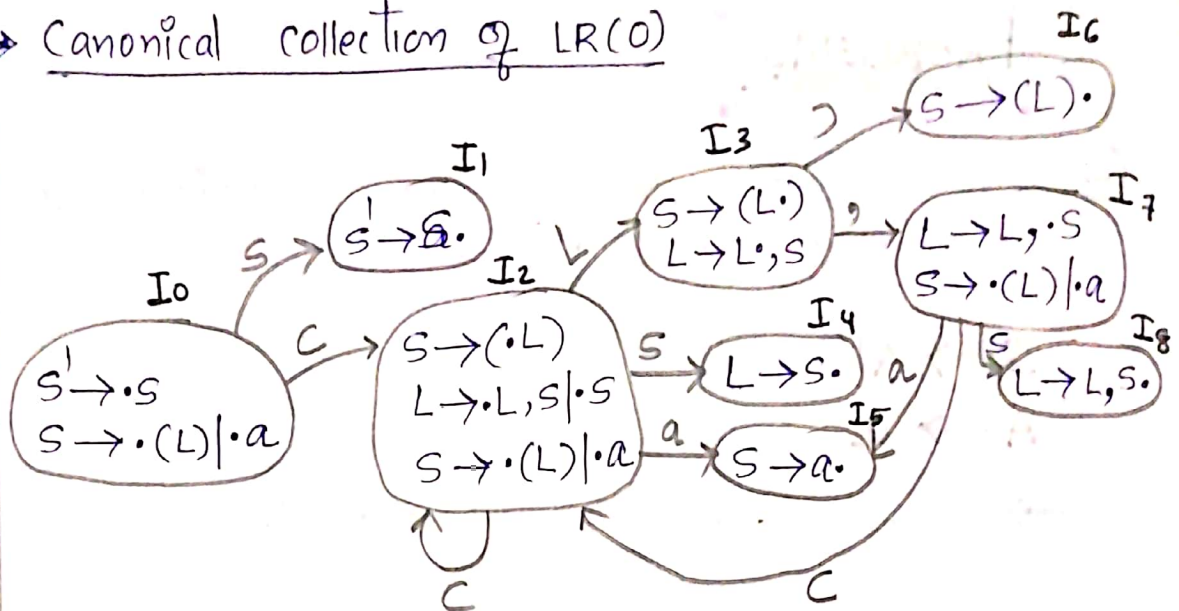
→ So this grammar is not parsed under any of the parser. Therefore the grammar is ambiguous.

- ② Check whether the given grammar is suitable for LR(0) and SLR(1) parser.

$S \rightarrow (L)|a$

$L \rightarrow L, S | S$

→ Canonical collection of LR(0)



→ No conflicts. So it is in LR(0) and SLR(1).

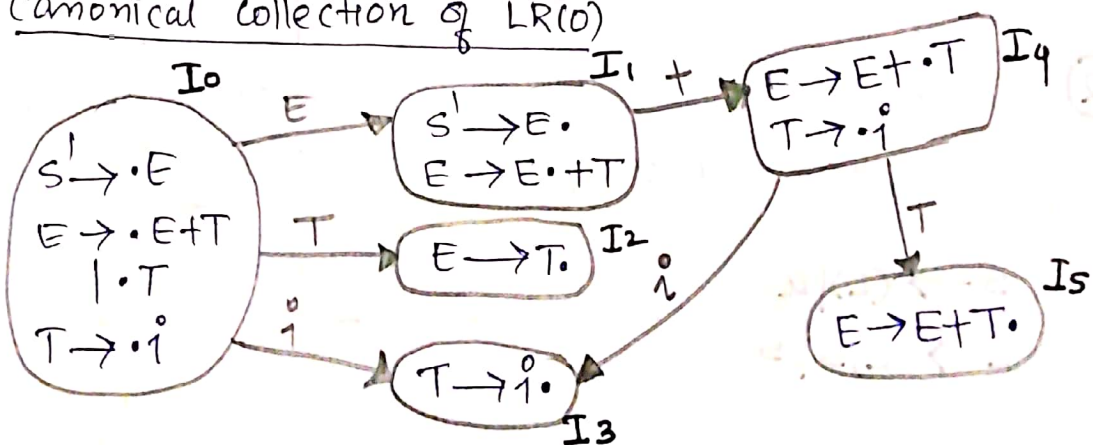
③ check whether the grammar is suitable for LR(0) and SLR(1)?

$E \rightarrow E + T \mid T$   
 $T \rightarrow i$

→ Augmented Grammar

$S' \rightarrow \cdot E$   
 $E \rightarrow \cdot E + T \mid \cdot T$   
 $T \rightarrow \cdot i$

Canonical collection of LR(0)



→ No conflicts. So the grammar is in LR(0) and SLR(1).



④ construct an SLR Parsing table for the following grammar.

$A \rightarrow aAa$

$A \rightarrow bAb$

$A \rightarrow ba$

→ Augmented Grammar

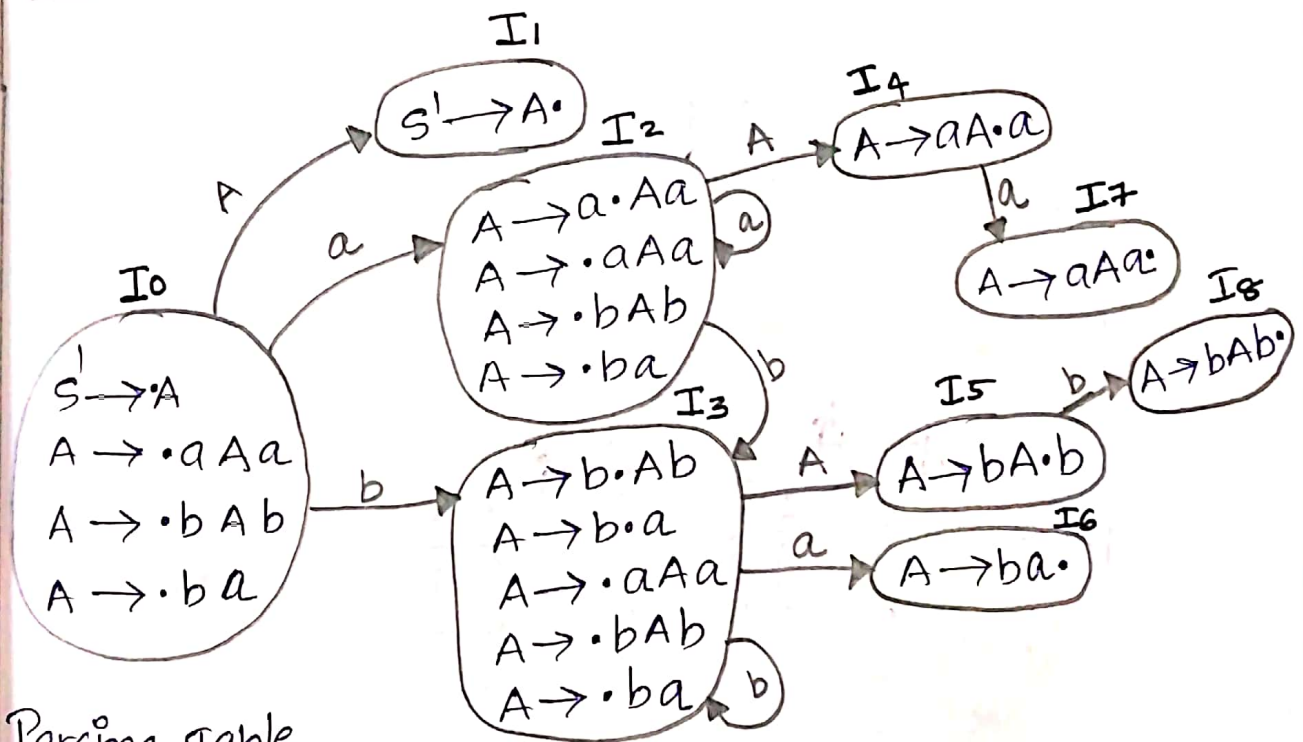
$S' \rightarrow A$

$A \rightarrow aAa$

$A \rightarrow bAb$

$A \rightarrow ba$

Canonical collection of LR(0) items:



Parsing Table

state	action		goto	
	a	b		
0	S2	S3	A	1
1				accept
2	S2	S3	A	4
3	S6	S3		5
4	S7			
5		S8		
6	S2/R3	S3/R3	A	4
7	R1	R1		
8	R2	R2		

Follow( $S'$ )  $\rightarrow \{ \$ \}$   
 Follow( $A$ )  $\rightarrow \{ a, b, \$ \}$

5) Construct the SLR Parsing Table for the following grammar.

$S \rightarrow xAy$

$S \rightarrow xBy$

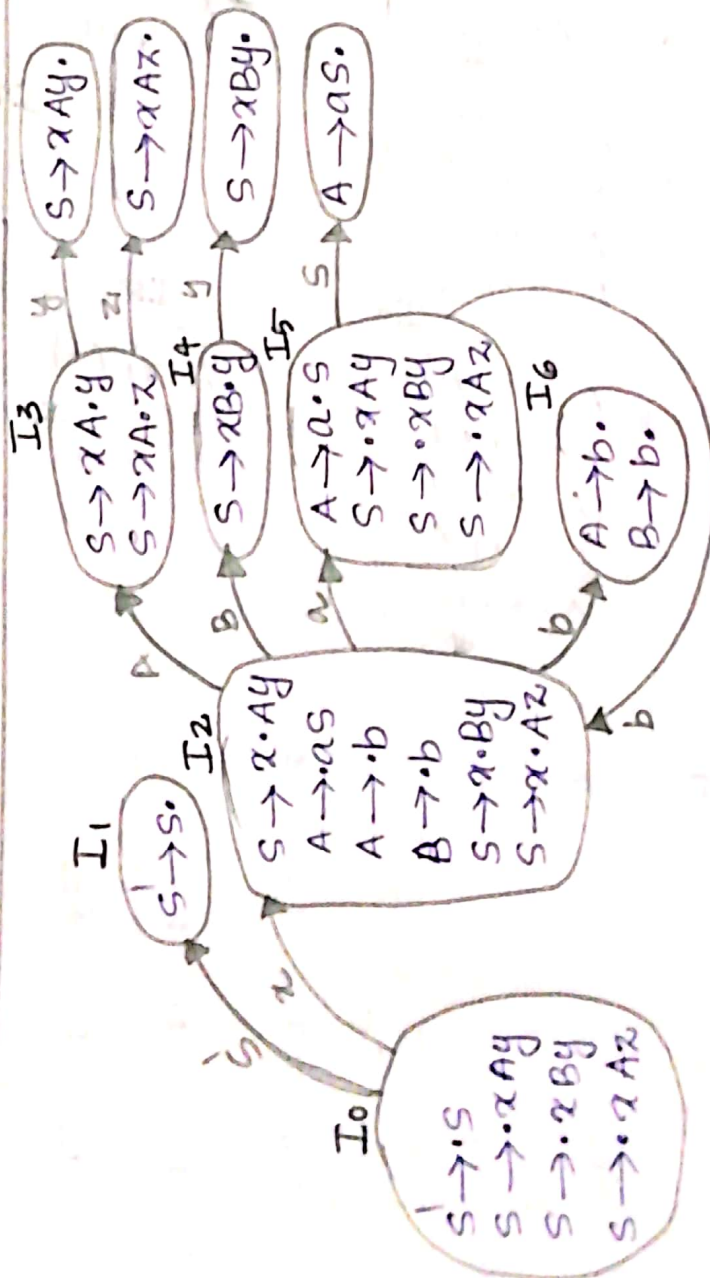
$S \rightarrow xAz$

$A \rightarrow aS$

$A \rightarrow b$

$B \rightarrow b$

→ Canonical collection of LR(0) items:



$Follow(S) = \{x, y, z\}$   
 $Follow(A) = \{y, z\}$   
 $Follow(B) = \{y\}$

state	action						goto		
	x	y	z	a	b	\$	S	A	B
0	S2						1		
1						accept			
2				S5	S6			3	4
3		S7	S8						
4		S9							
5	S2						10		
6		r5/r6	r5						
7		r1	r1						
8		r3	r3						
9		r2	r2						
10		r4	r4						

⑥ Construct an LALR(1) parsing table for the following grammar.

$S \rightarrow Ba$

$S \rightarrow bBc$

$S \rightarrow dc$

$S \rightarrow bda$

$B \rightarrow d$

→ Augmented Grammar

$S' \rightarrow S$

$S \rightarrow Ba$

$S \rightarrow bBc$

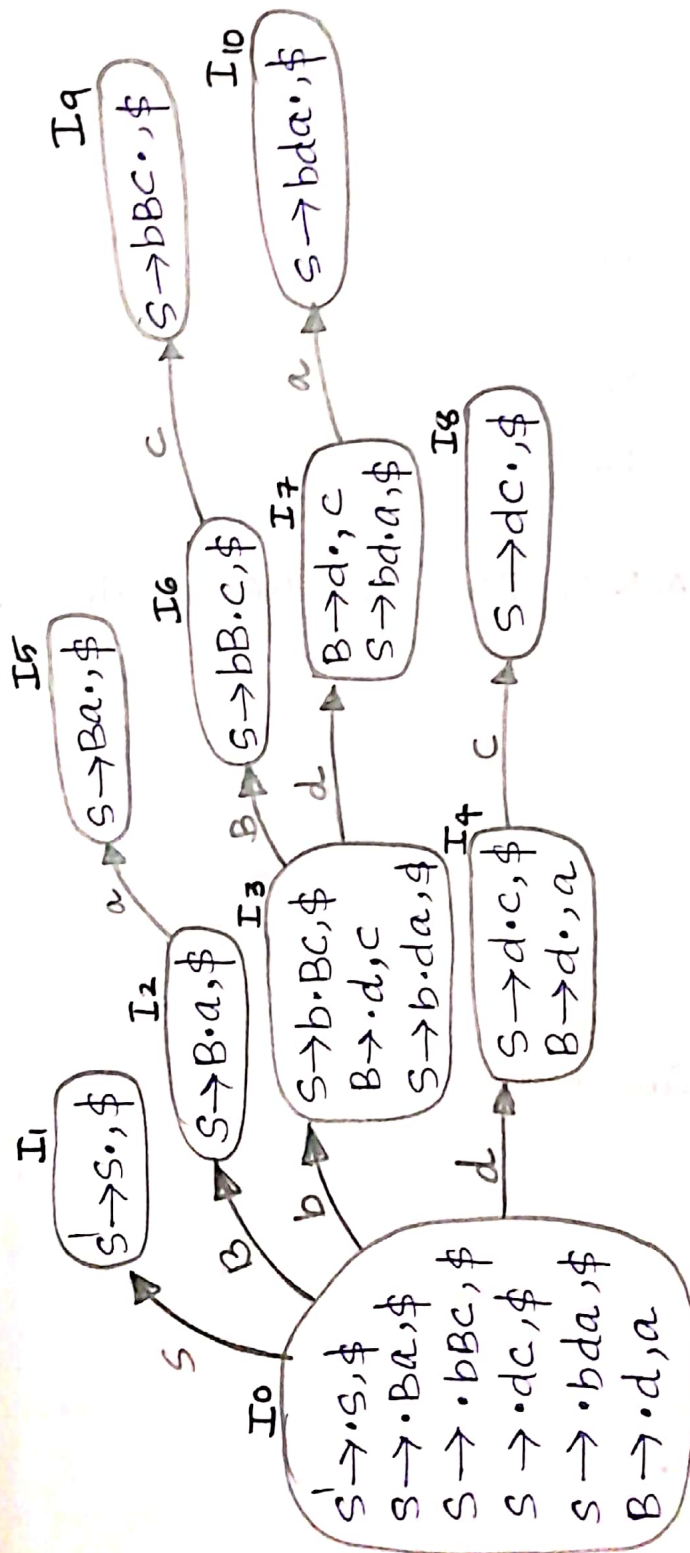
$S \rightarrow dc$

$S \rightarrow bda$

$B \rightarrow d$



## Canonical collection of LR(1) items:



→ There is no LR(1) items in the Canonical collection that have same LR(0) items & differ only in lookahead symbols.

## LALR(1) Parsing Table

state	Action					goto	
	a	b	c	d	\$	S	B
0		S3		S4		1	2
1					accept		
2	S5						
3				S7			6
4	R5		S8				
5					R1		
6			S9				
7	S10		R5				
8					R3		
9					R2		
10					R4		

⊕ Construct SLR and LALR parsing table for the given grammar.

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T F$$

$$T \rightarrow F$$

$$F \rightarrow F *$$

$$F \rightarrow a$$

$$F \rightarrow b$$

# MODULE IV

## Syntax Directed Translation:

Syntax directed Translation, Bottom-up evaluation of S-attributed definitions, L-attributed definitions, Top-down translation, Bottom-up evaluation of inherited attributes.

## Type checking:

Type systems, Specification of a simple type checker.

## Syntax — Directed Translation

→ There are 2 notations for associating semantic rules with productions:

① Syntax-directed Definition.

② Translation Schemes.

→ Syntax-Directed Definitions are high-level specification for translation. They hide many implementation details and free the user from having to specify explicitly the order in which translation takes place.

→ Translation Schemes indicate the order in which semantic rules are to be evaluated, so they allow some implementation details to be shown.

→ Both syntax-directed definition and translation schemes, we parse the input token stream, build the parse tree, and then traverse the tree as needed to evaluate the semantic rules at the parse tree nodes.



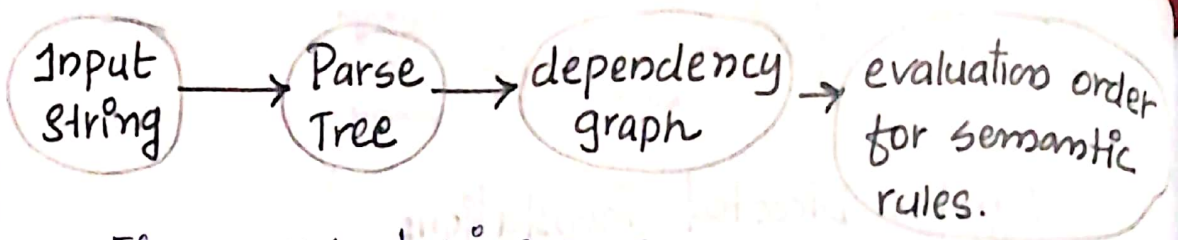


Fig: conceptual view of Syntax-directed Translation

- Evaluation of the semantic rules may generate code, save information in a symbol table, issue error messages, or perform any other activities. The translation of the token stream is the result obtained by evaluating the semantic rules.

### ■ SYNTAX-DIRECTED DEFINITIONS

- A syntax-directed definition is a generalization of a context-free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called the synthesized and inherited attributes of that grammar symbol.
- An attribute can represent anything we choose:
  - ① a string
  - ② a number
  - ③ a type
  - ④ a memory location.
- The value of an attribute at a parse-tree node is defined by a semantic rule associated with the production used at that node.

- The value of Synthesized attributes at a node is computed from the values of attributes at the children of that node in the parse tree.
- The value of inherited attribute is computed from the value of attributes at the Siblings and Parent of that node.
- A parse tree showing the value of attribute at each node is called an annotated parse tree.
- Form of a Syntax-Directed Definition:

→ In a Syntax-directed definition, each grammar production  $A \rightarrow \alpha$  has associated with it a set of semantic rules of the form  $b := f(c_1, c_2, \dots, c_k)$  where 'f' is a function, and either

- ① 'b' is a synthesized attribute of A, and  $c_1, c_2, \dots, c_k$  are attributes belonging to the grammar symbols of the production, or
- ② 'b' is an inherited attribute of one of the grammar symbols on the right side of the production, and  $c_1, c_2, \dots, c_k$  are attributes belonging to the grammar symbols of the production.

### EXAMPLE

PRODUCTION	SEMANTIC ACTION
$E \rightarrow E_1 + T$	$\{ E.val = E_1.val + T.val \}$



## EXAMPLE

Syntax-Directed definition for a desk-top calculator Program.

PRODUCTION	SEMANTIC RULES
$L \rightarrow E_n$	Print ( $E.val$ )
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow digit$	$F.val := digit.lexval$

- The definition associates an integer-valued synthesized attribute called 'val' with each of the nonterminals: 'E', 'T' and 'F'.
- For each E, T and F-production, the semantic rule computes the value of attribute 'val' for the nonterminal on the left side from the values of 'val' for the nonterminals on the right side.
- The token digit has a synthesized attribute 'lexval' whose value is assumed to be supplied by the lexical analyzer.



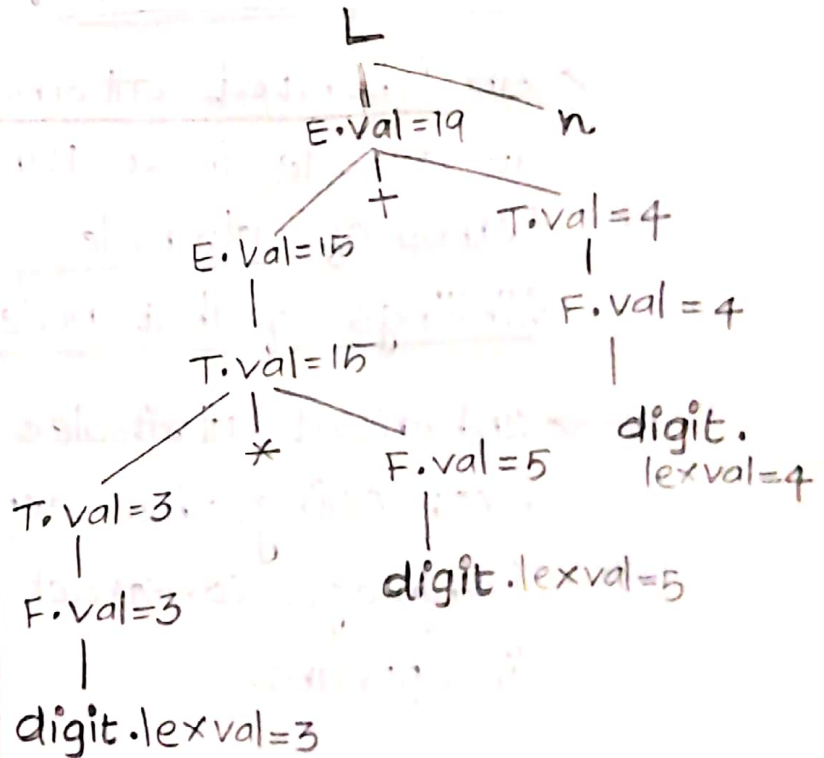
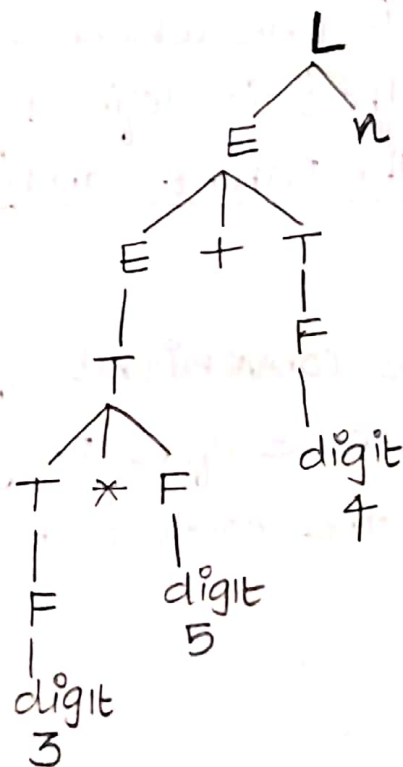
→ The rule associated with the production  $L \rightarrow E n$  for the starting nonterminal  $L$  is just a procedure that prints as output the value of the arithmetic expression generated by  $E$ .

## Synthesized Attributes:

→ A syntax-directed definition that uses synthesized attributes exclusively is said to be an S-attribute definition.

→ A Parse Tree for an S-attribute definition can always be annotated by evaluating the semantic rules for the attributes at each node bottom-up, from the leaves to root.

**EXAMPLE** Parse Tree for  $3 \times 5 + 4n$



$A \rightarrow BCD$ ,  $A$  be a parent node,  $B, C, D$  are children node.  
 $A.S = B.S$   
 $A.S = C.S$   
 $A.S = D.S$  } Parent node  $A$  taking values from its children  $B, C, D$ .

- The attribute values are computed from left to right bottom-up method.
- The bottom-most node production is  $F \rightarrow \text{digit}$ . The semantic rule corresponding to it is  $F.val = \text{digit.lexval}$  which is defined by the attribute  $F.val$  at that node and have the value 3.
- The value of the attribute  $T.val$  has the value 3. The value of the node having production  $T \rightarrow T * F$  is defined by the semantic rule  $T.val = T.val * F.val$ . The value is 15.
- The starting non-terminal  $L \rightarrow E_n$  prints the value of expression generated by  $E$ .

### Inherited Attributes:

- An inherited attribute is one whose values at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of that node.
- Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which it appears.

### EXAMPLE

An inherited attribute distributes type information to the various identifiers in a declaration.

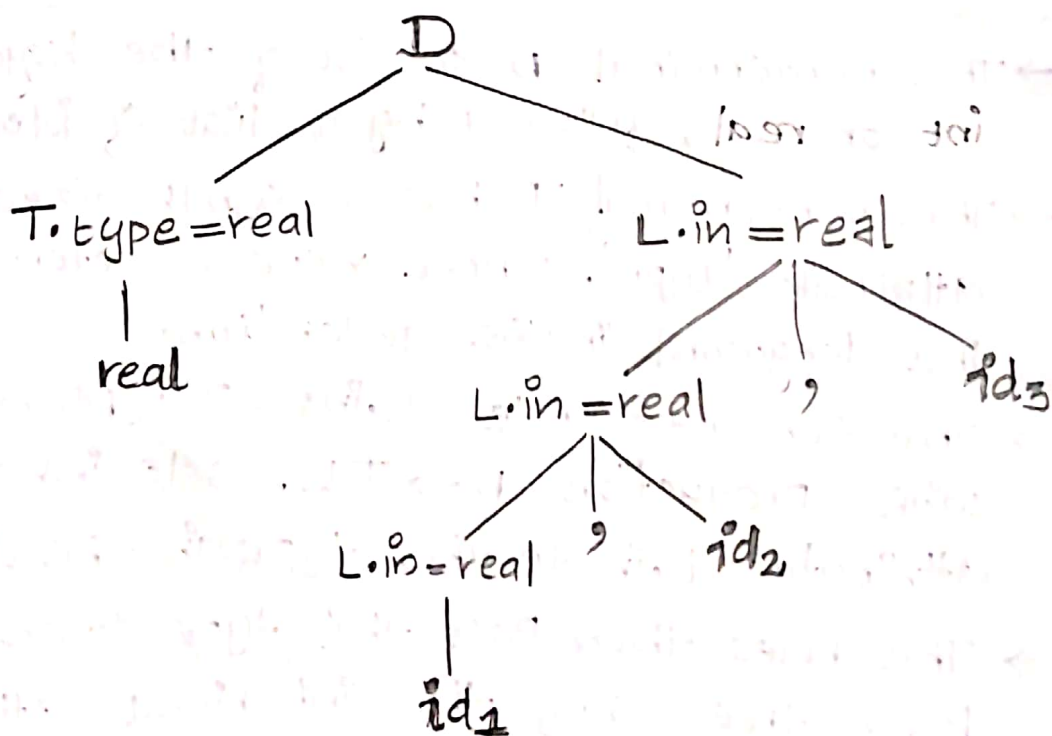
PRODUCTION	SEMANTIC RULES
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow \text{int}$	$T.type := \text{integer}$
$T \rightarrow \text{real}$	$T.type := \text{real}$
$L \rightarrow L_1, id$	$L_1.in := L.in$ $\text{addtype}(id.entry, L.in)$
$L \rightarrow id$	$\text{addtype}(id.entry, L.in)$

- The nonterminal  $D$  consists of the keyword `int` or `real`, followed by a list of identifiers.
- The nonterminal  $T$  has a synthesized attribute 'type', whose value is determined by the keyword in the declaration.
- The semantic rule  $L.in := T.type$ , associated with production  $D \rightarrow TL$ , sets inherited attribute  $L.in$  to the type in the declaration.
- The rules then pass this type down the parse tree using the inherited attribute  $L.in$ .
- Rules associated with the production for  $L$  call procedure 'addtype' to add the type of each identifier to its entry in the symbol table.



→ An annotated parse tree for the sentence  $\text{real id}_1, \text{id}_2, \text{id}_3$ . The value of  $L.in$  at the three L-nodes given the type of the identifiers  $\text{id}_1, \text{id}_2$  and  $\text{id}_3$ .

→ These values are determined by computing the value of the attribute  $T.type$  at the left child of the root and then evaluating  $L.in$  top-down at the 3 L-nodes in the right subtree of the root. At each L-node we also call the procedure 'addtype' to insert into the symbol table the fact that the identifier at the right child of this node has type  $\text{real}$ .



**Fig:** Parse tree with inherited attribute  $in$  at each node labeled  $L$ .

## Dependency Graphs:

- If an attribute 'b' at a node in a parse tree depends on an attribute c, then the semantic rule for 'b' at that node must be evaluated after the semantic rule that defines c.
- The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a Dependency Graph.
- Before constructing a dependency graph for a Parse tree, we put each semantic rule into the form  $b := f(c_1, c_2, \dots, c_k)$ , by introducing a dummy synthesized attribute 'b' for each semantic rule that consist of the procedure call.
- The graph has a node for each attribute and an edge to the node for 'b' from the for 'c' if attribute 'b' depends on attribute 'c'.
- The dependency graph for a given parse tree is constructed as follows:

For each node n in the Parse tree do  
for each attribute a of the grammar symbol  
at n do

construct a node in the dependency  
graph for a;

for each node n in the Parse tree do

for each semantic rule  $b := f(c_1, c_2, \dots, c_k)$   
associated with the production used at n do

for  $i_s = 1$  to  $k$  do

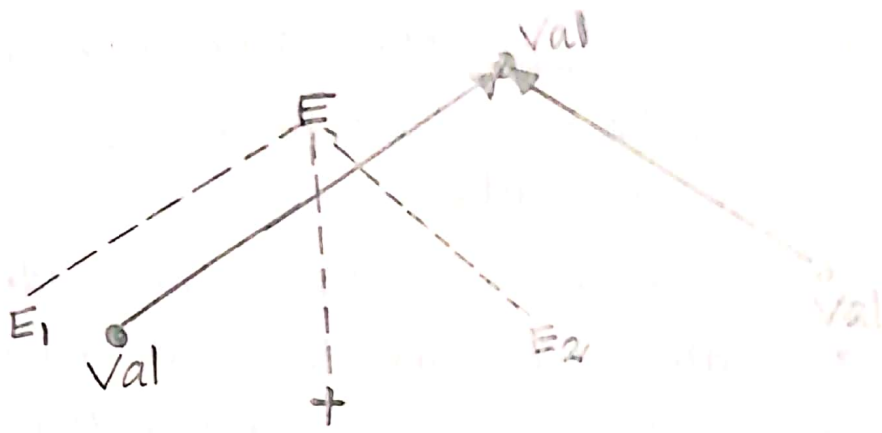
construct an edge from the node for  $q_i$   
to the node for  $k_i$ ;

- For example, suppose  $A.a := f(X.x, Y.y)$  is a semantic rule for the production  $A \rightarrow XY$ .
- The rule defines a synthesized attribute  $A.a$  that depends on the attributes  $X.x$  and  $Y.y$ .
- If this production is used in the parse tree, then there will be 3 nodes,  $A.a$ ,  $X.x$ , and  $Y.y$  in the dependency graph with an edge to  $A.a$  from  $X.x$  since  $A.a$  depends on  $X.x$ , and an edge to  $A.a$  from  $Y.y$  since  $A.a$  also depends on  $Y.y$ .
- If the production  $A \rightarrow XY$  has the semantic rule  $X.i := g(A.a, Y.y)$  associated with it, then there will be an edge to  $X.i$  from  $A.a$  and also an edge to  $X.i$  from  $Y.y$ , since  $X.i$  depends on both  $A.a$  and  $Y.y$ .

### EXAMPLE

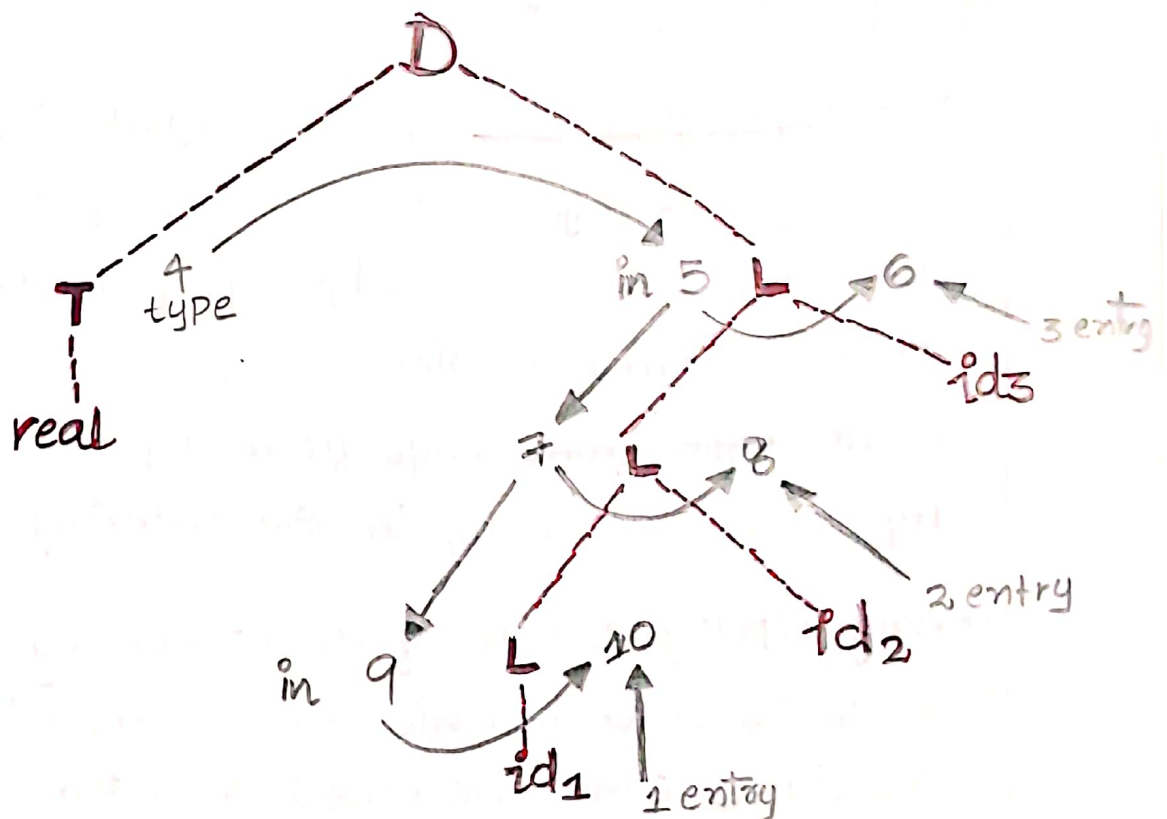
Consider the production  $E \rightarrow E_1 + E_2$  and semantic rule as  $E.val := E_1.val + E_2.val$ . The dependency graph representation of the production is as given below:





(Fig:  $E.val$  is synthesized from  $E_1.val$  and  $E_2.val$ .)

The dependency Graph:



→ Nodes in the dependency graphs are marked by numbers; There is an edge to node 5 for  $L.in$  from node 4 for  $T.type$  because the inherited attribute  $L.in$  depends on the attribute  $T.type$  according to the semantic rule  $L.in := T.type$  for the production  $D \rightarrow TL$ .

- The two downward edges into nodes 7 and 9 arise because  $L_1.in$  depends on  $L.in$  according to the semantic rule  $L_1.in := L.in$  for the production  $L \rightarrow L_1, id$ .
- Each of the semantic rules  $addtype(id.entty, L.in)$  associated with the  $L$ -productions leads to the creation of a dummy attribute.
- Nodes 6, 8 and 10 are constructed for these dummy attributes.

### Evaluation Order:

- A topological sort of a directed acyclic graph is any ordering  $m_1, m_2, \dots, m_k$  of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes; that is, if  $m_i \rightarrow m_j$  is an edge from  $m_i$  to  $m_j$ , then  $m_i$  appears before  $m_j$  in the ordering.
- Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated.
- That is, in the topological sort, the dependent attributes  $c_1, c_2, \dots, c_k$  in a semantic rule  $b := f(c_1, c_2, \dots, c_k)$  are available at a node before  $f$  is evaluated.

### EXAMPLE

Each of the edges in the above dependency graph goes from a lower-numbered node to a higher-numbered node. Hence, a topological sort of the dependency graph is obtained by writing down the nodes in the order of their numbers. From this topological sort, we obtain the following program.

→ We write ' $a_n$ ' for the attribute associated with the node numbered ' $n$ ' in the dependency graph.

```
a4 := real;  
a5 := a4;  
add-type(id3.entry, a5);  
a7 := a5;  
add-type(id2.entry, a7);  
a9 := a7;  
add-type(qd1.entry, a9);
```



## EXAMPLE

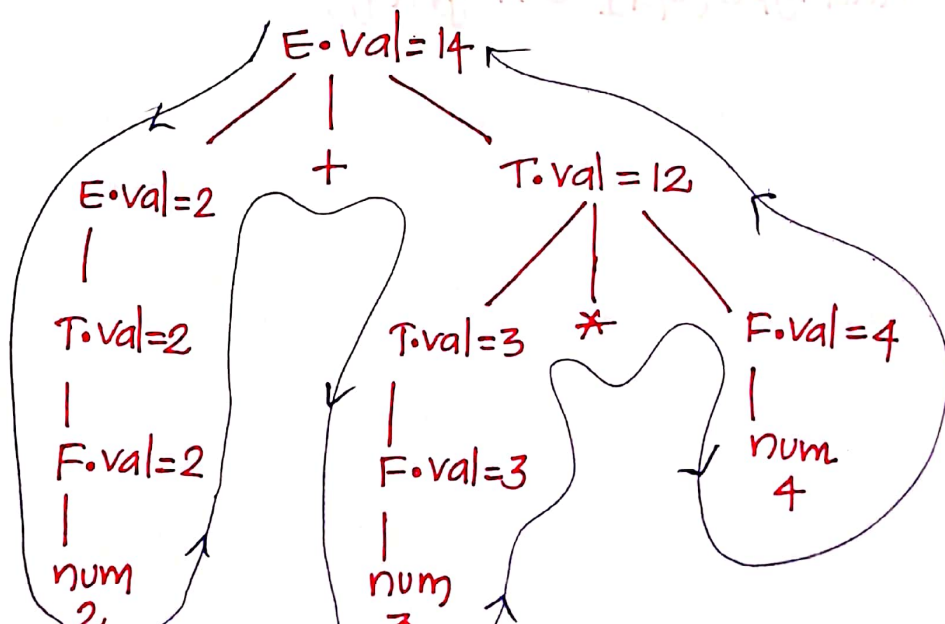
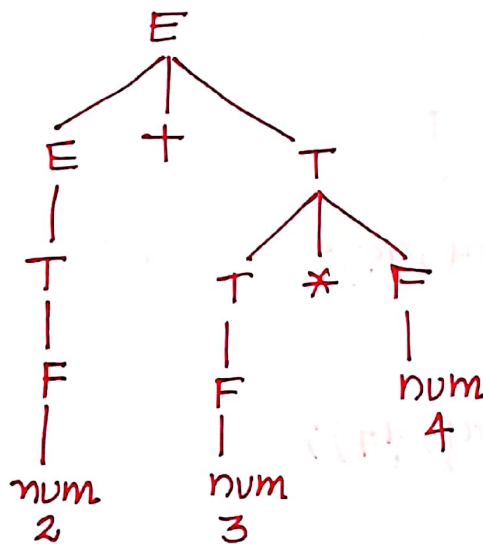
$$E \rightarrow E + T \quad \{ E.val = E.val + T.val \}$$
$$| T \quad \{ E.val = T.val \}$$

$$T \rightarrow T * F \quad \{ T.val = T.val * F.val \}$$
$$| F \quad \{ T.val = F.val \}$$

$$F \rightarrow num \quad \{ F.val = num.val \}$$

$$\left\{ \begin{array}{l} \text{String} := \\ 2 + 3 * 4 \\ \text{OP} \rightarrow 14 \end{array} \right.$$

→ Parse Tree



## EXAMPLE

$E \rightarrow E + T \quad \{ \text{Printf}(" + "); \} \quad (1)$

$| T \quad \{ \} \quad (2)$

$T \rightarrow T * F \quad \{ \text{Printf}(" * "); \} \quad (3)$

$| F \quad \{ \} \quad (4)$

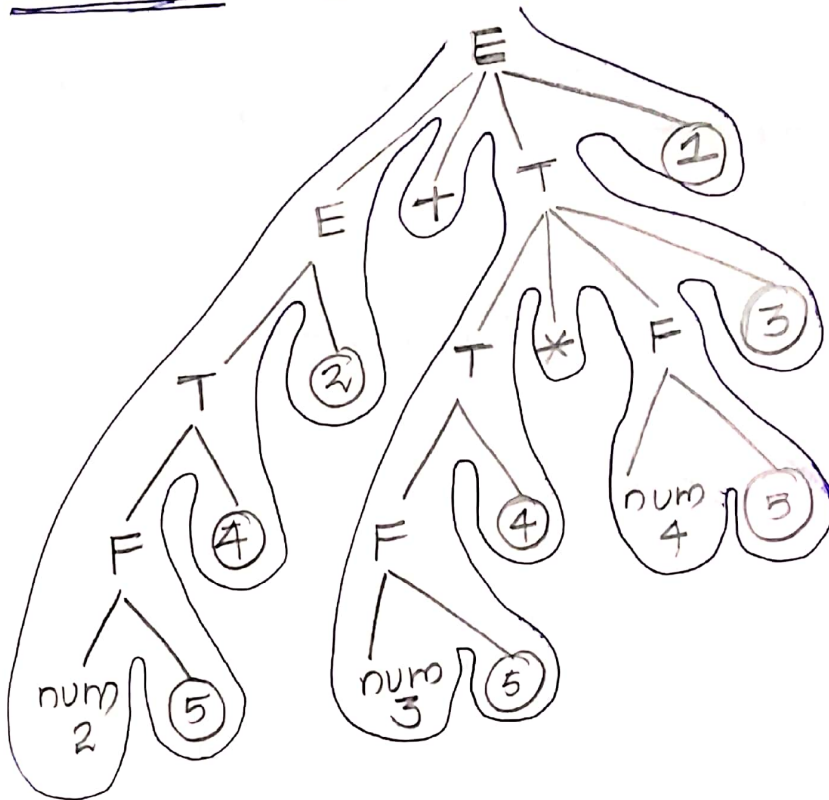
$F \rightarrow \text{num} \quad \{ \text{Printf}(\text{num.val}); \} \quad (5)$

String: -

2+3\*4

→ Parse Tree

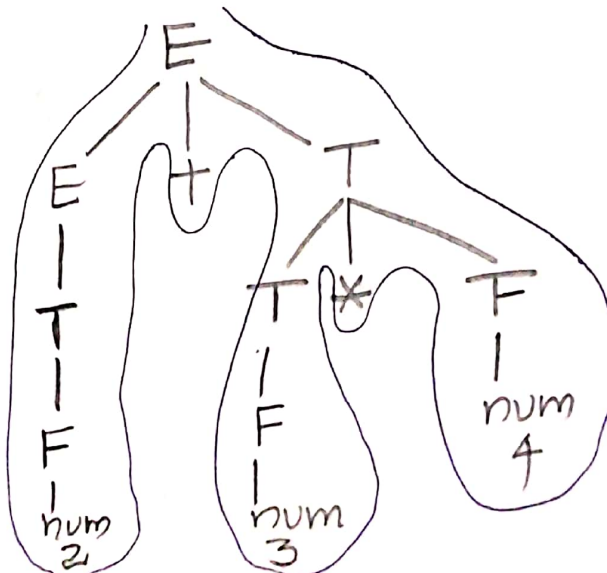
Top-down Parsing:



o/p  
234\*+

→ Parse Tree

Bottom-up Parsing



o/p  
234\*+

### EXAMPLE

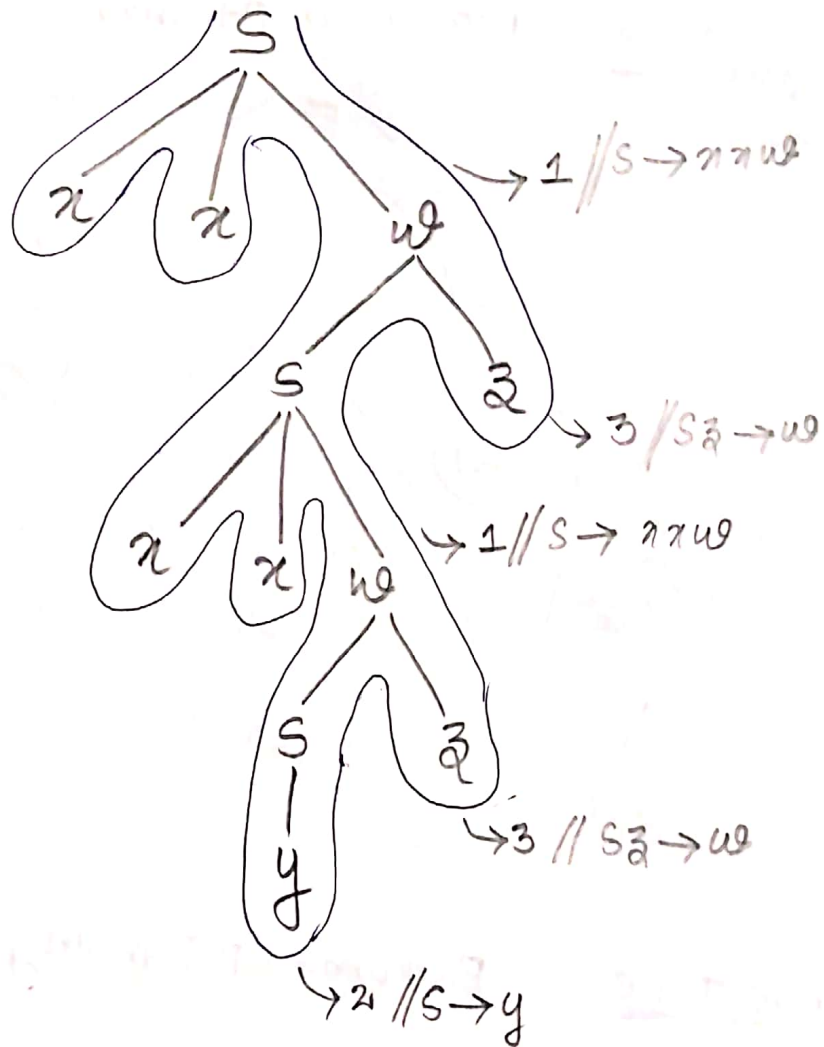
$S \rightarrow x x w \quad \{ \text{printf}(1); \}$

```
1 y { printf(2); }
```

$w \rightarrow S_2 \quad \{ \text{printf}(3); \}$

{ String:—  
xxxxxyzz

→ Parse Tree



Output  $\rightarrow$  23131



## BOTTOM-UP EVALUATION OF S-ATTRIBUTE DEFINITIONS

- S-attributed definitions, that is, the syntax-directed definition with only synthesized attributes.
- Synthesized attributes can be evaluated by a bottom-up parser as the input is being parsed.
- The Parser can keep the values of the synthesized attributes associated with the grammar symbols on its stack.
- Whenever a reduction is made, the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production.

### ■ Synthesized Attributes on the Parser Stack

- A translator for an S-attributed definition can often be implemented with the help of an LR-parser generator.
- The Parser generator can construct a translator that evaluates attributes as it parses the input.
- A bottom-up parser uses a stack to hold information about subtrees that have been parsed.
- We can use extra fields in the parser stack to hold the value of synthesized attributes.

state	val
...	...
x	$x.z$
y	$y.y$
z	$z.z$
...	...

top →

Fig: Parser stack with a field for synthesized attributes.

- Stack is implemented by a pair of arrays of 'state' and 'val'. Each state entry is a pointer to an LR(1) parsing table.
- If the  $i$ th state symbol is A, then  $val[i]$  will hold the value of the attribute associated with the Parse tree node corresponding to this A.
- The current top of the stack is indicated by the pointer top.

Assume synthesized attributes are evaluated before each reduction. Consider the production  $A \rightarrow xyz$  and the rule associated is

$A.a := f(x.z, y.y, z.z)$ . Before  $xyz$  reduce to A the value of attribute  $z.z$  is in  $val[top]$ .

$y.y$  is in  $val[top-1]$  and  $x.x$  is in  $val[top-2]$ .

- If a symbol has no attribute, then corresponding entry in the val array is undefined.

After the reduction the top is decremented by 2.

### Example:

consider the syntax-directed definition of the desktop calculator.

The Parser need to execute the code fragments before making the reduction.

Production	Code Fragment
$L \rightarrow E_n$	$\text{print}(\text{val}[\text{top}])$
$E \rightarrow E_1 + T$	$\text{val}[\text{ntop}] := \text{val}[\text{top}-2] + \text{val}[\text{top}]$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$\text{val}[\text{ntop}] := \text{val}[\text{top}-2] * \text{val}[\text{top}]$
$T \rightarrow F$	
$F \rightarrow (E)$	$\text{val}[\text{ntop}] := \text{val}[\text{top}-1]$
$F \rightarrow \text{digit}$	

When a production with  $n$  symbols on the right side is reduced, the value of ntop is set to  $\text{top}-n+1$ .

After each code fragment is executed, top is set to ntop.

The given below table shows the sequence of moves made by a parser on input  $3 * 5 + 4n$ .



Input	state	val	production used.
3*5+4n	—	—	
*5+4n	3	3	
*5+4n	F	3	$F \rightarrow \text{digit}$
*5+4n	T	3	$T \rightarrow F$
5+4n	T*	3-	
+4n	T*5	3-5	
+4n	T*F	3-5	$F \rightarrow \text{digit}$
+4n	T	15	$T \rightarrow T*F$
+4n	E	15	$E \rightarrow T$
4n	E+	15-	
n	E+4	15-4	
n	E+F	15-4	$F \rightarrow \text{digit}$
n	E+T	15-4	$T \rightarrow F$
n	E	19	$E \rightarrow E+T$
	En	19	
	L	19	$L \rightarrow En$

## L-Attributed Definition

The evaluation of L-attributed definition is performed by depth first order. The procedure is given below.

procedure dfvisit( $n$ :node);

begin

for each child  $m$  of  $n$ , from left to right do begin

evaluate inherited attributes of  $m$ ;

dfvisit( $m$ )

end;

evaluate synthesized attributes of  $n$ .

end

→ In the L-attributed definitions, 'L' stands for Left. i.e., in the L-attributed definition, the attribute flow is from left to right.

→ A syntax directed definition is L-attributed if each inherited attribute of  $x_j$   $1 \leq j \leq n$ , on the right side of  $A \rightarrow x_1 x_2 \dots x_j \dots x_n$  depends on

① The attributes of the symbols  $x_1, x_2, \dots, x_{j-1}$  to the left of  $x_j$  in the production and

② The inherited attributes of  $A$ .

→ Every S-attributed definition is L-attributed definition

PRODUCTION	SEMANTIC RULE
$A \rightarrow LM$	$L.i := l(A, i)$ $M.i := m(L.s)$ $A.s := f(M.s)$
$A \rightarrow QR$	$R.i := r(A, i)$ $Q.i := q(R.s)$ $A.s := f(Q.s)$

The above production and semantic rule is not L-attributed definition because in the 2<sup>nd</sup> production inherited attribute of 'Q' depends on the synthesized attribute of 'R' which is a right sibling of Q.

### TRANSLATION Scheme

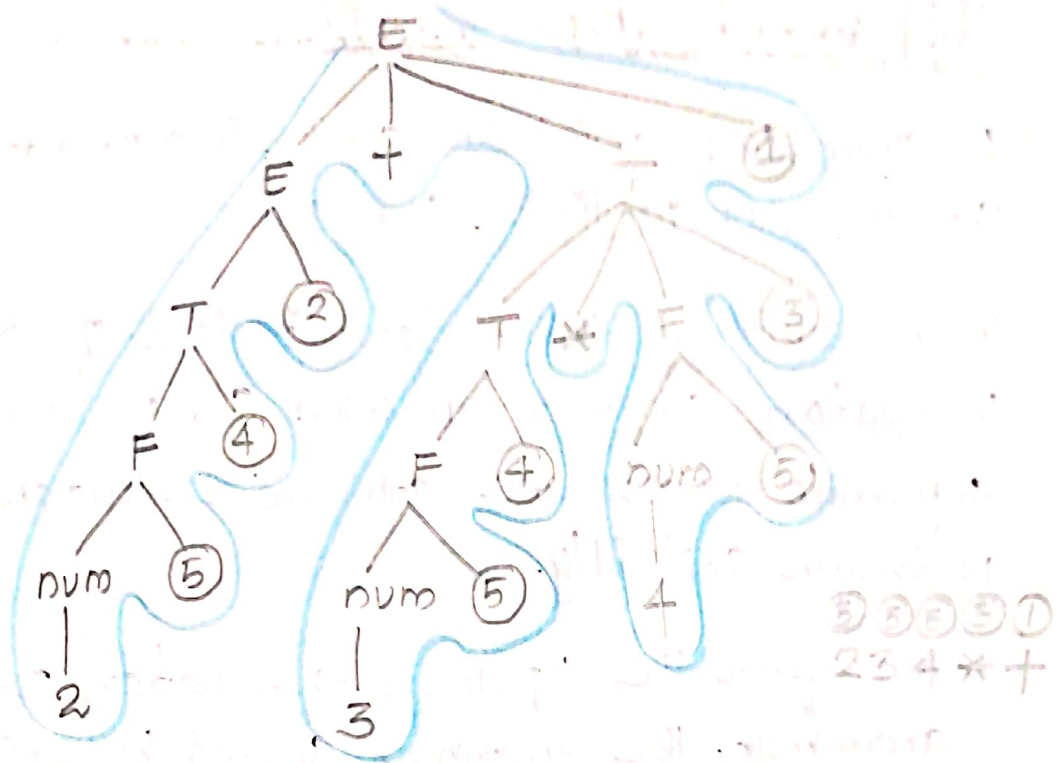
A translation scheme is a CFG in which the attribute are associated with grammar symbols and semantic actions enclosed b/w braces  $\{ \}$  are inserted in the RHS of the production.

#### Example

$E \rightarrow E + T$	$\{ \text{printf}(" + "); \}$	①
T	$\{ \}$	②
$T \rightarrow T * F$	$\{ \text{printf}(" * "); \}$	③
F	$\{ \}$	④
$F \rightarrow \text{num}$	$\{ \text{printf}(\text{num.lval}); \}$	⑤

To generate the input  $2+3*4$  below is the parse tree.





→ The above grammar is to convert infix to prefix.  
In the translation scheme we are embedded the semantic action to the RHS of each non-terminal in the parse tree.

→ If both synthesized and inherited attributes are present in the translation scheme. Then the following rules must be taken.

- ① An inherited attribute for a symbol on RHS must be computed in an action before the computation of symbol.
- ② An action must not refer to a synthesized attribute of a symbol to the right of the action.
- ③ A synthesized attribute for a nonterminal on the left can only be computed after all attributes it references have been computed.

# TOP DOWN TRANSLATION

L-attributed definition can be implemented by Top-down or predictive Parsers.

In the topdown translations instead of SDT, translation schemes are used so, the order in which semantic actions and attribute evaluation should be shown explicitly.

→ To perform the top down translation for the grammar, the grammar should be free from left Recursion. So, first we want to eliminate the left Recursion from the translation grammar.

## ■ Elimination of LR from a Translation Scheme:

Consider the following given grammar and semantic rule

$E \rightarrow E_1 + T$        $\{ E.val := E_1.val + T.val \}$

$E \rightarrow E_1 - T$        $\{ E.val := E_1.val - T.val \}$

$E \rightarrow T$        $\{ E.val := T.val \}$

$T \rightarrow (E)$        $\{ T.val := E.val \}$

$T \rightarrow num$        $\{ T.val := num.val \}$

The left recursion from the above grammar can be avoided by the following rule

$$A \rightarrow A\alpha_1 | A\alpha_2 | B$$

then

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \epsilon$$

The left recursion elimination of the above grammar is as follows:

$$E \rightarrow TR$$

$$R \rightarrow +TR$$

$$R \rightarrow -TR$$

$$R \rightarrow \epsilon$$

$$T \rightarrow (E)$$

$$T \rightarrow \text{num}$$

The semantic action for the left recursion eliminate grammar is as follows:

$$E \rightarrow T \quad \{ R.i := T.val \}$$

$$R \quad \{ E.val := R.s \}$$

$$R \rightarrow +$$

$$T \quad \{ R_1.i = R.i + T.val \}$$

$$R_1 \quad \{ R.s = R_1.s \}$$

$$R \rightarrow -$$

$$T \quad \{ R_1.i = R.i - T.val \}$$

$$R \quad \{ R.s = R_1.s \}$$

$$R \rightarrow \epsilon \quad \{ R.s = R.i \}$$

$$T \rightarrow ($$

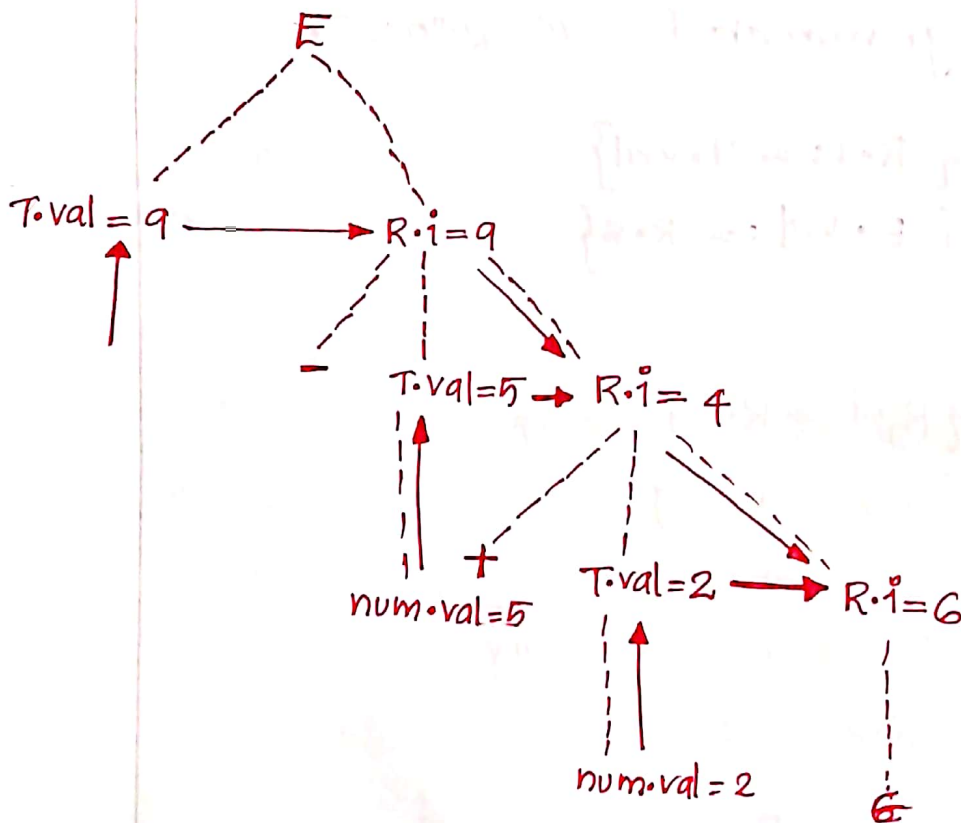
$$E \quad \{ T.val = E.val \}$$

)

$$T \rightarrow \text{num} \quad \{ T.val = \text{num.val} \}$$



- For top down parsing we assume an action is executed at the time that a symbol in the same position would be expanded.
- This is the second production, the 1<sup>st</sup> action (assignment to  $R.i$ ) is done after  $T$  has been fully expanded.
- In the L-attributed definition, the symbol must be computed by the action only after the computation of all of its inherited value.
- The Parse tree for the evaluation of the expression  $9-5+2$  is given below:



→ The translation scheme uses inherited attributes for the proper evaluation of digits separated by '+' and '-' sign.

→ The given translation scheme sends the final result to  $R \rightarrow E$  node on the bottom of the chain. The final result should be send to the root of the parse tree, so that the translation scheme used synthesized attribute for moving the final from  $R \rightarrow E$  node back to the root of the  $R$  nodes.

### BOTTOM-UP EVALUATION OF INHERITED ATTRIBUTES

Bottom up evaluation of inherited attribute is capable of handling all L-attributed definition based on LR(1) Grammars.

#### ■ Removing Embedding Actions from Translation Schemes:

→ Inherited attributes can be handled bottom up by introducing a transformation that makes all embedded actions in a translation scheme at the right ends of their productions.

→ The Transformation inserts a new marker nonterminals generating  $\epsilon$  into the grammar. Each embedded action is replaced by a distinct marker non terminal 'M' and attach the action to the end of the production  $M \rightarrow \epsilon$ .

For example: consider the following translation Scheme.

$$E \rightarrow TR$$

$$R \rightarrow +T \{ \text{Print}(' + ') \} R \mid -T \{ \text{Print}('-') \} R \mid \epsilon$$

$$T \rightarrow \text{num} \{ \text{Print}(\text{num.val}) \}$$

is transformed using marker ~~as~~ nonterminals  $M$  and  $N$  into

$$E \rightarrow TR$$

$$R \rightarrow +TMR \mid -TNR \mid \epsilon$$

$$T \rightarrow \text{num} \{ \text{Print}(\text{num.val}) \}$$

$$M \rightarrow \epsilon \{ \text{Print}(' + ') \}$$

$$N \rightarrow \epsilon \{ \text{Print}('-') \}$$

Actions in the transformed translation Scheme terminate productions, so they can be performed just before the right side is reduced during bottom-up parsing.

### ■ Inherited Attributes on the Parse Stack:

→ A bottom-up parser reduces the right side of Production  $A \rightarrow XY$  by removing  $X$  and  $Y$  from the top of the parser stack and replacing them by  $A$ . Suppose  $X$  has a synthesized attribute  $x.s$  and is kept along with  $X$  on the parser stack.



→ The value of  $x.s$  is already on the parser stack before any reductions take place in the subtree below  $y$ , this value can be inherited by  $y$ . i.e., if inherited attribute  $y.i$  is defined by the copy rule  $y.i := x.s$ , then the value of  $x.s$  can be used where  $y.i$  is called for. Copy rules play an important role in the evaluation of inherited attribute during bottom up parsing.

→ consider an example: real p, q, r

$$D \xrightarrow{L} T \quad \{ L.in := T.type \}$$

$$T \rightarrow \text{int} \quad \{ T.type := \text{integer} \}$$

$$T \rightarrow \text{real} \quad \{ T.type := \text{real} \}$$

$$L \rightarrow L_1, id \quad \{ L_1.in = L.in \}$$

$$\{ \text{add type}(id.entry, L.in) \}$$

$$L \rightarrow id \quad \{ \text{add type}(id.entry, L.in) \}$$

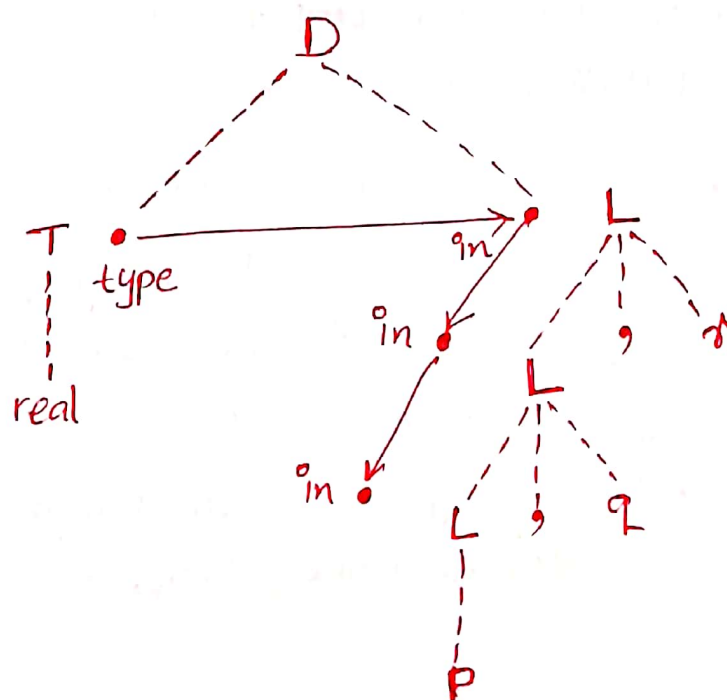


Fig: At each node for  $L$ ,  $L.in = T.type$

Suppose if the parser stack is implemented as a pair of arrays *state* and *val*. The production and code fragment is as given below.

Production	code fragment
$D \rightarrow TL;$	
$T \rightarrow \text{int}$	$\text{val}[\text{ntop}] = \text{integer}$
$T \rightarrow \text{real}$	$\text{val}[\text{ntop}] = \text{real}$
$L \rightarrow L, \text{id}$	$\text{addtype}(\text{val}[\text{top}], \text{val}[\text{top}-3])$
$L \rightarrow \text{id}$	$\text{addtype}(\text{val}[\text{top}], \text{val}[\text{top}-1])$

■ Algorithm: Bottom-Up Parsing and Translation with Inherited Attributes

Input: An L-attributed definition with an underlying LL(1) grammar.

Output: A parser that computes the values of all attributes on its parsing stack

Method: Let us assume that every non-terminal 'A' has one inherited attribute  $A.i$  and every grammar symbol  $X$  has a synthesized attribute  $X.s$ . If  $X$  is a terminal then its synthesized attribute is really the lexical value returned with  $X$  by the lexical analyzer; that the lexical value appears on the stack, in an array *val*.

For every Production  $A \rightarrow x_1 \dots x_n$ , introduce a new marker non-terminals,  $M_1, M_2, \dots, M_n$ , for the production  $A \rightarrow x_1, x_2, \dots, x_n$  and replace the production by  $A \rightarrow M_1 x_1, \dots, M_n x_n$ . The synthesized attribute  $x_j$ 's will go on the parser stack in the val array entry associated with  $x_j$ . The inherited attribute  $x_j.i$  appears in the same array but associated with  $M_j$ .

As we parse, the inherited attribute  $A.i$  exists, it is found in the position of the val array immediately below the position for  $M_1$ . If there is an inherited attribute for start symbol, it could be placed below the bottom of the stack.

There are two cases for the computation of attribute in the bottom up parser.

(i) If we reduce an inherited attribute to a marker non-terminal  $M_j$ , we know which production  $A \rightarrow M_1 x_1 \dots M_n x_n$  that marker belong to. Therefore we know the position of any attributes that the inherited attribute  $x_j.i$  needs for its computation.

(ii) When we reduce to a nonmarker symbol by Production  $A \rightarrow M_1 x_1 \dots M_n x_n$ . Then we have to compute the synthesized attribute  $A.s$ , note that  $A.i$  was already computed, and lives at the position on the stack just below the position into which we insert  $A$  itself.



The following two simplifications reduce the no. of markers the 2nd simplification avoids parsing conflict in left recursive grammar.

- ① If  $X_j$  has no inherited attribute, we need not use marker  $M_j$ . The expected position for attributes on the stack will change if  $M_j$  is omitted, and this change can be easily incorporated easily into the parser.
- ② If  $X_1.i$  exists and is computed by a copy rule  $X_1.i = A.i$ , then we can omit  $M_1$  since  $A.i$  will already be located just below  $X_1$  on the stack and this can serve value to  $X_1.i$ .

## TYPE CHECKING

→ Type checking is the part or component of semantic analyzer of the compiler. Type checker need to verify that the source program follows the syntactic (validated with respect to parser) and semantic (validated with respect to semantic analyzer) convention. Type checker detected and report the programming errors.

→ Following are the examples of static checks.

- ① Type Checks: A compiler should report an error if an operator is applied to an incompatible operand.

example:

```
int op(int), op(float);
```

```
int f(float);
```

```
int a, c[10], d;
```

```
d = c + d    # fail
```

```
*d = a      # fail
```

② Flow-of-control checks: Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control.

For example, A break statement in C causes control to leave the smallest enclosing while, for or switch statements, ~~an~~ an error occurs if such an enclosing statement does not exist.

```
myfunc()
{
    while(n)
    {
        if(i > 10)
            break; # Here break executes and control
    }              come outside loop.
}
```

```
myfunc()
{
    break;
}
```

③ Uniqueness Check: An object must be defined exactly once. For example, in Pascal an identifier must be declared uniquely, labels in a case statement must be distinct, and elements in a scalar type may not repeated.

```
main()
```

```
{ int i, j, i; # error
```

```
}
```

```
myfunct (int a, int a) # error
```

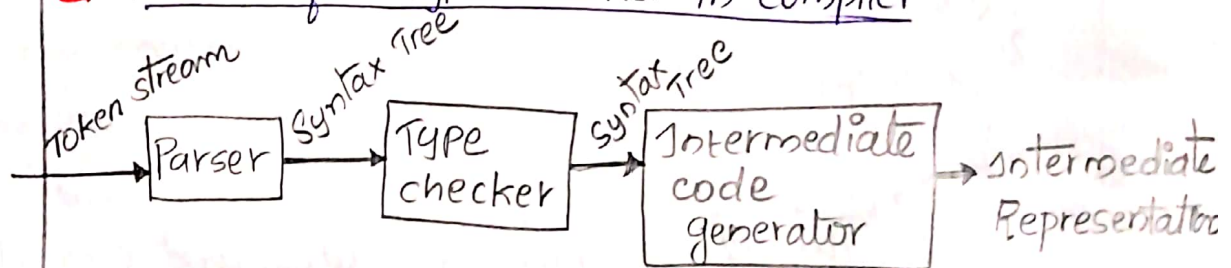
```
{
```

```
}
```

④ Name related checks: sometimes, the same name must appear two or more times.

Example; In ada, a loop or block may have a name that appears at the beginning and end of the construct. The compiler must check that the same name is used at both places.

#### ■ Position of a type checker in compiler



Type checker verifies whether the syntax tree generated by parser is correct or not for a given input. Type checking is done by adding Semantic rule based on grammar. Based on the Semantic rule, type checking is performed on the parse tree and verifies it is correct or not. The input and output of the type checker is the Syntax tree.



## ■ Type Systems:

A Type system is a collection of rules for assigning type expression to the various part of the program.

A type checker implements type system. Different compilers and processors of the same language use different type system.

The design of a type checker for a language is based on the information about the syntactic construct in the language, the notion of the types and the rules for assigning types to language construct.

eg: if both operands of the arithmetic operators of addition, subtraction and multiplication are of integer type then the result is of type integer.

## ■ Type Expression:

The type of the language construct is denoted by a type expression.

→ A <sup>type</sup> ~~language~~ expression is either a basic type or is formed by applying an operator called a type constructor or other type expressions.

→ The set of basic types and constructor depend on the language to be checked.

- ① A basic type is a type expression. The basic types are integer, char, real and boolean. A special type called type error will signal an error during type checking and the basic type.

void denotes the absence of value allows the statements to be checked.

② Type name is a type expression.

③ A type constructor applied to type expressions is a type expression.

Constructors include:

a) Arrays:— If  $T$  is a type expression, then  $\text{array}[I, T]$  is a type expression denoting the type of an array with elements of type  $T$  and index set  $I$ .  $I$  is often a range of Integer. eg: In Pascal declaration  $\text{Var } A: \text{array}[1..10] \text{ of Integer};$

associates the type expression  $\text{array}(1..10, \text{integer})$  with  $A$ .

b) Products:— If  $T_1$  and  $T_2$  are type expression, then their cartesian Product  $T_1 \times T_2$  is a type expression.

c) Records:— The records and Products are similar but records have names for the different fields.

d) Pointers:— If  $T$  is a type expression, then  $\text{pointer}(T)$  is a type expression denoting the type "pointer to an object of type  $T$ ".

eg:- In Pascal declaration  $\text{var } p: \uparrow \text{row}$  declares variable  $p$  to have type pointer (row).



e) Functions:— A function maps elements of the set domain to another set range. Function in programming languages map domain type  $D$  to range type  $R$  and is denoted by  $D \rightarrow R$ .

eg: In Pascal built-in function `mod` of Pascal has domain type `int x int` i.e., a pair of integers, and a range type `int` i.e., `int x int  $\rightarrow$  int`.

4) Type expression may contain variables whose values are type expression.

→ checking done by a compiler is called static checking while checking done when the target program runs is called dynamic checking.

### ■ Specification of a Simple Type Checker

The type checker is a translation scheme that synthesizes the type of each expression from the type of its subexpression. The type checker can handle array, pointer, statements and functions.

#### Simple Language:

The following given grammar generates programs, represented by the nonterminal  $P$ , consisting of a sequence of declarations  $D$  followed by a single expressions  $E$

$P \rightarrow D; E$

$D \rightarrow D; D \mid id : T$



$$T \rightarrow \text{char} \mid \text{integer} \mid \text{array}[\text{num}] \text{ of } T \mid \uparrow T$$

$$E \rightarrow \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E(E) \mid E \uparrow$$

→ Simple program generated by the grammar is:

key: integer;

key mod 1999

- The basic type are char and integer.
- The type type-error is used to signal errors.
- array[256] of char leads to the type expression  $\text{array}(1..256, \text{char})$  consisting of the ~~is~~ constructor Pointer applied to the range  $1..256$  and the type char.

→ As in Pascal, the Prefix Operator  $\uparrow$  in declarations builds a pointer type, so

$\uparrow \text{integer}$

leads to the type expression  $\text{pointer}(\text{integer})$ , consisting of the constructor Pointer applied to the type integer.

### Type checking of Expression

- In the following given rules the synthesized attribute 'type' for E gives the type expression assigned by the type system to the expression generated by E.

$E \rightarrow \text{literal}$	$\{E.\text{type} = \text{char}\}$
$E \rightarrow \text{num}$	$\{E.\text{type} = \text{integer}\}$
$E \rightarrow \text{id}$	$\{E.\text{type} = \text{lookup}(\text{id}, \text{entry})\}$
$E \rightarrow E_1 \text{ mod } E_2$	$\{E.\text{type} = \text{if } E_1.\text{type} = \text{integer and } E_2.\text{type} = \text{integer then int else type-error}\}$

$E \rightarrow E_1 \uparrow$	$\{E.\text{type} = \text{if } E_1.\text{type} = \text{pointer}(t) \text{ then } t \text{ else type-error}\}$
------------------------------	--

→ A function lookup is used to fetch the type of an identifier saved in the symbol table.

→ In the array sequence reference  $E_1[E_2]$ , the index expression  $E_2$  must have type integer, in which the result is the element type  $t$  obtained from the type array (st) of  $E_1$ .

$E \rightarrow E_1[E_2]$	$\{E.\text{type} = \text{if } E_2.\text{type} = \text{integer and } E_1.\text{type} = \text{array}(s, t) \text{ then } t \text{ else type-error}\}$
--------------------------	---

### Type checking of Statements

→ A special basic type void is assigned to the statement since the statements do not have values.

→ If an error is detected in the statement then type-error is assigned to the statements.

→ The statement can be assignment, conditional, while statement and the sequence of statements separated by semicolons.

$P \rightarrow D; S$  // Program consists of declaration followed by statements.

$S \rightarrow id := E \quad \{ s.type = \text{if } id.type = E.type \text{ then void else type\_error} \}$

$S \rightarrow \text{if } E \text{ then } S_1 \quad \{ s.type = \text{if } E.type = \text{boolean} \text{ then } S_1.type \text{ else type\_error} \}$

$S \rightarrow \text{while } E \text{ do } S_1 \quad \{ s.type = \text{if } E.type = \text{boolean} \text{ then } S_1.type \text{ else type\_error} \}$

$S \rightarrow S_1; S_2 \quad \{ s.type = \text{if } S_1.type = \text{void and } S_2.type = \text{void} \text{ then void else type\_error} \}$

→ The rules for checking statements are given above. The 1st rule checks that the left and right side of an assignment statement have some type. The 2nd & 3rd rules specify that expressions in conditional and while statements must have the type boolean. In the last rule a mismatch of type produces the type-error otherwise it produces void if each substatement has type void.



## Type checking of Functions

→ The application of a function to an argument can be given by the production

$$E \rightarrow E(E)$$

in which an expression is applying to another expression.

→ The rule for checking the type of a function application is,

$$E \rightarrow E_1(E_2) \quad \{ \begin{array}{l} E.\text{type} = t \text{ if } E_2.\text{type} = s \text{ and} \\ E_1.\text{type} = s \rightarrow t \text{ then } t \\ \text{else type\_error} \end{array} \}$$

The above rule says that in an expression formed by applying  $E_1$  to  $E_2$ , the type of  $E_1$  must be a function  $s \rightarrow t$  from the type  $s$  of  $E_2$  to some range type  $t$ , then the type of  $E_1(E_2)$  is  $t$ .

# MODULE V

## Run-Time Environment

**Source** Language issues, Storage Organization, Storage allocation Strategies

## Intermediate Code Generation (ICG):

Intermediate languages — Graphical representations, Three-address code, Quadruples, Triples.

Assignment Statements, Boolean expressions.

## RUN TIME ENVIRONMENT

A compiler must implement data objects (abstraction of source language) in the source language definition. These abstraction includes names, scopes, bindings, data types, Operators, Procedures, Parameters and flow of control constructs. The compiler must cooperate with OS and other system software to support these abstraction on the target machine.

To implement the source language abstraction, compiler creates and manages a run time environment in which it assures its target programs are being executed.

The environment deals with several issues such as

- ① Allocation of storage location for object names in source program.
- ② Mechanism used by target program to access variable.
- ③ Linkage b/w the procedures
- ④ Mechanism for Passing Parameters.
- ⑤ Interfaces to the OS.
- ⑥ Input & Output devices.



## Source language Issues:

→ For the description of issues, suppose that a program is made up of procedures in Pascal and the following sections distinguishes b/w the source text of a procedure and its allocation activation at the run time.

## Procedures:

- A Procedure definition is a declaration that associates an identifier with a statement.
- The identifier is the procedure name, and the statement is the Procedure body.
- Procedures that return values are called functions in many languages; however, it is convenient to refer them as procedures. A complete program will also be treated as a Procedure.
- When a procedure name appears within an executable statement, we say that the procedure is called at the point.
- The Procedure call executes the Procedure body.
- Identifier appears inside the called procedure is called actual parameters or arguments.
- The identifier appears in the procedure definition is called Parameters.



## Activation Trees:

The following assumptions are made about the flow of control among procedures during the execution of a program.

- ① Control Flows sequentially; that is, the execution of a program consists of sequence of steps and the control flow sequentially.
- ② Each execution of a procedure starts beginning of the procedure body and eventually returns control to the point immediately following the place where the procedure was called.

Each execution of a procedure body is referred to as an activation of the procedure.

The lifetime of an activation of a procedure 'P' is the sequence of steps between 1st and last steps in the execution of procedure body, including time spent on executing the procedures called by P, the procedure called by them and so on. The term 'lifetime' refers to a consecutive sequence of steps during the execution of a program.

A procedure is recursive if a new activation can begin before an earlier activation of the same procedure has ended.

The Activation Trees are used to depict the way control enters and leaves the activation (procedures).

In an activation tree:—

- ① Each node represents an activation of a procedure,
- ② The root represents the activation of main program.



- Date / /
- ③ the node for 'a' is the parent of the node for 'b' iff control flows from activation 'a' to 'b'
  - ④ the node for 'a' is to the left of the node for 'b' iff the lifetime of 'a' occurs before the lifetime of 'b'.

Consider the following sequence of output for the activation of quicksort procedure.

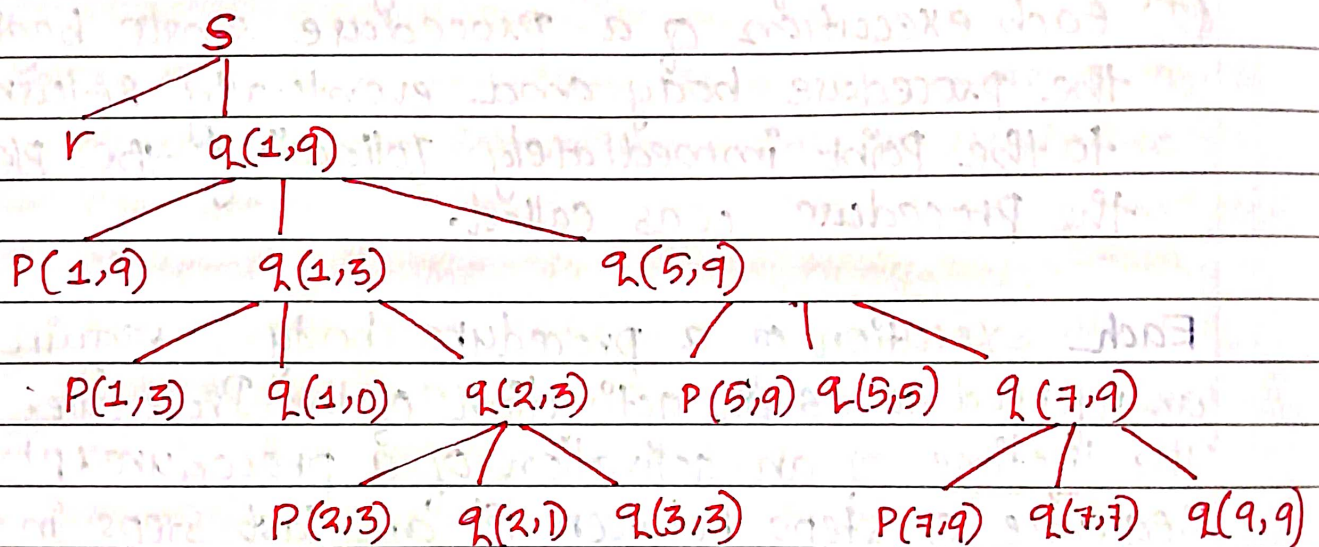


Fig: Activation Tree

### Control Stacks:

→ The Flow of control in a program corresponds to a depth-first traversal of the activation tree that starts at the root, visits a node before its children, and recursively visits children at each node in a left-to-right order.

→ A control stack is used to keep track of live procedure activations.



→ The Idea is to push the node for an activation onto the control stack as the activation begins and to Pop the node when the activation ends.

→ The content of control stack are related to paths to the root of the activation tree. When node 'n' is at the top of the control stack, the stack contains the nodes along the Path from 'n' to the root.

Program main:

Procedure P;

var a: real;

Procedure q;

var b: integer;

begin ... end;

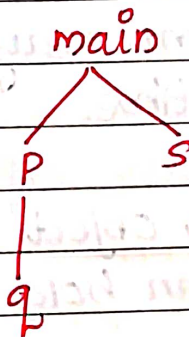
begin q; end;

Procedure s;

var c: integer;

begin ... end;

begin p; s; end;



main
P
a:
q
b:
s
c:

Stack



## Scope of Declaration:

A declaration in a language associates information with a name. Declaration may be implicit and explicit in Pascal.

Explicit Declaration: Var i: integer;

Implicit Declaration: Any variable name starting with I is assumed to denote an integer.

→ The Scope Rules of a language determine which declaration of a name applies when the name applies more number of the time in the text of a program.



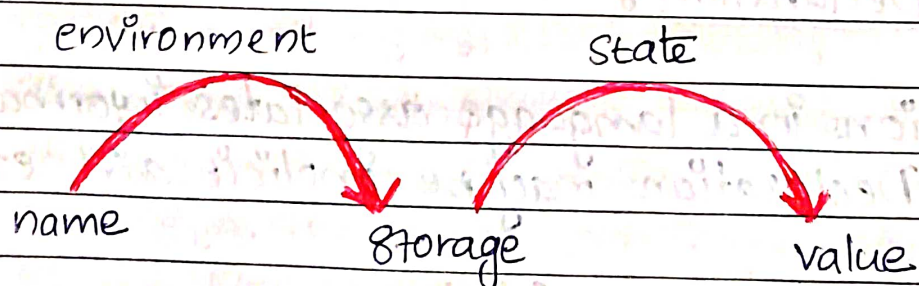
→ The portion of the Program to which a declaration applies is called the scope of that declaration. An occurrence of a name in a procedure is said to be local to the Procedure if it is in the scope of declaration within the procedure; otherwise, the occurrence is said to be nonlocal.

### Binding Of Names:

→ Even if each name is declared once in a program, the same name may denote different data objects at run time.

→ The term "data object" corresponds to a storage location that can hold values.

→ In the programming language semantics, the term "environment" refer to a function that maps a name to a storage location, and the term "state" refer to a function that maps a storage location to the value.



→ Environments and states are different; an assignment changes the state, but not the environment.

Example: Suppose that storage address 100, associated with variable  $\pi$ , holds 0. After the assignment  $\pi = 3.14$ , the same storage address is associated with  $\pi$ , value is 3.14.



## Storage Organization:

The organization of run-time storage are given below can be used for languages such as Fortran, Pascal, and C.

### Subdivision of Run-Time Memory:

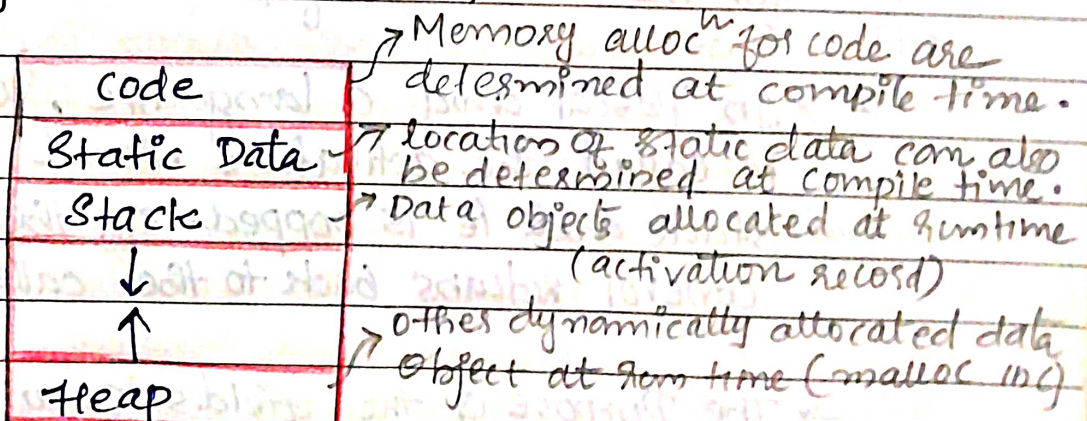
→ The compiler obtains a block of storage from the Operating System for the compiled program to run in.

→ The run-time storage might be subdivided into

- ① the generated target code
- ② data objects
- ③ Control stack to keep track of Procedure activation.

→ The size of the generated target code is fixed at compile time, so the compiler can place it in a statically determined area, perhaps in the low end of memory.

→ The size of some of the data objects may also be known at compile time, and there too can be placed in a statically determined area.



### Fig: Typical Subdivision of Run-time memory into code and data areas.

→ One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into a target code.



→ When a procedure call occurs in the source Program, execution of an activation is interrupted and information about the status of the machine, such as the value of the Program counter and machine registers, is saved on the stack.

→ When control returns from the call, the activation can be restarted after restoring the values of relevant registers and setting the Program counter to the point immediately after the call.

→

A separate area of run-time memory, called a heap, holds all other information.

→ The sizes of the stack and the heap can change as the program executes.

### Activation Record:-

→ Activation Record or frame is a contiguous block of storage which contains the information needed for the execution of procedure.

→ In Pascal and C language, when the procedure is called the activation record is pushed into the stack and it is popped off the stack when the control returns back to the caller.

→ The Purpose of the fields of an activation record is as follows

① Temporary values, such as those arising in the evaluation of expressions, are stored in the field for temporaries.



- 2 The Field for local data holds data that is local to an execution of a procedure.

Layout of this field:

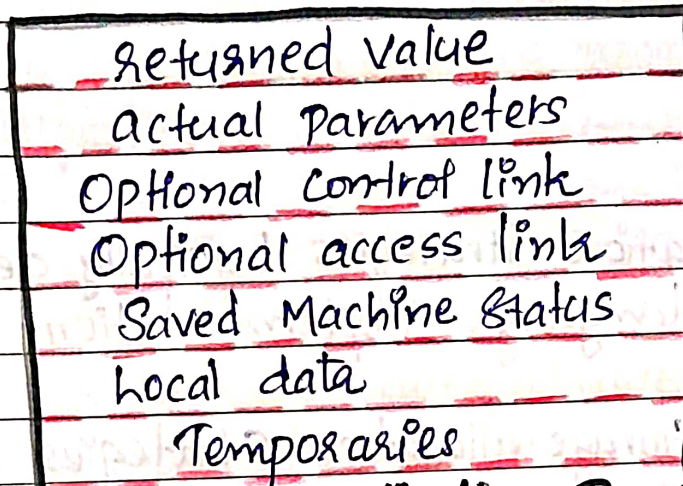


Fig: A general Activation Record

- 3 The field for saved machine status holds information about the state of the machine just before the procedure is called. This information includes the values of the Program Counter and machine registers that have to be restored when control returns from the procedure.
- 4 The Optional ~~control~~ access link is used to refer to nonlocal data held in other activation records.
- 5 The optional control link points to the activation record of the caller.
- 6 The field of actual Parameters is used by the calling procedure to supply parameters to the called procedure. But in practice parameters are passed through machine register for efficiency.
- 7 The field for the returned value is used by the called procedure to return a value to the calling procedure.



→ The size of each field in the activation record can be determined at the time a procedure is called.

## STORAGE ALLOCATION STRATEGIES

Storage Allocation Strategies basically depend on the Programming language implementation.

Three basic Storage allocation strategies are given below:

① STATIC ALLOCATION

② STACK ALLOCATION

③ HEAP ALLOCATION

### Static Allocation:

→ static allocation lays out storage for all data objects at compile time.

→ In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.

→ Since the bindings do not change at run time, every time a procedure is activated, its names are bound to the same storage locations. This property allows the values of local names to be retained across activations of a procedure.

→ That is, when control returns to a procedure, the values of the locals are the same as they were when control left the last time.



- From the type of a name, the compiler determines the amount of storage to set aside for that name.
- The address of this storage consists of an offset from an end of the activation record for the procedure.
- The compiler must eventually decide where the activation records go, relative to the target code and to one another.
- Once this decision is made, the position of each activation record, and hence of the storage for each name in the record is fixed.
- At compile time we can therefore fill in the addresses at which information is to be saved when a procedure call the target code can find the data it operates on.
- Similarly, the addresses at which information is to be saved when a procedure call occurs are also known at compile time.

### Limitations:

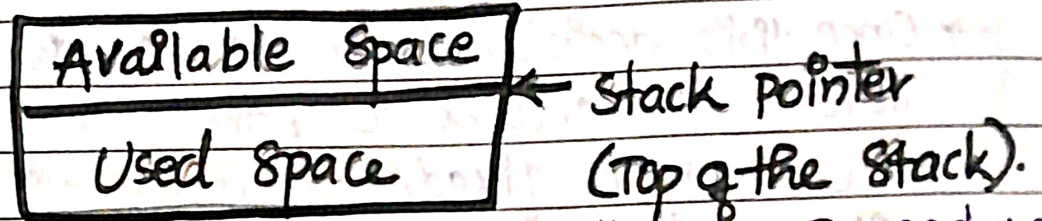
- ① The size of a data object and constraints on its position in memory must be known at compile time.
- ② Recursive Procedures are restricted, because all activations of a Procedure use the same bindings for local names.
- ③ Data structures cannot be created dynamically, since there is no mechanism for storage allocation at runtime.

→ Fortran was designed to permit static storage allocation.



## Stack Allocation:

→ Stack allocation is based on the idea of a Control Stack; storage is organized as a stack, and activation records are Pushed and Popped as activations begin and end, respectively.



- Storage for locals in each call of a procedure is contained in the activation record for that call.
- Thus locals are bound to fresh storage in each activation, because a new activation record is pushed onto the stack when a call is made.
- Furthermore, the values of locals are deleted when the activation ends; that is, the values of local are lost because the storage for locals disappears when the activation record is popped.
- The register "top" marks the top of the stack.
- At run time, an activation record can be allocated and deallocated by incrementing and decrementing "top", respectively, by the size of the record.
- If Procedure 'q' has an activation record of size 'a', then "top" is incremented by 'a' just before the target code of 'q' is executed.
- When control returns from 'q', "top" is decremented by 'a'.



POSITION IN ACTIVATION TREE	ACTIVATION RECORD ON THE STACK	REMARKS
S	<div>S</div> <div>a: array</div>	Frame for S
<pre>       S      /     r           </pre>	<div>S</div> <div>a: array</div> <div>r</div> <div>i: integer</div>	r is activated
<pre>       S      / \     r  q(1,9)           </pre>	<div>S</div> <div>a: array</div> <div>q(1,9)</div> <div>i: integer</div>	frame for r has been popped and q(1,9) is Pushed.
<pre>       S      / \     r  q(1,9)        / \       p(1,9) q(1,3)            / \           p(1,3) q(1,0)           </pre>	<div>S</div> <div>a: array</div> <div>q(1,9)</div> <div>i: integer</div> <div>q(1,3)</div> <div>i: integer</div>	Control has just returned to q(1,3).

Calling Sequences:

- Procedure calls are implemented by generating what are known as calling sequences in the target code.
- A call sequence allocates an activation record and enters information into its fields.
- A return sequence restores the state of the machine so the calling procedure can continue execution.



→ A call sequence does the following things:

- i Allocates the activation record for the called procedure.
- ii Loads Actual Parameter.
- iii Saves machine status.
- iv Transfer control to the callee.

→ A Return sequence does the following things:

- i Deallocate activation record of the called Procedure.
- ii Sets up return value.
- iii Restore Machine state. (PC and Stack Pointer)

→ Activation Record and calling sequence differ from machine to machine, the calling sequence are often divided b/w the caller and the procedure being called.

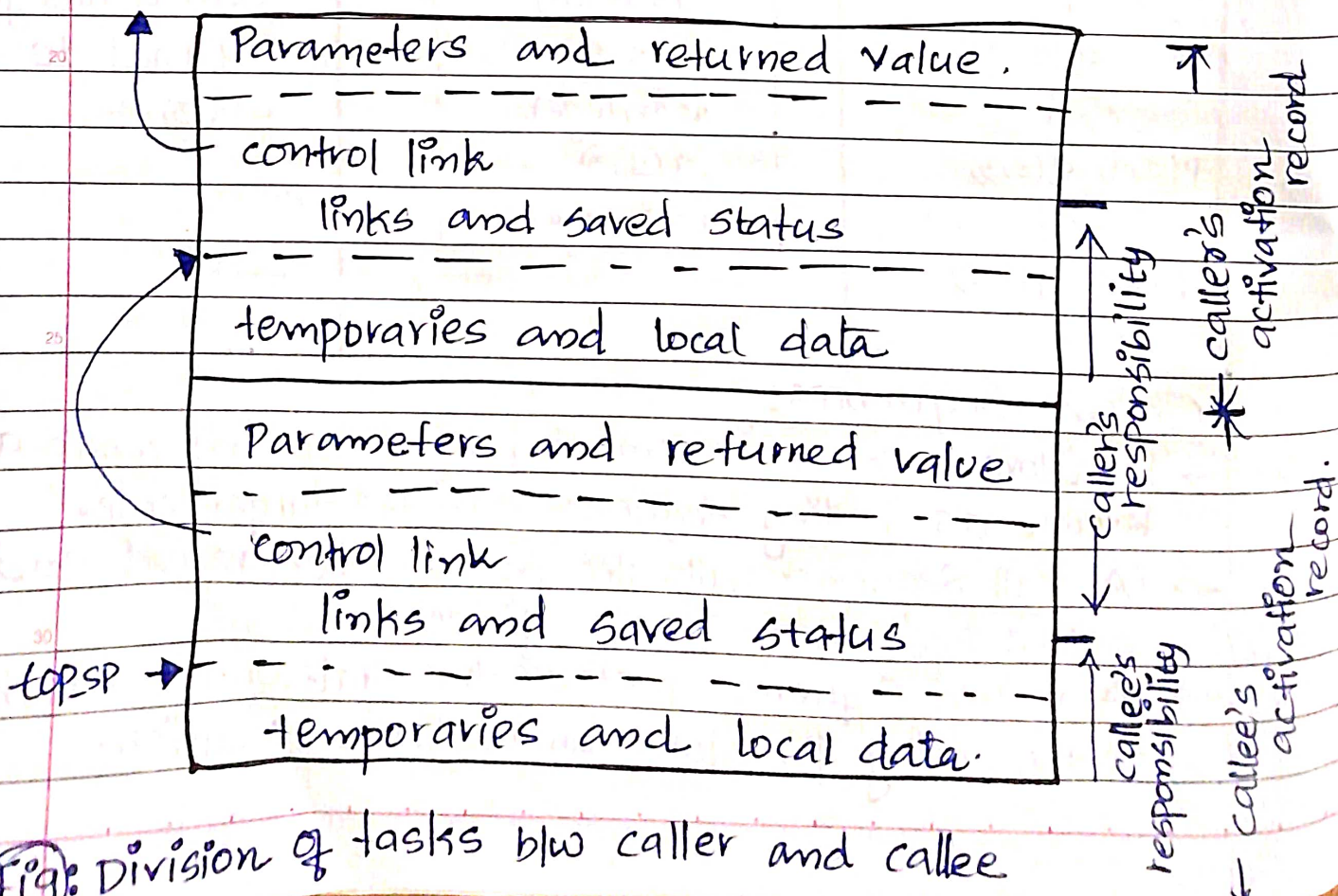


Fig: Division of tasks b/w caller and callee



Date / /

→ In the above figure, the register 'top-sp' points to the end of the machine status field in the activation record. This position is known to the caller, so it can be made responsible for setting 'top-sp' before control flows to the called Procedure.

→ The code for the callee can access its temporaries and local data using offsets from 'top-sp'.

→ The call sequence is:

- ① The caller evaluates actuals. (actual Parameters).
- ② The caller stores a return address and the old value of 'top-sp' into the callee's activation record.
- ③ The callee saves register values and other status information.
- ④ The callee ~~initiates~~ initialize its local data and begins execution.

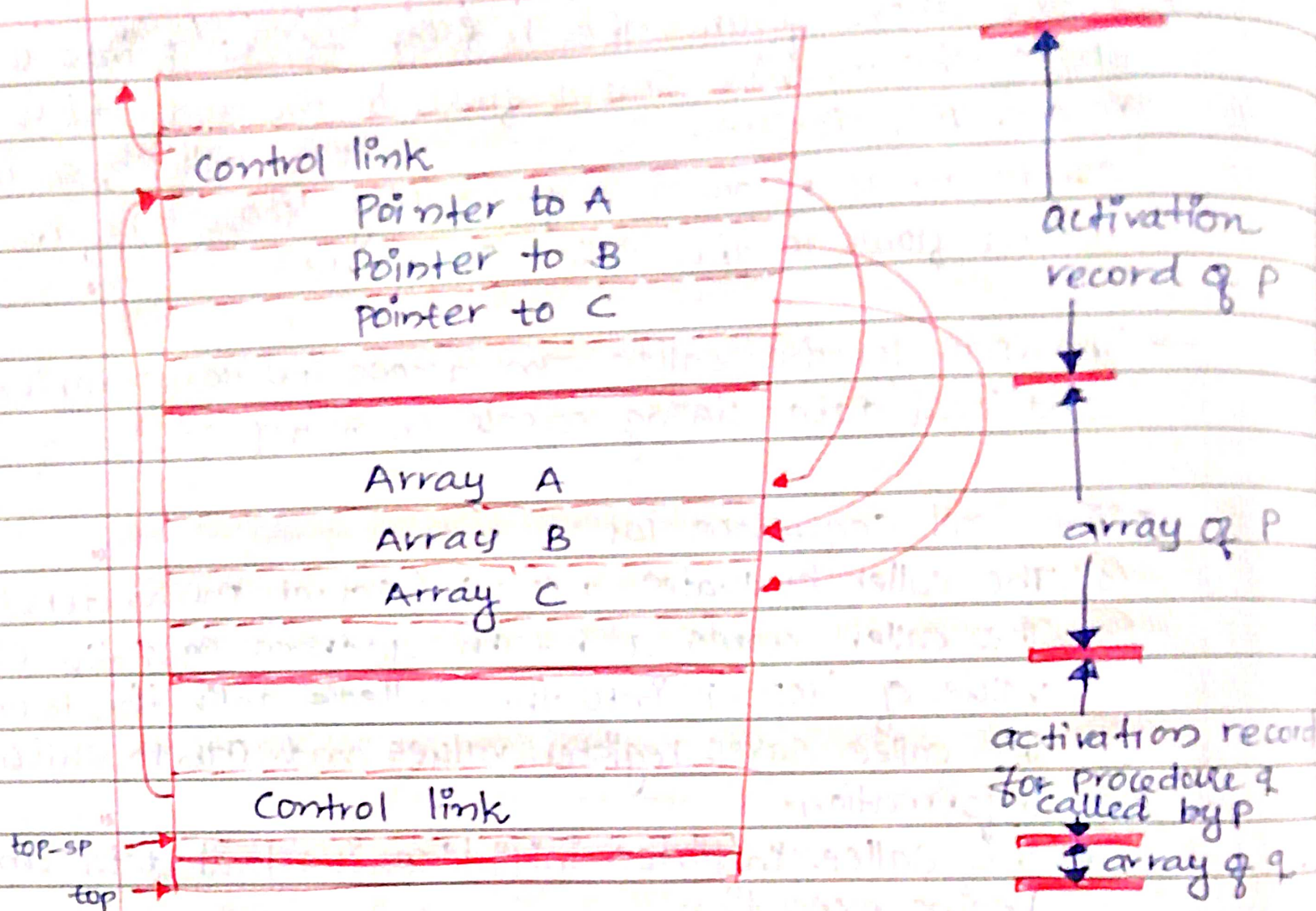
→ A Possible return sequence is:

- ① The callee places a return value next to the activation record of the caller.
- ② Using the information in the status field, the callee restores 'top-sp' and other registers and branches to a return address in the caller's code.
- ③ Although 'top-sp' has been decremented, the caller can copy the returned value into its own activation record and use it to evaluate an expression.

### Variable Length Data

A common strategy for handling variable length data is shown in figure.





(Fig): Access to dynamically Allocated Array.

- Procedure ' $P$ ' has 3 local arrays.
- The storage of these arrays is not part of the activation record for ' $P$ '; only a pointer to the beginning of each array appears in the activation record.
- The relative address of these pointers are known at compile time; so the target code can access array elements through the pointers.
- Access of data on the stack is through two pointers, ' $top$ ' and ' $top-sp$ '.
- The ' $top$ ' → Actual top of the stack.
  - It points to the position at which the next activation record will begin.



- The 'top\_sp' → Used to find local data.
- In Figure, 'top\_sp' points to the end of ~~the~~ this field in the activation record for Q. Within the field is a control link to the Previous value of 'top\_sp' when control was in the calling activation of P.

## Dangling References

- Whenever storage can be deallocated, the Problem of dangling references arises.
- A dangling reference occurs when there is a reference to storage that has been deallocated.
- Dangling Reference is a logical error, ~~to~~ since the value of deallocated storage is undefined according to the semantic of most languages.

**Example:** Program that leaves a pointing to deallocated storage.

```
main()
{
    int *P;
    P = dangle();
}
```

```
int *dangle()
{
    int i = 23;
    return &i;
}
```

3



## HEAP ALLOCATION:

→ The stack allocation strategy cannot be used if either of the following is possible.

① The value of local names must be retained when an activation ends.

② A called activation outlives the caller.

→ In each of the above cases, the deallocation of activation records need not occur in a last-in-first-out fashion, so storage cannot be organized as a stack.

→ Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects. Pieces may be deallocated in any order, so overtime the heap will consist of alternate areas that are free and in use.

→ In the following figure, the record for an activation of Procedure 'r' is retained when the activation ends. The record for the new activation  $q(1, q)$  therefore cannot follow that for 's' physically.

→ If the retained activation record for 'r' is deallocated, there will be free space in the heap b/w the activation records for 's' and  $q(1, q)$ . It is left to the heap manager to make use of this space.

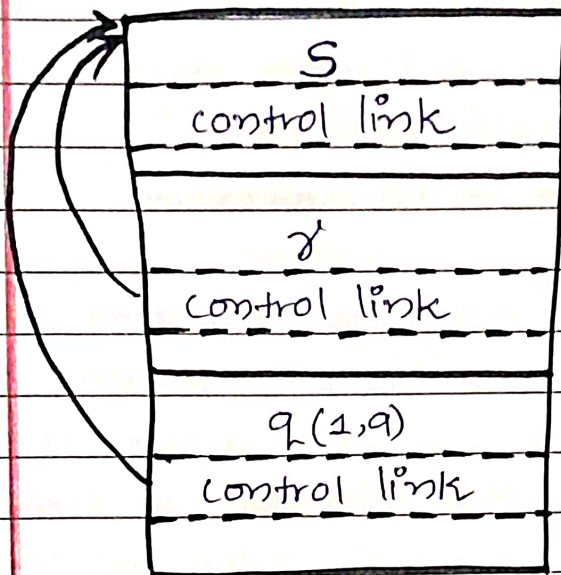
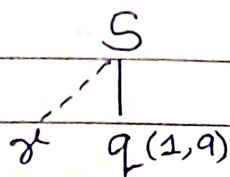
→ Time and space overhead are present in heap management.



Position in the  
Activation Tree

Activation Records in  
the Heap

Remarks



Retained  
activation  
record for  $x$ .

**Fig 8** Records for live activations need not be adjacent in a heap.

→ There is generally some time and space overhead associated with using a heap manager.

→ For efficiency reasons, it may be helpful to handle small activation records or records of a predictable size as a special case, as follows:

- 1 For each size of interest, keep a linked list of free blocks of that size.
- 2 If possible, fill a request for size 's' with a block of size  $s'$ , where  $s'$  is the smallest size greater than or equal to 's'. When the block is eventually deallocated, it is returned to the linked list it came from.
- 3 For large blocks of storage use the heap manager.

→ The above 3 approaches results in fast allocation and deallocation of small amount of storage, since taking and returning a block from a linked list are efficient operations.



# INTERMEDIATE CODE GENERATION

In the analysis - synthesis model of a compiler, the front end translates a source program into an intermediate representation from which the back end generates target code.

Benefits of using a machine-independent intermediate form are:

- 1 Retargetting is facilitated; a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
- 2 A machine-independent code optimizer can be applied to the intermediate representation.

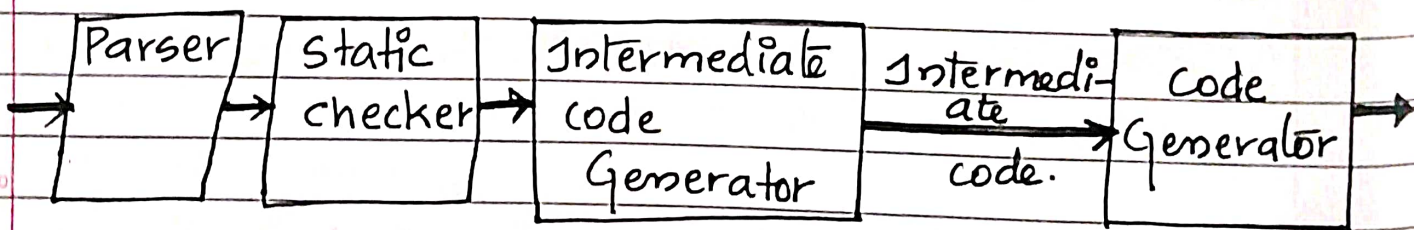


Fig: Position of Intermediate code Generator.

## INTERMEDIATE LANGUAGES:

Types of Intermediate Representations:

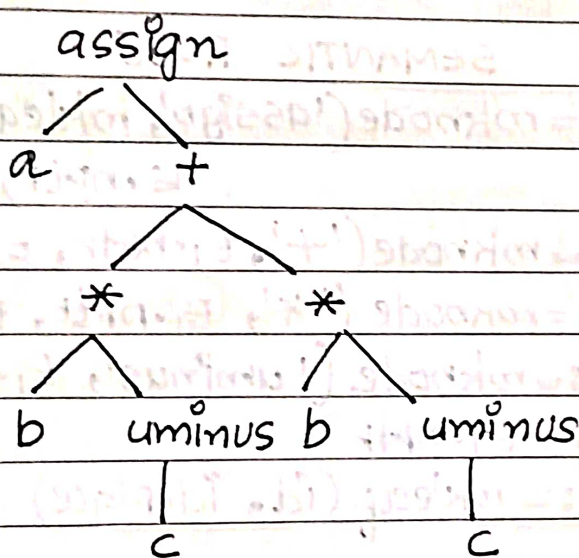
- 1 Syntax Tree
- 2 Postfix Notation
- 3 Three-address code



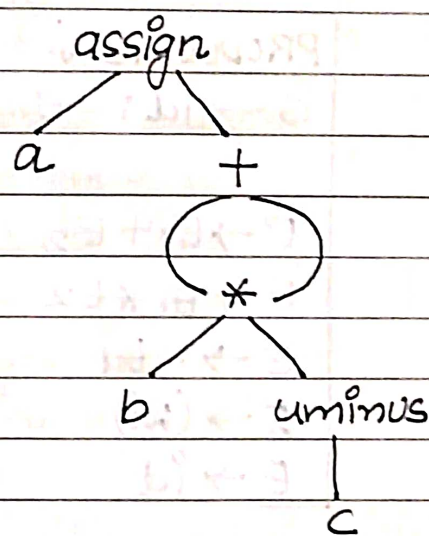
## Graphical Representation:

→ A Syntax Tree depicts the natural hierarchical structure of a source program. A dag gives the same information but in a more compact way because common subexpressions are identified.

→ A Syntax tree and dag for the assignment statement  $a := b * -c + b * -c$ .



(a) Syntax Tree



(b) Dag

**Fig:** Graphical representations of  $a := b * -c + b * -c$ .

→ Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children.

→ The Postfix notation for the syntax tree of above statement is

$a \ b \ c \ \text{uminus} \ * \ b \ c \ \text{uminus} \ * \ + \ \text{assign}$



- The edges in a Syntax tree do not appear explicitly in Postfix notation. They can be recovered from the order in which the nodes appear and the number of operands that the operator at node expects.
- The syntax trees for assignment statements are produced by the syntax-directed definition.
- Nonterminal  $S$  generates an assignment statement.
- The two binary operators  $+$  and  $*$  are examples of the full operator set in a typical language.

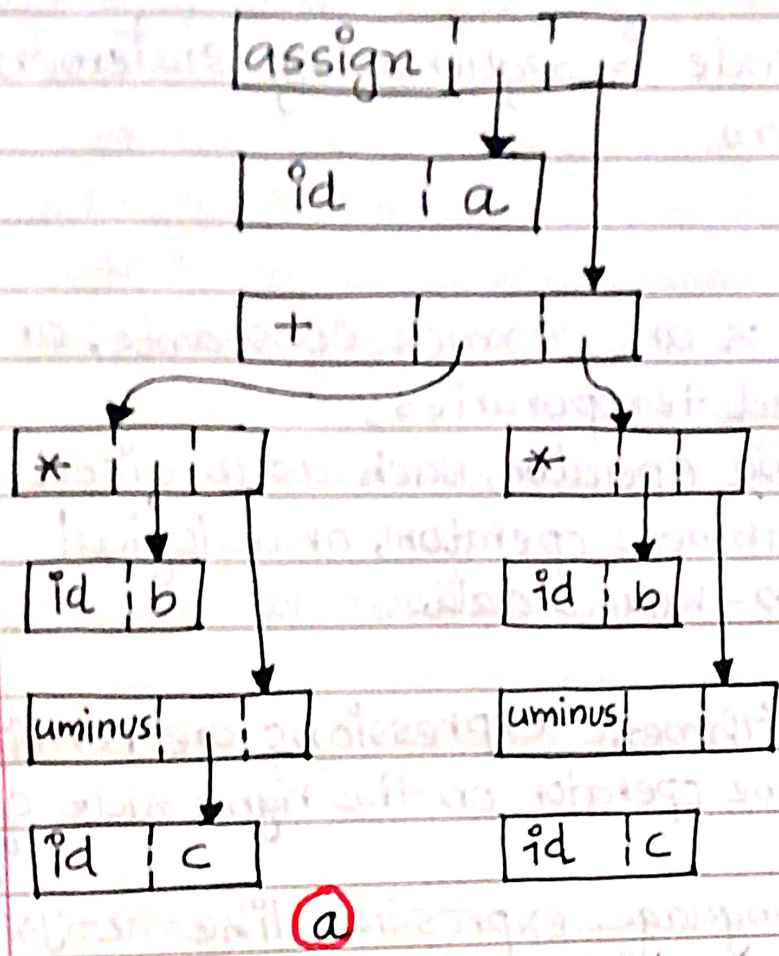
PRODUCTION	SEMANTIC RULE
$S \rightarrow \text{id} := E$	$S.\text{nptr} := \text{mknode}(\text{'assign'}, \text{mkleaf}(\text{id}, \text{id.place}), E.\text{nptr})$
$E \rightarrow E_1 + E_2$	$E.\text{nptr} := \text{mknode}(\text{'+'}, E_1.\text{nptr}, E_2.\text{nptr})$
$E \rightarrow E_1 * E_2$	$E.\text{nptr} := \text{mknode}(\text{'*'}, E_1.\text{nptr}, E_2.\text{nptr})$
$E \rightarrow -E_1$	$E.\text{nptr} := \text{mknode}(\text{'uninus'}, E_1.\text{nptr})$
$E \rightarrow (E_1)$	$E.\text{nptr} := E_1.\text{nptr}$
$E \rightarrow \text{id}$	$E.\text{nptr} := \text{mkleaf}(\text{id}, \text{id.place})$

**Fig:** Syntax-directed definition to produce syntax trees for assignment statement.

- The same syntax-directed definition will produce the dag if the function  $\text{mkunode}(\text{op}, \text{child})$  and  $\text{mknode}(\text{op}, \text{left}, \text{right})$  return a pointer to an existing node whenever possible, instead of constructing new nodes.
- The token  $\text{id}$  has an attribute 'Place' that points to the symbol-table entry for the identifier.



→ Two representations of the syntax tree is given below.



id	b	
id	c	
uminus	1	
*	0	2
id	b	
id	c	
uminus	5	
*	4	6
+	3	7
id	a	
assign	9	8
.....		

**Fig:** Two representations of the syntax tree.

- In Fig(a), each node is represented as a record with a field for its operator and additional fields for Pointers to its children.
- In Fig(b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node.



## Three - Address code :

→ Three - address code is sequence of statements of the general form

$$x := y \text{ op } z$$

where  $x, y$  and  $z$  are names, constants, or compiler-generated temporaries;

'op' stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data.

→ No built-up arithmetic expressions are permitted, as there is only one operator on the right side of a statement.

→ Thus a source language expression like  $x + y * z$  might be translated into a sequence

$$t_1 := y * z$$

$$t_2 := x + t_1$$

where  $t_1$  and  $t_2$  are compiler-generated temporary names.

→ Three - address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph.

→ Three - address code of the assignment statement  $a = b * -c + b * -c$  for syntax tree and dag is given below.



$$t_1 = -c$$

$$t_2 = b * t_1$$

$$t_3 = -c$$

$$t_4 = b * t_3$$

$$t_5 = t_2 + t_4$$

$$a = t_5$$

$$t_1 = -c$$

$$t_2 = b * t_1$$

$$t_5 = t_2 + t_2$$

$$a = t_5$$

Code for dag.

Code for Syntax Tree.

→ Each statement of the three-address code usually contains three addresses, two for the operand and one for the result.

### Types Of Three-Address Statements :

→ Three-address statements are akin to assembly code. Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of a three-address statement in the array holding intermediate code.

→ Common three-address statements used:

- ① Assignment statements of the form  $x := y \text{ op } z$ , where op is a binary arithmetic or logical operation.
- ② Assignment instructions of the form  $x := \text{op } y$ , where op is a unary operation.

Essential unary operations include:

- \* unary minus
- \* logical negation
- \* shift operators
- \* conversion operators.

- ③ Copy statements of the form  $x := y$  where the value of y is assigned to x.



- ④ The unconditional jump goto L. The three-address statement with label L is the next to be executed.
- ⑤ conditional jumps such as if  $x \text{ relop } y$  goto L. This instruction applies a relational operator ( $<, =, >, \text{etc}$ ) to  $x$  and  $y$ , and executes the statement with label L next to if  $x$  stands in relation relop to  $y$ . If not, the three-address statement following if  $x \text{ relop } y$  goto L is executed next, as in the usual sequence.
- ⑥ Param  $x$  and call(P,  $n$ ) for procedure calls and return  $y$ , where  $y$  representing a returned value is optional.

Their typical use is as the sequence of three-address statements.

Param  $x_1$

Param  $x_2$

Param  $x_n$

Call P,  $n$

generated as part of a call of the Procedure  $P\{x_1, x_2, \dots, x_n\}$ .

- ⑦ Indexed assignments of the form  $x[i] = y$  and  $x[i] := y$ . The 1<sup>st</sup> assignment of these sets  $x$  to the value in the location  $i$  memory units beyond location  $y$ . The statement  $x[i] := y$  sets the contents of the location  $i$  units beyond  $x$  to the value of  $y$ . In both these instructions  $x, y, \&i$  refer to data objects.

- ⑧ Address and Pointer assignments of the form  $x = \&y$ ,  $x := *y$ , and  $*x = y$ . The 1<sup>st</sup> of these sets the value of  $x$  to be the location of  $y$ .



In the statement  $x = *y$ , Presumably  $y$  is a pointer or a temporary whose r-value is a location.

## Syntax - Directed Translation into Three-Address Code

- When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree.
- The value of nonterminal  $E$  on the left side of  $E \rightarrow E_1 + E_2$  will be computed into a new temporary  $t$ .
- In general, the three-address code for  $id := E$  consist of code to evaluate  $E$  into some temporary  $t$ , followed by the assignment  $id.place := t$ .

PRODUCTION	SEMANTIC RULES
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place := E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel$ $gen(E.place := E_1.place + E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel$ $gen(E.place := E_1.place * E_2.place)$
$E \rightarrow -E_1$	$E.place := newtemp;$ $E.code := E_1.code \parallel gen(E.place :=$ $'uminus' E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow id$	$E.place := id.place;$ $E.code := ''$

**Fig:** Syntax-directed definition to produce three-address code for assignments.



→ The S-attributed definition in figure generates three-address code for assignment statements.

→ The synthesized attribute S.code represents the three-address code for the assignment S.

→ The nonterminal E has two attributes:

- ① E.place, the name that will hold the value of E.
- ② E.code, the sequence of three-address statements evaluating E.

→ The function newtemp returns a sequence of distinct names  $t_1, t_2, \dots$  in response to successive calls.

→ The notation  $\text{gen}(x := 'y' + 'z')$  represent the three-address statement  $x := y + z$ .

→ Expressions appearing instead of variables like  $x, y$  and  $z$  are evaluated when passed to gen, and quoted operators or operands, like  $'+'$ , are taken literally.

→ The semantic rules generating code for a while statement:

PRODUCTION	SEMANTIC RULES
$S \rightarrow \text{while } E$	$S.\text{begin} := \text{newlabel};$
$\text{do } S_1$	$S.\text{after} := \text{newlabel};$
	$S.\text{code} := \text{gen}(S.\text{begin} ':') \parallel E.\text{code} \parallel$
	$\text{gen}('if' E.\text{place} '=' '0' \text{'goto' } S.\text{after}) \parallel$
	$S_1.\text{code} \parallel \text{gen}('goto' S.\text{begin}) \parallel$
	$\text{gen}(S.\text{after} ':')$

**Fig** Semantic rules generating code for a while statement.



→ The Flow of control Statement  $S \rightarrow \text{while } E \text{ do } S_1$  is generated using new attributes  $S.\text{begin}$  and  $S.\text{after}$  to mark the first statement in the code for  $E$  and the statement following the code for  $S$ , respectively

$S.\text{begin}:$	$E.\text{code}$
	if $E.\text{Place} = 0$ goto $S.\text{after}$
	$S_1.\text{code}$
	goto $S.\text{begin}$
$S.\text{after}:$	.....

→ The function 'newlabel' that returns a new label every time is called.

→ ' $S.\text{after}$ ' become the label of the statement that comes after the code for the while statement.

→ Expressions that govern the flow of control may in general be boolean expressions containing relational and logical operators.

### Implementations of Three-Address Statements:

→ A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands.

→ Three such representations are:

1. Quadruples
2. Triples
3. Indirect Triples.



## Quadruples

→ A quadruple is a record structure with 4 fields, which we call OP, arg1, arg2, and result.

→ The op field contains an internal code for the Operator.

→ The three-address ~~code~~ statement  $x := y \text{ OP } z$  is represented by placing  $y$  in arg1,  $z$  in arg2, and  ~~$x$  in arg3~~  $x$  in result.

→ Statements with unary operators like  $x := -y$  or  $x := y$  do not use arg2.

→ operators like Param use neither arg2 nor result.

→ conditional and unconditional jumps put the target label in result.

	OP	arg 1	arg 2	arg 3
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Fig: Quadruples.

→ The quadruples in figure are for the assignment  $a := b * -c + b * -c$ .

→ They are obtained from the three-address code.

→ The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields.

→ If so, temporary names must be entered into the symbol table as they are created.



# Triples

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- The three-address statements can be represented by records with only three fields: op, arg 1 and arg 2.

	OP	arg 1	arg 2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Fig: Triples

- The fields arg 1 and arg 2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure.
- Since three fields are used, the intermediate code format is known as triples.
- Parenthesized numbers represent pointers into the triple structure, while symbol-table pointers are represented by the names themselves.
- The ternary operation like  $x[i] := y$  requires two entries in the triple structure. ↪ also  $x := y[i]$

	OP	arg 1	arg 2		OP	arg 1	arg 2
(0)	[ ] =	x	i	(0)	= [ ]	y	i
(1)	assign	(0)	y	(1)	assign	x	(0)

Fig: Triple Representation (a)  $x[i] := y$  (b)  $x := y[i]$



## Indirect Triples

→ Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called indirect triples.

→ consider an example, use an array statement to list pointers to triples in the desired order.

→ The triples for the statement  $a = b * -c + b * -c$  can be represented by indirect triples as follows:

	Statements		OP	arg 1	arg 2
(0)	(14)	(14)	uminus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(16)	(16)	uminus	c	
(3)	(17)	(17)	*	b	(16)
(4)	(18)	(18)	+	(15)	(17)
(5)	(19)	(19)	assign	a	(18)

**Fig:** Indirect triples representation of three-address statements.



# ASSIGNMENT STATEMENTS

Translation of assignment statement into 3-address code includes the access of names in the symbol table and how elements of arrays and records can be accessed.

## Names in the Symbol Table:

→ Names in the three-address code stood for pointers to their symbol-table entries.

→

$S \rightarrow id := E \quad \{ P := \text{lookup}(id.name);$   
if  $P \neq \text{nil}$  then

emit ( $P := E.place$ )  
else error }

$E \rightarrow E_1 + E_2 \quad \{ E.place := \text{newtemp};$

emit ( $E.place := E_1.place + E_2.place$ ) }

$E \rightarrow E_1 * E_2 \quad \{ E.place := \text{newtemp};$

emit ( $E.place := E_1.place * E_2.place$ ) }

$E \rightarrow -E_1$

$\{ E.place := \text{newtemp};$

emit ( $E.place := \ominus E_1.place$ ) }

$E \rightarrow (E_1)$

$\{ E.place := E_1.place \}$

$E \rightarrow id$

$\{ P := \text{lookup}(id.name);$

if  $P \neq \text{nil}$  then

$E.place := P$

else error }

**Fig:** Translation Scheme to Produce three-address code for assignments.

→ The translation scheme in above figure shows how such symbol-table entries can be found.



- The lexeme for the name represented by  $id$  is given by attribute  $id.name$ .
- Operation  $lookup(id.name)$  checks if there is an entry for this occurrence of the name in the symbol table. If so, a pointer to the entry is returned; otherwise,  $lookup$  returns  $nil$  to indicate that no entry was found.
- The Semantic actions in figure use Procedure emit to emit three-address statements to an output file, rather than building up code attributes for nonterminals.

### ■ Reusing Temporary Names:

- $newtemp$  generates a new temporary name each time a temporary is needed.
- Temporaries are used to hold intermediate values in expression calculation. Temporaries can be reused by changing  $newtemp$ .
- The temporaries generated during the syntax directed translation of the expression  $E \rightarrow E_1 + E_2$  has the following general form:

evaluate  $E_1$  into  $t_1$   
 evaluate  $E_2$  into  $t_2$

$t := t_1 + t_2$

- consider the assignment statement

$x = a * b + c * d - e * f.$

The sequence of three-address statements of the above statement that modify the  $newtemp$  is given below.



STATEMENT	VALUE OF C
	0
	1
$\$0 := a * b$	2
$\$1 := c * d$	1
$\$0 := \$0 + \$1$	2
$\$1 := e * f$	1
$\$0 := \$0 - \$1$	0
$z := \$0$	

**Fig** Three-address code with stacked temporaries.

→ Let  $C$  be the counter initialized to zero. Whenever a temporary name is used as an operand it is decremented by 1 and whenever a new temporary name is generated it is incremented by 1.

→ Temporary values are stored and load into the procedures data area at the run time.

### Addressing Array Elements:

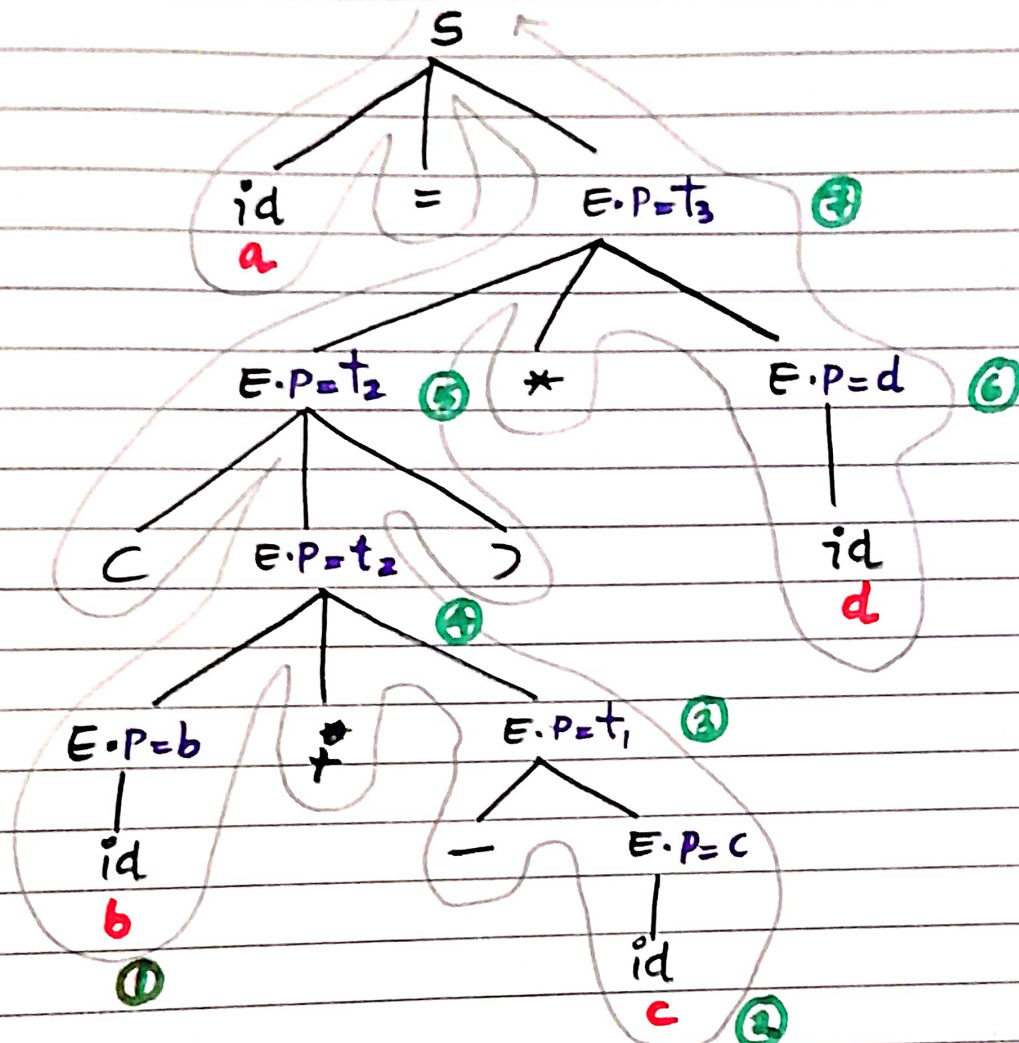
→ Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is  $w$ , then the  $i^{\text{th}}$  element of array  $A$  begins in location

$$\text{base} + (i - \text{low}) \times w$$

where 'low' is the lower bound on the subscript and 'base' is the relative address of  $A[\text{low}]$



EXAMPLE :  $a = (b + -c)d$



①  $S \rightarrow id = E$   
 $\hookrightarrow P = E.P \text{ place}$

②  $E \rightarrow - E_1$   
 $\hookrightarrow E.P = \text{new temp}$   
 $\text{o/p} \rightarrow E.P = - E_1$

③  $E \rightarrow E_1 + E_2$   
 $E \rightarrow E_1 * E_2$   
 $\hookrightarrow E.P = \text{new temp}$

$\text{o/p} \rightarrow E.P = E_1.P + E_2.P$   
 $E.P = E_1.P * E_2.P$

④  $E \rightarrow (E)$   
 $\hookrightarrow E.P = E_1.P$

3-address code:

$t_1 = -c$   
 $t_2 = b + t_1$   
 $t_3 = t_2 * d$   
 $a = t_3$



# BOOLEAN EXPRESSIONS

In Programming languages, boolean expressions have two binary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then, if-then-else, or while-do statements.

→ Boolean expressions are composed of the boolean operators (and, or, and not) applied to elements that are boolean variables or relational expressions.

→ Relational expressions are of the form  $E_1 \text{ relop } E_2$ , where  $E_1$  and  $E_2$  are arithmetic expressions.

## EXAMPLE:

$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id relop id} \mid$   
 $\text{true} \mid \text{false}.$

→ We use the attribute  $op$  to determine which of the comparison operators  $<, \leq, =, \neq, >$  or  $\geq$  is represented by  $rel\text{op}$ .

## ■ Methods Of Translating Boolean Expressions:

→ There are 2 principal methods of representing the value of a boolean expression

① To encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression.



→ Often 1 is used to denote true and 0 to denote false, although many other encodings are also possible.

② Implementing boolean expressions is by flow of control, that is, representing the value of a boolean expression by a position reached in a program.

This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

■ Syntax-Directed Definition to Produce 3-address code for booleans:

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1 \text{ or } E_2$	$E_1 \cdot \text{true} := E \cdot \text{true};$ $E_1 \cdot \text{false} := \text{newlabel};$ $E_2 \cdot \text{true} := E \cdot \text{true};$ $E_2 \cdot \text{false} := E \cdot \text{false};$ $E \cdot \text{code} := E_1 \cdot \text{code} \parallel \text{gen}(E_1 \cdot \text{false} ':') \parallel E_2 \cdot \text{code}$
$E \rightarrow E_1 \text{ and } E_2$	$E_1 \cdot \text{true} := \text{newlabel};$ $E_2 \cdot \text{true} := E \cdot \text{true};$ $E_1 \cdot \text{false} := E \cdot \text{false};$ $E_2 \cdot \text{false} := E \cdot \text{false};$ $E \cdot \text{code} := E_1 \cdot \text{code} \parallel \text{gen}(E_1 \cdot \text{true} ':') \parallel E_2 \cdot \text{code}$
$E_1 \rightarrow \text{not } E_2$	$E_1 \cdot \text{true} := E \cdot \text{false};$ $E_1 \cdot \text{false} := E \cdot \text{true};$ $E \cdot \text{code} := E_2 \cdot \text{code}$



$E \rightarrow (E_1)$	$E_1 \cdot \text{true} := E \cdot \text{true};$ $E_1 \cdot \text{false} := E \cdot \text{false};$ $E \cdot \text{code} := E_1 \cdot \text{code}$
-----------------------	--

$E \rightarrow \text{true}$	$E \cdot \text{code} := \text{gen}('goto' E \cdot \text{true})$
-----------------------------	---

$E \rightarrow \text{false}$	$E \cdot \text{code} := \text{gen}('goto' E \cdot \text{false})$
------------------------------	--

~~Ch~~  
01/10/2020



# MODULE VI

Code Optimization:

Principal sources of Optimization, Optimization of Basic blocks.

Code Generation:

Issues in the design of a code generator. The target machine, A simple code generator.

## CODE OPTIMIZATION

→ compilers should produce target code that is as good as can be written by hand. The code produced by straightforward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called Optimization.

→ compilers that apply code-improving transformations are called Optimizing Compilers.

## OPTIMIZATION

↓  
Machine Independent Optimization

→ The compiler takes in the intermediate-code & transforms a part of the code that does not involve any CPU registers and/or absolute memory locations.

↓  
Machine Dependent Optimization

→ It is done after the target code has been generated & when the code is transformed according to the target architecture.

→ It involves CPU registers & may have absolute memory



## ■ Code-Improving Transformations Criteria:

→ Simply stated, the best program transformations are those that yields the most benefit for the least effort.

→ The transformation provided by an optimizing compiler should have several properties:

① The transformation must preserve the meaning of a program.

↳ That is, an 'OPTIMIZATION' must not change the output produced by a program for a given input, or cause an error, such as a division by zero.

② A transformation must, on the average, speed up programs by a measurable amount.

↳ Sometimes we are interested in reducing the space taken by the compiled code, although the size of code has less importance than it once had.

↳ Of course, not every transformation succeeds in improving every program, and occasionally an "Optimization" may slow down a program slightly, as long as on the average it improves things.

③ The transformation must be worth the effort

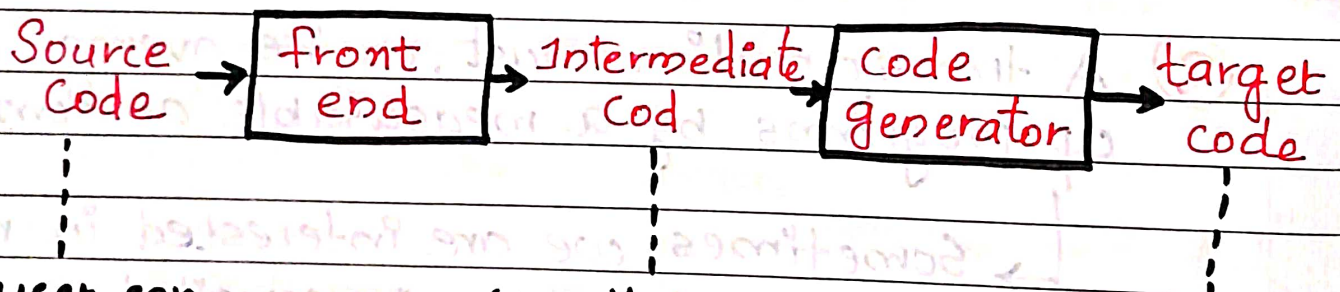
↳ It does not make sense for a compiler



writer to expand the intellectual effort to implement a code improving transformation and to have the compiler expand the additional time compiling source programs if this effort is not repaid when the target programs are executed.

## ■ Getting Better Performance :

→ Dramatic improvements in the running time of a program — Such as cutting the running time from a few hours to a few seconds — are usually obtained by improving the program at all levels, from the source level to the target level.



user can

→ Profile Pgm  
change algorithm  
transform loops.

compiler can

→ Improve loops  
Procedure calls  
address calculation

compiler can

→ Use registers  
Select instructions  
do peephole  
transformation.

**Fig:** Places for potential improvements by the user and the compiler.

→ At each level, the available options fall between the two extremes of finding a better algorithm and of implementing a given algorithm so that fewer operations are performed.



→ Algorithmic transformations occasionally produce spectacular improvements in running time.

→ Unfortunately, no compiler can find the best algorithm for a given program. Sometimes, however, a compiler can replace a sequence of operations by an algebraically equivalent sequence, and thereby reduce the running time of a program significantly. Such savings are more common when algebraic transformations are applied to programs in very-high level languages, eg:- query languages for databases.

**Example:** Effect of various code-improving transformation in QUICK SORT

```
void quicksort(m,n)
```

```
int m,n;
```

```
{
```

```
    int i,j;
```

```
    int v,x;
```

```
    if (n <= m) return;
```

```
    /* fragments begin here */
```

```
    i = m - 1; j = n; v = a[n];
```

```
    while (1) {
```

```
        do i = i + 1; while (a[i] < v);
```

```
        do j = j - 1; while (a[j] > v);
```

```
        if (i >= j) break;
```

```
        x = a[i]; a[i] = a[j]; a[j] = x;
```

```
    /* fragments ends here */
```

```
    quicksort(m,j);
```

```
    quicksort(i+1,n);
```

```
}
```

$x = a[i]; a[i] = a[j]; a[j] = x;$



→ Consider the three-address code for determining the value of  $a[i]$ , assuming that each array element takes 4 bytes;

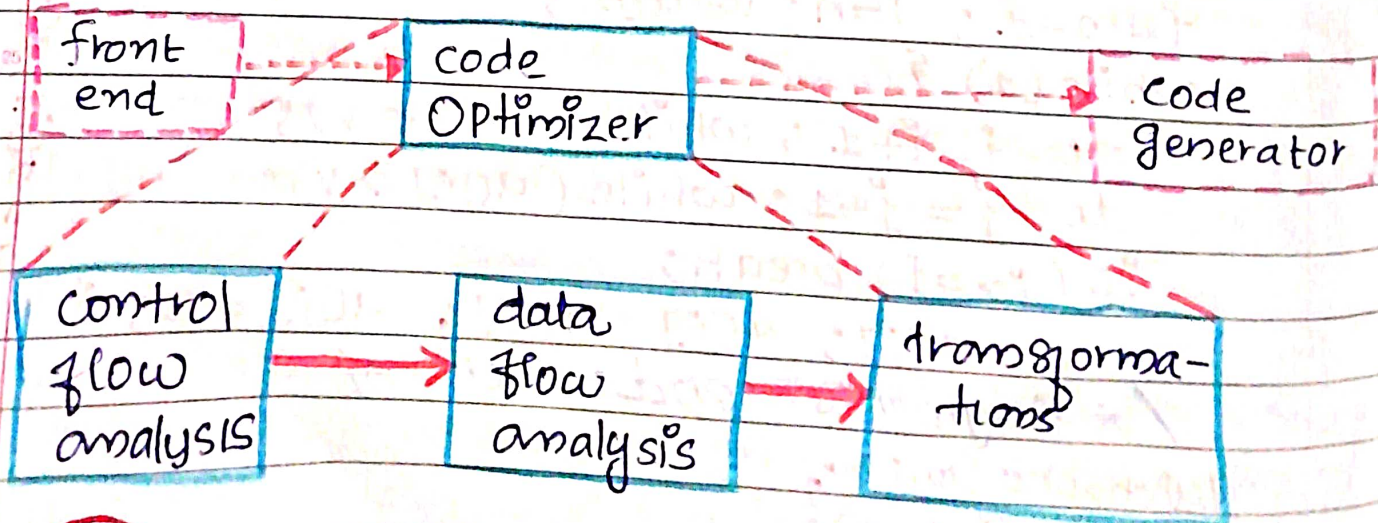
$t_1 := 4 * i; \quad t_2 := a[t_1];$

→ Naïve intermediate code will recalculate  $4 * i$  every time  $a[i]$  appears in the source Program, and the programmer has no control over the redundant address calculations, because they are implicit in the implementation of the language, rather than being explicit in the code written by the user.

→ At the level of the target machine, it is the compiler's responsibility to make good use of the machine's resources.

### ■ An Organization for an Optimizing Compiler:

The code-improvement phase consists of control-flow and data-flow analysis followed by the application of transformations.



**Fig:** Organization of the Code Optimizer.



## Advantages of Organization:

- ① The Operations needed to implement high-level constructs are made explicit in the intermediate code, so it is possible to optimize them.
- ② The intermediate code can be (relatively) independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for a different machine.

## Basic Blocks and Flow Graphs:

→ A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code generation algorithm.

### ① Basic Blocks:

→ A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

→ A three-address statement  $x := y + z$  is said to define  $x$  and to use (or reference)  $y$  and  $z$ . A name in a basic block is said to be live at a given point if its value is used after that point in the program, perhaps in another basic block.



## ② Flow Graphs:

→ The flow-of-control information to the set of basic blocks making up a program by constructing a directed graph called a flow graph.

→ The nodes of the flow graph are the basic blocks.

→ One node is distinguished as initial; it is the blocks whose leader is the 1<sup>st</sup> statement.

→ There is a directed edge from block  $B_1$  to block  $B_2$  if  $B_2$  can immediately follow  $B_1$  in some execution sequence, that is,

① there is a conditional or unconditional jump from the last statement of  $B_1$  to the 1<sup>st</sup> statement of  $B_2$ , or

②  $B_2$  immediately follows  $B_1$  in the order of the program, and  $B_1$  does not end in an unconditional jump.

$B_1$  is a predecessor of  $B_2$ , and  $B_2$  is a successor of  $B_1$ .

→ In the code optimizer, programs are represented by flow graphs, in which edges indicate the flow of control and nodes represent basic blocks.



$$(1) i \% = m - 1$$

$$(2) j \% = n$$

$$(3) t_1 \% = 4 * n$$

$$(4) v \% = a[t_1]$$

$$(5) i \% = i + 1$$

$$(6) t_2 \% = 4 * i$$

$$(7) t_3 \% = a[t_2]$$

$$(8) i \% \text{ if } t_3 < v \text{ goto}(5)$$

$$(9) j \% = j - 1$$

$$(10) t_4 \% = 4 * j$$

$$(11) t_5 \% = a[t_4]$$

$$(12) i \% \text{ if } t_5 > v \text{ goto}(9)$$

$$(13) i \% \text{ if } i \geq j \text{ goto}(23)$$

$$(14) t_6 \% = 4 * i$$

$$(15) x \% = a[t_6]$$

$$(16) t_7 \% = 4 * i$$

$$(17) t_8 \% = 4 * j$$

$$(18) t_9 \% = a[t_8]$$

$$(19) a[t_7] \% = t_9$$

$$(20) t_{10} \% = 4 * j$$

$$(21) a[t_{10}] \% = x$$

$$(22) \text{goto}(5)$$

$$(23) t_{11} \% = 4 * i$$

$$(24) x \% = a[t_{11}]$$

$$(25) t_{12} \% = 4 * 1$$

$$(26) t_{13} \% = 4 * n$$

$$(27) t_{14} \% = a[t_{13}]$$

$$(28) a[t_{12}] \% = t_{14}$$

$$(29) t_{15} \% = 4 * n$$

$$(30) a[t_{15}] \% = x.$$

Fig: Three - Address code for fragment of Quick Sort.



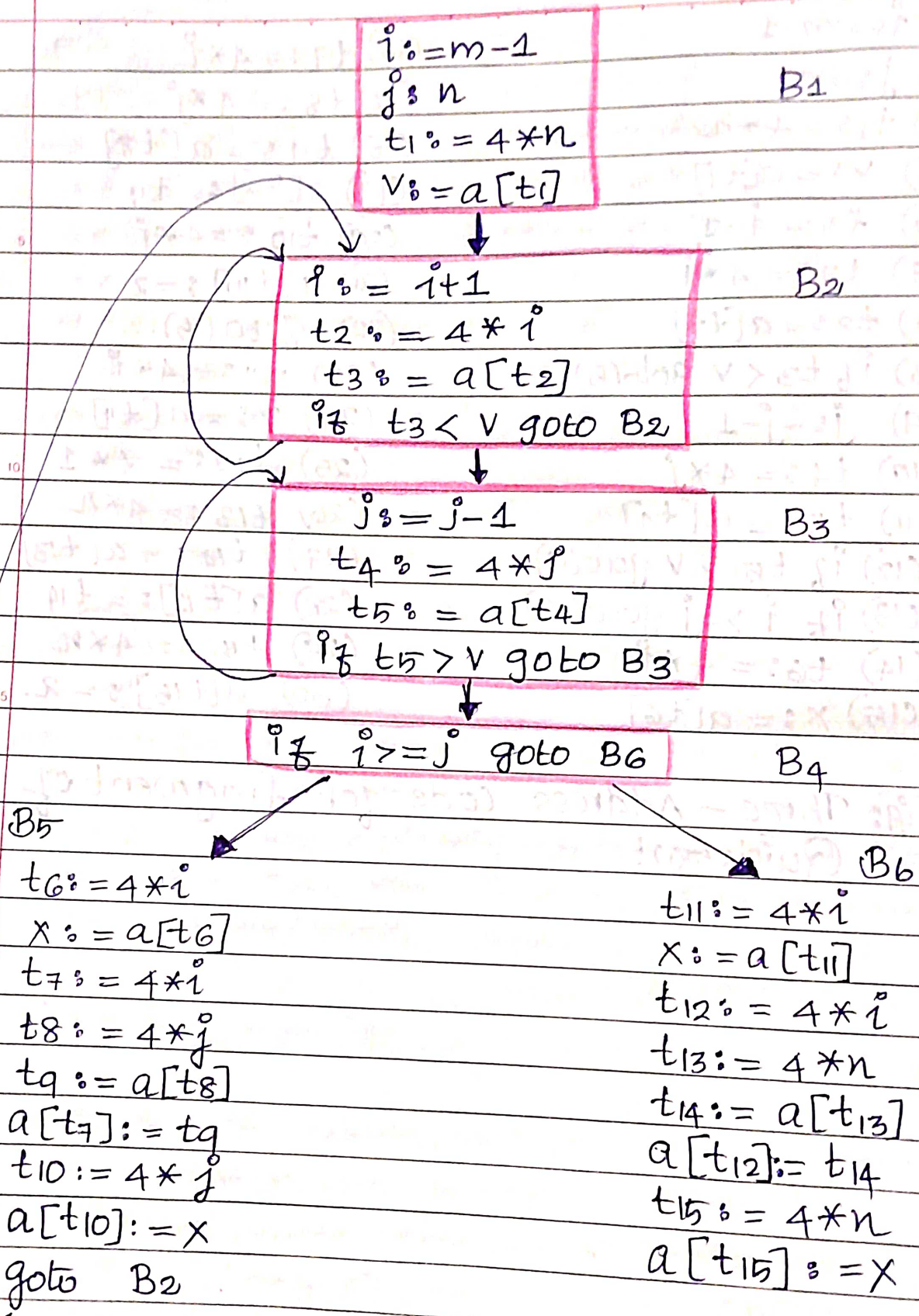


Fig: Flow graph.



A transformation of a program is called local if it can be performed by looking only at the statements in a basic blocks: otherwise, it is called global.

Many transformations can be performed at both local and global levels. Local transformations are usually performed first.

### Function — Preserving Transformations:

→ There are a number of ways in which a compiler can improve a program without changing the function it computes.

→ EXAMPLES:

- ① Common Subexpression Elimination.
- ② Copy Propagation.
- ③ Dead-code elimination.
- ④ Constant folding.

### Common Subexpressions

→ An occurrences of an expression  $E$  is called a "Common Subexpression" if  $E$  was previously computed, and the values of variables in  $E$  have not changed since the previous computation.

→ We can avoid recomputing the expression if we can use the previously computed value.

→ For example, the assignments to  $t_7$  and  $t_{10}$  have the common subexpressions  $4*i$  and  $4*j$ , respectively on the right side of the figure (c)



B5

```

t6 := 4 * i
x := a[t6]
t7 := 4 * i
t8 := 4 * j
t9 := a[t8]
a[t7] := t9
t10 := 4 * j
a[t10] := x
goto B2

```

B5

```

t6 := 4 * i
x := a[t6]
t8 := 4 * j
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto B2

```

(a) Before

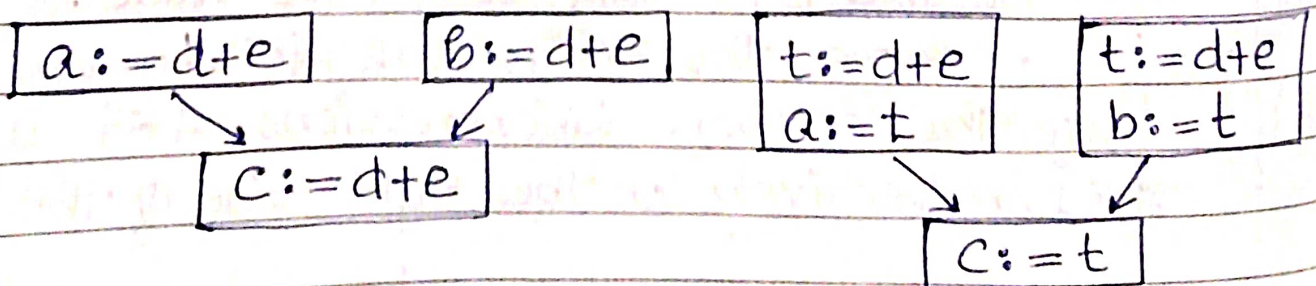
(b) After

(Fig) Local Common Subexpression elimination.

- They have been eliminated in fig(b), by using  $t_6$  instead of  $t_7$  and  $t_8$  instead of  $t_{10}$ .
- This change is what would result if we reconstructed the intermediate code from the DAG for the basic block.

## Copy Propagation

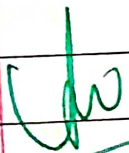
- Block B5 can be further improved by eliminating 'x' using 2 new transformations.
- One concerns assignments of the form " $f := g$ " called "Copy Statements" or "Copies" for short.



(Fig) Copies introduced during common subexpression elimination.

## Dead-Code Elimination

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
- A related ~~data~~ idea is dead or useless code, statements that compute values that never get used.
- While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformation.
- 

15  
  
13/10/2020



## ANTLR

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.

ANTLR takes as input a grammar that specifies a language and generates as output source code for a recognizer of that language. While Version 3 supported generating code in the programming languages Ada95, ActionScript, C, C#, Java, JavaScript, Objective-C, Perl, Python, Ruby, and Standard ML, the current release at present only targets Java, C#, C++, JavaScript, Python, Swift, and Go. A language is specified using a context-free grammar expressed using Extended Backus–Naur Form (EBNF).

ANTLR can generate lexers, parsers, tree parsers, and combined lexer-parsers. Parsers can automatically generate parse trees or abstract syntax trees, which can be further processed with tree parsers. ANTLR provides a single consistent notation for specifying lexers, parsers, and tree parsers.

By default, ANTLR reads a grammar and generates a recognizer for the language defined by the grammar (i.e., a program that reads an input stream and generates an error if the input stream does not conform to the syntax specified by the grammar). If there are no syntax errors, the default action is to simply exit without printing any message. In order to do something useful with the language, actions can be attached to grammar elements in the grammar. These actions are written in the programming language in which the recognizer is being generated. When the recognizer is being generated, the actions are embedded in the source code of the recognizer at the appropriate points. Actions can be used to build and check symbol tables and to emit instructions in a target language, in the case of a compiler.

Other than lexers and parsers, ANTLR can be used to generate tree parsers. These are recognizers that process abstract syntax trees, which can be automatically generated by parsers. These tree parsers are unique to ANTLR and help processing abstract syntax trees.

ANTLR 3 and ANTLR 4 are free software, published under a three-clause BSD License. Prior versions were released as public domain software. Documentation, derived from Parr's book *The Definitive ANTLR 4 Reference*, is included with the BSD-licensed ANTLR 4 source.

Various plugins have been developed for the Eclipse development environment to support the ANTLR grammar, including ANTLR Studio, a proprietary product, as well as the "ANTLR 2" and "ANTLR 3" plugins for Eclipse hosted on [SourceForge](#)