POORNIMA

COLLEGE OF ENGINEERING

Name of faculty	**Reena Sharma**
Class- VII SEM	**B.Tech – VII SEM**
Branch	**Computer Science & Engineering**
Course Code	**7CS5A**
Course Name	**Compiler Construction**
Session	**2019-2020**

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA COLLEGE OF ENGINEERING, JAIPUR
DEPARTMENT OF COMPUTER ENGINEERING

Vision & Mission of Poornima College of Engineering

Vision

To create knowledge based society with scientific temper, team spirit and dignity of labor to face the global competitive challenges.

Mission

To evolve and develop skill based systems for effective delivery of knowledge so as to equip young professionals with dedication and commitment to excellence in all spheres of life

Vision & Mission of Department of Computer Engineering

Vision

Evolve as a centre of excellence with wider recognition and to adapt the rapid innovation in Computer Engineering.

Mission

1. To provide a learning-centered environment that will enable students and faculty members to achieve their goals empowering them to compete globally for the most desirable careers in academia and industry.

2. To contribute significantly to the research and the discovery of new arenas of knowledge and methods in the rapid developing field of Computer Engineering.

3. To support society through participation and transfer of advanced technology from one sector to another.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA COLLEGE OF ENGINEERING, JAIPUR

DEPARTMENT OF COMPUTER ENGINEERING

PROGRAM EDUCATIONAL OBJECTIVES (PEO'S)

1. Graduates will work productively as skillful engineers playing the leading roles in multifaceted teams
2. Graduates will identify the solutions for challenging issues inspiring the upcoming generations leading them towards innovative, creative, and sophisticated technologies.
3. Graduates will implement their pioneering ideas practically to create products and the feasible solutions of research oriented problems

PROGRAM OUTCOMES (POs)

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.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

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 (PSOs)

1. The ability to understand and apply knowledge of mathematics, system analysis & design, Data Modelling, Cloud Technology, and latest tools to develop computer based solutions in the areas of system software, Multimedia, Web Applications, Big data analytics, IOT, Business Intelligence

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

and Networking systems.

2. The ability to understand the evolutionary changes in computing, apply standards and ethical practices in project development using latest tools & Technologies to solve societal problems and meet the challenges of the future.

3. The ability to employ modern computing tools and platforms to be an entrepreneur, lifelong learning and higher studies.

POORNIMA COLLEGE OF ENGINEERING, JAIPUR

DEPARTMENT OF COMPUTER ENGINEERING

MAPPING OF KEY PHRASES OF THE INSTITUTES MISSION STATEMENT WITH THE KEY PHRASES OF INSTITUTES VISION STATEMENT

(Institution Mission Vs Institute Vision)

Key Phrases of the Mission Statement of the Institute	**Key Phrases of the Vision Statement of the Institute**		
	IV_1: To create knowledge based society with scientific temper	**IV_2:** Team spirit	**IV_3:** To face the global competitive challenges
IM_1: Skill based systems for effective delivery of knowledge	√		√
IM_2: To equip young professionals with dedication		√	√
IM_3: Excellence in all spheres of life	√		√

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

MAPPING OF KEY PHRASES OF THE DEPARTMENTSVISION STATEMENT WITH THE KEY PHRASES OF INSTITUTES MISSION STATEMENT

(Department Vision Vs Institution Mission)

Key Phrases of the Vision Statement of the Department	**Key Phrases of the Mission Statement of the Institute**		
	IM_1: Skill Based Systems	**IM_2:** Delivery of Knowledge	**IM_3:** Excellence in all spheres of life
DV_1: Centre of Excellence	√	√	√
DV_2: Wider recognition	√	√	√
DV_3: Rapid innovation.	√	√	

MAPPING OF KEY PHRASES OF THE DEPARTMENTS MISSION STATEMENT WITH THE KEY PHRASES OF DEPARTMENTS VISION STATEMENT

(Department Mission Vs Department Vision)

Key Phrases of the Mission Statement of the Department	**Key Phrases of the Vision Statement of the Department**		
	DV_1: Centre of Excellence	**DV_2:** Wider recognition	**DV_3:** Rapid innovation.
DM_1 Learning-centered environment	√		√
DM_2: Research and Discovery	√		√

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

DM_3: Social Responsibility	√	√	√

MAPPING OF PEOS WITH KEY PHRASES OF DEPARTMENTS MISSION STATEMENT

(PEO Vs Department Mission)

PEOs	PEO Statements	Key Phrases of the Mission of the Department		
		DM_1: Learning-centered environment	**DM_2:** Research and Discovery	**DM_3:** Social Responsibility
PEO 1	Graduates will work productively as skillful engineers playing the leading roles in multifaceted teams	√	√	
PEO 2	Graduates will identify the solutions for challenging issues inspiring the upcoming generations leading them towards innovative, creative, and sophisticated technologies	√		√
PEO 3	Graduates will implement their pioneering ideas practically to create products and the feasible solutions of research oriented problems.	√	√	

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

MAPPING OF PSO WITH KEY PHRASES OF DEPARTMENTS MISSION STATEMENT

(PSO Vs Department Mission)

PSO Statements	Key Phrases of the Mission Department		
	DM_1: Learning-centered environment	**DM_2:** Research and Discovery	**DM_3:** Social Responsibility
PSO1: The ability to understand and apply knowledge of mathematics, system analysis & design, Data Modelling, Cloud Technology, and latest tools to develop computer based solutions in the areas of system software, Multimedia, Web Applications, Big data analytics, IOT, Business Intelligence and Networking systems	√	√	
PSO2: The ability to understand the evolutionary changes in computing, apply standards and ethical practices in project development using latest tools & Technologies to solve societal problems and meet the challenges of the future.	√	√	√
PSO3: The ability to employ modern computing tools and platforms to be an entrepreneur, lifelong learning and higher studies.	√		

MAPPING OF PEO WITH KEY PHRASES OF PO (PEO Vs PO)

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

PO/PEO	1. Engineering knowledge:	2. Problem analysis:	3. Design/development of solutions:	4. Conduct investigations of complex problems:	5. Modern tool usage:	6. The engineer and society:	7. Environment and sustainability:	8. Ethics:	9. Individual and team work:	10. Communication:	11. Project management and finance:	12. Life-long learning:
PEO 1: Graduates will work productively as skillful engineers playing the leading roles in multifaceted teams	√	√	√	√	√		√	√	√	√	√	√
PEO 2: Graduates will identify the solutions for challenging issues inspiring the upcoming generations leading them towards innovative, creative, and sophisticated technologies		√	√	√		√	√	√				

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

PEO 3: Graduates will implement their pioneering ideas practically to create products and the feasible solutions of research oriented problems	√	√	√	√	√			√	√	√	√	√

MAPPING OF PSO WITH PEO (PSO Vs PEO)

PEO / PSO	**PSO1:** The ability to understand and apply knowledge of mathematics, system analysis & design, Data Modelling, Cloud Technology, and latest tools to develop computer based solutions in the areas of system software, Multimedia, Web Applications, Big data analytics, IOT, Business Intelligence and Networking systems	**PSO2:** The ability to understand the evolutionary changes in computing, apply standards and ethical practices in project development using latest tools & Technologies to solve societal problems and meet the challenges of the future.	**PSO3:** The ability to employ modern computing tools and platforms to be an entrepreneur, lifelong learning and higher studies.
PEO 1: Graduates will work productively as skillful engineers playing the leading roles in multifaceted teams	√	√	√
PEO 2: Graduates will identify the solutions for challenging issues inspiring the upcoming generations leading them towards innovative, creative,	√	√	√

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

and sophisticated technologies			
PEO 3: Graduates will implement their pioneering ideas practically to create products and the feasible solutions of research oriented problems		√	√

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA COLLEGE OF ENGINEERING, JAIPUR

DEPARTMENT OF COMPUTER ENGINEERING

PERSON'S TIME TABLE

Faculty: **Reena Sharma**

Day/ Period	I 8.00-9.00	II 9.00-10.00	III 10.00-11.00	11.00-11.40	IV 11.40-12.40	V 12.40-1.40	VI 1.40-2.30
MON	CC 7CS-B,CF-03				PTS 7CS-A2 AF07-B		
TUE						CC 7CS-B,AF-04	
WED	PROJECT LAB 7CS-B3, AF-8A					CLOUD RS CF-05	
THU		7CS1A RS AF-04			CC 7CS-B, AF-04		PTS 7CS-A2 AF07-B
FRI						7CS1 RS CF-05	
SAT	CGMT LAB RS,5CS B1, AF07-A						

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA COLLEGE OF ENGINEERING, JAIPUR

DEPARTMENT OF COMPUTER ENGINEERING

RAJASTHAN TECHNICALUNIVERSITY, KOTA

SYLLABUS

4 Year - VII Semester: B. Tech. (Computer Science & Engineering)

7CS5A : Compiler Construction

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

6CE4A: DESIGN OF CONCRETE STRUCTURES – I (L-3)

B.Tech. (Civil) 6th semester **Max. Marks: 80**
Exam Hours: 3

UNIT	CONTENTS	CONTACT HOURS
I	Objective and fundamental concepts of design of RC members, Types and function of reinforcement. Introduction to various related IS codes. Design Philosophies: Working stress, ultimate strength and limit states of design. Analysis and Design of singly reinforced rectangular beam section for flexure using Working Stress Method and Limit State Method.	8
II	Analysis and design of singly reinforced, flanged beams and doubly reinforced rectangular beams for flexure using Limit State Method. Limit state of serviceability for deflection, control of deflection as per codal provisions of empirical coefficients.	8
III	**Limit state of collapse in shear**: analysis and design of prismatic sections for shear using LSM. **Limit state of collapse in bond**: concept of bond stress, anchorage length and development length, curtailment of reinforcement as per codal provisions.	8
IV	Analysis and design of one way and two way slabs using LSM and Flat slab using direct design method as per code, Detailing of reinforcement.	8
V	Columns: Short and long columns, their structural behaviour. Analysis and design of axially loaded short columns, using LSM. Analysis of uniaxially eccentrically loaded short columns. Introduction to Pu-Mu interaction curves and their use for eccentrically loaded columns. Design of Column Footings: Analysis and design of Isolated column footing and combined footing for two columns (without central beam) for axial loads using LSM.	8
	TOTAL	**40**

Unit No.	Contents	Hours
1	Introduction : Objective , scope , outcome of course	1
II	Compiler, Translator, Interpreter definition, Phase of compiler introduction to one pass & Multipass compilers, Bootstrapping, Review of Finite automata lexical analyzer, Input, buffering, Recognition of tokens, Idea about LEX: A lexical analyzer generator, Error handling	6

III	Review of CFG Ambiguity of grammars, Introduction to parsing. Bottom up parsing Top down parsing techniques, Shift reduce parsing, Operator precedence parsing, Recursive descent parsing predictive parsers. LL grammars & passers error handling of LL parser. LR parsers, Construction of SLR, Conical LR & LALR parsing tables, parsing with ambiguous grammar. Introduction of automatic parser generator: YACC error handling in LR parsers.	10
IV	Syntax directed definitions; Construction of syntax trees, L-attributed definitions, Top down translation. Specification of a type checker, Intermediate code forms using postfix notation and three address code, Representing TAC using triples and quadruples, Translation of assignment statement. Boolean e xpression and control structures.	10
V	Storage organization, Storage allocation, Strategies, Activation records, Accessing local and non local names in a block structured language, Parameters passing, Symbol table organization, Data structures used in symbol tables	8
VI	Definition of basic block control flow graphs, DAG representation of basic block, Advantages of DAG, Sources of optimization, Loop optimization, Idea about global data flow analysis, Loop invariant computation, Peephole optimization, Issues in design of code generator, A simple code generator, Code generation from DAG	7
	Total	42

POORNIMA COLLEGE OF ENGINEERING, JAIPUR

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Campus: Poornima College of Engineering
Course: B.Tech. **Year/ Section – 4 /B**
Name of Faculty: Reena Sharma **Name of Subject :Compiler Construction** **Code: 7CS5A**

COURSE PLAN (Deployment)

Lecture No.	Lect No.	Topics , Problems , Applications	CO	Target date of coverage	Actual Date of Coverage	Ref. Book / Journal with Page No.
1.	1	Storage organization, Storage allocation, Strategies		30-09-19	30-09-19	Alfred V. Aho , Ravi Sethi , Ullman
2.	**2**	Activation records, Accessing local and non local names in a block structured language, Parameters passing,		1-10-19	1-10-19	Alfred V. Aho , Ravi Sethi , Ullman
3.	**3**	Symbol table organization, Data structures used in symbol tables.		3-10-19	1-10-19	Alfred V. Aho , Ravi Sethi , Ullman
4.	**4**	Definition of basic block control flow graphs, DAG representation of basic block, Advantages of DAG		7-10-19	3-10-19	Alfred V. Aho , Ravi Sethi , Ullman
5.	**5**	Sources of optimization, Loop optimization, Idea about global data flow analysis, Loop invariant computation		10-10-19	10-10-19	Alfred V. Aho , Ravi Sethi , Ullman
6.	**6**	Peephole optimization, Issues in design of code generator,		14-10-19	14-10-19	Alfred V. Aho , Ravi Sethi , Ullman
7.	**7**	A simple code generator, Code generation from DAG		15-10-19	15-10-19	Alfred V. Aho , Ravi Sethi , Ullman

Reference book :

Aho, Ullman and Sethi: Compilers, Addison

Holub, Compiler Design in C,PHI

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA COLLEGE OF ENGINEERING, JAIPUR
DEPARTMENT OF Computer Science &Engineering

Campus: Poornima College of Engineering
Course: B.Tech. **Year/ Section – 4 / B**
Name of Faculty: Reena Sharma **Name of Subject :Compiler Construction** **Code: 7CS5A**

COURSE PLAN (Zero Lecture)

Session: 2019-20(Odd Sem.)

1). a) Name of subject with Code : Compiler Construction (7CS5A))
b) Compulsory/Elective : Compulsory

2). Self-Introduction:
a). ***Name:***Reena Sharma
b). ***Qualification:***B.Tech, M.Tech
c). ***Designation:***Assistant Professor
d). ***Research Area:*** Image Processing
e). ***E-mail Id:*** *reena.sharma@poornima.org*; +91- 8233912546
f). **Other details:**.

1. Areas of proficiency/expertise:
1.1 Subjects taken:
1.1.1 DCCN
1.1.2 Operating System
1.1.3 Software Engineering
1.1.4 Compiler Construction
1.1.5 Network Programming
1.1.6 Artificial Intelligence
1.1.7 Discrete Mathematics & Structure
1.1.8 Fundamental of C
1.1.9 OOPs
1.2 Laboratories Taken
1.2.1. C Programming Lab
1.2.2 Internet Programming lab
1.2.3 C++ Lab
1.2.4 Socket Programming Lab
1.2.5 Mobile Application Development Lab
1.2.6 Computer Graphics Lab

1.4 Academic Proficiency
1.4.1 English
1.4.2 Hindi

1.5 Book Authored
None

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

1.6 Papers published in National/ International Conferences/ Journals

S. No.	Authors name	Paper Title	journal name	DOI	page nos.	year of publication	indexing
1							
2							

3).Introduction of Students:

a*). Records of students in 7th semester*

Sr. No.	Average result	Name of student scored highest marks	Marks 60% above (No. of students)	Marks between 40%-60% (No. of students)	English Medium Students (No.)	Hindi Medium Students (No.)	No. of Hostellers	No. of Day Scholar

b.) Semester Result Target: 100%

c*). Name of 05 best students based on previous results:*
1.Suraj,
2. Jatin Kumawat ,
3. Himanshu Aggarwal,
4. Suraj Singh Deora ,
5. Bharti Singh

4). Instructional Language: - 100%English

5). Introduction to subject: -
***a) Relevance to Branch*:**

The main purpose of this subject is to study the compiler which translates the code written in one language to some other language without changing the meaning of the program

a) *Relevance to Society*:

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Design of compiler in required format , we can apply our idea in application and use with any other application.

b). Relevance to Self:

This subject has its own importance, for the personal growth this is must to have the knowledge the Compiler Design. By the design of many types of compiler we can generate any type of target language and we can use in many applications..

c). Connection with previous year and next year:

In previous semester Theory of Computation , is partially related to this subject, for further study good concept of this subject, will be very helpful in further related subjects which will come in preceeding years.

d). Connection with Laboratory:

This subject is implemented practically in the compiler design lab by performing different practical and the results obtained theoretically can be verified. It also gives idea that how to implement tokens .

b). ABC analysis (RGB method) of unit & topics:

S.no.	Category A	Category B	Category C	Preparedness for "A" topics
1	Finite automata.	Bootstrapping, Input, buffering, Recognition of tokens, Idea about LEX: A lexical analyzer generator, Error handling.	Compiler, Translator, Interpreter definition, Phase of compiler introduction to one pass & Multipass compilers	PPT
2.	Bottom up parsing Top down parsing techniques, Shift reduce parsing, Operator precedence parsing, Recursive descent parsing predictive parsers. LL grammars & passers error handling of LL parser. LR parsers, Construction of SLR,	Introduction of automatic parser generator: YACC error handling in LR parsers.	Introduction to parsing	Video lecture , ppt

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

	Conical LR & LALR parsing tables, parsing with ambiguous grammar.			
3.	Intermediate code forms using postfix notation and three address code, Representing TAC using triples and quadruples, Translation of assignment statement.	Construction of syntax trees, L-attributed definitions, Top down translation. Specification of a type checker	Boolean e xpression and control structures.	
	Activation records , Symbol table organization, Data structures used in symbol tables	Accessing local and non local names in a block structured language, Parameters passing,	Storage organization, Storage allocation, Strategies	**Video Lecture, PPT**
	Loop optimization, Idea about global data flow analysis, Loop invariant computation, Peephole optimization, Code generation from DAG	Definition of basic block control flow graphs, DAG representation of basic block, Advantages of DAG, Sources of optimization	Issues in design of code generator, A simple code generator	**Video Lecture**

7). Books/ Website/Journals & Handbooks/ Association & Institution

S.N.	**Title of Book**	**Authors**	**Publisher**
Text Books			
T1	**Compiler : principles and tools**	**Aho, Ullman , sethi**	
1	**http://nptel.ac.in/courses/105106142/**	**Web**	
2	**http://nptel.ac.in/courses/105104131/**	**Videos**	

.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

OUTCOMES:

• Able to think and develop a new mobile application.
• As Able to take any new technical issue related to this new paradigm and come up with a solution(s).
• Able to develop new ad hoc network applications and/or algorithms/protocols.
• Able to understand & develop any existing or new protocol related to the mobile environment

OBJECTIVE:

• To make the student understand the concept of the mobile computing paradigm, its novel applications, and limitations.
• To understand the typical mobile networking infrastructure through a popular GSM protocol
•Understand the issues and solutions of various layers of mobile networks, namely MAC layer, Network Layer & Transport Layer
• To understand the database issues in mobile environments & data delivery models.
• Understand the ad hoc networks and related concepts.
• To understand the platforms and protocols used in the mobile environment.

8). Syllabus Deployment: -

a). Total weeks available for academics (excluding holidays) as per Poornima Foundation calendar-

Semester	VI
No. of Working days available(Approx.)	
No. of Weeks (Approx.)	

- Total weeks available for special activities (as mentioned below)- 1 weeks (Approx.)

b). Special Activities (To be approved by HOD & Dean & must be mentioned in deployment):

- Open Book Test- Once in a semester
- Class Test- After completion of Unit.
- Quiz - Once in a semester
- Special Lectures (SPL)- Minimum 10% of total no. of lectures including following
    i. Smart Class by the faculty, who is teaching the subject
    ii. SPL by expert faculty at PGC level
    iii. SPL by expert from industry/academia (other institution)
- Revision classes (Solving Important Question Bank):- 1 class before Mid Term and 2 classes before End Term Exam

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

- Video Lecture & PPT

c). Lecture schedule per week

i). University scheme (L+T+P) = 3+0+0

ii). PGC scheme (L+T+P) = 4+0+0

S. No.	Name of Unit	No. of lectures	Broad Area	Degree of difficulty (High/Medium/Low)	No. of Question in RTU Exam.	Text/ Reference books
1.	Zero Lecture	1	-	-	-	Compilers: principles, Techniques,and tools -
2.	Introduction	7	Introduction of Subject	Low	2	Compilers: principles, Techniques,and tools -
3.	Review of CFG Ambiguity of grammars	9	Design	High	5	Compilers: principles, Techniques,and tools -
4.	Syntax directed definitions	7	Analysis and design	High	4	Compilers: principles, Techniques,and tools -
5.	Storage Organization	9	Analysis and Design	Medium	2	Compilers: principles, Techniques,and tools -

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

6	Definition of basic block control graph	10	Analysis and Design	High ,Medium	4	Compilers: principles, Techniques,and tools -

d). Introduction & Conclusion: Each subject, unit and topic shall start with introduction & close with conclusion. In case of the subject, it is Zero lecture.

e). Time Distribution in lecture class: - Time allotted: 60 min.

i. First 5 min. should be utilized for paying attention towards students who were absent for last lecture or continuously absent for many days + taking attendance by calling the names of the students and also sharing any new/relevant information.
ii. Actual lecture delivery should be of 50 min.
iii. Last 5 min. should be utilized by recapping/ conclusion of the topic. Providing brief introduction of the coming up lecture and suggesting portion to read.
iv. After completion of any Unit/Chapter a short quiz should be organized.
v. During lecture student should be encouraged to ask questions**.**

9). Tutorial and Home assignment: - An essential component of Teaching- Learning process in Professional Education.

Objective: - To enhance the recall mechanism.
To promote logical reasoning and thinking of the students.
To interact personally to the students for improve numerical solving ability.

a). *Tutorial processing:* - Not applicable for the subject**.**

b). *Home assignment shall comprise of two parts:*

Part (i) Minimum essential questions, which are to be solved and submitted by all within specified due date.

Part (ii) other important questions, which may also be solved and submitted for examining and guidance by teacher.

10). University Examination systems: -

Sr. No.	Name of the Exam	Max. Marks	% of passin	Nature of paper Theory	Syllabus coverage	Conducted by

			g marks		**(in %)**	
1.	1st mid term	10	10%	T	60%	College
2.	2nd mid term	10	10%	T	40%	College
3.	RTU main exam	80	80%	T	100%	RTU

Place : Jaipur

Reena Sharma
Assistant Professor

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA COLLEGE OF ENGINEERING, JAIPUR
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Campus: Poornima College of Engineering
Course: B.Tech. **Year/ Section -4/ B**
Name of Faculty: Reena Sharma **Name of Subject :Compiler Construction** **Code: 7CS5A**

COURSE PLAN –BLOWN UP

SNo.	TOPIC AS PER SYLLABUS	BLOWN UP TOPICS (up to 10 Times Syllabus)
1.	Zero Lecture	Introduction to the subject and its significance.
2.	**Introducton**	Compiler, Translator, Interpreter definition, Phase of compiler introduction to one pass & Multipass compilers, Bootstrapping, Review of Finite automata lexical analyzer, Input, buffering, Recognition of tokens, Idea about LEX: A lexical analyzer generator, Error handling
3.	Review of CFG Ambiguity of grammars	Introduction to parsing. Bottom up parsing Top down parsing techniques, Shift reduce parsing, Operator precedence parsing, Recursive descent parsing predictive parsers. LL grammars & passers error handling of LL parser. LR parsers, LR(0)items, Construction of SLR, LR(1) items , Conical LR & LALR parsing tables, parsing with ambiguous grammar. Introduction of automatic parser generator: YACC error handling in LR parsers.
4.	Syntax directed definitions	Construction of syntax trees, L-attributed definitions, Top down translation. Specification of a type checker, Intermediate code forms using postfix notation and three address code, Representing TAC using triples and quadruples, Translation of assignment statement. Boolean expression and control structures.
5.	Storage organization	Storage allocation, Strategies, Activation records, Accessing local and non local names in a block structured language, Parameters passing, Symbol table organization, Data structures used in symbol tables.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

6.	Definition of basic block control flow graphs	DAG representation of basic block, Advantages of DAG, Sources of optimization, Loop optimization, Idea about global data flow analysis, Loop invariant computation, Peephole optimization, Issues in design of code generator, A simple code generator, Code generation from DAG

Reena Sharma

Assistant Professor

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA COLLEGE OF ENGINEERING, JAIPUR

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Campus: Poornima College of Engineering
Course: B.Tech. **Year/ Section -4/ B**
Name of Faculty: Reena Sharma **Name of Subject :Compiler Construction** **Code: 7CS5A**

COURSE OUTCOMES

After completion of course

7CS05A.1 (CO1): Explain the concepts and different phases of compilation with compile time error handling..

7CS05A.2 (CO2) Represent language tokens using regular expression , context free grammer and finite automata and design lexical analyzer for a language

7CS05A.3 (CO3) Design syntax directed translation scheme for a given context free grammer.

7CS05A.4(CO4)**:** Implement various storage allocation strategies , parameter passing and data structure using symbol table.

MAPPING OF CO WITH PO AND PSO

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3												3		
CO2		3											2		

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

CO3				3									3		
CO4			3										1		

a.) PO Strongly Mapped:

PO1: **Apply** the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: **Identify**, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: 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

PO4: 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.

a. PO Moderately Mapped:

Not Applicable

b. PO Low Mapped:

Not Applicable

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

c. PSO Strongly Mapped:

PSO1: The ability to understand and apply knowledge of mathematics, system analysis & design, **Data Modelling**, Cloud Technology, and latest **tools to develop computer based solutions** in the areas of system software, Multimedia, Web Applications, Big data analytics, IOT, Business Intelligence and Networking systems

PSO Moderately Mapped:

PSO1: The ability to understand the evolutionary changes in computing, apply standards and ethical practices in project development using latest tools & Technologies to solve societal problems and meet the challenges of the future

PSO Low Mapped:

PSO1: The ability to employ modern computing tools and platforms to be an entrepreneur, lifelong learning and higher studies.

RULES FOR CO/LO ATTAINMENT LEVELS: (TARGETS)

Course Category	Level3	Level2	Level1
A	60 % of students getting > 60% marks	50-60 % of students getting > 60% marks	40-50 % of students getting > 60% marks

END TERM RTU COMPONENT: CO ATTAINMENT LEVELS

Course Category	Level3	Level2	Level1
A	50 % of students getting > 60% marks	40-50 % of students getting > 60% marks	30-40 % of students getting > 60% marks

S. No.	Course Type	Attainment	Attainment	Attainment

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

		Level=1	Level=2	Level=3
1	**Theory Courses** Mid Semester Exams	40-50 % of students getting > 60% marks	50-60 % of students getting > 60% marks	60 % of students getting > 60% marks
2	**Theory Courses** University Exam	30-40 % of students getting > 60% marks	40-50 % of students getting > 60% marks	50 % of students getting > 60% marks
3	**Assignments/Unit Test**	40-50 % of students getting > 60% marks	50-60 % of students getting > 60% marks	60 % of students getting > 60% marks
4	**Any other**	NA	NA	NA

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA COLLEGE OF ENGINEERING, JAIPUR

DEPARTMENT OF COMPUTER ENGINEERING

Campus: Poornima College of Engineering
Course: B.Tech. **Year/ Section – 4/B**
Name of Faculty: Reena Sharma **Name of Subject :Compiler Construction** **Code: 7CS5A**

CO WISE ASSESSMENT ACTIVITIES (AS MENTIONED IN SESSION PLAN)

CO	Mid 2
CO1	Y
CO2	Y
CO3	Y
CO4	Y

CO-PO/PSO MAPPING AND TARGETS

CO	PO												Avg.	PSO		
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	CO Targets	PSO1	PSO2	PSO3
CO1	3												3	3		
CO2		3											3	2		
CO3				3									3	3		
CO4			3										3	1		

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA COLLEGE OF ENGINEERING, JAIPUR

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Campus: Poornima College of Engineering **Class/Section: IV**
Course: B.Tech. **Year/ Section - B**
Name of Faculty: Reena Sharma **Name of Subject :compiler construction** **Code: 7CS5A**

ACTIVITY WISE ASSESSMENT TOOLS

Sr. No.	Activity	Assessment Method	Tools	Weightage Marks	Recommendation
1.	MidTerm2	Direct	Marks	40	CO1-CO4

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA COLLEGE OF ENGINEERING, JAIPUR

DEPARTMENT OF COMPUTER ENGINEERING

Campus: Poornima College of Engineering **Year/Section: 4th**
Course: B.Tech. **Semister/ Section – 7 B**
Name of Faculty: Reena Sharma **Name of Subject :Compiler Construction** **Code: 7CS5A**

ATTAINMENT OF CO THROUGH MIDTERM-II EXAM

Attainment of CO: 7CS5A Compiler Construction				
S.No.	Student	Midterm II Marks 60	% of Marks	Level of Attainment
1	ABHISHEK SHUKLA .	51	85	3
2	AKSHITA GUPTA .	45	75	3
3	ARSHAD QURESHI .	44	73	3
4	BHAVIKA ARORA .	47	78	3
5	ESHAN TALWADIYA .	33	55	2
6	MOHIT AGARWAL .	47	78	3
7	MOHIT AREN .	52	87	3
8	MOHIT PATWA .	50	83	3
9	MOHIT SAINI .	45	75	3
10	MUDIT VYAS .	44	73	3
11	NAMIT JAIN .	44	73	3

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

12	PANKAJ JHALANI .	**44**	73	3
13	PIYUSH VAISHYA .		0	1
14	PRAKHAR SINGHAL .	**42**	70	3
15	PRANJAL SHAH .	**39**	65	3
16	PRATEEK KUMAWAT .	**41**	68	3
17	PRAVESH POONIA .	**41**	68	3
18	PREM SINGH RATHORE .	**50**	83	3
19	PRIYANSHI AGRAWAL .	**32**	53	2
20	PULKIT AGRAWAL .	**51**	85	3
21	PUNIT LAKHOTIA .	**34**	57	2
22	RAGHAV KAUSHIK .	**39**	65	3
23	RAJENDRA SINGH DEVARA .	**46**	77	3
24	RANU AGARWAL .	**51**	85	3
25	REENA BADESRA .	**39**	65	3
26	RISHINA DARGAR .	**34**	57	2

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

27	SAHIL CHOPRA .	**53**	88	3
28	SAKSHI AGARWAL .	**28**	47	2
29	SANJOLI JAIN .	**47**	78	3
30	SARTHAK SARRAF .	**52**	87	3
31	SHIVANSH MATHUR .	**50**	83	3
32	SHRI KRISHNA SINGH .	**46**	77	3
33	SHRUTI GANERIWAL .	**18**	30	1
34	SHUBHAM KUMAR .	**51**	85	3
35	SHUBHAM MATHUR .	**51**	85	3
36	SHUBHAM TIWARI .	**42**	70	3
37	SIDHANT RAI SHARMA .	**45**	75	3
38	SORABH MIRCHONIA .	**51**	85	3
39	SPARSH RAJ .	**48**	80	3
40	SUDHANSHU JAIN .	**56**	93	3
41	SUDHANSHU KUMAR .	**48**	80	3

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

42	SURAJ KUMAR .	**52**	87	3
43	SWAPNIL SINGH .	**53**	88	3
44	TAMANNA UPADHYAY .	**40**	67	3
45	TANIYA GUPTA .	**36**	60	3
46	TANMAY SHARMA .	**53**	88	3
47	TANVI SHARMA .	**46**	77	3
48	TARUN JOSHI .	**48**	80	3
49	TILAK MAHAWAR .	**48**	80	3
50	TUSHAR BANSAL .	**45**	75	3
51	VAIBHAV DWIVEDI .	**41**	68	3
52	VIKAL YADAV .	**45**	75	3
53	VIKRAM SINGH BHATI .	**38**	63	3
54	VINAMRA SHARMA .	**50**	83	3
55	VIVEK SINGH RAJPUT .	**48**	80	3
56	YASH BHARADWAJ .	**45**	75	3
57	YASH SHARMA .	**45**	75	3

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

58	YOGITA KHATRI .	**46**	77	3
59	YOGITA SHARMA .	**41**	68	3
60	JAYATI SONI .	**47**	78	3
61	AMIT KUMAR	**50**	83	3
62	ANOOP KUMAR	**48**	80	3
63	PANKAJ KUMAR	**47**	78	3
64	JAYA SACHDEV	**46**	77	3
65	SHALU KUMARI	**39**	65	3
No. of Students attained level 3=		63	% of Students Attained Level 3=	88.73
No. of Students attained level 2=		5	% of Students Attained Level 2=	7.04
No. of Students attained level 1=		3	% of Students Attained Level 1=	4.23
CO Attainment =				2.35
Mark X for absent- Take avg. of all present				

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

CO: 7CS5A Compiler Construction	
Target	**2.25**
Achieved	**2.35**
Gap	**-0.10**

ATTAINMENT OF PO THROUGH CO (MIDTERM-II) COMPONENT

Attainment of PO through CO(MIDTERM-II) Component															
CO	**PO**												**PSO**		
	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**	**PSO3**
CO1	3	-	-	-	-	-	-	-	-	-	-	-	3	-	-
CO2	-	**3**		-	-	-	-	-	-	-	-	-	2	-	-
CO3	-		-	**3**	-	-	-	-	-	-	-	-	3	-	-
CO4	-	-	**3**			-	-	-	-	-	-	-	1	-	-

Attainment of PO through CO(MIDTERM-II) Component															
6CE4A	**PO**												**PSO**		
	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**	**PSO3**
Targets	**3**	**3**	**3**	**3**									**2.4**		
Achieved	**2.9855**	**2.5313**	**2.9275**	**2.4615**									**1.7283**		
Gap	**0.0145**	**0.4688**	**0.0725**	**0.5385**									**0.5217**		

Gaps in PO through CO from MIDTERM-II component: its numerical table is too lengthy .

Action to be taken: Revision and Question Bank will be given

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA COLLEGE OF ENGINEERING, JAIPUR

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Campus: Poornima College of Engineering **Class/Section: IV**
Course: B.Tech. **Year/ Section - B**
Name of Faculty: Reena Sharma **Name of Subject :compiler construction** **Code: 7CS5A**

ATTAINMENT OF CO THROUGH RTU EXAM

Attainment of CO: 7CS5A Compiler Construction				
S.No.	Student	RTU exam Marks 80	% of Marks	Level of Attainment
1	ABHISHEK SHUKLA .	58	72.5	3
2	AKSHITA GUPTA .	78	97.5	3
3	ARSHAD QURESHI .	34	42.5	2
4	BHAVIKA ARORA .	78	97.5	3
5	ESHAN TALWADIYA .	11	13.75	1
6	MOHIT AGARWAL .	44	55	2
7	MOHIT AREN .	34	42.5	2
8	MOHIT PATWA .	8	10	1

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

9	MOHIT SAINI .	56	70	3
10	MUDIT VYAS .	0	0	1
11	NAMIT JAIN .		0	1
12	PANKAJ JHALANI .	23	28.75	1
13	PIYUSH VAISHYA .	7	8.75	1
14	PRAKHAR SINGHAL .	44	55	2
15	PRANJAL SHAH .	33	41.25	2
16	PRATEEK KUMAWAT .	4	5	1
17	PRAVESH POONIA .	6	7.5	1
18	PREM SINGH RATHORE .	8	10	1
19	PRIYANSHI AGRAWAL .	5	6.25	1
20	PULKIT AGRAWAL .	43	53.75	2
21	PUNIT LAKHOTIA .	22	27.5	1
22	RAGHAV KAUSHIK .	67	83.75	3
23	RAJENDRA SINGH DEVARA .	9	11.25	1
24	RANU AGARWAL .	78	97.5	3

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

25	REENA BADESRA .	56	70	3
26	RISHINA DARGAR .	45	56.25	2
27	SAHIL CHOPRA .	65	81.25	3
28	SAKSHI AGARWAL .	43	53.75	2
29	SANJOLI JAIN .	43	53.75	2
30	SARTHAK SARRAF .	54	67.5	3
31	SHIVANSH MATHUR .	23	28.75	1
32	SHRI KRISHNA SINGH .	54	67.5	3
33	SHRUTI GANERIWAL .	64	80	3
34	SHUBHAM KUMAR .	23	28.75	1
35	SHUBHAM MATHUR .	23	28.75	1
36	SHUBHAM TIWARI .	78	97.5	3
37	SIDHANT RAI SHARMA .	33	41.25	2
38	SORABH MIRCHONIA .	12	15	1
39	SPARSH RAJ .	67	83.75	3

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

40	SUDHANSHU JAIN .	65	81.25	3
41	SUDHANSHU KUMAR .	43	53.75	2
42	SURAJ KUMAR .	22	27.5	1
43	SWAPNIL SINGH .	42	52.5	2
44	TAMANNA UPADHYAY .	13	16.25	1
45	TANIYA GUPTA .	66	82.5	3
46	TANMAY SHARMA .	66	82.5	3
47	TANVI SHARMA .	76	95	3
48	TARUN JOSHI .	34	42.5	2
49	TILAK MAHAWAR .	32	40	2
50	TUSHAR BANSAL .	12	15	1
51	VAIBHAV DWIVEDI .	44	55	2
52	VIKAL YADAV .	65	81.25	3
53	VIKRAM SINGH BHATI .	56	70	3
54	VINAMRA SHARMA .	80	100	3
55	VIVEK SINGH RAJPUT .	77	96.25	3

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

56	YASH BHARADWAJ .	34	42.5	2
57	YASH SHARMA .	56	70	3
58	YOGITA KHATRI .	34	42.5	2
59	YOGITA SHARMA .	23	28.75	1
60	JAYATI SONI .	23	28.75	1
61	AMIT KUMAR	53	66.25	3
62	ANOOP KUMAR	78	97.5	3
63	PANKAJ KUMAR	32	40	2
64	JAYA SACHDEV	58	72.5	3
65	SHALU KUMARI	34	42.5	2
No. of Students attained level 3=		23	% of Students Attained Level 3=	36.51%
No. of Students attained level 2=		18	% of Students Attained Level 2=	28.57%
No. of Students attained level 1=		22	% of Students Attained Level 1=	34.92%
CO Attainment =				2.02
Mark X for absent- Take avg. of all present				

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

CO: 7CS5A Compiler Construction	
Target	**2.25**
Achieved	**2.02**
Gap	**0.20**

ATTAINMENT OF PO THROUGH CO (MIDTERM-II) COMPONENT

Attainment of PO through CO(RTU) Component															
CO	**PO**												**PSO**		
	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**	**PSO3**
CO1	3	-	-	-	-	-	-	-	-	-	-	-	3	-	-
CO2	-	**3**		-	-	-	-	-	-	-	-	-	2	-	-
CO3	-		-	**3**	-	-	-	-	-	-	-	-	3	-	-
CO4	-	-	**3**			-	-	-	-	-	-	-	1	-	-

Attainment of PO through CO(RTU) Component															
6CE4A	**PO**												**PSO**		
	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**	**PSO3**
Targets	**3**	**3**	**3**	**3**									**2.4**		
Achieved	**2.9855**	**2.5313**	**2.9275**	**2.4615**									**1.7283**		
Gap	**0.0145**	**0.4688**	**0.0725**	**0.5385**									**0.5217**		

Gaps in PO through CO from RTU component: complete theoretical subject

Action to be taken: Revision and Question Bank will be given.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA COLLEGE OF ENGINEERING, JAIPUR

DEPARTMENT OF COMPUTER ENGINEERING

Campus: Poornima College of Engineering **Class/Section: IV**
Course: B.Tech. **Year/ Section –7CS- B**
Name of Faculty: Reena Sharma **Name of Subject: Compiler Construction** **Code: 7CS5A**

OVERALL ATTAINMENT OF PO & PSO: THROUGH CONTINUOUS ASSESSMENT & RTU

		RTU Component			
S. No.	**Student**	**RTU Overall CO % (75% Weightage)**	**Level**	**Mid II (10% Weightage)**	**Level**
1	ABHISHEK SHUKLA .	72.5	3	70.00	3
2	AKSHITA GUPTA .	97.5	3	71.67	3
3	ARSHAD QURESHI .	42.5	2	66.67	3
4	BHAVIKA ARORA .	97.5	3	53.33	3
5	ESHAN TALWADIYA .	13.75	1	33.33	3
6	MOHIT AGARWAL .	55	2	70.00	3
7	MOHIT AREN .	42.5	2	25.00	3
8	MOHIT PATWA .	10	1	61.67	3
9	MOHIT SAINI .	70	3	1.67	3

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

10	MUDIT VYAS .	0	1	68.33	3
11	NAMIT JAIN .	0	1	18.33	3
12	PANKAJ JHALANI .	28.75	1	0.00	3
13	PIYUSH VAISHYA .	8.75	1	46.67	3
14	PRAKHAR SINGHAL .	55	2	65.00	3
15	PRANJAL SHAH .	41.25	2	38.33	3
16	PRATEEK KUMAWAT .	5	1	46.67	3
17	PRAVESH POONIA .	7.5	1	48.33	3
18	PREM SINGH RATHORE .	10	1	55.00	3
19	PRIYANSHI AGRAWAL .	6.25	1	51.67	3
20	PULKIT AGRAWAL .	53.75	2	43.33	3
21	PUNIT LAKHOTIA .	27.5	1	56.67	3
22	RAGHAV KAUSHIK .	83.75	3	36.67	3
23	RAJENDRA SINGH DEVARA .	11.25	1	68.33	3
24	RANU AGARWAL .	97.5	3	56.67	3

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

25	REENA BADESRA .	70	3	45.00	3
26	RISHINA DARGAR .	56.25	2	28.33	3
27	SAHIL CHOPRA .	81.25	3	73.33	3
28	SAKSHI AGARWAL .	53.75	2	68.33	3
29	SANJOLI JAIN .	53.75	2	46.67	3
30	SARTHAK SARRAF .	67.5	3	90.00	3
31	SHIVANSH MATHUR .	28.75	1	0.00	3
32	SHRI KRISHNA SINGH .	67.5	3	36.67	3
33	SHRUTI GANERIWAL .	80	3	76.67	3
34	SHUBHAM KUMAR .	28.75	1	60.00	3
35	SHUBHAM MATHUR .	28.75	1	10.00	3
36	SHUBHAM TIWARI .	97.5	3	20.00	3
37	SIDHANT RAI SHARMA .	41.25	2	76.67	3
38	SORABH MIRCHONIA .	15	1	60.00	3
39	SPARSH RAJ .	83.75	3	43.33	3
40	SUDHANSHU JAIN .	81.25	3	21.67	3

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

41	SUDHANSHU KUMAR .	53.75	2	93.33	3
42	SURAJ KUMAR .	27.5	1	76.67	3
43	SWAPNIL SINGH .	52.5	2	75.00	3
44	TAMANNA UPADHYAY .	16.25	1	0.00	3
45	TANIYA GUPTA .	82.5	3	68.33	3
46	TANMAY SHARMA .	82.5	3	55.00	3
47	TANVI SHARMA .	95	3	0.00	3
48	TARUN JOSHI .	42.5	2	43.33	3
49	TILAK MAHAWAR .	40	2	30.00	3
50	TUSHAR BANSAL .	15	1	11.67	3
51	VAIBHAV DWIVEDI .	55	2	75.00	3
52	VIKAL YADAV .	81.25	3	28.33	3
53	VIKRAM SINGH BHATI .	70	3	50.00	3
54	VINAMRA SHARMA .	100	3	18.33	3
55	VIVEK SINGH RAJPUT .	96.25	3	40.00	3

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

56	YASH BHARADWAJ .	42.5	2	3.33	3
57	YASH SHARMA .	70	3	30.00	3
58	YOGITA KHATRI .	42.5	2	15.00	3
59	YOGITA SHARMA .	28.75	1	55.00	3
60	JAYATI SONI .	28.75	1	18.33	3
61	AMIT KUMAR	66.25	3	25.00	3
62	ANOOP KUMAR	97.5	3	35.00	3
63	PANKAJ KUMAR	40	2	48.33	3
64	Jaya Kumari	70	3	55	3
65	Shalu	81	3	35	3

No. of Students attained level 3=	**20**
No. of Students attained level 2=	**29**
No. of Students attained level 1=	**12**
2.13	

ATTAINMENT OF CO THROUGH OVERALL COMPONENT

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

CO: 7CS5A Compiler Design	
Target	**3**
Achieved	**2.38**
Gap	**0.62**

Gaps for CO attainment through MIDTERM-I Component: Numerical based subject.

Action to be taken: Revision and class test.

OVERALL PO & PSO ATTAINMENT THROUGH COURSE

Attainment & Gap of Overall PO Session 2019-20															
3CS4-06	**PO**												**PSO**		
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
Targets	3	3	3	3									2.4		
Achieved	2.56	2.96	2.25	2.34									2.15		
Gap													0.64		

Overall Gaps for Course taught: numerical based subjects

Action to be taken: Revision and class test

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA
COLLEGE OF ENGINEERING

LECTURE NOTES

Campus: ..... Course: ..... Class/Section: ..... Date: .....

Name of Faculty: Roona Sharma Name of Subject: Compiler Construction Code: 7CS05A

Date (Prep.): ..... Date (Del.): .....Unit No./Topic: I .....Lect. No: .....

OBJECTIVE: To be written before taking the lecture (Pl. write in bullet points the main topics/concepts etc.. which will be taught in this lecture)

Introduction of Compiler, Translator & Interpreter definition

IMPORTANT & RELEVANT QUESTIONS:

- What is Bootstrapping.
- Explain LEX : A lexical analyzer generator

FEED BACK QUESTIONS (AFTER 20 MINUTES):

- What is Recognition of token.
- Explain Review of Finite automata

OUTCOME OF THE DELIVERED LECTURE: To be written after taking the lecture (Pl. write in bullet points about students' feedback on this lecture, level of understanding of this lecture by students etc.)

Student understand about Translator, Buffering & Error handling

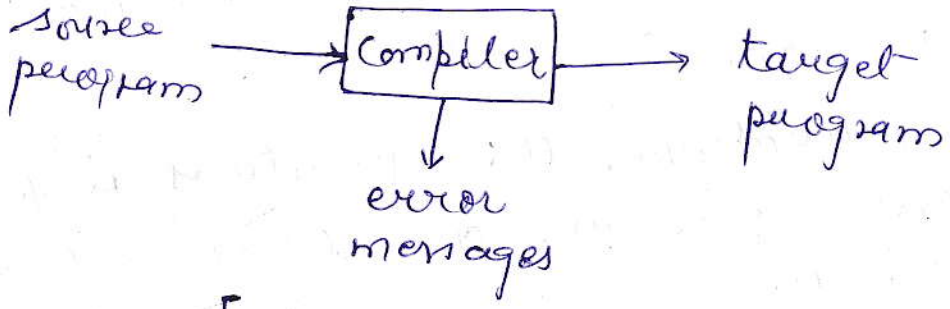
REFERENCES: Text/Ref. Book with Page No. and relevant Internet Websites:

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Unit I

Compiler :

A compiler is a program that reads a program written in one language - the source language - and translate it into an equivalent program in another language - the target language. Compiler reports to its user the presence of errors in the source program.



[A Compiler]

Source language :- programming Language
Target language :- machine language of any computer b/w a microprocessor and a supercomputer.

→ first compilers started to appear in the early 1950's

→ Fortran compiler implement in 1957 (took 18 staff-years)

During compilation, many systematic techniques for handling many of the

The Analysis - Synthesis Model of Compilation

There are two parts to compilation: analysis and synthesis.

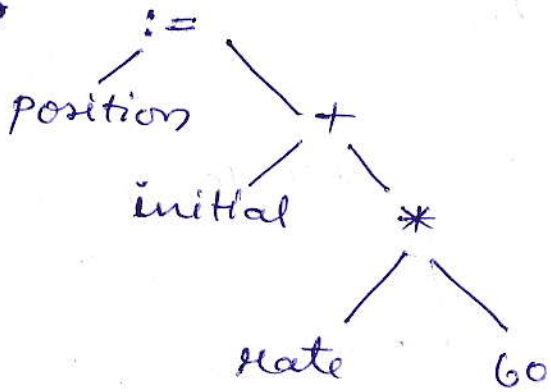
The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.

The synthesis part constructs the desired target program from the intermediate representation. Synthesis requires the most specialized techniques.

→ During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a tree. Often, a special kind of tree called a syntax tree is used, in which each node represents an operation and the children of a node represent the arguments of the operation.

eg:- position := initial + rate * 60

⇒



Many software tools that manipulate source programs first perform some kind of analysis.

1. Structure editor : A structure editor takes as input a sequence of commands to build a source program. Structure editor analyzes the program text, putting an appropriate hierarchical structure on the source program. eg. it can check that the i/p is correctly formed, can supply keywords automatically (while - do)
The o/p of such an editor is often similar to the o/p of the analysis phase of a compiler.

2. Pretty Printers : It analyzes a program and prints it in such a way that the structure of the program becomes clearly visible.
eg: comments may appear in a special font.

3. Static checkers : A static checker reads a program, analyzes it and attempts to discover potential bugs without running the program.
eg: a static checker may detect that parts of the source program can never be executed.
It can catch logical errors such as trying to use a real variable as a pointer.
The type checking techniques.

4. Interpreters: Interpreter performs the operations implied by the source program. eg. Systems Interpreters are frequently used to execute command languages, since operator executed in a command language is usually an invocation of a complex routine such as an editor or compiler.

The analysis portion in each of the following eg is similar to that of a conventional compiler.

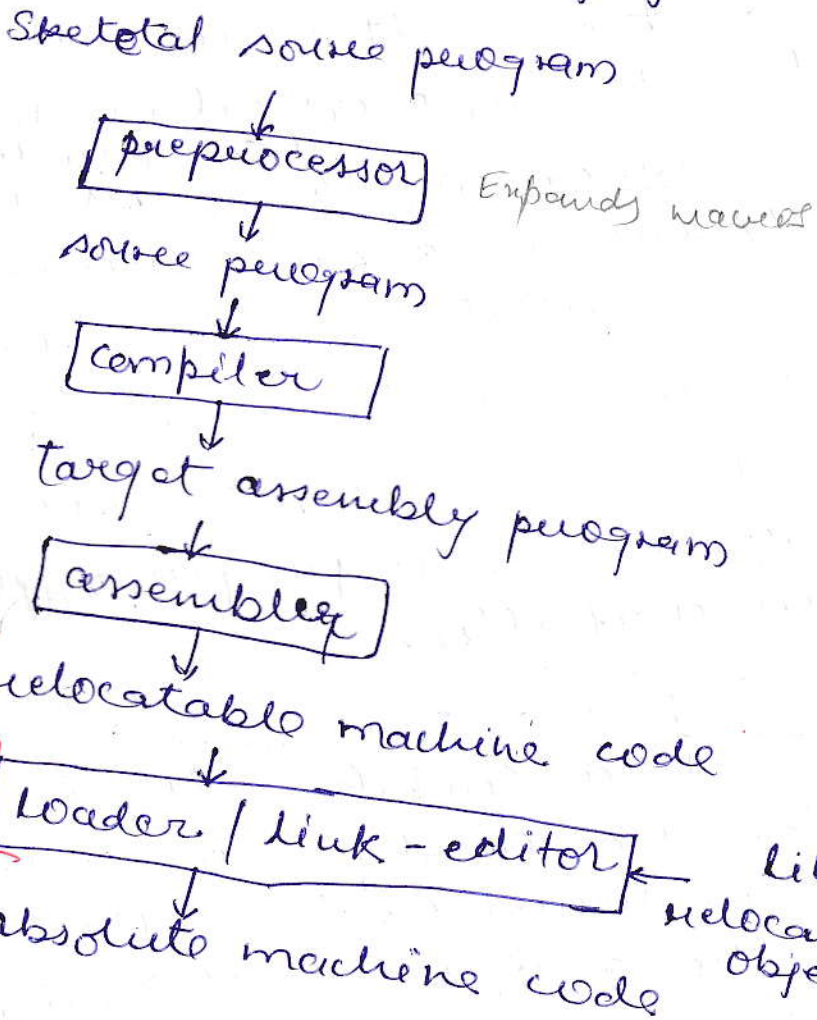
1. Text formatters: A text formatter takes input that is a stream of characters, most of which is text to be typeset, but some which includes commands to indicate paragraph, figures or mathematical structures like subscripts and superscripts.

2. Silicon compilers: A silicon compiler has a source language that is similar or identical to a conventional programming language.

The content of a Compiler :

In addition to a compiler, several other programs may be required to create an executable target 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.

The preprocessor may also expand shorthands, called macros, into source language statements.



relocatable machine code is combined with library files and object files from other sources by linker and loader.

[A language processing system]

This figure shows "compilation". The

further processing before it can be run. Compiler creates assembly code and then linked together with some library routines into the code that actually runs on the machine.

Analysis of the Source Program:

In compiling, analysis consists of three phases:

1. Linear analysis (lexical 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.
2. Hierarchical analysis (Syntax analysis) (is called parsing): in which characters or tokens are grouped hierarchically into nested collections with collective meaning.
3. Semantic analysis: in which certain checks are performed to ensure that the components of a program fit together meaningfully.

→ **Lexical Analysis:**

In a compiler, linear analysis is called lexical ~~anal~~ analysis or scanning.

eg: in lexical analysis the characters in the assignment statement

position := initial + rate * 60

would be grouped into the 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

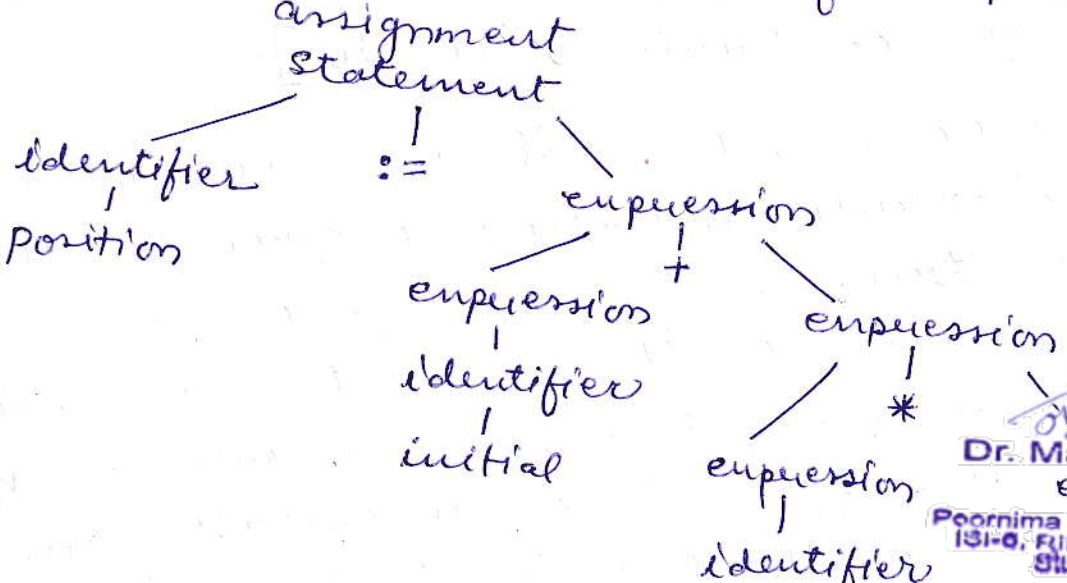
[The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.]

Syntax Analysis:

Hierarchical analysis is 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 o/p.

Usually, the grammatical ~~phar~~ phrases of the source program are represented by a parse ~~tree~~ as the tree.

This is the second phase of compiler.



The hierarchical structure of a program is usually expressed by recursive rules. 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$_1$ and expression$_2$ are expressions, then so are
   expression$_1$ + expression$_2$
   expression$_1$ * expression$_2$
   (expression$_1$)

Rule (1) and (2) are (nonrecursive) basis rules. while (3) defines expressions in terms of operators applied to other expressions.
By rule (1) initial and rate are expressions.
By rule (2), 60 is an expression.
By rule (3), we can first infer that rate * 60 is an expression and finally that initial + rate * 60 is an expression.

[Lexical construct do not require recursion, while syntactic constructs often do.]

A more common internal representation of this syntactic structure is given by the syntax tree. 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.

Semantic Analysis : This 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.

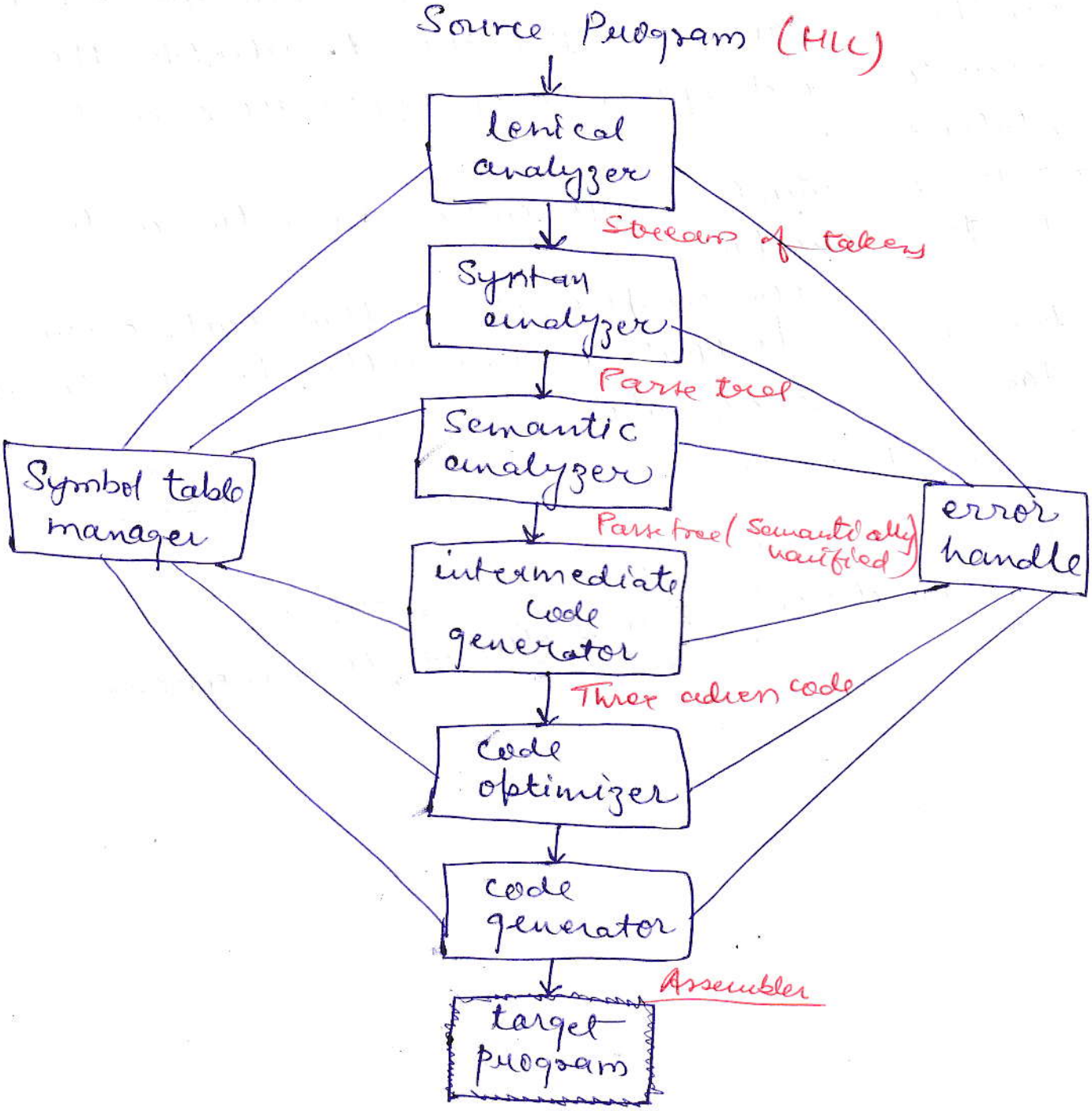
→ Scope Resolution : Symbol table [int a; int a;]

→ array bound check → a[32] etc.

Syntax directed translation → It is mechanism.

The phases of a Compiler:

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



Phases of Compiler

→ First three phases, comes in analysis portion of compiler.

→ two activities symbol table manager, error handler

Symbol Table Management:

An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier.

These attributes may provide information about the storage allocated for an identifier, its type, its scope and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (by reference) and the type returned.

A ~~symbol~~ © symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure 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 detected by the lexical analyzer, the identifier is entered into the symbol table. The attributes of an identifier cannot normally be determined during lexical analysis.

Error Detection and Reporting:

Each phase can encounter errors. After detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected. A compiler that stops when it finds the first error.

→ The syntax and semantic analysis ph usually handle a large fraction of the erro detectable by the compiler.

Intermediate Code Generation:

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program.

This intermediate representation should have two important properties; it should be easy to produce, and easy to translate into the target program.

→ The intermediate representation can have a variety of forms. eg. "three-address-code", which is like the assembly language for a machine in which every memory location can act like a register. Three-address-code consists of a sequence of instructions, each of which has at most three operands.

```
temp 1 := int to real (60)
temp 2 := id3 * temp 1
temp 3 := id2 + temp 2
id1 := temp 3
```

Code optimization:

The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result. Some optimizations are trivial.

```
temp 1 := id3 * 60.0
id1 := id2 + temp 1
```

There is great variation in the amount of code optimization different compilers perform.

"Optimizing compilers", a significant fraction of the time of the compiler is spent on this phase.

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 for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task.

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

→ F in each instruction tells us that instructions deal with floating-point numbers.

→ # signifies that 60.0 is to be treated as a constant.

Position := initial + rate * 60

↓

Lexical analyzer

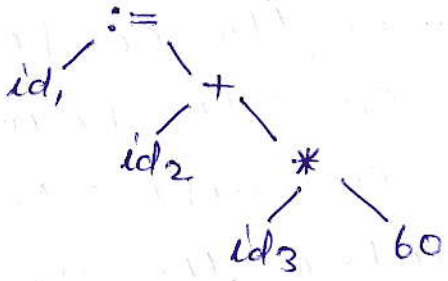
↓

$id_1 := id_2 + id_3 * 60$

↓

Syntax analyzer

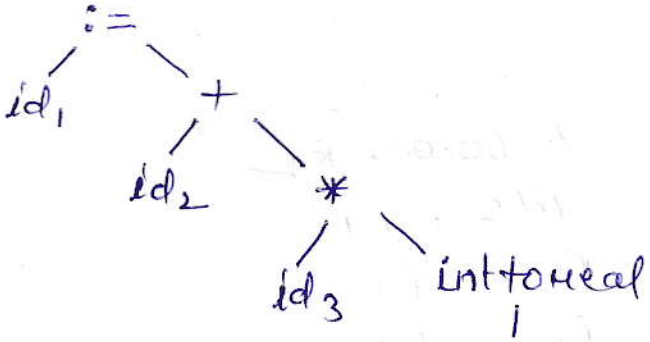
↓



↓

Semantic analyzer

↓



(Assembler depend on the platform)

↓

intermediate code generator

↓

temp1 := inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3

↓

code optimizer

↓

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

↓

Code generator

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

Assembler: An assembler is a program that converts assembly language into machine code. It takes the basic commands and operations from assembly code and converts them into binary code that can be recognized by a specific type of processor.

→ Assembler are similar to compilers in that they produce executable code.

→ Assembler convert low-level code (assembly language) to machine code.

⇒ Most programs are written in high-level programming languages and are compiled directly to machine code using a compiler.

↳ In some cases, assembly code may be used to customize functions and ensure they perform in a specific way. Therefore, IDEs often include assemblers so they can build programs from both high and low-level languages.

IDE → Integrated device Electronics (hardware)
Integrated development Environment (s/w)

↳ IDE is an application that developers used to create computer programs. In this usage case, "integrated" refers to the way multiple development tools are combined into a single program.

Application :- word processor, browser, e

System s/w :- it consist of programs

background,

Applications are said to reside on top of the system s/w.

Loader is the part of an OS that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepare them for execution.

Link-editing process creates an output file from one or more input files.

Lexical analyzers :

Lexical analyzer produces tokens.

Lexical analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens.

→ What is a token ?

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming language.

Lexical analyzer that reads and converts the i/p into a stream of tokens to be analyzed by the parser.

→ A sequence of input characters that comprises a single token is called a lexeme.

⇒ A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

A token is a pair consisting of a token name and an optional attribute value.

- Lexical analysis can be implemented with the DFA.
- The O/P is a sequence of tokens that is sent to the parser for syntax analysis.

eg. of token :-

Keywords : for, while, if etc.

Identifier : Variable name, function name

Operators : +, ++, - etc.

Example of non-tokens:

- comments, preprocessor directive, macros, blanks, tabs, newline, etc.

eg. of lexeme:

float, abs_zero, =, -

How lexical Analyzer functions:

1. Tokenization i.e. Dividing the Program into valid tokens.
2. Remove white space characters.
3. Remove comments.
4. It also provides help in generating error ~~bug messaging~~ message by providing row numbers and column numbers.

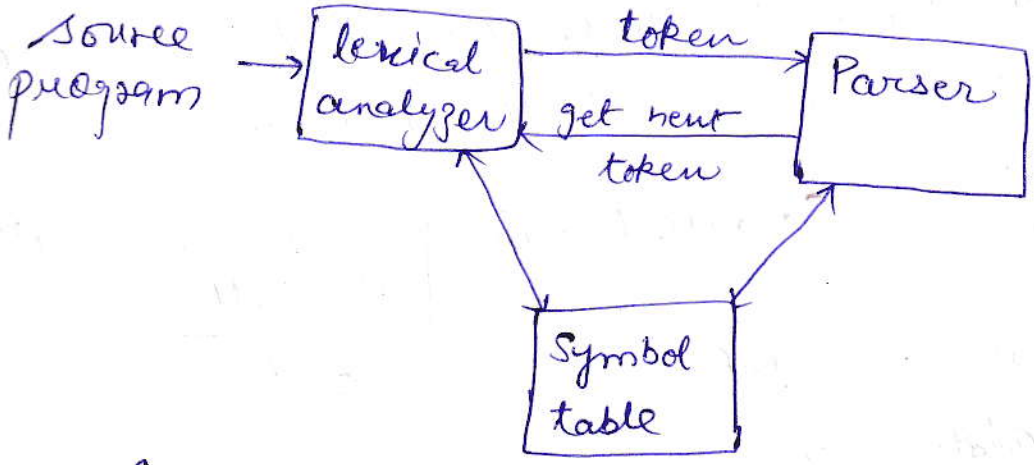
The lexical analyzer identifies the error with the help of the automation machine and the grammer of the given language on which it is based like C, C++ and gives row number and column number of the error.

eg:-

Pass a statement through lexical analyzer - a = b + c ; it will generate token sequence like: id = id + id ; where each id refers to it's variable in the symbol table referencing all details.

Printf ("Geeks Quiz") ;

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR



Interaction of lexical analyzer with parser

Sometimes, lexical analyzer are divided into a cascaded of two phases, the first called "scanning" and the second "lexical analysis". The scanner is responsible for doing simple task, while the lexical analyzer proper does the more complex operations.

→ Issues in Lexical Analysis:

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing.

1. Simplex design is the most important consideration. The seperation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.
2. Compiler efficiency is improved.
3. [illegible]

Pattern :

A pattern is a rule describing a set of lexemes that can represent a particular token in source program. The pattern for the token const in the table is just the single string const that spells out the keyword.

Token	Sample lexemes	Informal description of pattern
const	const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.14, 0, 6.02	any numeric constant
literal	" "	any character between " and "

Attributes of tokens :

The lexical analyzer collects information about tokens into their associated attributes.

The attributes influence the translation of tokens:

1. Constant : value of the constant
2. Identifiers : pointer to the corresponding symbol table entry

~~The Role of Lexical Analyzer~~

Error Recovery Strategies in Lexical Analysis:

1. Deleting an extraneous character.
2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. Transforming two adjacent characters.
5. Panic mode recovery: Deletion of successive characters from the token until error is resolved.

Input Buffering:

Input is necessary to identify tokens. There are three general approaches to the implementation of a lexical analyzer.

1. Use a lexical-analyzer generator, such as the Lex compiler ~~discussed~~, to produce the lexical analyzer from a regular-expression-based specification. In this case, the generator provides routines for reading and buffering the i/p.
2. Write the lexical analyzer in a conventional systems-programming language, using the I/O facilities of that language to read the i/p.
3. Write the lexical analyzer in assembly language and explicitly manage the reading of i/p.

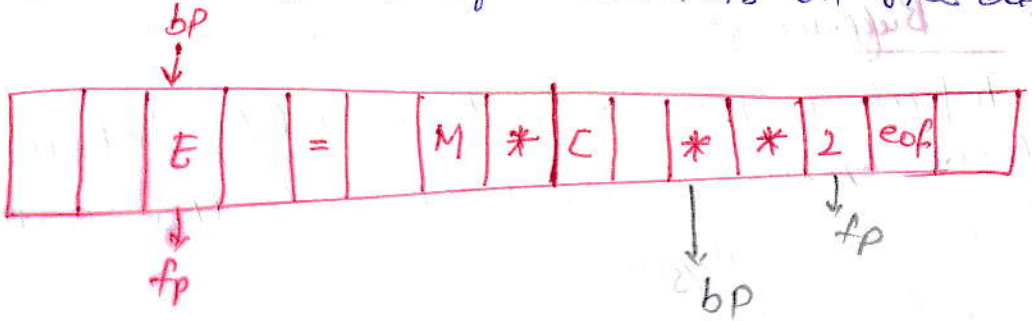
⇒ Lexical analyzer ~~only~~ is the only phase of the compiler that reads the source character-by-character.

Buffer Pairs:

The lexical analyzer used a function ungetc to push lookahead characters back into the input stream. Because a large amount of time can be consumed moving characters specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

→ We use a buffer divided into two N-character halves.

N→ is the no. of characters on one disk block.

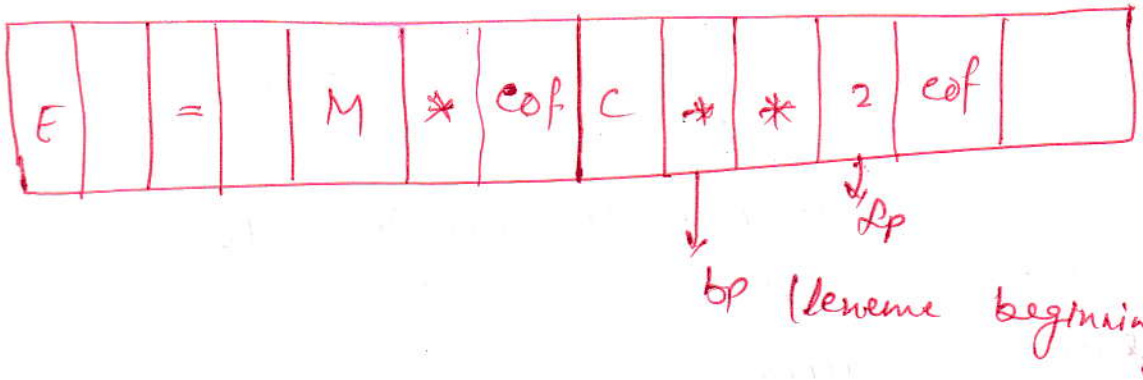


We read N input characters into each half of the buffer with one system read command.

: If fewer than N character remain in the i/p, then a special character eof is read into the buffer after the input characters.

: Two pointers to the i/p buffer are maintained. The string of characters between the two pointers is the current lexeme.

Sentinels: We can reduce the two tests 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 part of the source program, eg. eof



Specification of Tokens:

There are 3 specifications of tokens:

1. Strings
2. Language
3. Regular expression

Strings and languages:

An alphabet or character class is a finite set of symbols.

A string over an alphabet is a finite sequence of symbols drawn from that alphabet.

A language is any countables set of strings over some fined alphabet.

→ In language theory, the term "sentences" and "word" are often used as ~~Dr. Mahesh Bundele~~ "string". The length of a string

eg. banana is a string of length 6. The empty string, denoted ϵ, is the string of length zero.

→ If x and y are strings, then the concatenation of x and y, written xy, is the string formed by appending y to x.

eg, x = dog, y = house, then
xy = doghouse.

→ The empty string is the identity element under concatenation. That is, $s\epsilon = \epsilon s = s$.

Operations on strings:

The following string related terms are commonly used:

1. A prefix of string S is any string obtained by removing zero or more symbols from the end of string S. eg. ban is a prefix of banana.
2. A suffix of string s is any string obtained by removing zero or more symbols from the beginning of s. eg. nana is a suffix of banana.
3. A substring of s is obtained by deleting any prefix and any suffix from s. eg. nan is a substring of banana.
4. The proper prefixes, suffixes and substrings of a string s are those prefixes, suffixes and substrings, respectively of s that are not ϵ or not equal to s itself.
5. A subsequence of s is any string

eg. baan is a subsequence of banana.

Operations on language:

1. Union
2. Concatenation
3. Kleene closure
4. Positive closure

These are the operations that can be applied to languages.

1. **Union:** If two sets L and M then

$L \cup M = \{ s / s \text{ is in } L \text{ or } s \text{ is in } M \}$

$L = \{0, 1\}$ and $M = \{a, b, c\}$

$L \cup M = \{0, 1, a, b, c\}$

2. **Concatenation:** $LM = \{ st / s \text{ is in } L \text{ and } t \text{ is in } M \}$

$LS = \{0a, 1a, 0b, 1b, 0c, 1c\}$

3. **Kleene closure:** KC of L written L^*

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

L^* denotes "zero or more concatenations of" L.

$L^* = \{\epsilon, 0, 1, 00 \ldots\}$

4. positive closure of L written L^+:

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

L^+ denotes "one or more concatenations of"

$L^+ = \{0, 1, 00 \ldots\}$

Regular Expression:

Each regular expression r denotes a language $L(r)$.

Here are the rules that define the regular expressions over alphabet Σ. Associated with each rule is a specification of the language denoted by the regular expression being defined.

1. ϵ is a regular expression that denotes $\{\epsilon\}$, that is, the set containing the empty string.
2. If 'a' is a symbol in Σ, then 'a' is a regular expression, and $L(a) = \{a\}$, that is the language with one string of length one, with 'a' in its one position.
3. Suppose r and s are regular expressions denoting the languages $L(r)$ and $L(s)$. then,

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

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

c) $(r)^*$ is a regular expression denoting $(L(r))^*$.

d) (r) is regular expression denoting $L(r)$

Unnecessary parentheses can be avoided in regular expressions if we adopt the conventions that:

1) the unary operator * has the highest precedence and is left associative.
2) Concatenation has the second highest precedence and is left associative.
3) | has the lowest precedence and is left associative.

Regular set:

eg: Let $\Sigma = \{a, b\}$

1. The regular expression a/b denotes the set $\{a, b\}$.
2. The regular expression (a/b)(a/b) denotes $\{aa, ab, ba, bb\}$, the set of all strings of a's and b's of length two. another regular expression for this same set is aa/ab/ba/bb.
3. The regular expression a^* denotes the set of all strings of zero or more a's ie. $\{\in, a, aa, aaa \ldots\}$.
4. The regular expression $(a/b)^*$ denotes the set of all strings containing zero or more instances of an a or b, that is, the set of all strings of a's and b's. another expression is $(a^*b^*)^*$ for same set.
5. The regular expression a/a^*b denotes the set containing the string a and all strings consisting of zero or more a's followed by a b.

~~Q~~ too.

A language that can be defined by a regular expression is called a regular set. If two regular expressions r and s denote the same regular set, we say that they are equivalent and write $r = s$.

eg: (a/b) = (b/a)

Algebric properties of regular expressions:-

1. $r/s = s/r$ [/ is commutative]
2. $r/(s/t) = (r/s)/t$ [/ is associative]

4. $r(s|t) = rs|rt$ [concatenation distributes over |]
   $(s|t)r = sr|tr$

5. $\epsilon r = r$ [ϵ is the identity element for concatenation]
   $r\epsilon = r$

6. $r^* = (r|\epsilon)^*$ [relation between * and ϵ]

7. $r^{**} = r^*$ [* is idempotent]

Regular definitions:

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

- - -

$d_n \rightarrow r_n$

Where :-

→ Each d_i is a distinct name.

→ Each r_i is a regular expression over the symbols in $\Sigma \cup \{d_1, d_2 \text{ -- } d_{i-1}\}$

eg:- The set of Pascal identifiers is the set of strings of letters and digits beginning with a letter. Here is a regular definition for this set -

letter → A|B| - - - |Z|a|b| - - - |z

digit → 0|1|2| - - - |9

id → letter (letter | digit)*

Notational Shorthands :-

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. One or more instances : (+)
   - → The unary postfix operator + means "one or more instances of".
   - → If r is a regular expression that denotes the language L(r), then $(r)^+$ is a regular expression that denotes the language $(L(r))^+$.
   - → Thus the regular expression a^+ denotes the set of all strings of one or more a's.
   - → The operator + has the same precedence and associativity as the operator *.

2. Zero or One instance : (?)
   - → 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 $L(r) \cup \{\epsilon\}$.

3. Character classes :
   - → The notation [abc] where a, b, c are alphabet symbols denotes the regular expression a|b|c. An abbreviated character class such as [a-z] denotes the regular expression a|b|c---|z.
   - → Using character classes, we can descri

Non - Regular sets:

A language which cannot be described by any regular expression is a non-regular set.
Regular expressions cannot be used to describe balanced or nested constructs.
eg: the set of all strings of balanced parentheses cannot be described by a regular expression. On the other hand, this set can be specified by a context-free-grammer.

Recognition of Tokens:

We see how to recognize the tokens.
eg: Consider the following grammer fragment:

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | ∈
expr → term relop term
     | term
term → id
     | num
```

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

```
if → if
then → then
else → else
relop → <|<=|=|<>|>|>=
id → letter ( letter | digit )*
num → digit+ (.digit+)? (E(+|-)? digit+)?
```

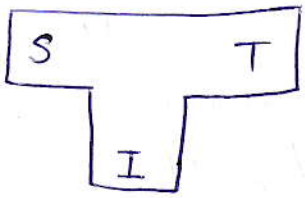
where letter and digit are digit

For this language fragment the lexical analyser will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, num. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers.

Bootstrapping :- Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program.

For bootstrapping purpose, a compiler is characterized by three languages: the source language S that is compiles, the target language T that it generates code for, and implementation language I that it is written in.

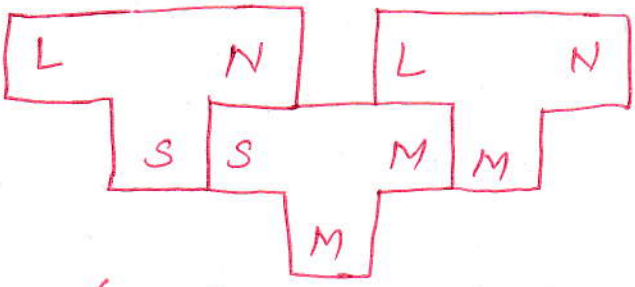
We represent the three languages using the following diagram, called a **T-diagram**, because of its shape.



Within text, we abbreviate the above T-diagram as S_I^T. The three languages S, I and T may all be quite different.

For example, a compiler may run on one machine and produce target code for another machine. Such a compiler is often called a **cross-compiler**.

eg:- We write a cross-compiler for a new language L in implementation language S to generate code for machine N; that is, we create L_S^N. If an existing compiler for S runs on machine M and generates code for M, it is characterized by S_M^M. If we run L_S^N through S_M^M, we get a compiler L_M^N that is a compiler from L to N, running on machine M.



(Compiling a Compiler)

When T-diagram put together, then the implementation language S of the compiler $L_S N$ must be the same as the source language of the existing $S_M M$ and that the target language M of the existing compiler must be that same as the implementation language of the translated form $L_M N$. A trio of T-diagrams such as can be write as

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

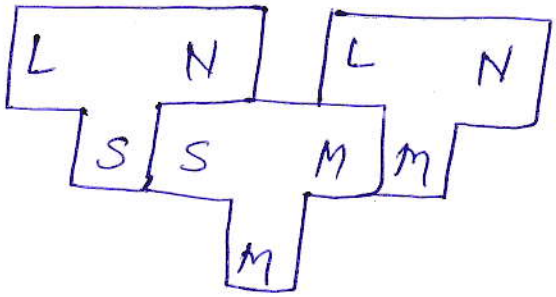
Eg of Transpilers:

Source language	Target lang.	Transpiler
C++	C	CFRONT
C#	JavaScript	ScriptSharp
PHP	C++	HIPHOP FOR PHP
COBOL	C	OpenCOBOL

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Q. Suppose we write a cross-compiler for a new language L in implementation language S to generate code for machine N; that is, we create L_SN. If an existing compiler for S runs on machine M and generates code for M, it is characterized by S_MM. If L_SN is run through S_MM, we get a compiler L_MN, that is, a compiler from L to N that runs on M.

Sol:



$$L_SN + S_MM = L_MN$$

Finite automata :-

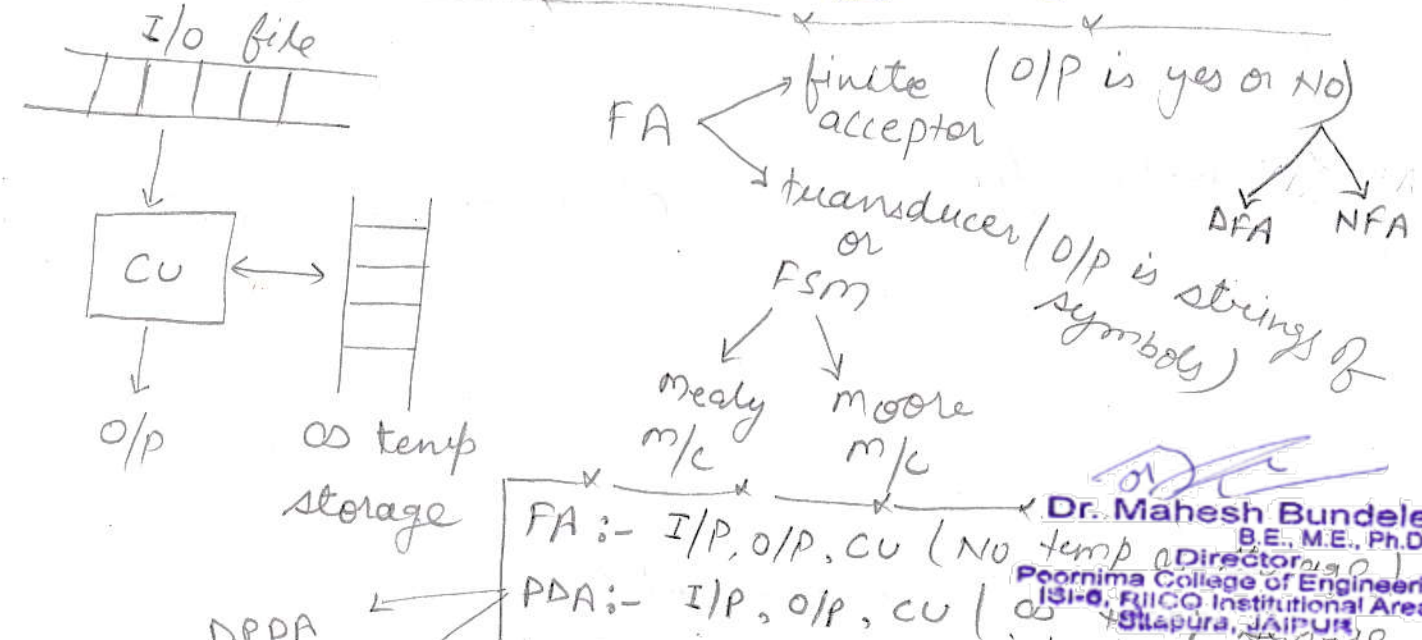
An automaton (Automata in plural) is an abstract self-propelled computing device which follow a predetermined sequence of operations automatically.

An automaton with a finite number of states is called a Finite automaton (FA) or Finite State Machine (FSM).

Formal definition of a Finite Automaton

An automaton can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where -

- Q is a finite set of states
- Σ is a finite set of symbols, called the alphabet of the automaton [set of i/p symbols]
- δ is the transition function $[Q \times \Sigma \to Q]$
- q_0 is the initial state from where any input is processed ($q_0 \in Q$)
- F is a set of final state $[F \subseteq Q]$



FA is characterized into two ways: S is diff NFA, DFA

1. Nondeterministic Finite Automata (NFA)
2. Deterministic Finite Automata (DFA)

1. NFA: NFA is a mathematical model that consists of
   i) a set of states S [Q]
   ii) a set of input symbols Σ | the input symbol alphabet
   iii) a transition function more that maps state-symbol pairs to set of states
   iv) a state S_0 that is distinguished as the start (or initial) state [$q_0 \in Q$]
   v) a set of states F distinguished as accepting (or final) states [$F \subseteq Q$]

→ An NFA can be represented diagrammatically by a labeled directed graph, called a transition graph, in which the nodes are states and the labeled edges represent the transition function.

If more than one starting states are present than it is called transition diagram. In this case, it is not NFA.

Transition function $S: Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$ or $P(Q)$

$\cup\{\lambda\} \to$ 

$Q = \{q_0, q_1\}$

$P(Q) = \{\phi, \{q_0\}, \{q_1\}, \{q_0, q_1\}\}$

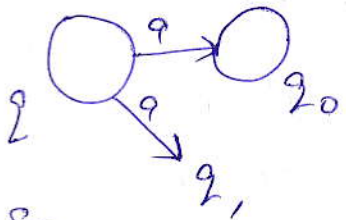
M: 

$F = \phi$

$(q, a) \rightarrow \{q_0, q_1\}$

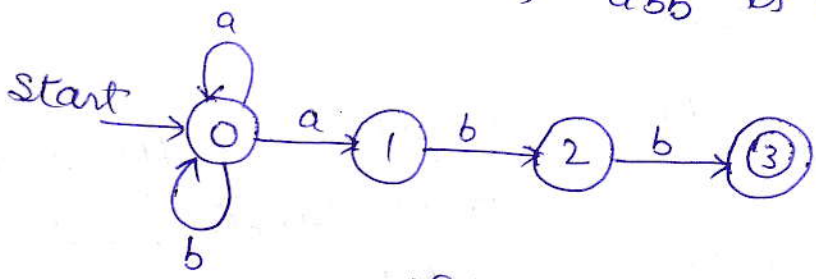
$(q, a) \rightarrow \phi$ machine on dead configuration

$(q, a) \rightarrow q_0$ no choice



So, in this way, when we have choice for a state like q_0, q_1 from initial state q and input is a. Then we can't determine the state. That's why it is called non-deterministic called. This is done by P(Q).

The transition graph for an NFA that recognizes the language $(a/b)^* abb$ is ~~shown~~ :



NFA

When describing an NFA, we use the transition graph representation.

easiest implementation of transition function of an NFA is a transition table in which ~~there~~ there is a row for each state and a column for each input symbol. and

State	Input Symbol	
	a	b
0	{0, 1}	{0}
1	–	{2}
2	–	{3}

[Transition table for the finite automaton]

Advantage of transition table :- it provides fast access to the transitions of a given state on a given character.

Disadvantage is that it can take up a lot of space when the input alphabet is large and most transitions are to the empty set.

→ An NFA accepts an input string x if and only if there is some path in the transition graph from the start state to some accepting state, such that the edge labels along this path spell out x.

↳ for example aabb is accepted by the path from 0, following the edge labeled a to state 0 again, then to state 1, 2, 3 via edges labeled a, b and b respectively.

↳ A path can be represented by a sequence of state transitions called moves. Like :

aabb :

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

⇒ Several other sequences of moves may be on the i/p string aabb.

$$0 \xrightarrow{a} 0 \xrightarrow{a}$$

Deterministic Finite Automata :

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence it is called "Deterministic Automation".

DFA can be represented by 5-tuple :

$Q \rightarrow$ finite set of states

$\Sigma \rightarrow$ finite set of symbols

$\delta \rightarrow$ transition function where $\delta: Q \times \Sigma \rightarrow Q$

$q_0 \rightarrow$ initial state

$F \rightarrow$ Finial state

eg:

Let a deterministic finite automaton be →

$Q = \{q, b, c\}$,

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

$q_0 = \{q\}$

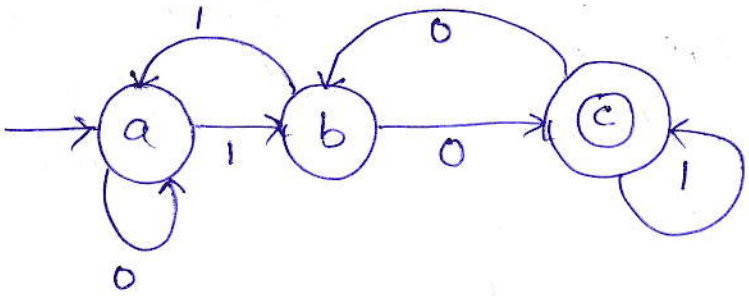
$F = \{c\}$

Transition function δ as shown by the following table :

Present State	Next state for input 0	Next state for input 1
q	q	b
b	c	q
c	b	c

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

its graphical representation would be as follows :-



Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

eg. of finite automata :-

	I/P	
SFL	②	q_2
FA	①	q_1
GFL	⓪	q_0

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

$\Sigma = \{0, 1, 2\}$

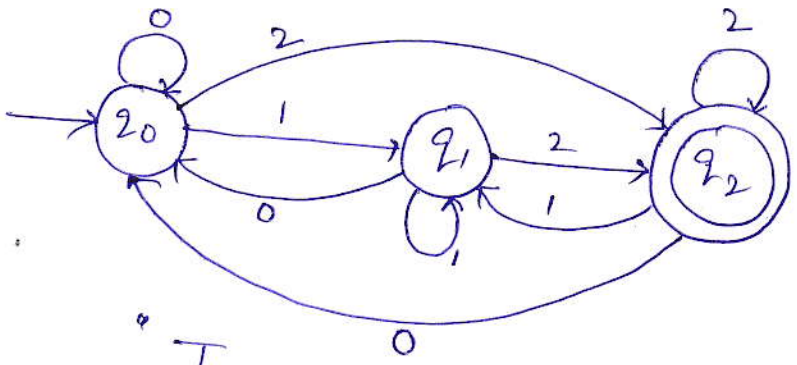
q_0 = initial state, $q_0 \in Q$

$F = \{q_2\}$, $F \subseteq Q$

$\delta(q_0, 0) = q_0$, $\delta(q_0, 1) = q_1$, $\delta(q_0, 2) = q_2$

$\delta(q_1, 0) = q_0$, $\delta(q_1, 1) = q_1$, $\delta(q_1, 2) = q_2$

$\delta(q_2, 0) = q_0$, $\delta(q_2, 1) = q_1$, $\delta(q_2, 2) = q_2$



"Transition System"

	0	1	2
→ q_0	q_0	q_1	q_2
q_1	q_0	q_1	q_2
q_2	q_0	q_1	q_2

"Transition table"

Regular Expression

→ The language accepted by finite automata can easily described by simple expression called Regular Expression.

→ It is most effective way to represent any Language.

→ The language accepted by some regular expression are referred to as Regular languages.

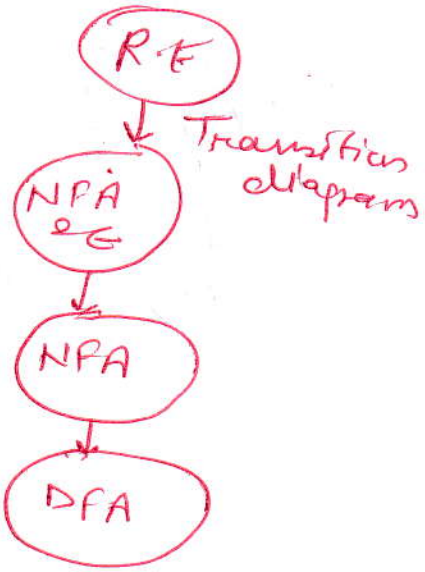
→ A regular expression can also be described as a sequence of pattern that defines a string.

Convert regular expression to finite automata.

Convert R.E. ~~into~~ NFA

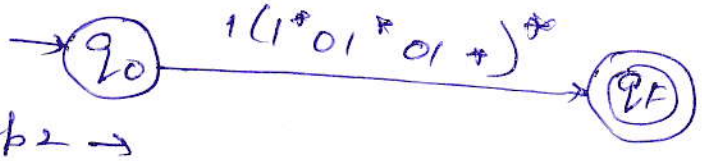
~~Design NFA~~

Design NFA from given R.E.

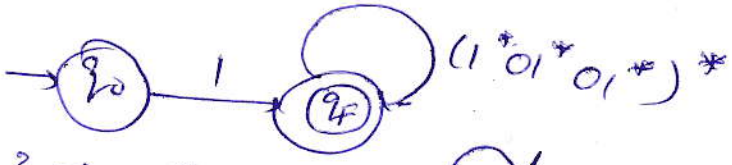


R.E. = $1(1^*01^*01^*)^*$

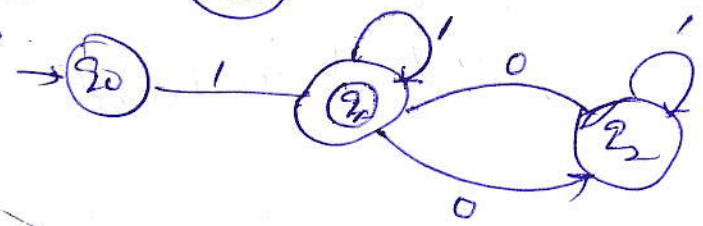
Step 1 →



Step 2 →



Step 3 →

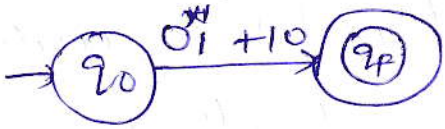


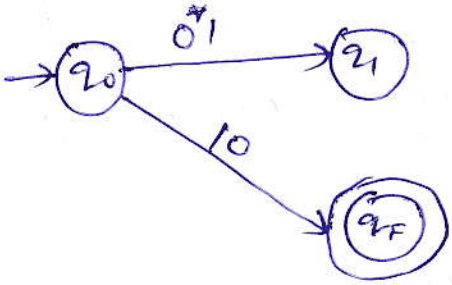
= NFA

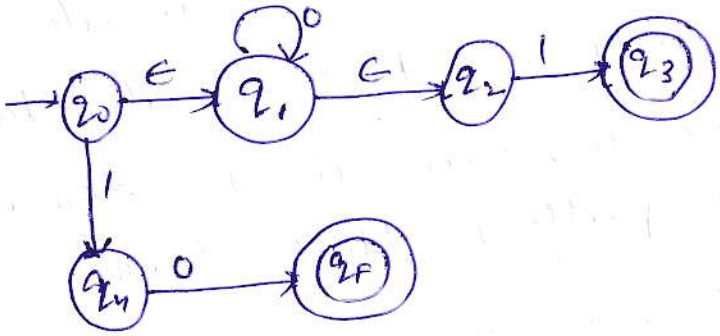
Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Construct FA for given R.E

RE. $0^*1 + 10$

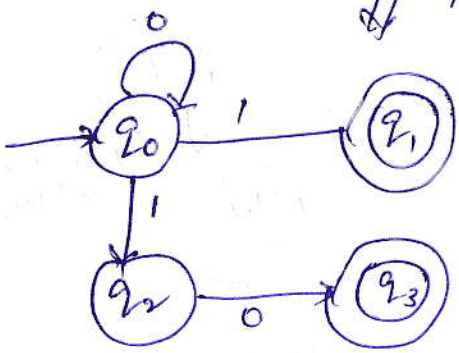






NFA with ϵ

// NFA without ϵ



Lang (Regular) → R.E. → Finite automata

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

→ The language that is accepted by regular expression is called regular language.

eg: of R.E.

1) Write R.E. for the language accepting all strings containing any number of a's & b's.

Sol

R.E. = $(a+b)^*$ [L = ∈, a, b, aa, bb, aba, ...]

This will given the set as L = { ∈, a, aa, b, bb, ab, ba ... } any combination of a & b.

The $(a+b)^*$ shows any combination with a & b even a null string also.

2) Write R.E. for the language accepting all the string which are starting with 1 & ending with 0, over Σ = {0, 1}

Sol The first symbol is 1

last symbol is 0

∴ R.E. = $1(0+1)^*0$

{ L = 10, 101, 100, 1000, 1001, ... }

3) Write R.E. for the lang starting with a but not having consecutive b's.

L = { a, aba, aaba, aaa, abab, ... } bbb ✗

R.E. = $\{a + ab\}^*$

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Write the R.E. for the lang. over $\Sigma = \{0\}$ having even length of the string.

Sol: $L = \{\in, 00, 0000, \ldots\}$

$R.E. = \{00\}^*$

Conversion of RE to FA

To convert RE to FA, we use a method called subset method.

Step 1: Design a transition dia for given RE, using NFA with ∈ moves

2: Convert NFA with ∈ to NFA without ∈

3: Convert the obtained NFA by equivalent DFA

Regular Grammer

A regular grammar is a mathematical object G, with four components, $G = (N, \Sigma, P, S)$, where N is nonempty, finite set of nonterminal symbols, or alphabet, symbols, P is a set of grammer rules, each of one having one of the form. $A \rightarrow aB$ or

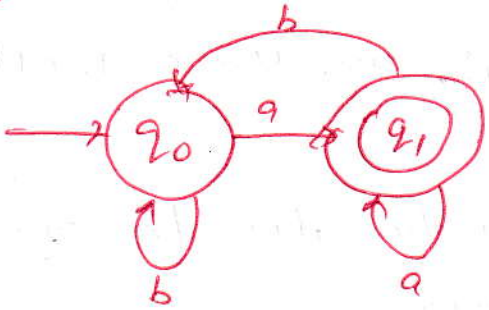
nonterminal symbols (N)

Σ (terminal symbol)

S (start symbol)

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Transform the DFA to regular grammar



Sol.

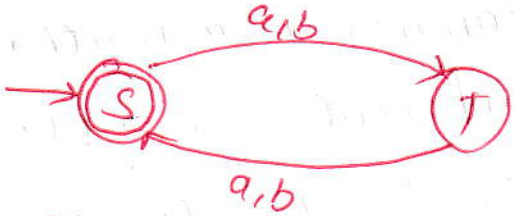
$Q_0 \rightarrow aQ_1 / bQ_0$

$Q_1 \rightarrow aQ_1 / bQ_0$

$L = \{w \in \{a,b\}^* : |w| \text{ is even}\}$

R.E. $(aa) \cup (ab) \cup (ba) \cup (bb))^*$

FSm.



NFSps

R.G. $G = (N, \Sigma, P, S)$ where

$N = \{S, T, a, b\}$

$\Sigma = \{a, b\}$

$R = \{ S \rightarrow \epsilon$

$S \rightarrow aT$

$S \rightarrow bT$

$T \rightarrow aS$

$T \rightarrow bS$

$\}$

11 → LEX TOOL: Lexical Analyzer Generator

LEX

LEX is a tool. It is also called Lex Compiler. It takes as a input the program called lex.l containing the specification of lexical analyzer.

lex.l → [lex - compiler] → lex.yy.c → This is in C programming

lex.yy.c → [C. compiler] → a.out [lexical analyzer]

i/p stream → [a.out] → sequence of tokens

lex.l is not written in C Program it is just set of specifications.

a.out is an object program which is a lexical analyzer that transforms an input stream into a sequence of tokens.

→ Lex program consist of three parts:

A lex program is separated into three sections by %% delimiters. The format of Lex source is as follows:

```
{ definitions }
%%
{ rules }
%%
{ user subroutines }
```

action i is program fragment written in C which describes what action the lexical analyzer should take when pattern P_i matches the lex

⇒ Lex is a tool for automatically generate

with the help of LEX compiler

If i/p stream a=b+c*d is going to lexical analyzer then it is easy to find out the tokens of i/p stream by all steps of lex program.

L-28

Bhavik Prog

Program to count the no. of vowels & constants in a given grammer.

```
%{
#include <stdio.h>
    int vowels = 0;
    int cons = 0;
%}
%%
    [aeiou AEIOU] {vowels++}
    [a-z A-Z]  {const++}
%%
int yywrap()
  {
   return 1;
  }
main()
{
printf("Enter the string at end press ^d \n");
 yylex();
printf(" no. of vowels = %d\n
        no. of constant = %d\n",
        vowels
```

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Unit - 2.9

YACC

Lex → lexical analyzer generator

YACC → Parser generator

It is a tool which generate LALR parser.

Lex → LA → first pass

YACC → SA → 2nd pass

O/P of YACC tool is parse tree.

Lex → always take R.E specification

YACC → always take Grammer

Step 1 → Yacc specification

Parser.y

↓

Yacc compiler → Y.tab.c

Step 2 → Y.tab.c → C compiler → a.out { syntax analyzer }

Step 3 :

I/P { token } LA → a.out → O/P { Parse tree }

Syntax :

definition

%%

Rules { list of grammer rules with sem }

%%

Supplementary code

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Definition includ declaration of constant, variable and regular definiton.

Rules defines the statement of form p1 {action1} p2 {action2} — pn {actionn}.
where pi describes the regular expression and action, describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

User subroutines: are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

yylex() → invoke the lexical analyzer by calling the yylex subroutine.

yywrap() → Returns the value 1 when the end of input occurs.

yymore() → Append the next matched string to the current value of the yytext array rather than replacing the contents of the yytext array.

| Operator indicates alternation [ab|cd { matches either ab or cd

a{1,5} [looks 1 to 5 occurrence of a]

[ ] Specifies any one character from the string in as it appears first character after the left bracket, indicates all characters in the standard set.

[A-Za-z0-9@&#]

[A-Za-z][0-9a-zA-Z]*

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Vowels and consonant

```
%{
    int v=0, c=0;
%}
%%
[aeiouAEIOU] v++;
[A-Za-z] c++;
%%
int yywrap(){ }
int main()
{
 printf("enter the string\n");
 yylex(); [It is used for take the input]
 printf("the number of vowels are %d\n", v);
 printf("------ consonant - %d\n", c);
 return 0;
}
```

lex prog.l
gcc lex.yy.c -lfl -ll
./a.out
Ctrl+d

letter = A|B|--|z
digit = 0|1|2 -- 9
identifier = letter(letter/digit)*

[it is call to invoke the lexer]

Positive and Negative numbers:

```
%{
    int pos=0, neg=0;
%}
^[-][0-9]+
{
 negative++;
    printf("negative number = %s\n", yytext);}
[0-9]+
 {
  pos++;
  printf(
```

```
int yywrap(){ }
int main()
{
yylex();
printf("no of %d,"
return 0;
}
```

POORNIMA
COLLEGE OF ENGINEERING

LECTURE NOTES

Campus: ..... Course: ..... Class/Section: ..... Date: .....

Name of Faculty: Reena Sharma Name of Subject: Compiler Construction Code: 7CS05

Date (Prep.): ..... Date (Del.): .....Unit No./Topic: II .....Lect. No: .....

OBJECTIVE: To be written before taking the lecture (Pl. write in bullet points the main topics/concepts etc.. which will be taught in this lecture)

Introduction of CFG Ambiguity of grammars.
&
Introduction of Parsing

IMPORTANT & RELEVANT QUESTIONS:

What is CFG Ambiguity of grammars

Explain LL grammars & passers error handling of LL.

FEED BACK QUESTIONS (AFTER 20 MINUTES):

- What is Construction of SLR.
- Explain YACC error handling in LR parsers.

OUTCOME OF THE DELIVERED LECTURE: To be written after taking the lecture (Pl. write in bullet points about students' feedback on this lecture, level of understanding of this lecture by students etc.)

Student understand LL grammers & passers error handling of LL parser.

REFERENCES: Text/Ref. Book with Page No. and relevant Internet Websites:

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

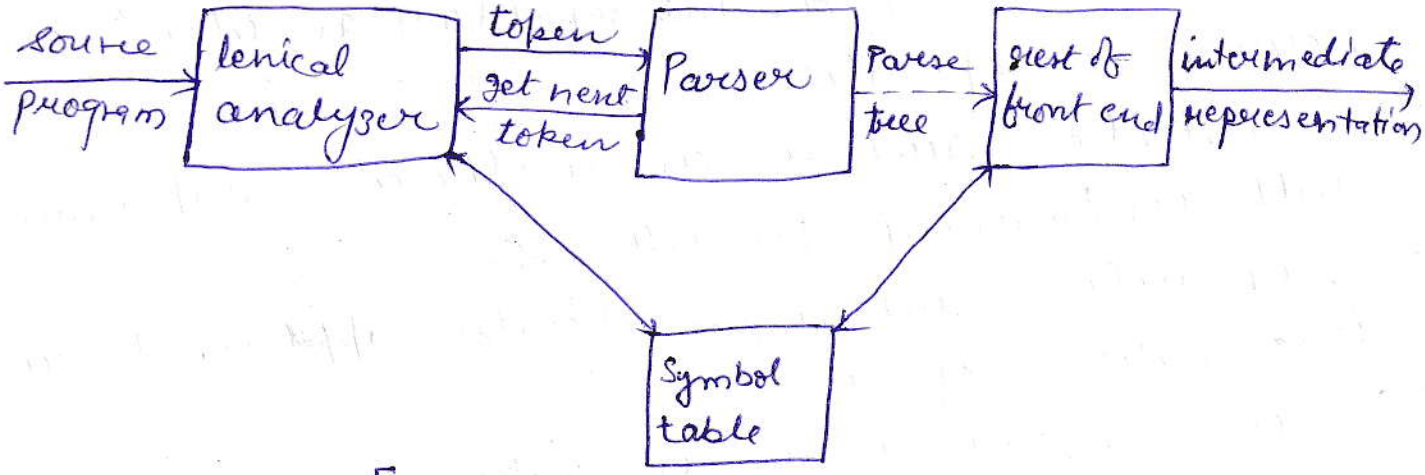
Unit 2

Review of CFG Ambiguity of grammers

Introduction to parsing:

In the syntax analysis phase, a Compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the Language. This is done by Parser.

Parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammer for the source Language.



[Position of parser in compiler model]

The O/P of pariser is some representation of the Parse tree for the stream of tokens produced by the lexical analyzer.

Syntax analyzers follow production ~~rules~~ defined by means of Context-free grammer. The

the production rules are implemented

→ top-down parsing
→ bottom-up parsing

Syntax Error Handling :

A good compiler should assist the programmer in identifing and locating errors. Most programming Language specifications do not describe how a compiler should respond to errors; the response is left to the compiler designer. Planning the error handling right from the start can both simplify the structure of a compiler and improve its response to errors.

Errors can be :

→ lexical, such as missepelling an identifier, keyword or operator
→ Syntactic, such as an arithmatic expression with unbalanced parentheses
→ semantic, such as an operator applied to an incompatible operand
→ logical, such as an infinitely recursive call

Much 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 la

methods; they can detect the presence of syntactic errors in programs very efficiently. Accurately detecting the presence of semantic and logical errors at compile time is a much more difficult task.

The error handler in a parser has simple-to-state goals:

→ 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 programs.

↳ How should an error handler report the presence of an error?

↳ Once an error is detected, how should the parser recover?

An inadequate job of recovery may introduce an annoying avalanche of "spurious" errors, those that were not made by the programmer, but were introduced by the changes made to the parser state during error recovery. In a similar vein, syntactic error recovery may introduce spurious semantic errors that will later be detected by the semantic analysis or generation phase.

Error - Recovery Strategies :

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

→ Panic mode
→ phrase level
→ error productions
→ global correction

Content-Free Grammers:

For specifying the syntax of a language, we use content-free-Grammer. A grammer naturally describes the hierarchical structure of many programming language constructs.

For eg:- an if-else statement in C has the form

if (expression) statement else statement

Using the variable expr to denote an expression and the variable stmt to denote a statement, this structure rule can be expressed as

Stmt → if (expr) stmt else stmt

Rule is called production. In a production lexical elements like the keyword if and the parentheses are called tokens.

Variable like expr and stmt represent sequences of tokens and are called nonterminals.

→ lexical analyzer cannot check the syntax of a given sentences due to the limitations of the regular expressions.

Regular expression cannot check balancing tokens, such as parentheses. Therefore, this phase uses content-free-grammer (CFG), which is recognized by push-down automata.

CFG is ~~the~~ a superset of Regular Grammer

CFG is a helpful tool in describing the syntax of programming languages.

→ A context-free grammer has four components:

1. A set of tokens, known as terminal symbols. (Σ)
2. A set of nonterminals. (V)
3. A set of productions where each production consists of a nonterminal, called the left side of the production, an arrow, and a sequence of tokens and/or nonterminals, called the right side of the production. (P)
4. A designation of one of the nonterminals as the start symbol. (S)

$$G = (V, \Sigma, P, S)$$

eg:- Palindrome language, which cannot be described by Regular expression.

That is $L = \{w \mid w = w^R\}$ is not a regular language. But it can be described by CFG.

$$G = (V, \Sigma, P, S)$$

where $V = \{Q, Z, N\}$ (variables/non-terminals)

$\Sigma = \{0, 1\}$ (set of terminal symbol)

$P = \{Q \to Z \mid Q \to N \mid Q \to \varepsilon \mid Z \to 0Q0 \mid N \to 1Q1\}$

$S = \{Q\}$ (set of production rules)

Difference between CFG and Regular [In production rule only]

$$\alpha \to \beta$$

CFG

Every Regular Grammar is a Context free Grammar.

Derivation Tree :-

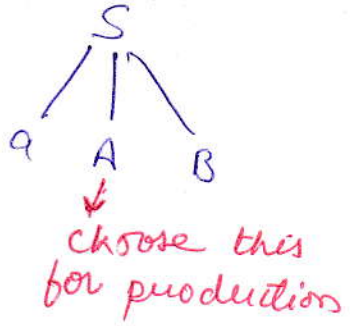
A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take decisions for some sentential form of input.

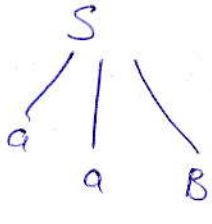
- → Deciding the non-terminal which is to be replaced.
- → Deciding the production rule, by which, the non-terminal will be replaced.

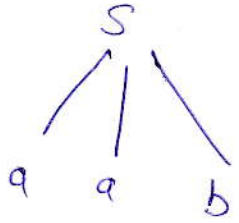
To decide which non-terminal to be replaced with production rule, we can have two options.

Left most Derivation : If the sentential form of input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

Eg:- $S \rightarrow aAB$, $A \rightarrow a$, $B \rightarrow b$



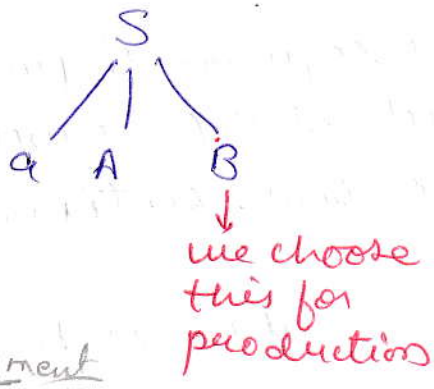


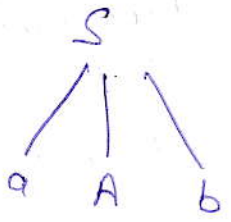


Right most Derivation : If we scan the input with production rules from right to left, it is known as right-most

right-most derivation is called the right-sentential form.

eg:- $S \rightarrow aAB$, $A \rightarrow a$, $B \rightarrow b$





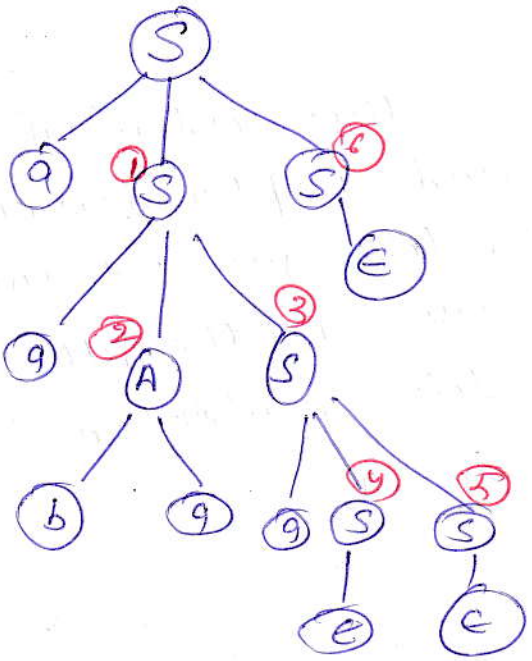
Assignment

eg:- $G = S \rightarrow aAS \mid aSS \mid \epsilon$, $A \rightarrow SbA \mid ba$

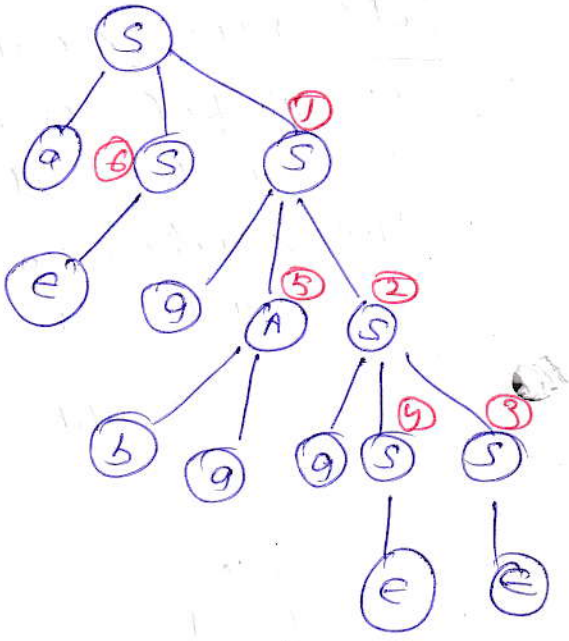
using this grammer obtain the string aabaa with the help of L.D.T and R.D.T.

Sol.

Left Derivation Tree:



Right Derivation Tree:



Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

$S \rightarrow S_0 S_1$

Grammer

(Context free Grammar)

Grammer Contain a set of rules

CFG Formal definition

$A \rightarrow A\alpha / \beta$

$A \rightarrow Ba / Aa / c$

$B \rightarrow Bb / Ab / d$

$A \rightarrow A\alpha / \beta$

$\beta a A' / c$

$\begin{cases} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' / \epsilon \end{cases}$

$E \rightarrow E + T / T$ (A, A, α, β)

$T \rightarrow T * F / F$ (A, A, α, β)

$\begin{cases} E \rightarrow TE' \\ E' \rightarrow +TE' / \epsilon \end{cases}$

$F \rightarrow (E) / id$

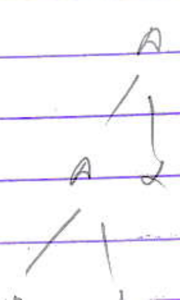
$A \rightarrow \alpha A / \beta$

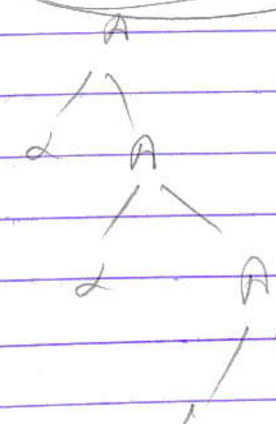
$A \rightarrow \alpha A / \beta$

$A \rightarrow \alpha A / \beta$

$A \rightarrow A\alpha / \beta$.

$A \rightarrow Aba / Aa / c$





PAGE NO.

formal definition of CFG / Context free Grammar

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

CFG Rules "Create strings and collection of strings is called language.

CFL → CFG Conf. Context free language → Grammar

Parser not accept ambiguous grammer.

POORNIMA
COLLEGE OF ENGINEERING
DETAILED LECTURE NOTES

Input
SA ← grammer

PAGE NO. ...

Grammers

Input to Syntax Analyzer is Grammer, and i/p will be checked according to grammer.

$G = (V, T, P, S)$ → Start Symbol, → Production, ↓ Variable, → Terminals

Automata Theory for Grammer

Grammer

$E \rightarrow E + E$
$\quad / E * E$
$\quad / id$

Grammer have 3 production

→ Variables are of left hand side (non-terminal) capital letter
$V = \{E\}$

→ Terminal are right hand side (whatever is not in LHS) small letter
$T = \{+, *, id\}$ $(V \cap T = \emptyset)$

→ Rules are production

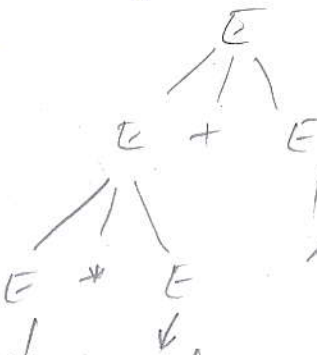
→ S is E.

for a given grammer and for a given string we got more than two LMD, or RMD, parse tree it means this grammer is ambiguous.

Left most Derivation

$id + id * id$

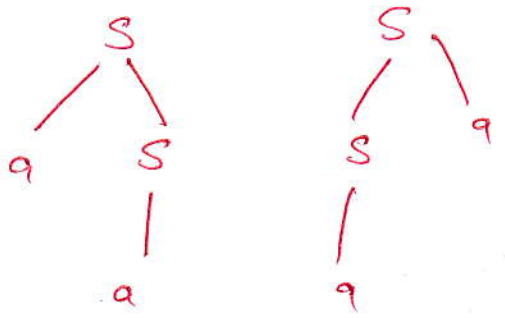
If I want to generate the above string using grammer.



Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

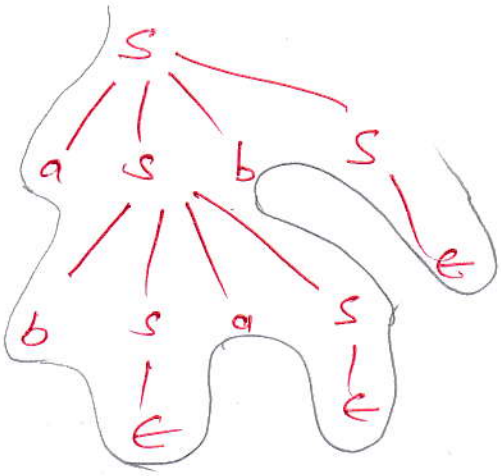
① $S \to aS \mid Sa \mid a$ → grammer (we get two parse tree ∴ that this grammer is ambiguous.)

$w = aa$ → Study

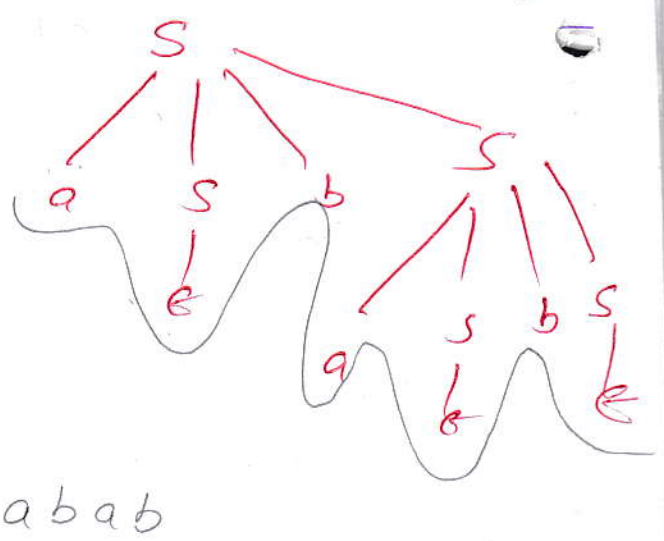


② $S \to aSb \; S \mid bSaS \mid \epsilon$ → ambiguous grammer

$w = abab$



abab



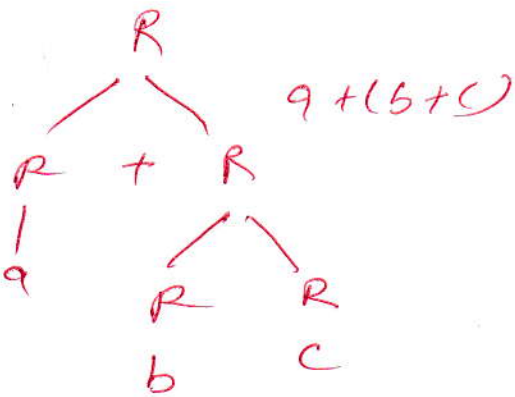
abab

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

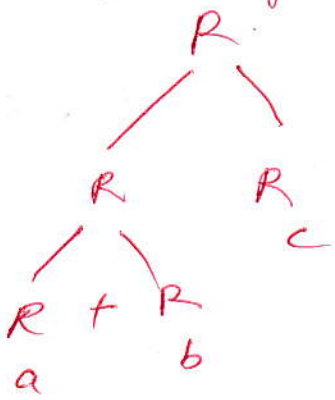
Primitive regular expression

ambiguous grammer.

$a+bc$



$a+(b+c)$



$(a+b)c$

Concatenation is getting higher precedence because in it's an

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

Parse Tree:- A parse tree pictorially shows how the start symbol of a grammer derives a string in the language. If nonterminal A has a production $A \rightarrow XYZ$, then a parse tree may have an interior node labeled A with three children labeled X, Y and Z from left to right.

A parse tree is a tree with the following properties:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a token or by ϵ.
3. Each interior node is labeled by a nonterminal.
4. If A is the nonterminal labeling some interior node and $X_1, X_2 -- X_n$ are the labels of the children of that node from left to right, then $A \rightarrow X_1 X_2 - X_n$ is a production. Here $X_1, X_2 - X_n$ stand for a symbol that is either a terminal or a nonterminal. As a special case, if $A \rightarrow \epsilon$ then a node labeled A may have a single child labeled ϵ.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Associativity: If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

eg:- (a) Addition, Multiplication, Subtraction and Division are left associative, if the expression contains:

id op id op id

it will be evaluated as:

(id op id) op id

(b) Exponentiation are right associative.

id op (id op id)

eg: id ^ (id ^ id)

Precedence of Operators:

If two different operators share a common operand, the precedence of operators decides which will take the operand.

Multiplication and division have higher predence than addition and subtraction.

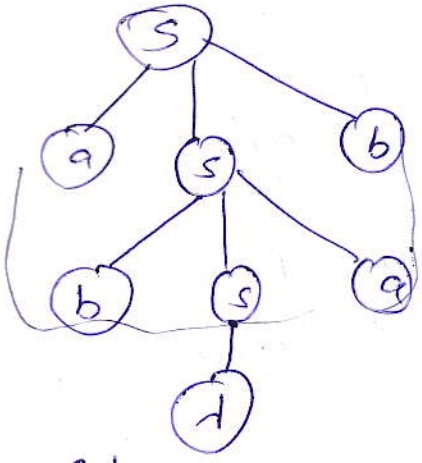
Ambiguous Grammer

A Grammer is said to be ambiguous if there exists two or more derivation tree for a string w.

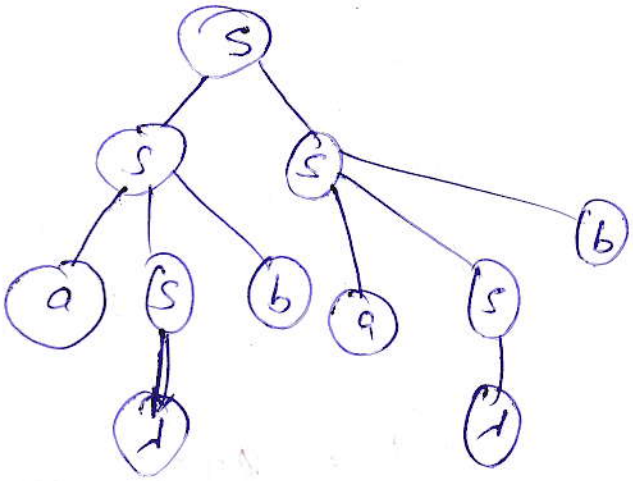
Q (b) $G = (\{S\}, \{a,b\}, S, S \to aSb \mid bSa \mid SS \mid \lambda)$

String = "abab"

(1)



abab

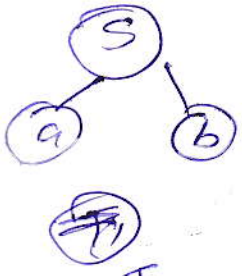


abab

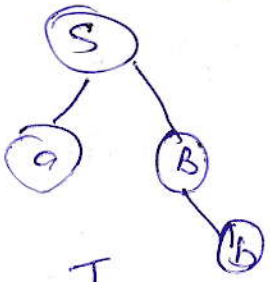
Q (2) [This G is ambiguous Grammer.]

$S \to aB \mid ab$, $A \to aAB \mid a$, $B \to Abb \mid b$, is ambiguous.

(Smallest string ab in this)



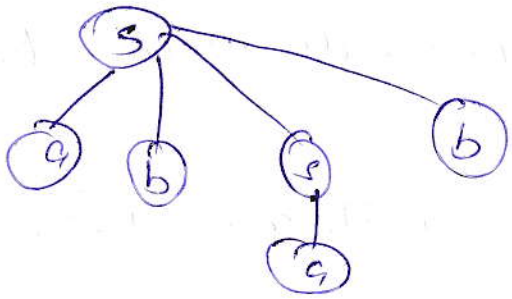
T_1



T_2

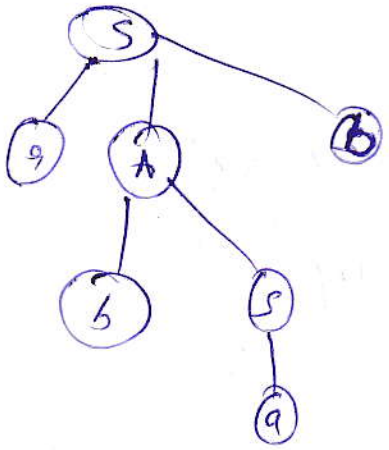
for string ab, It is Ambiguous Grammer.

(3) Prove that the Gramm $S \to a/aAb/abSb$, $A \to aAAb/bS$ is ambiguous. abab



abab

T1



abab

T2

for string abab we have more than two D.T. , that's why it is ambiguous grammar.

CFG are classified based on:

→ Number of Derivation Tree

→ Number of string

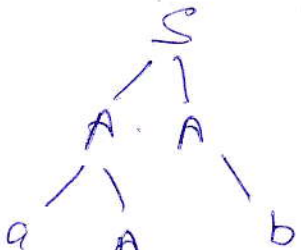
① Number of D.T. CFG are sub divided into 2 types

→ Ambiguous grammers

→ Unambiguous grammer.

Unambiguous eg: → $S \to AA$, $A \to aA$, $A \to b$

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



abb

POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

Left Recursion (16)

A grammer becomes a left recursive if it has any non-terminal 'A' whose derivation contains 'A' itself as the left-most symbol. Left recursive grammer is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the start symbol, which is itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

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

In this we see that how the left-recursive pair of production $A \rightarrow A\alpha / \beta$ could be replaced by the non-left recursive productions.

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' / \epsilon$$

without changing the set of strings derivable

eg:- Consider the following grammer for arithmatic expressions.

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

Sol. Eliminating the immediate left recursion (productions of the form $A \to A\alpha$) to the production for E and then for T, we obtain

1) $E \to TE'$
   $E' \to +TE' \mid \epsilon$
2) $T \to FT'$
   $T' \to *FT' \mid \epsilon$
3) $F \to (E) \mid id$

Algorithm : Eliminate left recursion

1. Arrange the nonterminals in some order $A_1, A_2 \ldots A_n$
2. for i := 1 to n do begin
   for j := 1 to i-1 do begin
   replace each production of the form $A_i \to A_j\gamma$
   by the productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
   where $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j - productions;
   end
   eliminate the ~~terminal~~ immediate left recursion among the A_i - productions
   end

Elimination of ~~left~~ left recursion and left factoring the grammer :-

①

Recursion

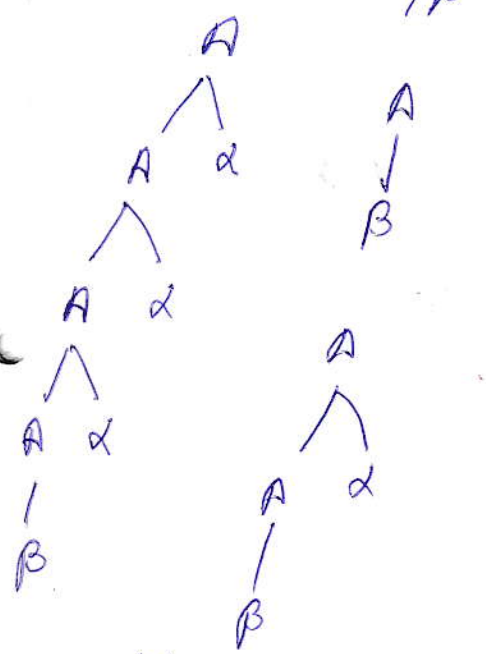
- Left recursion ($A \to A\alpha/\beta$)
- ~~Rit~~ Right recursion $A \to \alpha A/\beta$

→ If the leftmost symbol of R.H.S. is equal to L.H.S. then it is called <u>left recursion</u>.

→ If the Rightmost symbol of R.H.S. is equal to L.H.S. then it is called <u>Right recursion</u>.

<u>Left recursion:-</u>

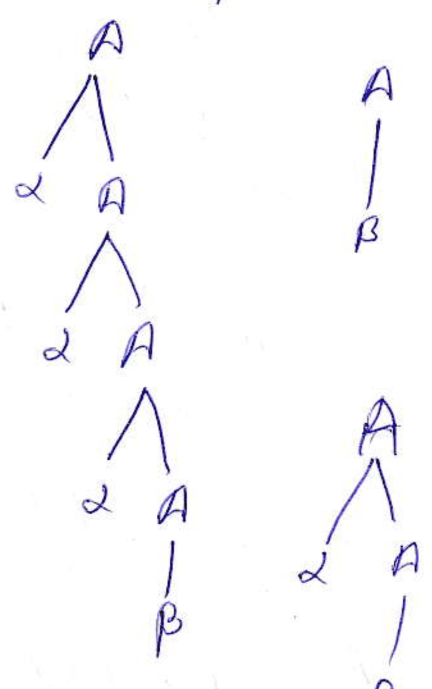
$A \to A\alpha/\beta$



obtained grammer is $= \beta\alpha^*$

<u>Right recursion:</u>

$A \to \alpha A/\beta$



obtained grammer is $\alpha^*\beta$

→ problem is with L.R. that calling the ... infitine loop. A() { A()

→

Top down parser can't work with left recursion

→ Eliminate left recursion without changing grammar

eg: $A\alpha^*$

$A \to A\alpha^*$

$A \to \beta A'$

$A' \to \epsilon / \alpha A'$ ⟷ ~~$A \to A\alpha/\beta$~~ $A \to A\alpha/\beta$

eg: $E \to E+T/T$ ($A \to A\alpha/\beta$)

$E \to TE'$ ($A \to \beta A'$)

$E' \to \epsilon/+TE'$ ($A' \to \epsilon/\alpha A'$)

eg:- $S \to S0S1S/01$ ($A \to A\alpha/\beta$)

~~$S \to S_0 S_1 S/01$~~

$S \to 01S'$

$S' \to \epsilon/0S1SS'$

eg: $S \to (L)/x$

$L \to L,S/S$ ($A \to A\alpha/\beta$)

Ans. $A \to \beta A'$, $A' \to \alpha A'/\epsilon$ | $L \to SL'$, $L' \to \epsilon/SL'$

$A \to \beta A'$, $A' \to \alpha A'/\epsilon$ | $L \to SL'$, $L' \to ,SL'/\epsilon$

eg: $A \to A\alpha_1/A\alpha_2/A\alpha_3/\ldots$ ($\beta_1, \beta_2, \beta_3$)

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

$A' \to \alpha_1 A'/\alpha_2 A'/\alpha_3 A' \ldots$

7, 9, 9, 10, 11, 16, ~~18, 20, 21~~, 26, 28, ~~29~~, ~~30~~, 31, 44, 47, 58, 59, 65, 66,

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA
COLLEGE OF ENGINEERING
DETAILED LECTURE NOTES

Left Factoring : (27)

If more than one grammer production rules has a common prefix string, then the top-down parser cannot make a choice as to as to which of the production it should take to parse the string in hand.

eg:- if a top-down parser encounters a production like

$A \Rightarrow \alpha\beta \mid \alpha\gamma \mid \ldots$

then it cannot determine which production to follow to parse the string as both productions are starting from the same terminals (or non-terminal). To remove this confusion, we use a technique called the left factoring.

In left factoring, we make one production for each common prefixes and the rest of the derivation is added by new productions.

eg:- The above production can be written as

$A \rightarrow \alpha A'$

$A' \rightarrow \beta \mid \gamma \mid \ldots$

Now the parser has ...

prefix which makes it easier to take decisions.

eg:- $S \to iEtS \mid iEtSeS \mid a$ (S: A, $iEtS$: α)
$E \to b$

~~Sol:~~ It is non-deterministic grammer.

Sol: $S \to iEtSS' \mid a$
$S' \to \epsilon \mid eS$
$E \to b$

$A \to \alpha A'$
$A' \to \beta \mid \gamma \mid .$

eg:- $E \to T + E \mid T$
$T \to int \mid int * T \mid (E)$

Sol: $E \to TE'$
$E' \to +E \mid \epsilon$
$T \to int\ T' \mid (E)$
$T' \to \epsilon \mid *T$

$A \to \alpha \mid \alpha\beta \mid a$
$S \to aeSS' \mid a$
$S' \to \epsilon \mid eS$

$E \to TE'$

$T \to int\ T'$

$E \to +E'$
$E \to TE'$
$E' \to \epsilon +$

$E \to TE'$
$E' \to +E \mid \epsilon$

~~E~~ $A \to \alpha A'$
$E \to TE'$
$E' \to +E \mid \epsilon$
$T \to int\ T' \mid (E)$
$T' \to \epsilon \mid *T$

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

G

Deterministic grammer

Non deterministic

eg: $A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3$

Left factoring

$A \to \alpha A_1$

$A_1 \to \beta_1 / \beta_2$

Left factoring

$A \to \alpha A'$

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

$A \to \alpha\beta_1 / \alpha\beta_2 / \ldots / \alpha\beta_m / \gamma$

$A \to \alpha A' / \gamma$

$A' \to \beta_1 / \beta_2 / \ldots / \beta_m$

$\alpha\beta_3$

$A \to \alpha A'$

$A \to \beta_1 / \beta_2 / \beta_3$

} without left factoring

Problem with

It is problem of non-deter grammer

⇒ Conversion of Non-deterministic grammer in deterministic grammer using Left factoring :-

eg: $S \to iEtS$
$/ iEtSeS$
$/ a$
$E \to b$

} is non-deterministic grammer

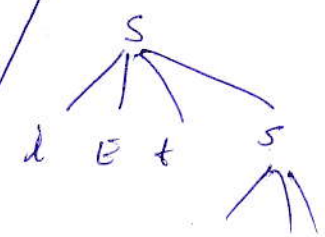
↓ ambiguous grammer

$iEtiEtSeS$

$S \to iEtSS' / a$

$S' \to \epsilon / eS$

$E \to b$



S

S

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Applying left factoring and elimination left recursion, by this we don't get ambiguous grammer.

$A \to \alpha\beta_1 / \alpha\beta_2$
$A \to \alpha A'$
$A' \to \beta_1 / \beta_2$

① Eg:- $S \to iEtS / iEtSeS / a$
$E \to b$

$S \to iEtSS' / a$
$S' \to eS / \epsilon$
$E \to b$

②
$x \to x+x / x*x / D$
$D \to 1/2/3$
$x \to xY / D$
$Y \to +x / *x$
$D \to 1/2/3$

③ $E \to T+E / T$ $\quad \alpha = T,\ \beta_1 = +E,\ \beta_2 = \epsilon$
$T \to int / int * T / (E)$
$E \to TE'$
$E' \to +E / \epsilon$
$T \to int\ T' / (E)$
$T' \to \epsilon / *T / \epsilon$

④ $S \to aSSbS / aSaSb / abb / b$
$S \to aS' / b$ —①
$S' \to SSbS / SaSb / bb \Rightarrow S' \to SS'' / bb$ —②
$S'' \to SbS / aSb$ —③

$S \to aSS' / abb / b \Rightarrow S \to aS'' / b$ —①
$S' \to SbS / aSb$ —②
$S'' \to SS' / bb$ —③

POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

Top Down parsing:-

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

Recursive descent Parsing: It is common form of top-down parsing. It is called recursive as it uses recursive procedure to procces the i/p. Recursive descent parsing suffers from backtracking.

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

Top down parser

↓

Recursive Descent

↙ ↘

Back Tracking | Non Back Tracking

↓

Predictive

Recursive descent Parsing: Top down parsing can be viewed as an attempt to find a leftmost derivation for an input string. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammer associated with it (if not left factored) cannot avoid backtracking.

A form of recursive descent parsing that does not require any back-tracking is known as predictive parsing.

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

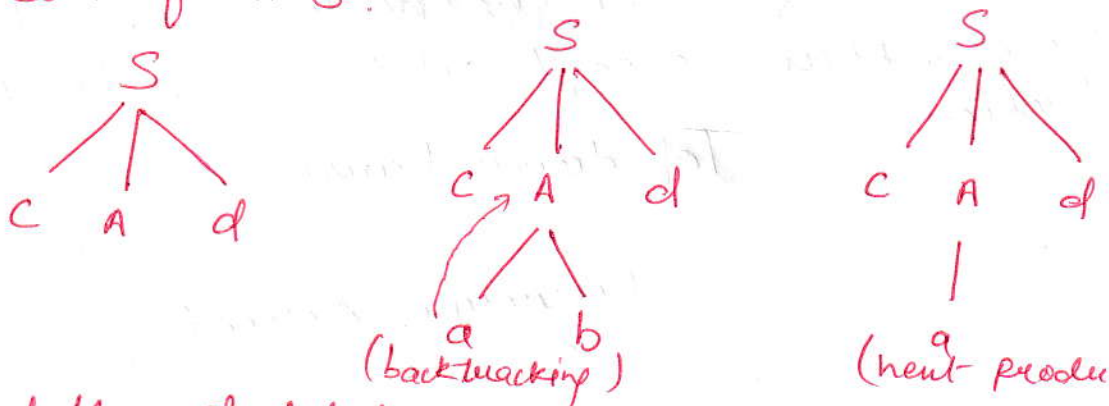
eg:- Consider the grammer

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

and the input string is $w = cad$.

Start from S.



The leftmost leaf, labeled c, matches the first symbol of w, so we now advance the input pointer to a. the second step and consider the next leaf, labeled

POORNIMA
COLLEGE OF ENGINEERING
DETAILED LECTURE NOTES

for the second input symbol so we advance the input pointer to d, the third input symbol, and compare d against the next leaf, labeled b. Since does not match d, we report failure and go back to A to see whether there is another alternative for A that we have not tried but that might produce a match.

A left-recursive grammer can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop.

Predictive Parser: [Transition diagram remaining]

In many cases, when writing a grammer, eliminating left recursion from it and left factoring the resulting grammer, we can obtain a grammer that can be parsed by a recursive-descent Parser that needs no backtracking is called predictive parser.

Predictive Parser is a recursive descent Parser, which has the capability to predict which production is to be used to replace the i/p string. The predictive parser d from backtracking.

To construct a predictive parse

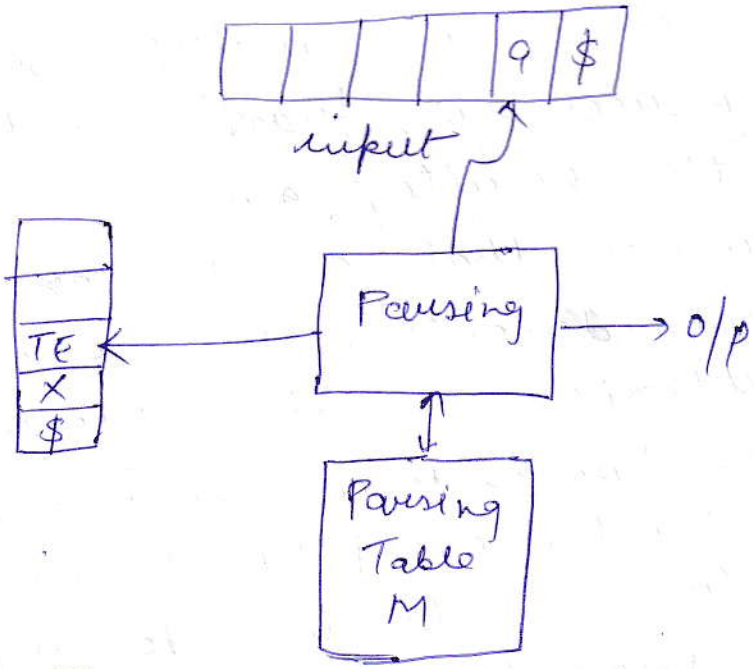
the alternatives of production $A \to \alpha_1 | \alpha_2 | \ldots$ is the unique alternative that derives a string beginning with a.

To make the parser backtracking free, the predictive parser puts some constraints on the grammer and accepts only a class of grammer known as LL(k) grammer.

Non recursive Predictive Parsing: It is possible to build a non-recursive 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.

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



[Model of nonrecursive predictive pars

POORNIMA
COLLEGE OF ENGINEERING
DETAILED LECTURE NOTES

Predictive parsing uses a stack and a parsing table to parse the input and generates a parse tree. Both the stack and the input contains an end symbol $ to denote that the stack is empty and the input is consumed. The ~~parsing~~ parser refers to the parsing table to take any decision on the input and stack element combination.

LL Parser :

An LL Parser accepts LL grammer. LL grammer is a subset of context free grammer but with some restriction to get the simplified version, in order to achieve easy implementation. LL grammer can implemented by both algorithms namely, recursive descent or table-driven.

A grammer whose Parsing table has no multiply-defined entries is said to be LL(1). The first L in LL(1) stands for scanning the input from left to right, the second L for producing a leftmost derivation and 1 for using one input symbol of lookahead at each step to make parsing

LL(1) grammers have several distinctive properties

No ambiguous or left recursive grammer can be LL(1). It can be shown that grammer G is LL(1) if and only if whenever $A \to \alpha | \beta$ are two distinct productions of G the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \overset{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A).

FIRST and FOLLOW:

An important part of parser table construction is to create first and follow set. These set can provide the actual position of any terminal in the derivation.

FIRST: If α is any string of grammer symbol, Let FIRST(α) be the set of terminals that begin the strings derived from α. If $\alpha \overset{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α).

eg: $\alpha \to t\beta$

That is α derives t in the very first position. So $t \in$ FIRST(α).

Look at the definition of FIRST(α) set

1. If α is a terminal, then FIRST(α) = $\{\alpha\}$
2. If α is a non-terminal and $\alpha \to \epsilon$

POORNIMA
COLLEGE OF ENGINEERING
DETAILED LECTURE NOTES

3. If α is a non-terminal and $\alpha \rightarrow Y_1 Y_2 Y_3 \dots Y_n$ and any FIRST(Y) contains t then t is in FIRST(α).

FOLLOW : We calculate what terminal symbol immediately follows a non-terminal α in production rules. We do not consider what the non-terminal can generate but instead, we see what would be the next terminal symbol that follows the production of a non-terminal.

Rules for calculating FOLLOW set :

1. If S is the start symbol, then FOLLOW() = \$
2. If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) except for ϵ is placed in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

how to find first and follow in LL(1)

	First	Follow
$S \to ABCDE$	$\{a, b, c\}$	$\{\$\}$
$A \to a/\epsilon$	$\{a, \epsilon\}$	$\{b, c\}$
$B \to b/\epsilon$	$\{b, \epsilon\}$	$\{c\}$
$C \to c$	$\{c\}$	$\{d, e, \$\}$
$D \to d/\epsilon$	$\{d, \epsilon\}$	$\{e, \$\}$
$E \to e/\epsilon$	$\{e, \epsilon\}$	$\{\$\}$

because c does not have ϵ.

(2)

$S \to Bb/cd$	$\{a, b, c, d\}$	$\{\$\}$
$B \to aB/\epsilon$	$\{a, \epsilon\}$	$\{b\}$
$C \to cC/\epsilon$	$\{c, \epsilon\}$	$\{d\}$

(3)

$E \to TE'$	$\{id, (\}$	$\{\$,)\}$
$E' \to +TE'/\epsilon$	$\{+, \epsilon\}$	$\{\$,)\}$
$T \to FT'$	$\{id, (\}$	$\{+, \$,)\}$
$T' \to *FT'/\epsilon$	$\{*, \epsilon\}$	$\{+, \$,)\}$
$F \to id / (E)$	$\{id, (\}$	$\{*, +, \$,)\}$

(4)

$S \to ACB/CbB/Ba$	$\{d, g, h, \epsilon, b, a\}$	$\{\$\}$
$A \to da/BC$	$\{d, g, h, \epsilon\}$	$\{h, g, \$\}$
$B \to g/\epsilon$	$\{g, \epsilon\}$	
$C \to h/\epsilon$	$\{h, \epsilon\}$	

Follow ()

abcd$

Grammer string is follow by $

S → ABCD
A → b/ε
B → c
C → d
D → e

S - {$}
A - {c}
B - {d}
C → {e}
D → $

1) S कहीं दीवार में ... Right side में than we get
2) अगर RHS में Vertex हो तो ~~उसे prod~~ उसका Production check करना है।

(2) A → BC

Aabc$
BCabc

1) Types of DT. < ~~RD~~ LDT, RDT
2) ~~Ab~~ Ambi Gram

First

First (terminal) = terminal
~~First~~ (ε) = ε

4) E → TE' {id, ε}
E' → ~~*~~ TE'' / ε {*, ε}
T → FT' {id, (}
T' → ε / +FT' {ε, +}
F → id / (E {id, (}

1) S → abc / def / ghi
S → {a, d, g}

2) S → ABC / ghi / jkl — {a, b, c, g, j}
A → a/b/c — {a, b, c}
B → b — {b}
D → d — {d}

3) S → ABC — {a, b, c, d, e, ~~d~~, e, f, ε}
A → a/b/ε — {a, b, ε}
B → c/d/ε — {c, d, ε}
C → e/f/ε

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

$S \to aABb$	$\{a\}$	$\{\$\}$
$A \to c/\epsilon$	$\{c, \epsilon\}$	$\{d, b\}$
$B \to d/\epsilon$	$\{d, \epsilon\}$	$\{b\}$

≠

$S \to aBDh$	$\{a\}$	$\{\$\}$
$B \to cC$	$\{c\}$	$\{g, f, h\}$
$C \to bC/\epsilon$	$\{b, \epsilon\}$	$\{g, f, h\}$
$D \to EF$	$\{g, f, \epsilon\}$	$\{h\}$
$E \to g/\epsilon$	$\{g, \epsilon\}$	$\{f, h\}$
$F \to f/\epsilon$	$\{f, \epsilon\}$	$\{h\}$

Follow

① $S \to ACD$
$C \to a/b$

Follow(A) = {a, b}
Follow(D) = {$}
Follow(S) = {$}

② $S \to aSbS/bSaS/\epsilon$

Follow never contain ϵ

Follow(S) = {$, b, a}

③ $S \to AaAb / BbBa$
$A \to \epsilon$
$B \to \epsilon$

Follow(A) = {a, b}
Follow(B) = {b, a}
Follow(S) = $

④ $S \to ABC$
$A \to DEF$
$B \to \epsilon$

Follow(A) = {$}
Follow(D) = {$}

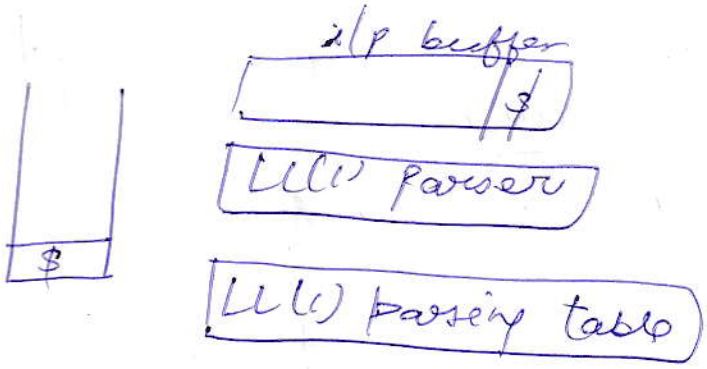
Bottom up parsing

Right most derivation.
in a reverse order
[derive from string]

aABe
aAde
aAbcde
abbcde

→ leftmost derivation
abcde →

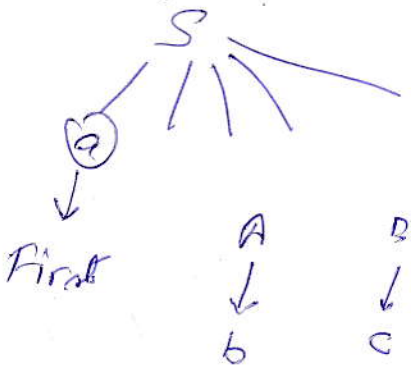
LL(1)
↓
Left to right → no. of look ahead (how many symbol)



First & Follow

First ( )

S → aABCD
A → b
B → c
C → d
A → e



First of S is a
in S → ABCD , first of S is b
S → ABCD
A → b/ε
first of S is b in case A → b
S is ε in case A → ε

If we have a variable or string from that variable if u try the string whatever

Top down

↓

Recursive Descent

↙ ↘

Back Tracking | Non Back tracking

↓

Predective Parser

↓

LL Parser

Recursive Descent parsing:

input is read from left to right.
and
parser is constructed from top.

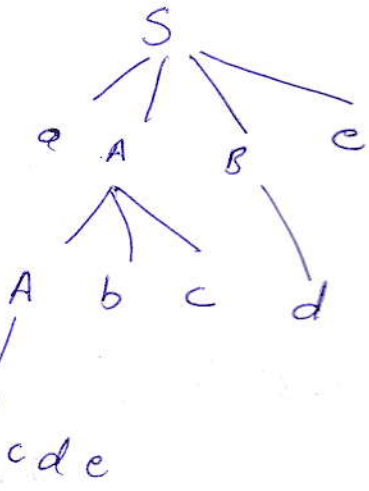
Ambiguous grammer is not used in any parser, except operator precedance parsing.

$S \rightarrow aABe$

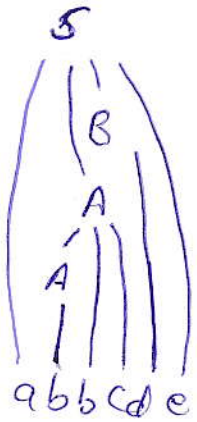
$A \rightarrow Abc / b$

$B \rightarrow d$

$W \rightarrow abbcde$



abbcde



TDP ①

What we use/choose it is main decision.

They following Left most derivation

$S \rightarrow aABe$

$aAbcBe$

When to reduce

POORNIMA
COLLEGE OF ENGINEERING
(1)
DETAILED LECTURE NOTES

Predictive Parsing Table:- (LL(1) Parser)

Input : Grammer G.

Output : Parsing table M.

Method :

1. For each production $A \to \alpha$ of the grammer, do steps 2 and 3.
2. For each terminal a in FIRST(α), add $A \to \alpha$ to M[A, a].
3. If ϵ is in FIRST(α), add $A \to \alpha$ to M[A, b] for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to M[A, \$].
4. Make each undefined entry of M be error.

eg:- Consider the grammer we make parsing table.

$E \to TE'$

$E' \to +TE' / \epsilon$

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to (E) \mid id$

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

this grammer have left-recursion

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

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

FIRST (E') = { +, ∈ }

FIRST (T') = { *, ∈ }

if we first we have ∈ then we take follow of that (E')

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

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

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

it is LL(1) grammer

NonTerminal	Input symbol					
	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ∈	E' → ∈
T	T → FT'			T → FT'		
T'		T' → ∈	T' → *FT'		T' → ∈	T' → ∈
F	F → id			F → (E)		

~~Parsing Table~~ [Predictive Parsing Table]

A Predictive parser is an efficient implementation of recursive descent parser and it by handling the stack of activation records explicitly.

eg :- with input id + id * id the predictive parser makes the sequence of moves as follows. and grammer is :

E → TE'

E' → +TE' / ∈

T → FT'

T' → FT' / ∈

F → (E) / id

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

28	L40	The viterbi Algorithm fee distance of a convolution codes	CO3	20/12/2021	20/12/2021	T1-213

If we get more than one entry in a cell then it means this grammer is not LL(1).

Study material

Textbook:

T1) Ranjan Bose: Information Theory, Coding and Cryptography ISBN 978-0-07-0669017, Tata McGraw-Hill.

T2) Simon Haykon: Communication Systems, ISBN- 0-471-17869-1, John Wiley & Sons, Inc.

Reference books:

R1) HWEI P. HSU, Ph.D. : Analog and Digital Communications: ISBN 0-07-140228-4, Schaum's Outline Series, McGraw-Hill.

Stack	Input	o/p (Production)
$E\$$	$id+id*id\$$	$E \to TE'$
$TE'\$$	$id+id*id\$$	$T \to FT'$
$FT'E'\$$	$id+id*id\$$	$F \to id$
$idT'E'\$$	$id+id*id\$$	
$T'E'\$$	$+id*id\$$	$T' \to \epsilon$
$E'\$$	$+id*id\$$	$E' \to TE'$
$+TE'\$$	$+id*id\$$	
$TE'\$$	$id*id\$$	
$FT'E'\$$	$id*id\$$	$T \to FT'$
$idT'E'\$$	$id*id\$$	$F \to id$
$T'E'\$$	$*id\$$	
$*FT'E'\$$	$*id\$$	$T' \to *FT'$
$FT'E'\$$	$id\$$	
$idT'E'\$$	$id\$$	$F \to id$
$T'E'\$$	$\$$	$T' \to \epsilon$
$E'\$$	$\$$	$E' \to \epsilon$
$\$$	$\$$	

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

28	L40	The viterbi Algorithm fee distance of a convolution codes	CO3	20/12/2021	20/12/2021	T1-213

If we get more than one entry in a cell then it means this grammer is not LL(1).

Study material

Textbook:

T1) Ranjan Bose: Information Theory, Coding and Cryptography ISBN 978-0-07-0669017, Tata McGraw-Hill.

T2) Simon Haykon: Communication Systems, ISBN- 0-471-17869-1, John Wiley & Sons, Inc.

Reference books:

R1) HWEI P. HSU, Ph.D. : Analog and Digital Communications: ISBN 0-07-140228-4, Schaum's Outline Series. McGraw-Hill.

Stack	Input	O/p (Production)
$E\$$	$id + id * id \$$	$E \to TE'$
$TE'\$$	$id + id * id \$$	$T \to FT'$
$FT'E'\$$	$id + id * id \$$	$F \to id$
$id\ T'E'\$$	$+ id * id \$$	$T' \to \in$
$T'E'\$$	$+ id * id \$$	$E' \to TE'$
$E'\$$	$+ id * id \$$	
$+TE'\$$	$+ id * id \$$	
$TE'\$$	$id * id \$$	$T \to FT'$
$FT'E'\$$	$id * id \$$	$F \to id$
$id\ T'E'\$$	$* id \$$	
$T'E'\$$	$* id \$$	$T' \to *FT'$
$*FT'E'\$$	$* id \$$	
$FT'E'\$$	$id \$$	$F \to id$
$id\ T'E'\$$	$id \$$	
$T'E'\$$	$\$$	$T' \to \in$
$E'\$$	$\$$	$E' \to \in$
$\$$	$\$$	

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Counting Sort (not comparison sort)

$(n \log n)$

→ Sorting acc to keys

→ Counting the element having distinct key values.

1 0 2 1 0 1 5 6 7 5 4 2 2 0 0 1

n = 17

0 — 7

k = 7 maximum value

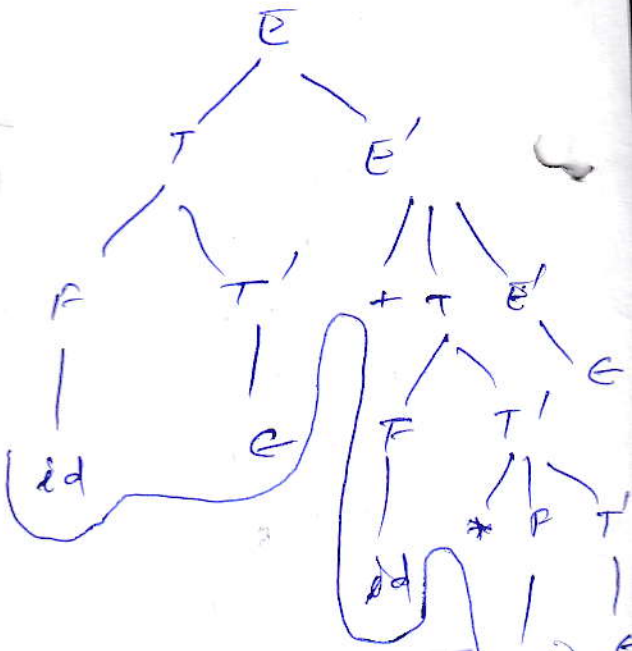
① $0 \le a[i] \le k$

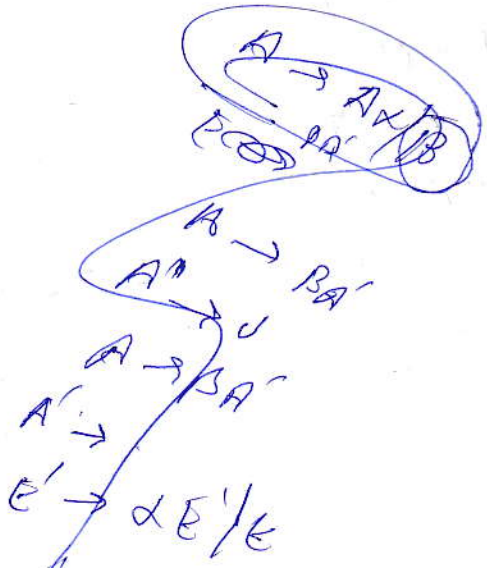
② $a[i] \in I$

count + 1 =

↕ TE (+TE'/ϵ)

Parse Tree





POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

Stack	Input	Output
$E	id + id * id $	
$ E'T	id + id * id $	$E \rightarrow TE'$
$ E'T'F	id + id * id $	$T \rightarrow FT'$
$ E'T' id	id + id * id $	$F \rightarrow id$
$ E'T'	+ id * id $	
$ E'	+ id * id $	$T' \rightarrow \epsilon$
$ E'T +	+ id * id $	$E' \rightarrow +TE'$
$ E'T	id * id $	
$ E'T'F	id * id $	$T \rightarrow FT'$
$ E'T' id	id * id $	$F \rightarrow id$
$ E'T'	* id $	
$ E'T'F *	* id $	$T' \rightarrow *FT'$
$ E'T' F	id $	
$ E'T' id	id $	$F \rightarrow id$
$ E'T'	$	
$ E'	$	$T' \rightarrow \epsilon$
$	$	$E' \rightarrow \epsilon$

When Top most symbol on stack & in input both are $ then it will exit from loop. This indicates successful parsing.

Predictive Parser means depending upon stack (or current situation) symbol and depending upon the input symbol, what is the next production rule, we are going to apply. It is called predictive parser.

Algorithm:- Nonrecursive Predictive Parsing.

I/P: A string w and a parsing table M for grammer G.

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

⇒ Set 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] = $X \rightarrow Y_1 Y_2 \cdots Y_k$ then begin

Pop X from the stack;

push $Y_k, Y_{k-1}, \cdots Y_1$ onto the stack, with Y_1 on top;

output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$

end

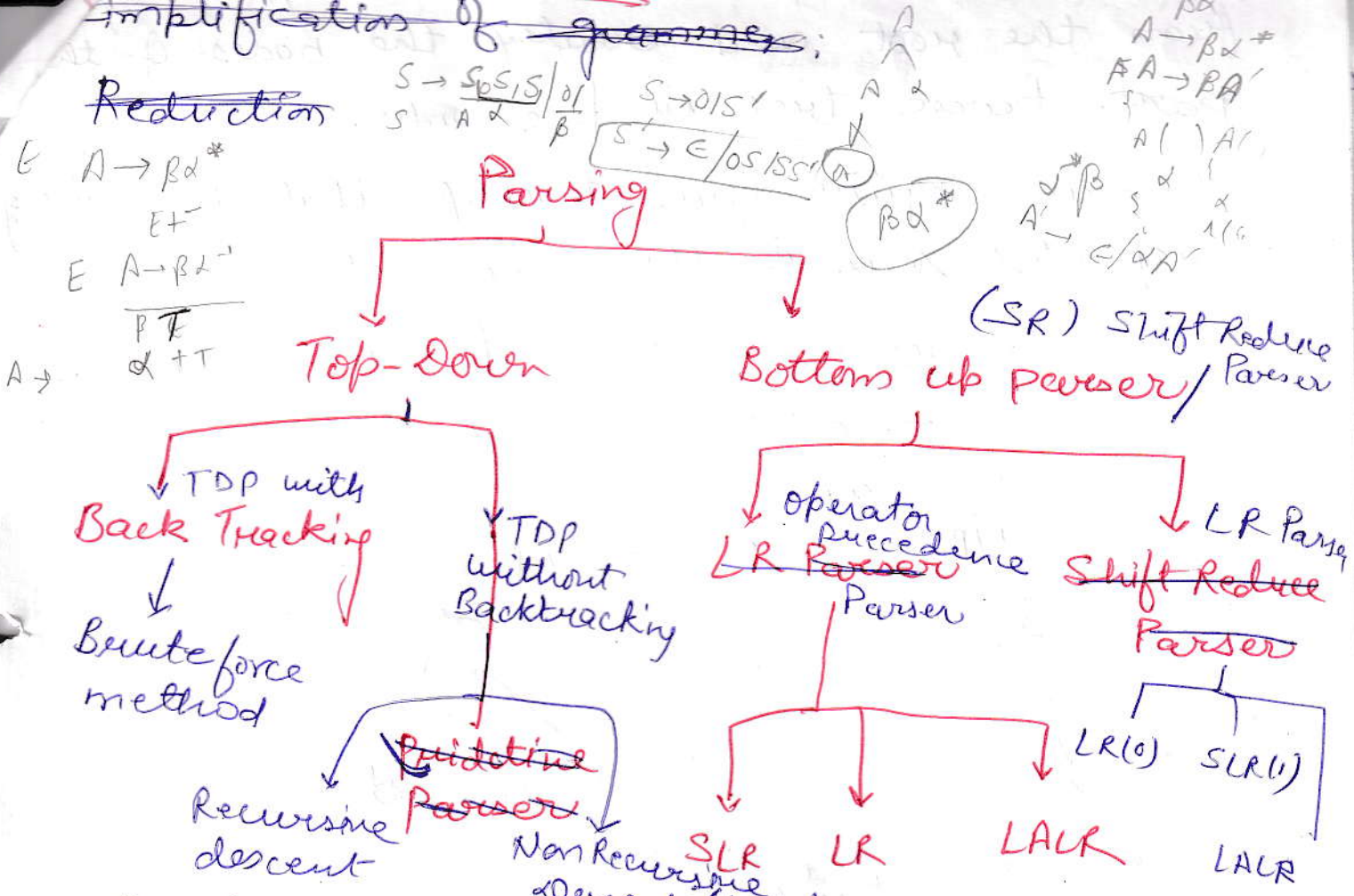
else error()

until X = $ / * stack is empty */

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

~~Simplification of grammar~~:

~~Reduction~~



Parser is s/w, which takes input stream as input and it will tell whether this string is in accordance with the grammer productions or not. that means it will either accept the string or reject the string with certain error.

Tutorial point video

Top down Parsing : Top down parsing attempts to build the parse tree from root to leaf. Top down parser will start from start symbol and proceeds to string. It follows leftmost derivation. In leftmost derivation the leftmost non-terminal in each ~~sentence~~ is always chosen.

constructs a parse tree for the i/p st
from the root and creating the nodes of
parse tree in pre-order.

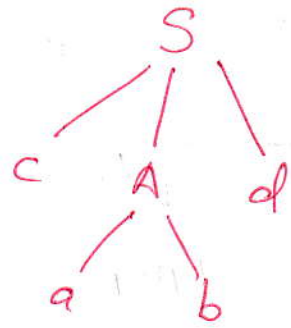
It may require back-tracking. that is, making repeated scan of the input string.

eg:- Let us consider the following grammar.

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

let i/p string $\omega = cad$



it is not correct.] it is called backtracking.

This is disadvantage in backtracking.

To resolve this backtracking we must have some method.

Recursive descent parser

TDP constructed for the grammer if it is free from ambiguity & left recursion.

Classification of TDP:-

with backtracking → brute force technique

without backtracking → Predictive parser

$G = \{ S, A \}, \{ a, b \}, S, \{ S \to aAb, aA \to aaAb, A \to \epsilon \}$

Lang string generate karni hai

$S \to aAb$

$\hookrightarrow S \to aaAbb \quad [aA \to aaAb]$

$S \to aaabbb \qquad S \to aabb$

leaves → terminal
Vertex → NT
Root → S

3, Sum, D...

Aakash Ka

V T S P

$V \to ABS$ ⌐SS

$T = 0, 1$

→ Regular Grammer

$G = \{ S, A \}, \{ a, b \}, S, \{ S \to aAb, aA \to aaAb, A \to \epsilon \}$

$S \to aAb$

$\hookrightarrow S \to aaAbb \quad [aA \to aaAb]$

$S \to aaaAbbb$

$aabb \quad [A \to \epsilon]$

$S \to aaabbb \quad [A \to \epsilon]$

$(a^n b^n)$

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

$S \to AB$

↳ $S \to aB$ [$A \to a$]

↳ $S \to ab$ [$B \to b$]

↳ desired Lang. for particular Grammar

$G = (\{S, A, B\}, (a, b), S, \{S \to AB, A \to aA \mid a, B \to bB \mid b\})$

1 → b

A → aA

A → a

① $S \to AB$

↳ $S \to aB$ [$A \to a$]

↳ $S \to ab$ [$B \to b$)

② $S \to AB$

$S \to aAB$ [$A \to aA$]

↳ $S \to aAbB$ [$B \to bB$]

↳ $S \to aabb$ [$A \to a$; $B \to b$]

③ $S \to AB$

$S \to aAB$

$S \to aab$

④ $S \to AB$

$S \to AbB$

$S \to abb$

$L = \{ab, a^2b^2, a^2b, ab^2 \ldots\}$

$L(G) = \{a^m b^n, m \geq 0, \text{ and } n \geq 0\}$

Grammar 01

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Nodes → ~~represent by~~ ~~left side~~

nodes are labeled with the left side of production and in which the children of a node represent its corresponding right-sides.

$G = [V, T, S, P\}, S \to OB, A \to 1AA \} | \in, B \to OAA$

$V = A, B, S$ → start symbol

$T = 0, 1$

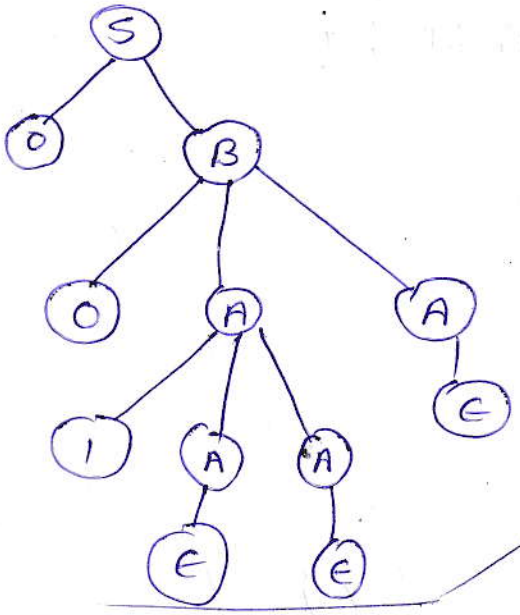
(S) Root vertex → Start symbol

(A, B) vertex → Non terminal

(child) Leaves → Terminal & symbol $\in$

(a, b - $\in$)

D.T.



We have to create D.T. for ~~D.T.~~ particular string.

(2) $G = (V, T, S, P)$, $S \to a$, $S \to aAS$, $A \to bS$

Obtain the D.T. for the string word →

$w = abaaba$

$S \to aAS$

$S \to abSS$ [$A \to bS$]

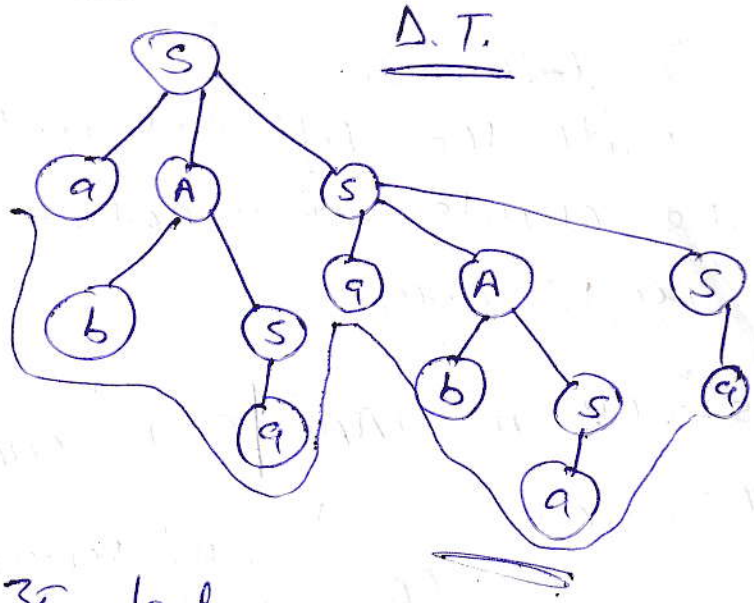
$S \to abaS$ [$S \to a$]

$S \to abaaAS$ [$S \to aAS$]

$S \to abaabSS$ [$A \to bS$]

$S \to$

D.T.



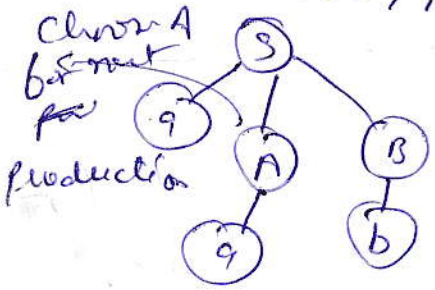
Yield

Types of D.T.

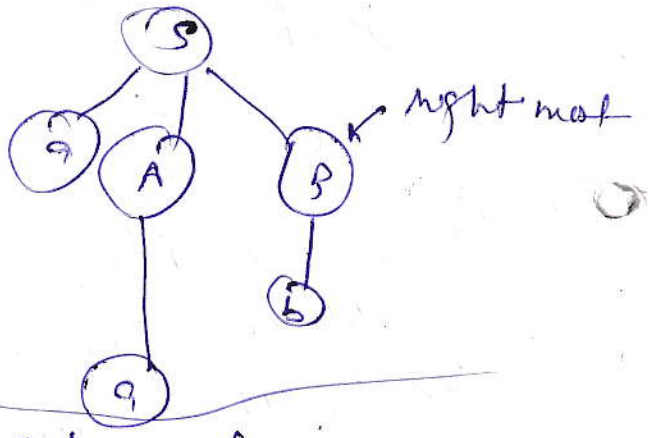
Left D.T.

→ Obtained by applying production to the leftmost variable in each step.

$S \to aAB,\ A \to a,\ B \to b$

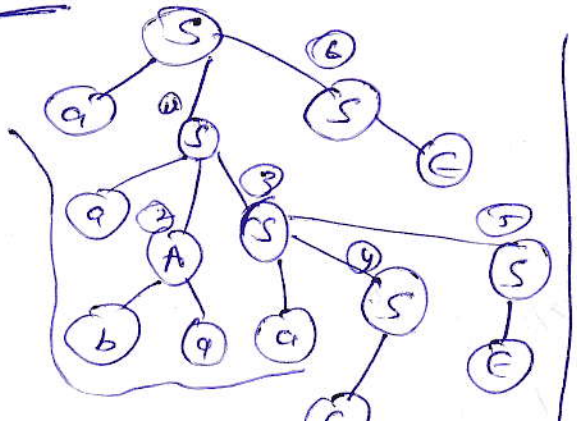


Right D.T.

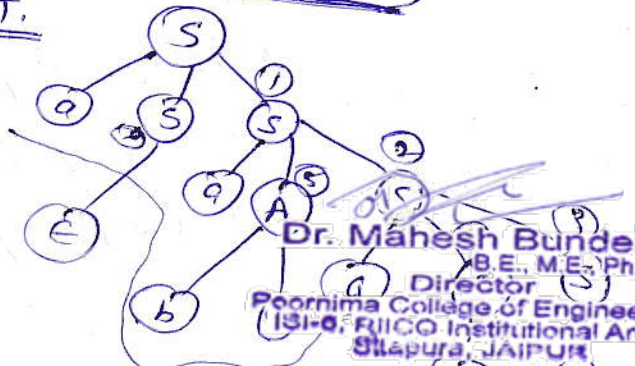


Q. $S \to aAS \mid aSS \mid \epsilon,\ A \to SbA \mid ba$ (aabaa)

L.D.T.



R.D.T.



POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

Bottom up Parsing: [LR / shift-reduce Parser:]

Bottom up Parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. It is known as the shift-reduce parsing. This process as one of "reducing" a string w to the start symbol of a grammer.

eg: Consider the grammer

$S \rightarrow aABe$

$A \rightarrow Abc/b$

$B \rightarrow d$

The sentence abbcde can be reduced to S by the following steps:

abbcde
aAbcde
aAde
aABe
S

Shift-Reduce Parsing:-

Shift-reduce Parsing uses two unique steps for bottom-up Parsing. These steps are known as shift-step and reduce-step.

→ Shift step: The shift step refers

input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.

→ Reduce step: when the parser finds a complete grammer rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which POPs off the handle and replaces it with LHS non-terminal symbol.

LR Parser

L → Left to Right scanning of the input
R → Right most derivation in reverse order

eg:-

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

id * id + id (grammer)

/

E * id + id

/

E

E * E + id

E * E + E

↓

E + E

↓

E

30 easy expl → Handles: A handle of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production

of a rightmost derivation.

In many cases the leftmost substring β that matches the right side of some production $A \to \beta$ is not a handle, because a reduction by the production $A \to \beta$ yields a string that cannot be ~~reduced~~ ~~reduction~~ reduced to the start symbol.

A handle of a right sentential form ($\gamma \equiv \alpha\beta w$) is a production rule $A \to \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

$$S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$$

If the grammer is ~~was~~ unambiguous, then every right-sentential form of the grammer has exactly one handle.

eg. $S \underset{rm}{\to} aABe \underset{rm}{\to} aAde \underset{rm}{\to} aAbcde \underset{rm}{\to} abbcde$

Production Rule:
$S \to aABe$
$A \to Abc \mid b$
$B \to d$

abbcde : $\gamma = abbcde$, $A \to b$, Handle = b

aAbcde : $\gamma = aAbcde$, $A \to Abc$, Handle = Abc

aAde : $\gamma = aAde$, $B \to d$, Handle = d

aABe : $\gamma = aABe$, Handle = aABe

3)

Handle Pruning:

A rightmost derivation in reverse can be obtained by "handle Pruning".

Removing the children of left hand side non-terminal from the Parse tree is called Pruning.

Steps to follow:-

→ Start with a string of terminals w that is to be parsed.

→ Let $w = \gamma_n$, where γ_n is the n^{th} right sentential form of unknown rightmost derivation (RMD).

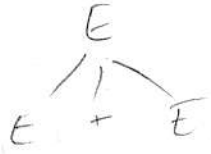
→ To reconstruct the RMD in reverse, locate handle β_n in γ_n; Replace β_n by LHS of some $A_n \to \beta_n$ to obtain Right-sentential form (RSF) γ_{n-1}. Repeat.

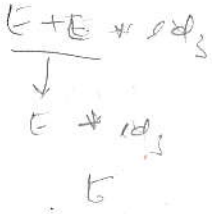
Right sentential form	Handle	Reducing Production
$id_1 + id_2 * id_3$	id_1	$E \to id$
$E + id_2 * id_3$	id_2	$E \to id$
$E + E * id_3$	id_3	$E \to id$
$E + E * E$	$E * E$	$E \to E * E$
$E + E$	$E + E$	$E \to E + E$
E		

where Grammer is $E \to E+E \mid E*E \mid id$

$$S \Rightarrow \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

here start with w and apply pro on - - -





POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

32 Stack implementation of Shift-Reduce Parsing:

A convenient way to implement a shift-reduce parser is to use a stack to hold grammer symbols and an input buffer to hold the string w to be parsed.

We use $ to mark the bottom of the stack and also the right end of the input.

Stack	Input
$	w$

→ Major Actions performed are:

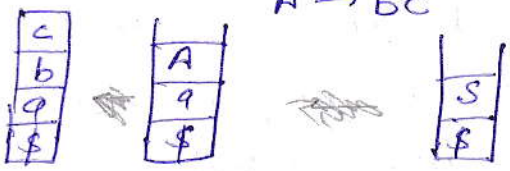
1. Shift: PUSHING
   The next input symbol is shifted onto the top of the stack.
2. REDUCE: POPPING
   Popping the handle whose right end is at Top of stack and replacing it with left side Non-terminal.
3. Accept: it denotes successful completion of Parsing.
4. Error: the parser discovers that a syntax error has occured and calls an error recovery routine.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Stack implementation of SR Parser :

* Shift input symbols onto stack untill a handle β is on Top of stack.
# Reduce β to left side Non Terminal of appropriate production.
* Repeat untill error or stack has the start symbol left and input is empty.

$S \rightarrow aA$
$A \rightarrow bc$

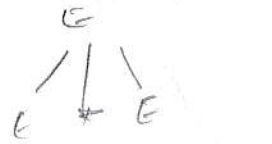


33. Perform Shift-Reduce Parsing Using a STACK

STACK CONTENT	INPUT (id + id * id)	ACTION
$	$id_1 + id_2 * id_3$ $	Shift
$ id_1	$+ id_2 * id_3$ $	reduce by $E \rightarrow id$
$ E	$+ id_2 * id_3$ $	shift
$ E +	$id_2 * id_3$ $	shift
$ E + id_2	$* id_3$ $	reduce by $E \rightarrow id$
$ E + E	$* id_3$ $	shift
$ E + E *	id_3 $	shift
$ E + E * id_3	$	reduce by $E \rightarrow id$
$ E + E * E	$	reduce by $E \rightarrow E * E$
$ E + E	$	reduce by $E \rightarrow E + E$
$ E	$	accept

$E + E \rightarrow E$
$E * id_3$
$E \rightarrow id_3$
$E * E$
$E \rightarrow E * E$

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



POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

34.

Why we use STACK for SR-parsing:

Any handle will always appear on the top of stack and the parser need not search within the stack at any time.

Conflicts in SR-Parsing:

two decisions decide a succeful SR-Parsing

- locate the substring to reduce that has to be reduce
- which production to chose when multiple productions with the selected substring on their right hand side exist.

SR Parser may reach a configuration in which knowing the contents of stack and input buffer, still the parser cannot decide:

- wheather to perform a shift or a Reduce operation (Shift-Reduce conflict)
- ~~Reduce~~ which out of the several possible reductions to make (Reduce-Reduce conflicts)

eg:- S → if E then S |
if E then S else S |
other

Stack

Input Buffe

if E then S

Q: What happen when shift operation occur and another case when reduce operation occur

35. This is the eg of SR conflict [How conflict occur in SR parsing?]

eg:- $E \to E+T/T$
$T \to T*F/F$
$F \to (E)/x$

STACK	INPUT	Action
$	x * x $	Shift
$ x	* x $	Reduce
$ F	* x $	Reduce
$ T	* x $	Shift o
$ T *	x $	shift
$ T * x	$	Reduce
$ T * F	$	Reduce.
$ T	$	Reduce
$ E	$	Accept

STACK	INPUT	ACTION
$	x * x $	Shift
$ x	* x $	Reduce
$ F	* x $	Reduce
$ T	* x $	Reduce
$ E	* x $	Shift
$ E *	x $	Shift
$ E * x	$	Reduce
$ E * F	$	Reduce
$ E * T	$	Reduce
$ E * E	$	ERROR

POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

Operator Precedence Parsing:

Shift Reduce Parsers can be built successfully using for 2 main classes of grammer

- → LR grammer
- → operator grammer.

Property of operator grammer:

1. No production in the grammer has ∈ on the right hand side.
2. No 2 Non terminals appear together on RHS of any production.

eg: 1. $E \to EAE \mid (E) \mid -E \mid id$

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

↓ Violates rule 2 → It is Not an operator grammer.

if we substitute for A each of its alternative, the we get,

$E \to E+E \mid E*E \mid E-E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid id$

This is an operator precedence grammer.

eg: 2 $S \to SAS \mid a$
$A \to bSb \mid b$ } not in the form of OPG.

$S \to SbSbS \mid a \mid SbS$ this is in the form of OPG.

1, 5, 6, 8, 9, 10, 16, 20, 25, 34

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

32.

In operator-precedence parsing, we define three disjoint precedence relations : $<\cdot$, $\doteq$, $\cdot>$ between certain pairs of terminals.

Use → They help us in selection of handles.

Meaning :-

Relation	Meaning
$a <\cdot b$	a has lower precedence than b
$a \cdot> b$	a has higher precedence than b
$a \doteq b$	a has equal precedence as b

How to assign relations :

↳ Using the concept of associativity and Precedence

↳ For all terminals (including $) we design an operator precedence table.

Operator precedence Table :

eg :- grammer is $E \to E+E \mid E*E \mid id$

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

[Right sentential form
id + id * id]

Operator - precedence relations

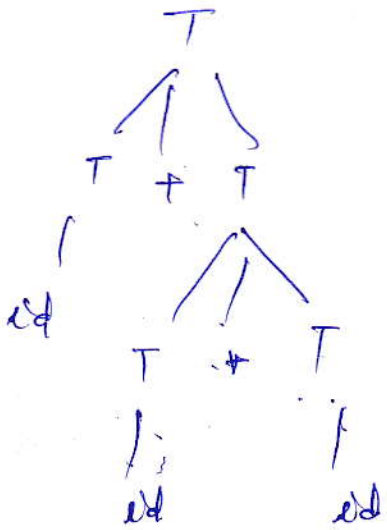
Symbol on TOS — Symbol in I/P string

If (I/P str sym) $\cdot>$ TOS : PUSH I/P symbols.

else (I/P str sym) $<\cdot$ TOS POP

else ERROR

STACK	Action	Input	Comment
$		id + id * id $	
$ id		+ id * id $	Shift id
$ T		+ id * id $	Reduce T → id
$ T +		id * id $	Shift +
$ T + id		* id $	Shift id
$ T + T		* id $	Reduce id
$ T + T *		id $	Shift *
$ T + T * id		$	Shift id
$ T + T * T		$	Reduce id
$ T + T		$	T → T * T
$ T	Acc	$	T → T + T



a + b * c * d

E → E + T / T

T → T * V / V

V → a / b / c / d

POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

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

I.S.> TOS : Push
I.S.< TOS : Pop & reduce

eg:-

Input String	Stack
id + id * id $	$
+ id * id $	$ id
+ id * id $	$
id * id $	$ +
* id $	$ + id
* id $	$ +
id $	$ + *
$	$ + * id
$	$ + *
$	$ +
$	$

—(1)
—(2)
—(3)

OPG is only grammer that is able to handle the ambiguous grammer.

~~OPG is only grammer that is handle the ambiguous grammer~~

2, 5, 6, 7, 8, 9, 10, 11, 12, 16, 20, 21, ~~22~~, ~~24~~ 25

42, ~~50~~, 5

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

39 Advantage of operator Precedence Parsing:

1. Simplicity of method itself
2. Error detecting capability of parser
3. Capability of parser to parse ambiguous grammar (single parse Tree)

Disadvantage of operator precedence Parsing:

1. Only a small class of grammer can be parsed using O.P. Technique.
2. Difficult to handle operators which may have different roles in different situations
3. As the number of terminal increase the size of the Precedence table also increases.

→ Solution of 3rd disadvantage is Operator Precedence Functions: Map terminals to integers (symbols)

In Operator Precedence function, select 2 functions of f and g such that for symbols a and b

(1) $f(a) < g(b)$ when $a < b$
(2) $f(a) = g(b)$ when $a = b$
(3) $f(a) > g(b)$ when $a > b$

Thus the precedence relation between a and b can be determined by a numerical comparison between $f(a)$ and $g(b)$.

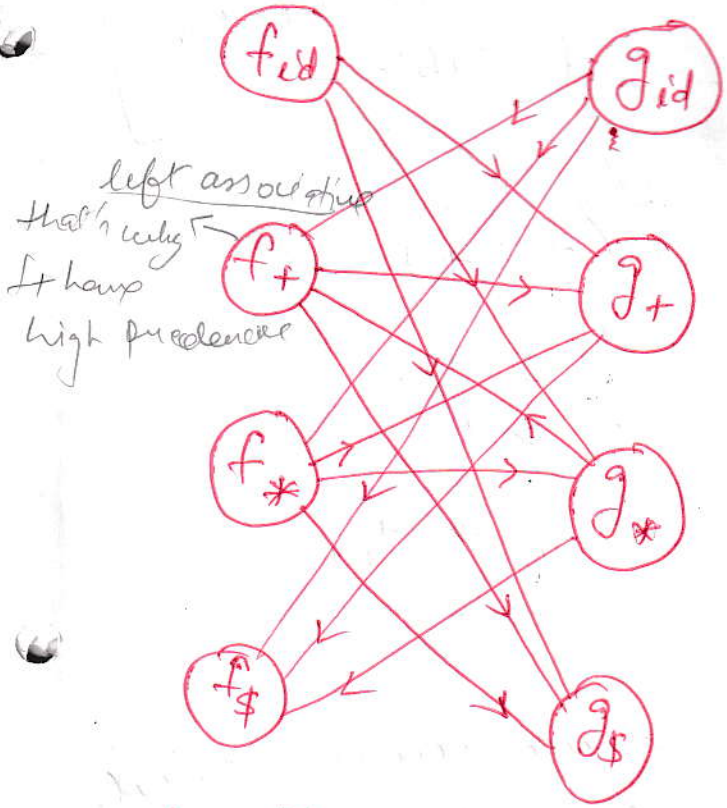
Not every table of precedence relations has precedence functions to encode it. But practical cases the functions usually

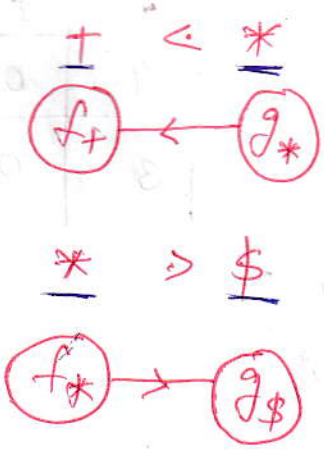
POORNIMA
COLLEGE OF ENGINEERING
DETAILED LECTURE NOTES

40 Q: Operator Precedence functions :-

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

set of terminal { $, +, *, id }





1. id have high precedence ~~to~~ ~~add~~ of it compare to other operator.
2. left + have high precedence according to left associative rule.
3. $ have least precedence.
4. * have high precedence
5. we do not [illegible]

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Now we find out a path from every node and check what is the maximum length / max. no. of edges.

* $f_{id} \rightarrow g_+ \rightarrow f_\$$

$g_{id} \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

$g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

$f_{id} \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

$g_{id} \rightarrow f_* \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

Precedence table for functions: [Operator precedence ~~table~~ function table :-]

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

$* > +$

$f_* > g_+$

$4 > 1$

$(+, +)$

$f_+ \geq g_+$

$2 \geq 1$

$+_{LHS} > +_{RHS}$.

There are no cycles, so precedence functions exist. As $f_\$$ and $g_\$$ have no out-edges, $f(\$) = g(\$) = 0$. The longest path from g_+ has length 1, so $g(+) = 1$.

41. LR Parsing :

It is Bottom up technique to perform syntax analysis.

L stands for left to right scanning of the input.

R stands for constructing a right most derivation in reverse.

K is the number of input symbols of the look ahead used to make number of Parsing decision.

Benefits of LR Parser :

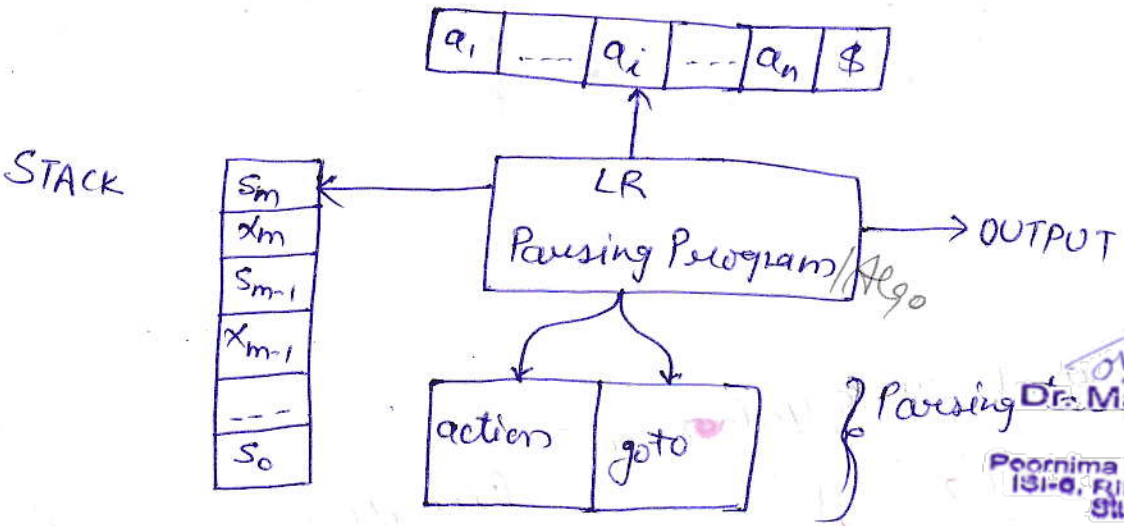
1. Most generic non Backtracking shift Reduce Parsing Technique.
2. These parser can recognize all programming Languages for which context-free-grammer can be written.
3. They are capable of detecting syntactic error as soon as possible in scanning of input.

Types of LR Parser :

(LR(0), LR(1), LALR(1), CLR(1))

1. Simple LR Parser (SLR) : LR(0) Parser, SLR(1) : Simple but least efficient to detecting errors
2. Canonical LR Parser (CLR) : most powerful to detect error. expensive also.
3. Lookahead LR Parser (LALR) : intermediate in terms of (Power & error), cost.

Component of LR Parser :



Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA
COLLEGE OF ENGINEERING
DETAILED LECTURE NOTES

42.

Component :

1. Input buffer
2. Stack
3. Parsing Algo
4. Parsing table

→ Parsing table changes from one LR parser to another.

→ Parsing algo ~~also~~ called the driver program, remains same for all LR parsers.

Behaviour of LR Parser :

→ Parsing algorithm reads the next unread i/p char from the input buffer.

→ Parsing algo also reads the character on the top of stack.

A stack can have grammer symbol (X_i) or state symbols (S_i).

→ Combination of i/p char and T.O.S. char is used to index parsing table

Parsing table consist of two parts :

1. Action
   - → Shift
   - → Reduces
   - → Error
2. go to takes a state and grammer symbol as argument and produces a state.

43. Parsing Action Performed by LR-Parser :

A configuration of an LR Parser is a pair whose first component is the stack contents and whose second component is the ~~unexpected~~ unexpended input :

$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$

The next move of the parser is determined by reading a_i, the current input symbol and s_m, the state on top of the stack, and then consulting the parsing action table entry action $[s_m, a_i]$.

Case 1 If action $[s_m, a_i]$ = shift s, the parser executes a shift move, entering the configuration

$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$)$

Case 2 if action $[s_m, a_i]$ = reduce $A \to \beta$, then the parser executes a reduce move, entering the configuration

$(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n \$)$

(· state s_{m-r} appears on T.O.S.
· Parser pushes A & goto $[s_{m-r}, A] = s$

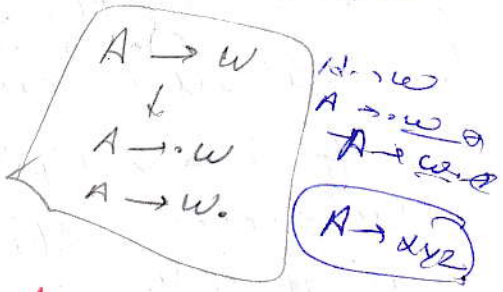
3. If action $[s_m, a_i]$ = accept, Parsing is completed.
4. If action $[s_m, a_i]$ = error, the ~~parser~~ parser has discovered an error and calls an error recovery routine.

[Parsing program/ driver program/

POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

LR(0) Items :- LR(0) item of a grammer G is a production of G with a dot at some position of the right side. eg:- a production $A \to XYZ$ have four items $A \to .XYZ$, $A \to X.YZ$, $A \to XY.Z$, $A \to XYZ.$



The production $A \to \epsilon$ generates only one item, $A \to .$

What does an item indicate :- It indicate, how much part of a production we have seen at a given point in parsing process.

Augumented Grammer : If G is a grammer with start symbol S, then G', the Augument grammer for G, is G with a new start symbol S' and production $S' \to S$.

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

Kernel items : which include the initial item, $S' \to .S$, and all items whose dots are not at the left end.

Non-kernel item : All those LR(0) items which have dot at the leftmost position (except augumented prod^n : $S' \to .S$)

45 Closure Function :

If I is a set of items for a grammer G, then closure (I) is the set of items constructed from I by the two rules :

1. Initially, every item in I is added to closure (I).
2. If $A \to \alpha . B\beta$ is in closure (I) and $B \to \gamma$ is a production, then add the item $B \to .\gamma$ to I, if it not already there. We apply this rule until no more new items can be added to closure (I).

eg: Consider the augumented expression grammer :

$E' \to E$
$E \to E+T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

Construct a set I of all LR(0) items for the given grammer.

If I is the set of one item $\{[E' \to .E]\}$, then closure (I) is

Closure (I) $\to$ $E' \to .E$
$E \to .E+T$ } closure
$E \to .T$
$T \to .T * F$ } closure
$T \to .F$
$F \to .(E)$ } closure

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

we compute closure whenever there is a dot to the immediate left of a non-terminal and the ~~NT~~ non terminal has not yet been expanded. Expansion of such ~~NT~~ non-terminal into items with dot at extreme left is called closure.

46.

Go-to operation

goto (I, X) where I is a set of items and X is a grammer symbol. goto (I, X) is defined to be the closure of the set of all items $[A \to \alpha X, \beta]$ such that $[A \to \alpha . X \beta]$ is in I. if I is the set of items that are valid ~~for~~ for some viable prefix γ, then goto (I, X) is the set of items that are valid for the viable prefix γX.

[We take augumented production always when we start closure or go to operation.]

In Goto function we shift the dot one step to the right. After computing we again check for closure.

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

I_1: $E' \to E \cdot$ — goto (I_0, E) no further closure
$E \to E \cdot + T$

goto (I_0, T)

I_2: $E \to T \cdot$ no further closure
$E \to T \cdot * F$

I_3: $T \to F \cdot$ — goto (I_0, F) no more closure

I_4: $F \to (\cdot E)$
$E \to \cdot E + T$ ← goto $(I_0, ()$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I_5: $F \to id \cdot$ goto (I_0, id)

I_6: $E \to E + \cdot T$ goto $(I_2, +)$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I_7: $T \to T * \cdot F$ goto $(I_2, *)$
$F \to \cdot (E)$

I_8: $F \to (E \cdot)$ goto (I_4, E)
$E \to E \cdot + T$

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

I_{10}: $T \to T * F \cdot$ goto (I_7, F)

I_{11}: $F \to (E) \cdot$ goto $(I_8,))$

file

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

47. How to find Canonical Collection of LR(0) items.

Grammer is $S \to AA$ - (1)
$A \to aA/b$ - (2)

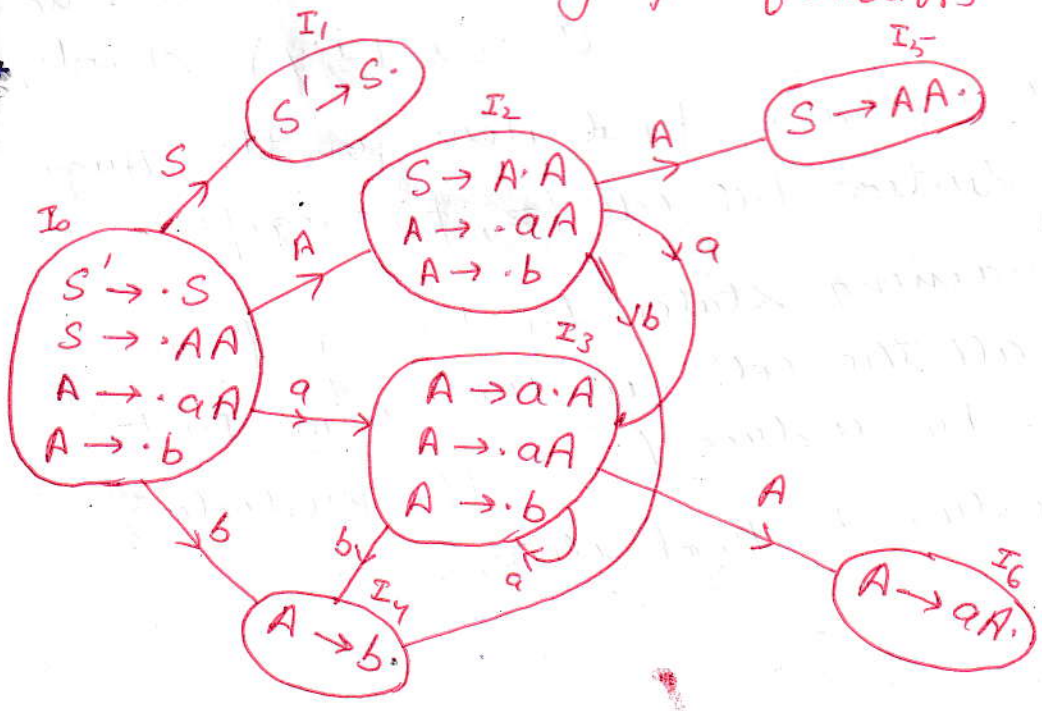
Step 1 augement Grammer is $S' \to S$
$S \to AA$
$A \to aA/b$

Step 2 LR(0) item of augement production

$S' \to .S$
$S \to .AA$
$A \to .aA$
$A \to .b$

now cannot apply closure because of ~~term~~ ~~subs~~ having terminal symbol.

Step 3 now apply goto function



[Having dot symbol at rightmost end, is called final item.

$A \to .aA$
$\Downarrow$
$A \to a.A$
$A \to .aA/.b$

Canonical collection is the set of all ...

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

48. **LR(0) Parsing table :**

	Action			GoTo	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Acc		
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3	R_3		
5	R_1	R_1	R_1		
6	R_2	R_2	R_2		

$S \rightarrow AA$ — ①
$A \rightarrow aA$ — ②
$A \rightarrow b$ — ③

Steps to construct parsing table :

1. Write all states numbers in leftmost column.
2. Divide the parsing table into two parts : Action and GoTo
3. For every state I_i if there is a transition on Non Terminal X, to state I_j, fill cell $(i, X) = j$ in PT.
4. For every state I_i if there is a transition to state I_y on terminal 'y' then fill cell $(i, y) = shift\ j$ (S_j)
5. For the state I_z having final item for the Augmented Production fill cell $(z, \$) = accept$
6. For all remaining states I_y having final items fill all the cells in the action part of row y, by reduce (n) → the production no. corresponding to final item.

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

SLR Parsing table:

(1) $S \rightarrow AA$ $

(2) $A \rightarrow aA$

(3) $A \rightarrow b$

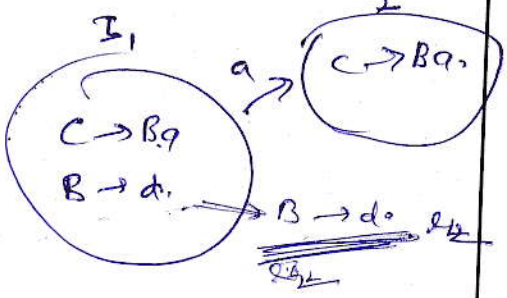
	Action			GoTo	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Acc		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5			r_1		
6	r_6	r_6	r_6		

Reduce

$A \rightarrow b$.

$S \rightarrow AA$.

$A \rightarrow aA$.

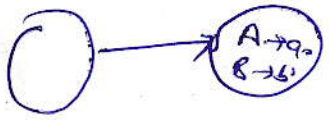


$A \rightarrow a$ (1)

$B \rightarrow b$ (2)

$C \rightarrow aB$ (3)

RR Reduce Reduce



	Action			GoTo
	a	b	$	
I_1	r_1/r_2	r_1/r_2	r_1/r_2	

SR Shift Reduce conflict

	Action			GoTo
	a	b	$	
I_1	S_2/r_2	r_2	r_2	
I_2				

Questions & Summary:

Ref.:-

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

49. Parsing a string using LR(0) Parsing Table:

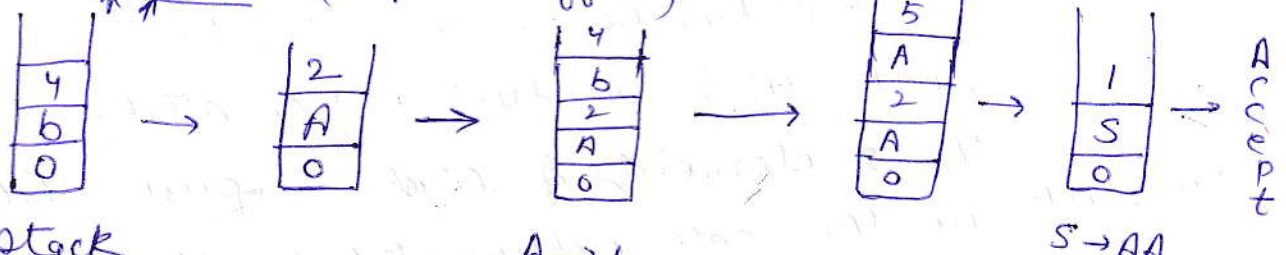
$S \to AA$ — (1)

$A \to aA/b$ — (3) (2)

We take string **bb**

Step 1 append $ symbol in input string

bb $ ← (input buffer)



Stack

$A \to b$ — 2

$A \to b$ — 2

$S \to AA$ — 2 ⇒ 2×2 = 4

1 → (2) twice the length → repeat the 2 element from stack

A	A A	S
|	| |	/ \
b b $	b b $	A A
		| |
		b b $

[1st element of stack should be always initial state.]
↳ top of the stack

In shift operation, we push element in the stack and move the input pointer forwards.

R_3 means reduction by production no. 3 [in this eg. $A \to b$]

→ When we get reduce move, we construct

Steps :-

Read top 2 elements of stack & push state present in the cell represented by these symbols.

- Push state 0 on stack.

  If I/p denotes the current input symbol, T.O.S. denotes top of stack symbol:

  Read cell (T.O.S., I/P)

  ↳ If entry is S_i, shift / push the I/P symbol followed by state i.

  ↳ If entry (T.O.S., I/P) is reduce j:

  ↳ see production no. j

  If RHS has n symbols, pop 2n symbols, push LHS of production j on stack.

  Read top 2 elements of stack & push state present in the cell represented by these symbols.

42

Canonical collection of LR(0) items for SLR Parser:

$E' \rightarrow .E$

$E \rightarrow .E+T$

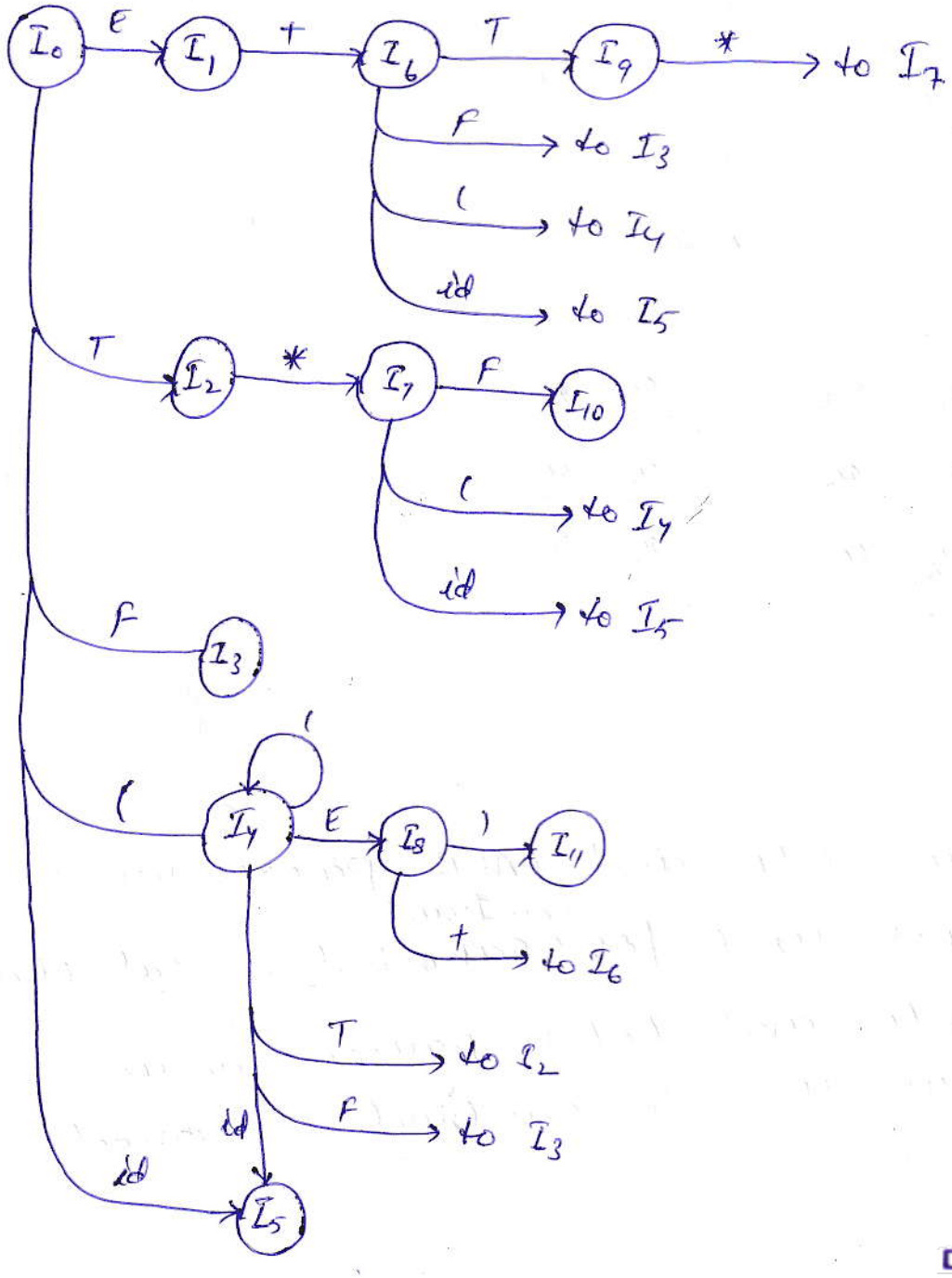
$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

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

By using follel we get:



Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

SLR(1) Parsing table:-

	id	+	*	(	)	$	E	P	T
0	S_5			S_4					
1		S_6				ACC			
2		R_2	S_7		R_2	R_2			
3		R_4	R_4		R_4	R_4			
4				S_4			8	2	3
5		R_6	R_6		R_6	R_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		R_1	S_7		R_1	R_1			
10		R_3	R_3		R_3	R_3			
11		R_5	R_5		R_5	R_5			

For construct CLR and LALR parser, we use LR(1) items used for construct canonical collection.
and SLR(1) and LR(0) parser, we use LR(0) items used for construct canonical collection.

LR(0) Parsing Table :-

	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				Acc			
2	R_2	R_2	S_7	R_2	R_2	R_2			
3	R_4	R_4	R_4	R_4	R_4	R_4			
4				S_4			8	2	3
5	R_6	R_6	R_6	R_6	R_6	R_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9	R_1	R_1	S_7	R_1	R_1	R_1			
10	R_3	R_3	R_3	R_3	R_3	R_3			
11	R_5	R_5	R_5	R_5	R_5	R_5			

Production no.

~~E' → E~~

E → E + T / T — (2)

T → T * F / F — (4)

F → (E) / id — (6)

(1), (3), (5)

POORNIMA
COLLEGE OF ENGINEERING
DETAILED LECTURE NOTES

53

Construction of LR(1) items:-

eg 1. grammer is $S \to CC$
$C \to cC / d$

for this grammer, we construct canonical collection of LR(1) items for CLR and LACR Parser then first we add augmented prod^{n}:-

$S' \to S$
$S \to CC$
$C \to cC / d$

I_0: $S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$

I_1: $S' \to S \cdot, \$$

I_2: $S \to C \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

I_3: $C \to c \cdot C, c/d$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$

I_4: $C \to d \cdot, c/d$

I_5: $C \to cC \cdot, c/d$

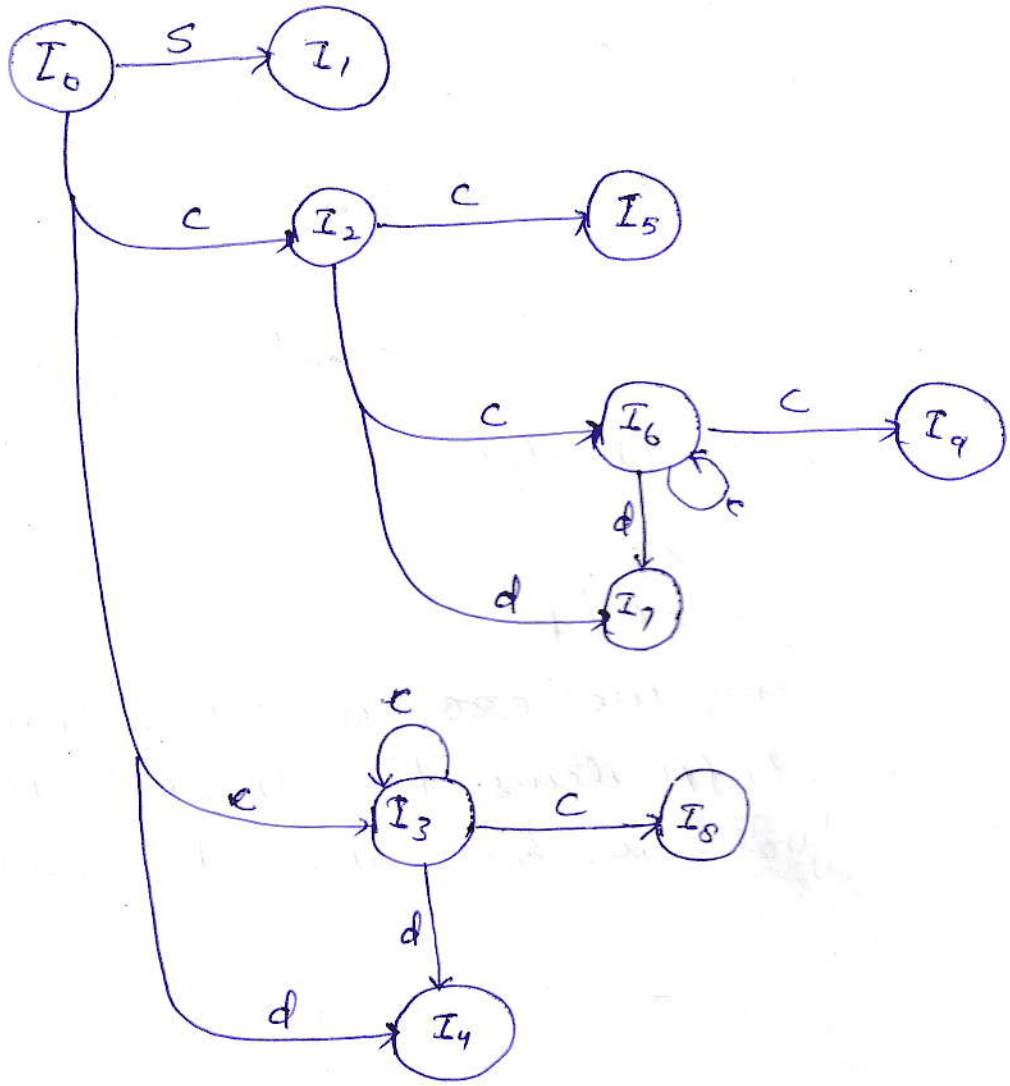
I_6: $C \to c \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

I_7: $C \to d \cdot, \$$

I_8: $C \to cC \cdot, c/d$

I_9: $C \to cC \cdot, \$$

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR



[canonical collection of LR(1) items.]

54.

eg:-2

$S \rightarrow A_a A_b / B_b B_a$

$A \rightarrow \in$

$B \rightarrow \in$

Solve it:

POORNIMA

$S \to CC$
$C \to CC/d$

$S' \to S$ → AC
$S \to CC$ — ①
$C \to CC/d$ — ②, ③

I_0: $S' \to \cdot S, \$$; $S \to \cdot CC, \$$; $C \to \cdot CC/\cdot d, (C,d)$
I_1: $S' \to S \cdot, \$$
I_2: $S \to C \cdot C, \$$; $C \to \cdot CC/\cdot d, \$$
I_3: $C \to C \cdot C, (C,d)$; $C \to \cdot CC/\cdot d, (C,d)$
I_4: $C \to d \cdot, (C,d)$
I_5: $S \to CC \cdot, \$$
I_6: $C \to C \cdot C, \$$; $C \to \cdot CC, \$$; $C \to \cdot d, \$$
I_7: $C \to d \cdot, \$$
I_8: $C \to CC \cdot, (C,d)$
I_9: $C \to CC \cdot, \$$

$S \to AA$ — ①
$A \to aA$ — ②
$A \to b$ — ③

	Action			GOTO	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Acc		
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3	R_3		
5	R_1	R_1	R_1		
6	R_2	R_2	R_2		

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

DETAILED LECTURE NOTES

55

CLR Parsing table for eg 1.

State	Action c	Action d	Action $	Goto S	Goto C
0	S_3	S_4		1	2
1			Acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	R_3	R_3			
5			R_1		
6	S_6	S_7			9
7			R_3		
8	R_2	R_2			
9			R_2		

25 (Shanbriya)

LALR Parser:

LALR Parser uses LR(1) items.

For LALR parsing, we can see CLR parsing, we see only ~~first~~ that production rule of CLR those ~~state~~ have first production

I_3 and I_6 have same first production

$I_3 + I_6 \Rightarrow I_{36}$
$I_4 + I_7 \Rightarrow I_{47}$
$I_8 + I_9 \Rightarrow I_8$
} have same productions and different lookahead.

	action			goto	
State	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			acc		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	R_3	R_3	R_3		
5			R_1		
89	R_2	R_2	R_2		

POORNIMA
COLLEGE OF ENGINEERING

LECTURE NOTES

Campus: ..... Course: ..... Class/Section: ..... Date: .....

Name of Faculty: Roona Sharma Name of Subject: Compiler Construct Code: 7CS05

Date (Prep.): ..... Date (Del.): .....Unit No./Topic: III .....Lect. No: .....

OBJECTIVE: To be written before taking the lecture (Pl. write in bullet points the main topics/concepts etc.. which will be taught in this lecture)

Introduction of Syntax directed, L-attributed definitions.

IMPORTANT & RELEVANT QUESTIONS:

What is Boolean expression & Contol Structures.

Explain Specification of a Type checker

FEED BACK QUESTIONS (AFTER 20 MINUTES):

What of Intermediate Code forms using postfix notation.

Explain TAC using Triples & quadruples

OUTCOME OF THE DELIVERED LECTURE: To be written after taking the lecture (Pl. write in bullet points about students' feedback on this lecture, level of understanding of this lecture by students etc.)

Student understand intermediate Code forms using postfix notation & three address Code

REFERENCES: Text/Ref. Book with Page No. and relevant Internet Websites:

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

(18)

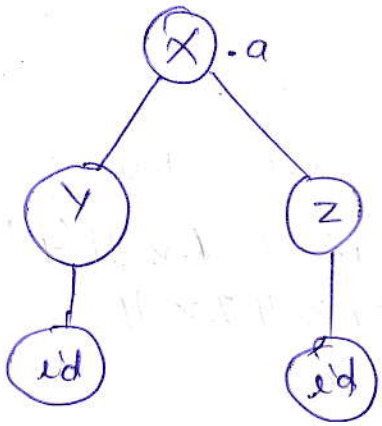
Unit # 3

Syntax Directed Translation

With every symbol present in grammer syntax directed definition associates a
- Set of attributes & with each production, a set of semantic rules for computing values of the attributes associated with the symbol appearing in that production. The grammer and the set of semantic rules make "Syntax Directed Definition".

eg. of Parse tree:

$X \to YZ$
$Y \to id_1$
$Z \to id_2$



X.a denotes that node X has an attribute called 'a'.
The value of X.a is found using semantic rule ~~for~~ for the X production used at this node.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Annotated Parse Tree: A parse tree with attribute values at each node is an Annotated Parse Tree.

Synthesized Attributes: An attribute is said to be a Synthesized Attribute if its value at a parse Tree node is determined from the attribute values at the children of the node.

Advantage: Synthesized attribute can be evaluated during a single Bottom-up Traversal.

Syntax Directed definitions Example: [Semantic Rules]

eg:

$E \rightarrow E_1 + T$
$E \rightarrow E_1 - T$
$E \rightarrow T$
$T \rightarrow 0$
$T \rightarrow 1$
:
$T \rightarrow 9$

} Infix to postfix expression

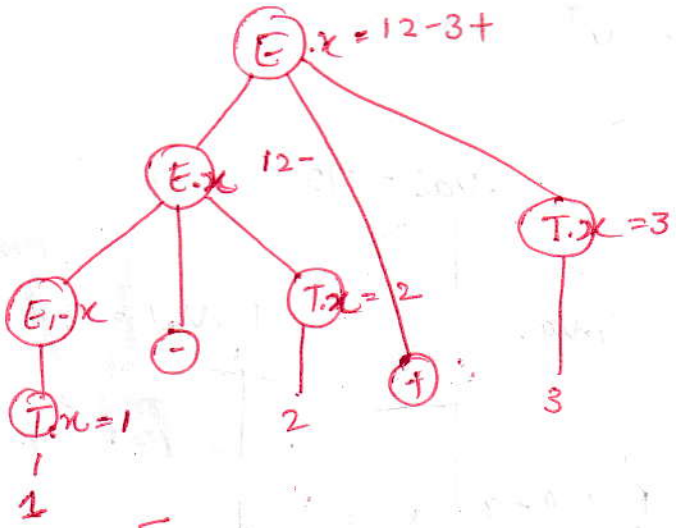
Sol:

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.x \Rightarrow E_1.x \mid\mid T.x \mid\mid '+'$
$E \rightarrow E_1 - T$	$E.x \Rightarrow E_1.x \mid\mid T.x \mid\mid '-'$
$E \rightarrow T$	$E.x \Rightarrow T.x$
$T \rightarrow 0$	$T.x \Rightarrow '0'$
$T \rightarrow 1$	$T.x \Rightarrow '1'$
$T \rightarrow 2$	$T.x \Rightarrow '2'$
$T \rightarrow 9$	$T.x \Rightarrow '9'$

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

(19) eg:- 12-3+

((1-2+3)) o/p



[Attribute values at nodes in a parse Tree.]

Or

[Syntax directed direction for given grammar]

[Annotated Parse tree]

eg:- $E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

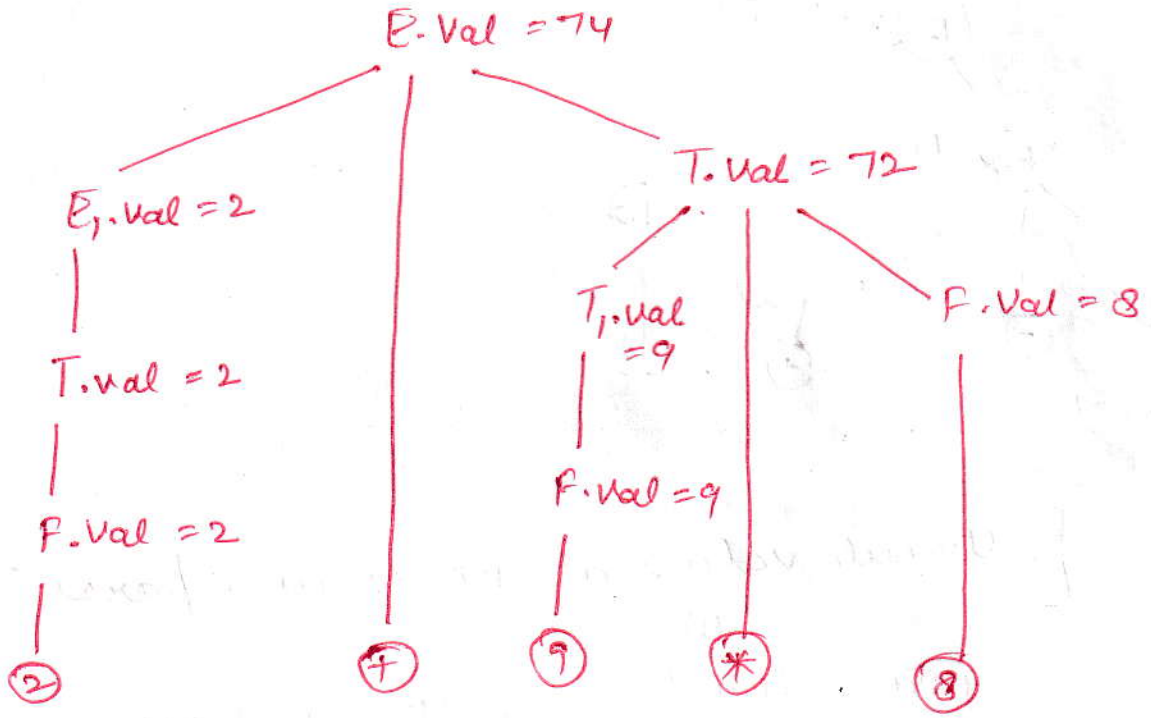
$F \rightarrow (E)$

$F \rightarrow$ digit

Sol:-

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$



2 + 9 * 8

Syntax tree

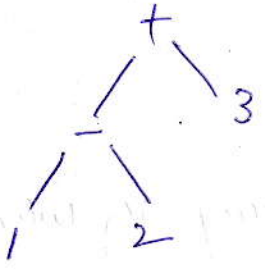
Abstract Syntax tree :- Each node in an AST represents an operator and the children of the operator are the operands (of this op^n).

eg:- 1+2

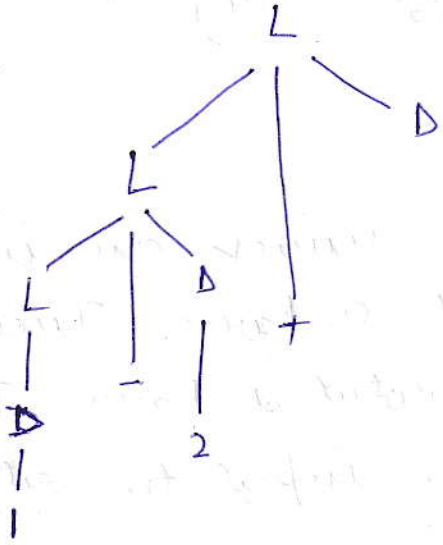
In AST no any unimportant details present.

Concrete Syntax tree :- It is a normal parse tree and the underlying grammer is called concrete syntax for the language.

eg:- 1 − 2 + 3



AST / ST (Syntax tree)



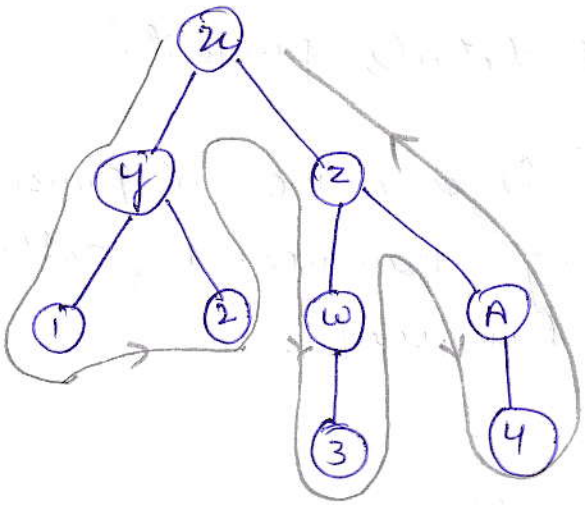
Concrete Syntax tree

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

SDD does not impose any specific order for the evaluation of attributes on Parse Tree. Any evaluation order that computes 'a' after all the attributes that 'a' depends on, is acceptable.

Traversal of a Tree :- Starts at the root and visit each node of the tree in some particular order.

Depth First Traversal :-



Start with the root and recursively visit children of each node in left to right manner.

21

Translation :- Given an input string x, when we construct a parse Tree for x and convert it into Annotated Parse Tree (o/p), this mapping from input to output is known as Translation.

Translation Scheme :- A translation scheme is a context-free grammer in which program fragments / pieces are embedded in RHS

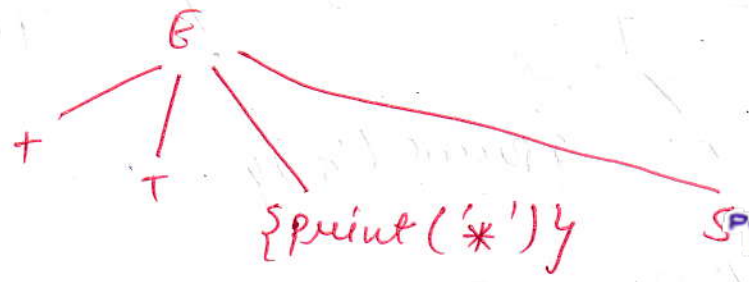
Translation Scheme is ~~like~~ similar to SDD except the fact that Translation Scheme also specifies explicitly: the evaluation order of semantic actions.

⇒ Translation Scheme generates an output for each string x present in the language generated by the grammer, by executing the actions in the order in which they appear in the order in ~~which~~ the Depth First Traversal of Parse Tree for x.

eg: $E \rightarrow +T \{ print('*') \} S$

For making a Parse Tree for a Translation Scheme :-

1. Make an extra node for semantic actions.
2. Nodes for semantic actions do not have children.
3. Evaluate node for SA whenever that node is seen.



Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
181-B, RIICO Institutional Area
Sitapura, JAIPUR

eg:- Consider a grammer

$E \rightarrow E+T$

$E \rightarrow E-T$

$E \rightarrow T$

$T \rightarrow 0$

$T \rightarrow 1$

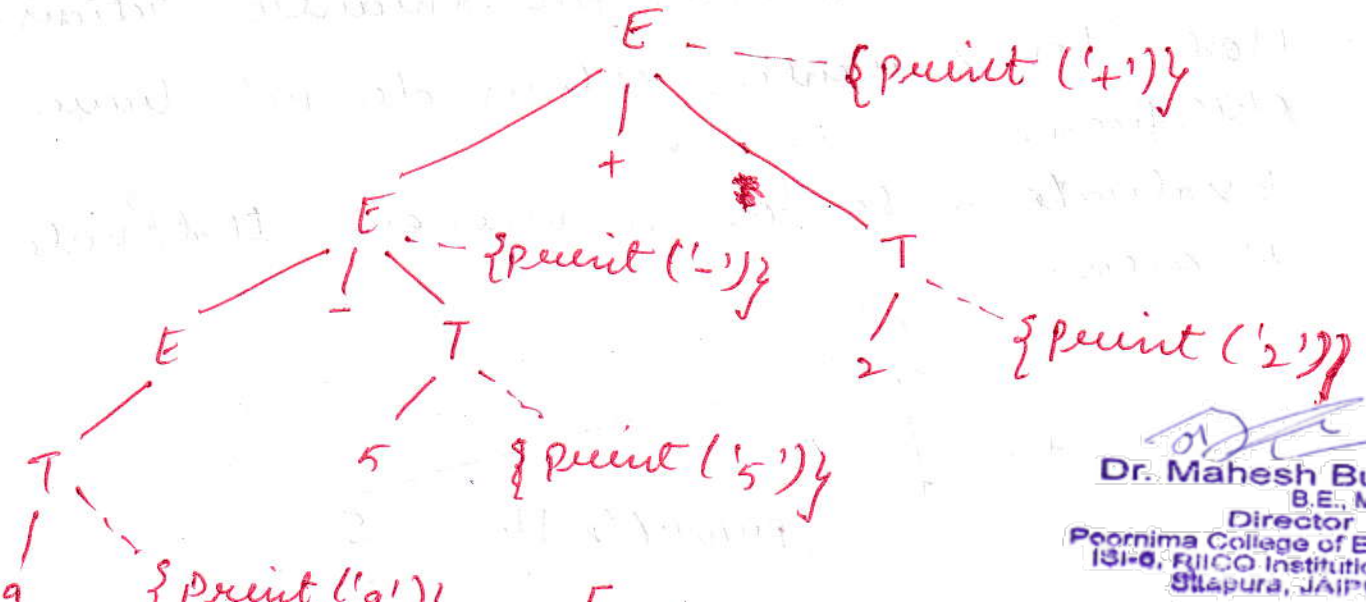
| |

$T \rightarrow 9$

Translation Scheme is

$E \rightarrow E+T$	{ print ('+') }
$E \rightarrow E-T$	{ print ('-') }
$E \rightarrow T$	
$T \rightarrow 0$	{ print ('0') }
$T \rightarrow 1$	{ print ('1') }
! !	
$T \rightarrow 9$	{ print ('9') }

Translation Scheme and a parse tree with actions for 9-5+2 is shown below:



[Infix to Postfix Translation]

eg:- How to associate semantic action code for any production and how to generate parse tree.

$E \rightarrow E_1 + T \quad \rightarrow E.x = E_1.x \,||\, T.x \,||\, '+'$

$E \rightarrow E_1 - T \quad \rightarrow E.x = E_1.x \,||\, T.x \,||\, '-'$

$E \rightarrow T \quad \rightarrow E.x = T.x$

$T \rightarrow 0 \quad \rightarrow T.x = '0'$

$T \rightarrow 1 \quad \rightarrow T.x = '1'$

| | |

$T \rightarrow 9 \quad \rightarrow T.x = '9'$

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

$E \rightarrow E_1 - T \quad : \{ print('-') \}$

$E \rightarrow T$

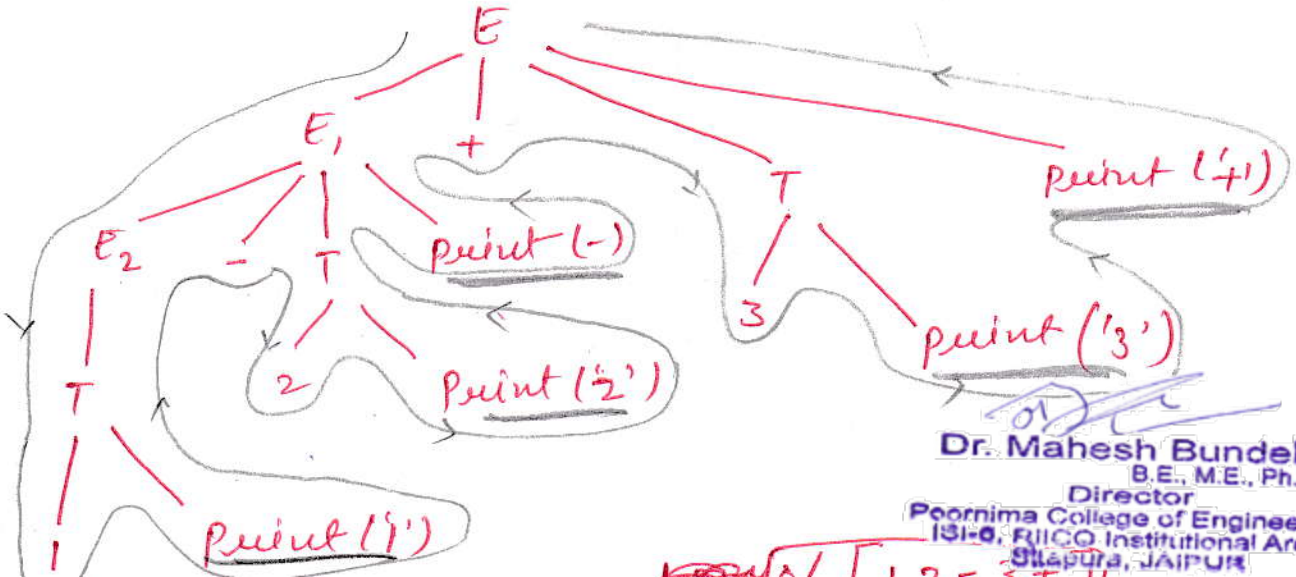
$T \rightarrow 0 \quad : \{ print('0') \}$

$T \rightarrow 1 \quad : \{ print('1') \}$

| | |

$T \rightarrow 9 \quad : \{ print('9') \}$

String : 1-2+3



Form of a Syntax - Directed Definition :

In a syntax - directed definition, each grammer production $A \to \alpha$ 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 grammer symbols of the production, or
2. b is an inherited attribute of one of the grammer symbols on the right side of the production, and $c_1, c_2 \dots c_k$ are attributes belonging to the grammer symbols of the production.

Pg 293 Eg 5.1

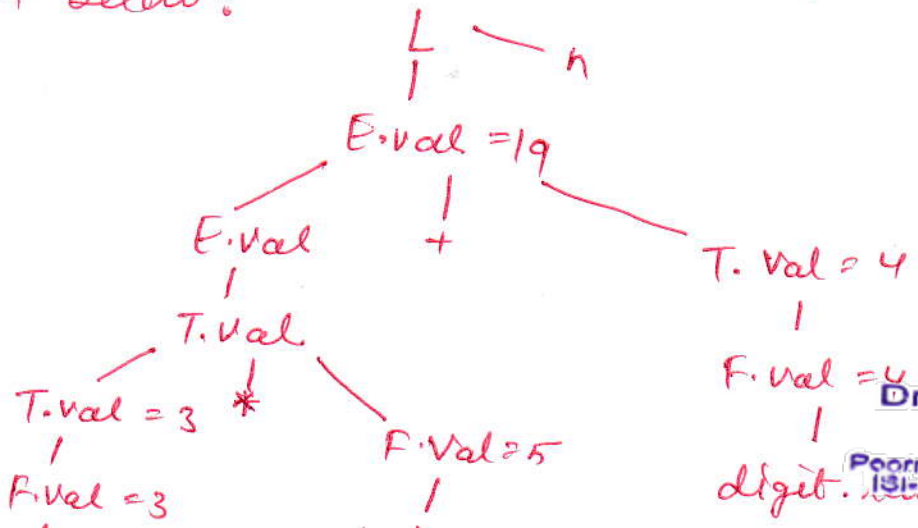
Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Synthesized Attributes: Synthesized attributes are used extensively. A Syntax-directed definition that uses synthesized attributes exclusively is said to be an S-attributed definition. A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node bottom up, from the leaves to the root.

Eg:- Syntax-directed definition of a simple desk calculator

Production	Semantic Rules
$L \rightarrow En$	~~Print~~ (E.Val) L.val := 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 X F.Val
$T \rightarrow F$	T.val := F.Val
$F \rightarrow (E)$	F.val := E.val
$F \rightarrow$ digit	F.Val := digit.lexval

→ An annotated parse tree for the input 3*5+4n shown below:



Inherited Attributes: An inherited attribute is one whose value at a node in a parse tree is defined in terms of attribute 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.

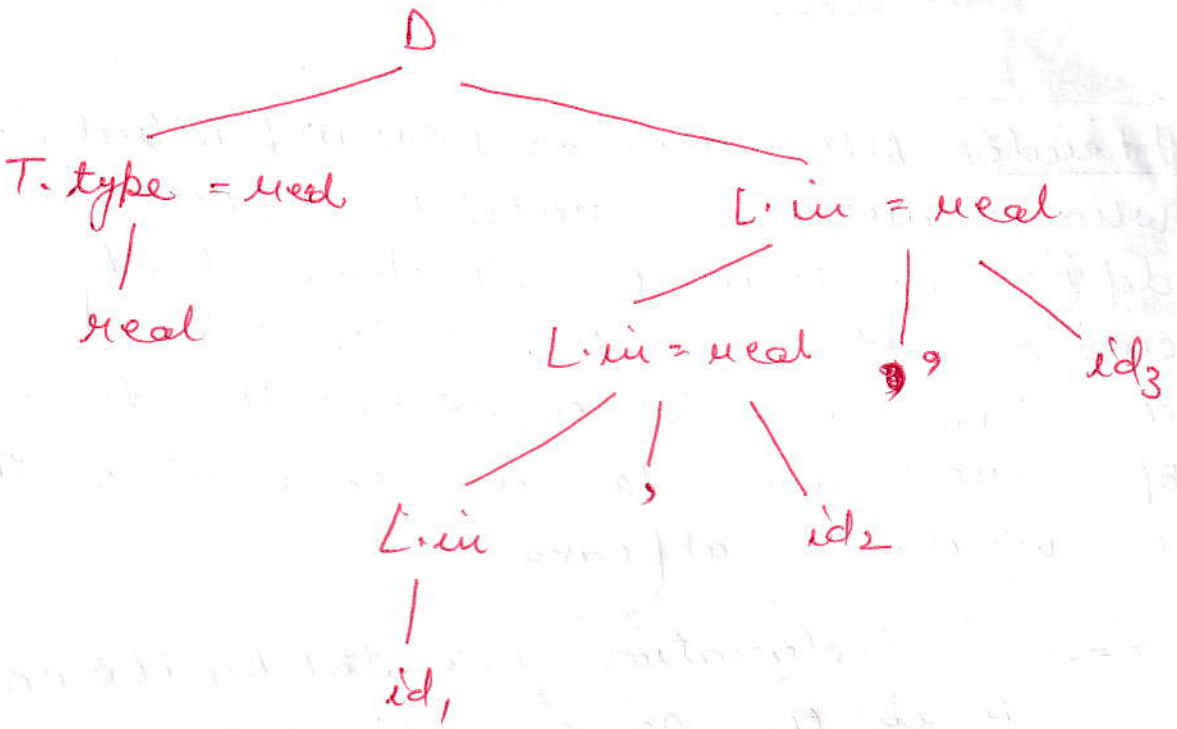
eg:- A declaration generated by the nonterminal D in the syntax directed definition consist of the keyword int or real, followed by a list of identifier.

Production	Semantic Rules
D → TL	L.in := T.type
T → int	T.type := integer
T → real	T.type := real
L → L1 , id	L1.in := L.in addtype (id.entry , L.in)
L → id	addtype (id.entry , L.in)

[Syntax directed definition with inherited attribute L.in]

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

An annotated parse tree for the sentence real id_1, id_2, id_3 shown below:



[Rules associated with the productions for L call procedure addtype to add the type of each identifier to its entry in the symbol table.]

⇒ In this tree,

⇒ 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 real.

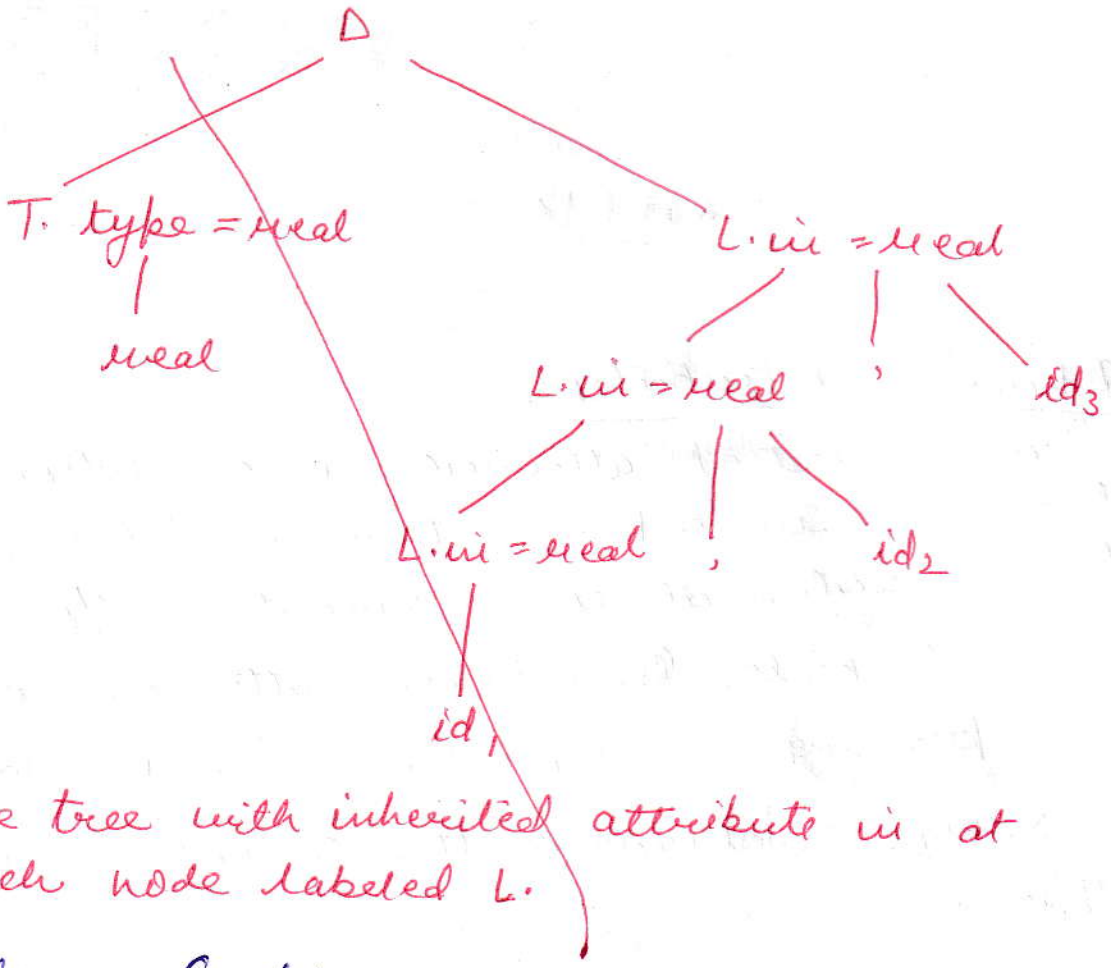
Inherited Attributes:

An inherited attribute is one whose value 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 content in which it appears.

eg:

Production	Semantic Rules
$D \rightarrow TL$	L.in := T.type
$T \rightarrow$ int	T.type := integer
$T \rightarrow$ real	T.type := real
$L \rightarrow L_1$, id	L_1.in := L.in addtype (id.entry, L.in)
$L \rightarrow$ id	addtype (id.entry, L.in)

draw annotated parse tree for the sentence real id_1, id_2, id_3.



Parse tree with inherited attribute in at each node labeled L.

Dependency Graph:- 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.

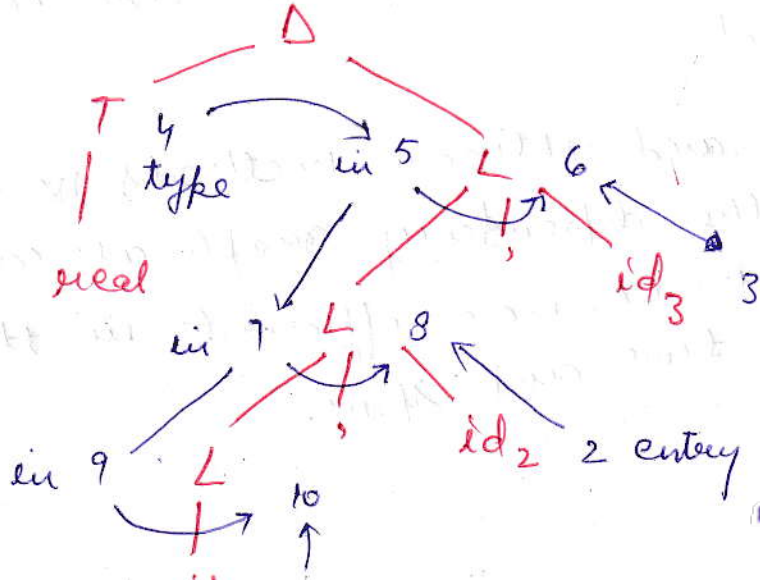
The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph.

The basic idea behind the dependency graph is for compiler to look for various kinds if dependence among statements to prevent their execution in wrong order i.e. the order that

eg: 2.97 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 $b := f(c_1, c_2, \ldots c_k)$ associated with the production used at n do
    for i := 1 to k do
      construct an edge from the node for c_i to the node for b;

eg:- for input string real id_1, id_2, id_3 dependency graph shown below:



:- addtype(id. entry, L.in) associated with the creation of dummy attributes L.in

:- Nodes 6, 5, 10 are constructed for

Several methods have been proposed for evaluating semantic rules:

1. Parse-Tree methods: At compile time, these methods obtain an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input. These methods will fail to find an evaluation order only if the dependency graph for the particular parse tree under consideration has a cycle.

2. Rule based methods: At compile-construction time, the semantic rules associated with productions are analyzed, either by hand, or by a specialized tool. For each production, the order in which the attributes associated with that production are evaluated is predetermined at compiler-construction time.

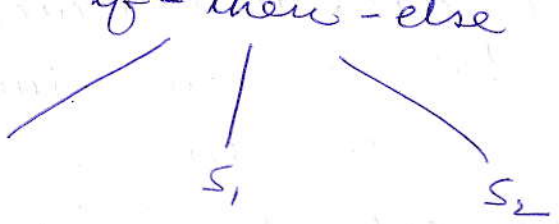
3. Oblivious method: An evaluation order is chosen without considering the semantic rules. An oblivious evaluation order restricts the class of syntax-directed definitions that can be implemented.

* Rule based and oblivious methods need not explicitly construct the dependency graph at compile time, so they can be more efficient in their use of compile time and space.

Construction of Syntax tree:

The use of Syntax tree as an intermediate representation allows translation to be decoupled from parsing.

Syntax Trees: An (abstract) Syntax tree is a condensed form of parse tree useful for representing language constructs. The production
$S \rightarrow$ if B then S_1 else S_2 might appear in a Syntax tree as



In a syntax tree, operators and keywords do not appear as leaves, but rather are associated with the interior node that would be the parent of those leaves in the parse tree.

Construction Syntax Trees for Expressions:

The construction of a syntax tree for an expression is similar to the translation of the expression into postfix form. We construct subtrees for the subexpressions by creating a node for each operator and operand.

Each node in a syntax tree can be implemented as a record with several fields. In the node for an operator, one field identifies the operator and the remaining fields contains pointers to the nodes for the operands. The operator is often called the label of the node.

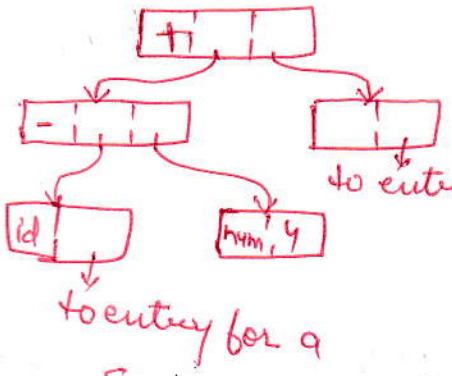
→ We use following functions to create the nodes of syntax trees for expressions with binary operators. Each function returns a pointer to a newly created node.

1. mknode (op, left, right) creates an operator node with label op and two fields containing pointers to left and right.
2. mkleaf (id, entry) creates an identifier node with label id and a field containing entry, a pointer to the symbol-table entry for the identifier.
3. mkleaf (num, val) creates a number node with label num and a field containing val, the value of the number.

eg:- Construction of syntax tree for $a - 4 + c$

Sol:

$P_1 := mkleaf(id, entry\ a)$;
$P_2 := mkleaf(num, 4)$;
$P_3 := mknode('-', P_1, P_2)$;
$P_4 := mkleaf(id, entry\ c)$;
$P_5 := mknode('+', P_3, P_4)$;

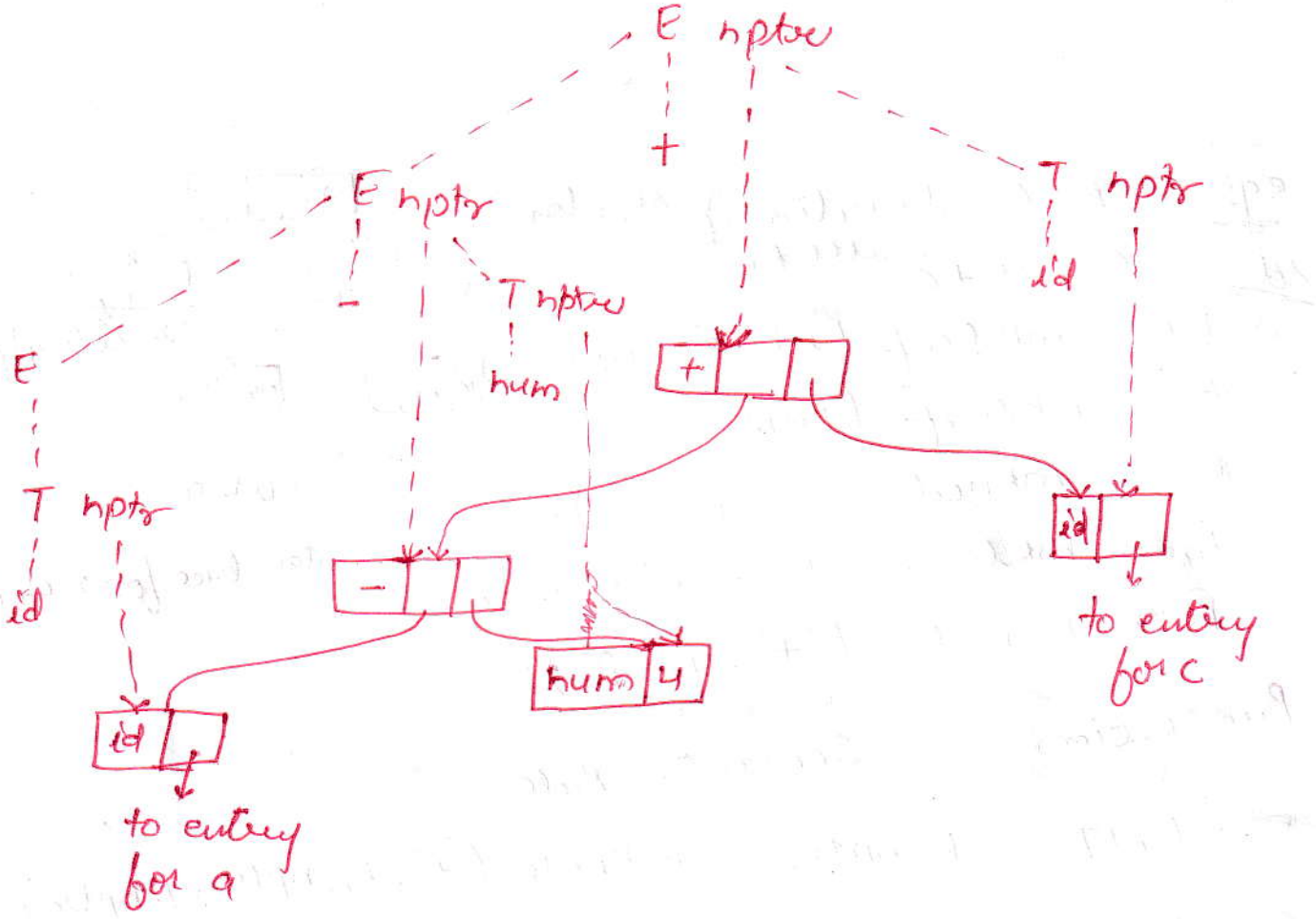


[Syntax tree for a-4+c]

Production	Semantic Rule
$E \rightarrow E_1 + T$	E.nptr := mknode ('+', E_1.nptr, T.nptr)
$E \rightarrow E_1 - T$	E.nptr := mknode ('-', E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow (E)$	T.nptr := E.nptr
$T \rightarrow id$	T.nptr := mkleaf (id, id.entry)
$T \rightarrow num$	T.nptr := mkleaf (num, num.val)

x * y - 5 + z

Syntax tree :-

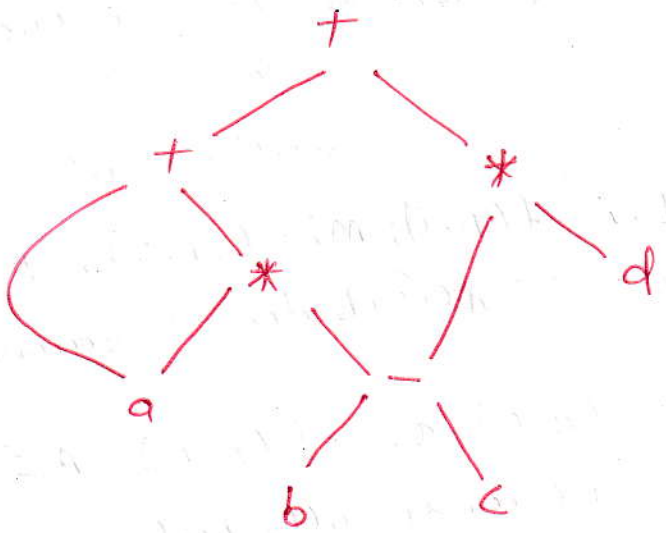


[Construction of a syntax tree for a-4+c]

Directed Acyclic Graphs for Expressions :

A directed acyclic graph (called a DAG) for an expression identifies the common sub-expression of the expression.

eg:- DAG for expression a+a*(b-c)+(b-c)*d



Instruction for constructing the dag :-

1. P_1 := mkleaf (id, a);
   P_2 := mkleaf (id, a);
   P_3 := mkleaf (id, b);
   P_4 := mkleaf (id, c);
   P_5 := mknode ('-', P_3, P_4);
   P_6 := mknode ('*', P_2, P_5);
   P_7 := mknode ('+', P_1, P_6);
   P_8 := mkleaf (id, b);

P_{10} := mknode ('-', P_8, P_9);

P_{11} := mkleaf (id, d#);

P_{12} := mknode ('*', P_{10}, P_{11});

P_{13} := mknode ('+', P_7, P_{12});

S-Attributes definition: If syntax directed definition uses only synthesized attributes, it is called S-attribute ~~definition~~. S attribute-SDT are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes. Semantic Rule placed at the Right end of production.

$A \rightarrow BCD \{ \ldots \}$

Semantic Rule

eg:- (1) $A \rightarrow LM \{ L.i = f(A.i);\ M.i = f(L.s);\ A.s = f(M.s); \}$

It have not S-Attributes, because of $L.i = f(A.i)$.

(2) $A \rightarrow QR \{ R.i = f(A.i);\ Q.i = f(R.i);\ A.s = f(Q.s); \}$

None → no any attributes.

L-attribute definition: A syntax-directed definition is L-attributed if each inherited attribute of X_j, $1 \le j \le n$, on the right side of $A \rightarrow X_1 X_2 \ldots X_n$, depends only on

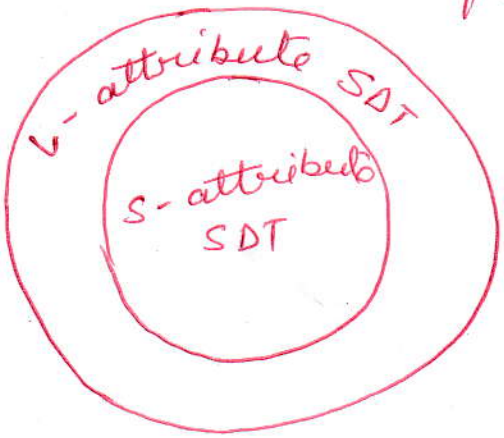
1. the attributes of the symbols $X_1, X_2 \ldots X_{j-1}$ to the left of X_j in the production and
2. the inherited attributes of A.

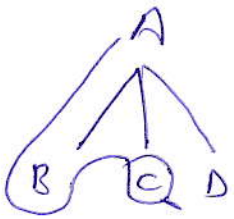
→ If an SDT uses both synthesized and inherited attributes, ...

from left sibling only, it is called as L-attributed SDT.

→ Attributes in L-attributed SDT are evaluated by depth-first and left-to-right parsing manner.

→ Semantic actions are placed anywhere in RHS.





Translation Schemes :- If we have both inherited and synthesized attributes, we must be more careful:

1. An inherited attribute for a symbol on the right side of a production must be computed in an action before that 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 the nonterminal on the left can only be computed after all attributes it references have been computed.

be placed at the end of the right side of the production.

Top down Translation: In this, L-attributed definitions is implemented during predictive parsing. In this, algorithm is extend for left-recursion elimination to translation schemes with synthesized attributes.

Eliminating left Recursion from a Translation Scheme :

Three Address code :-

Three - address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior node of the graph.

The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

eg:- $x := y \ op \ z$

here x, y, z are names, constants or compiler-generated temporaries; op stands for any operator. eg:- $x + y * z$ translate into

$t_1 := y * z$
$t_2 := x + t_1$

Where t_1 and t_2 are compiler-generated temporary names.

eg:- $a := b * -c + b * -c$

$t_1 := -c$
$t_2 := b * t_1$
$t_3 := -c$
$t_4 := b * t_3$
$t_5 := t_2 + t_4$
$a := t_5$

$t_1 := -c$
$t_2 := b * t_1$
$t_5 := t_2 + t_1$
$a := t_5$

a) code for the

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
181-B, RIICO Institutional Area
Sitapura, JAIPUR

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.

1. Assignment statements of the form $x := y$ op z, where op is a binary arithmatic or logical operation.
2. Assignment instructions of the form $x :=$ op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and 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 jump : such as if x relop y goto L. This instruction applies a relational operator ($<$, $=$, $>=$, etc) to x and y, and executes the statement with label L next if x stands in relation relop to y. If not, the three-address statement following if x relop y goto L is executed next, as in the usual sequence.
6. Param x and call p, n for procedures and return y

They use in the procedure call.

7. Indexed Assignments : of the form $x := y[i]$ and $x[i] := y$. The first 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 refers to data objects.

8. Address and pointer assignments : of the form $x := \&y$, $x := *y$ and $*x := y$.

Syntax - Directed Translation into Three - Address code :

When three - address code is generated, temporary names are made up for the interior node 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.

For input $x = a + b * c$

The S - attributes definition are as follows

Production	Semantic Rule
$S \rightarrow id := E$	S.code := E.code || gen('id.place' := E.place
$E \rightarrow E_1 + T$	E.place := newtemp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)
$E \rightarrow T$	E.place := T.place
$T \rightarrow T_1 * F$	T.place := newtemp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)
$T \rightarrow T_1 * F$	T.place := new temp; T.place := T.place

$S \rightarrow id = E$ { gen('id.name = E.place'); }

$E \rightarrow E_1 + T$ / T { E.place = newTemp(); gen(E.place = E_1.place + T.place); }

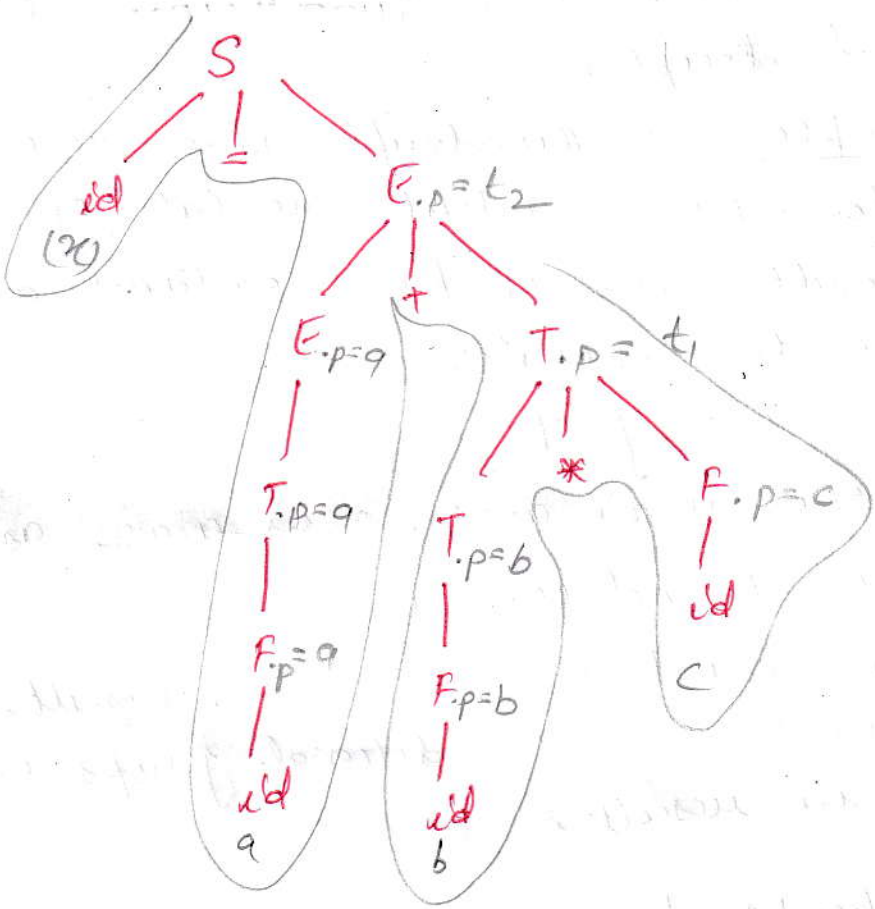
{ E.place = T.place }

$T \rightarrow T_1 * F$ / F { T.place = newTemp(); gen(T.place = T_1.place * F.place); }

{ T.place = F.place }

$F \rightarrow id$ { F.place = id.name }

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR



$x = a + b * c$ → i/p

$$\left.\begin{aligned} t_1 &= b * c \\ t_2 &= a + t_1 \\ x &= t_2 \end{aligned}\right\} \rightarrow o/p$$

By using this SDT, we convert the statement into three address code.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Implementation of Three - Address Statements :

In compiler, three address statements can be implemented as records with fields for the operator and operands. Three such representations are quadruples, triples and indirect triples.

Quadruples : A Quadruple is a record structure with four fields, which we call op, arg 1, arg 2 and result. The op field contains an internal code for the operator.

eg:- $x := y \ op \ z$

in this y in arg 1, z in arg 2 and x in result.

$x = -y$, do not use arg 2.

Param do not use arg 2 and result.

Conditional and unconditional jumps put the target label in result.

Quadruple for $a := b * -c + b * -c$

TAC

$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

	op	arg 1	arg 2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Quadruples

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. By doing this, three address statements can be represented by records with only three fields : op , arg1 , arg2 .

eg:- a := b * -c + b * -c

	op	arg1	arg2
(0)	Uminus	c	
(1)	*	b	(0)
(2)	Uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

triples

eg:-

	op	arg1	arg2
(0)	[ ] =	x	i
(1)	assign	(0)	y

x[i] := y

	op	arg1	arg2
(0)	=[ ]	y	i
(1)	assign	x	(0)

Indirect Triples: It is makes use of pointer to the listing of all references to computation which is made separately and stored.

eg:- $a := b * -c + b * -c$

	Statements
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	OP	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

[Indirect triples representation of Three add^{n} statements]

---

eg:- $A := -B * (C/D)$

$T_1 := -B$

$T_2 := C/D$

$T_3 := T_1 * T_2$

$A := T_3$

TAC

	OP	Arg1	Arg2	Result
(0)	uminus	B	-	T_1
(1)	/	C	D	T_2
(2)	*	T_1	T_2	T_3
(3)	:=	T_3	-	A

Quadruples

	OP	Arg1	Arg2
(0)	uminus	B	-
(1)	/	C	D
(2)	*	(0)	(1)
(3)	:=	A	(2)

Triples

	Statement
(0)	(21)
(1)	(22)
(2)	(23)
(3)	(24)

	OP	Arg1	Arg2
(21)	uminus	B	-
(22)	/	C	D
(23)	*	(21)	(22)
(24)	:=	A	(23)

Indirect triples

Boolean Expressions :

Boolean Expressions have two primary 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.

* or and and are left-associative and ~~they~~ that or has lowest precedence, then and then not.

Methods of translating Boolean expression :

Two methods of representing the value of a boolean expression.

1. The first method is to encode true and false numerically and to evaluate a boolean expression analogously to an arithmatic expression. 1 is used to denote true and 0 to denote false.
2. Boolean expression is implented by flow of control, that is, representing the value of a boolean expression by a position reached in a program. This method is convenient for control statements such as if-then, while etc.

Consider a grammer

$E \rightarrow E$ OR E

$E \rightarrow E$ and E

$E \rightarrow$ Not E

$E \rightarrow (E)$

$E \rightarrow$ id relop id

$E \rightarrow$ true

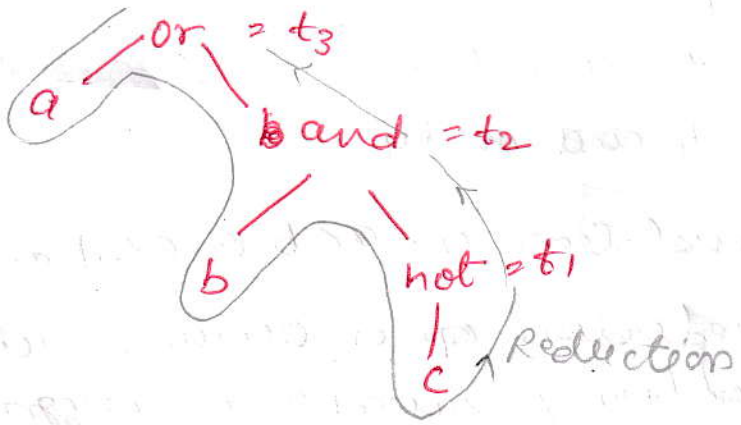
$E \rightarrow$ false

Production Rule	Semantic Rules
$E \rightarrow E_1$ or E_2	{ E.place := newtemp; emit (E.place ':=' E_1.place 'or' E_2.place
$E \rightarrow E_1$ and E_2	{ E.place := newtemp; emit (E.place := ' E_1.place 'and' E_2.place) }
$E \rightarrow$ not E_1	{ E.place := newtemp; emit (E.place := 'not' E_1.place) }
$E \rightarrow (E_1)$	{ E.place := E_1.place }
$E \rightarrow id_1$ relop id_2	{ E.place := newtemp; emit ('if' id_1.place relop.op id_2.place 'goto' nextstat +3); emit (E.place ':=' '0'); emit ('goto' nextstat +2); emit (E.place := '1') }
$E \rightarrow$ true	{ E.place := newtemp; emit (E.place ':=' '1') }
$E \rightarrow$ false	{ E.place := newtemp

eg:- if a > b then 1 else 0

```
100 : if a > b goto 103
101 : t1 = 0
102 : goto 104
103 : t1 = 1
104 :
```

eg:- a or b and not c

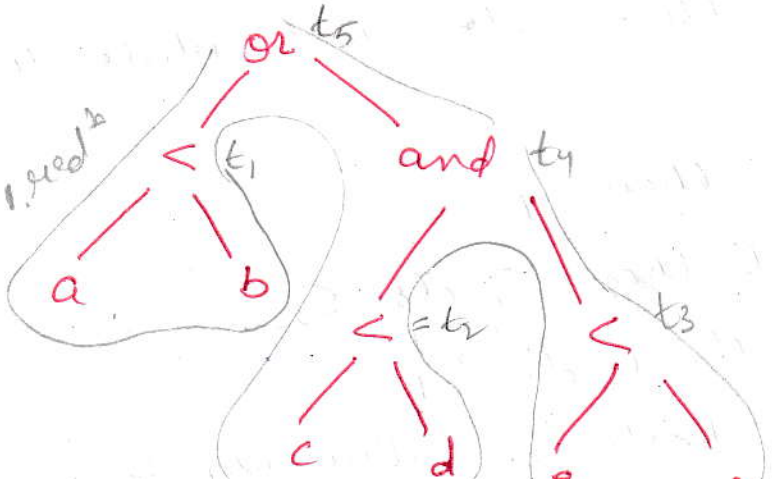


t_1 = not c

t_2 = b and t_1

t_3 = a or t_2

eg:- a < b or c < d and e < f



Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

1 red^{n} 100 : if a < b goto 103
101 : t_1 := 0
102 : goto 104
103 : t_1 := 1
2 red^{n} 104 : if c < d goto 107
105 : t_2 := 0
106 : goto 108
107 : t_2 := 1
3 red^{n} 108 : if e < f goto 111
109 : t_3 := 0
110 : goto 112
111 : t_3 := 1
4 112 : t_4 := t_2 and t_3
5 113 : t_5 := t_4 or t_4

Translation of a<b & c<d and e<f

* [Whenever reduction is occur, action is perfor and new temporary variable is assigned.]

Flow of control statements :

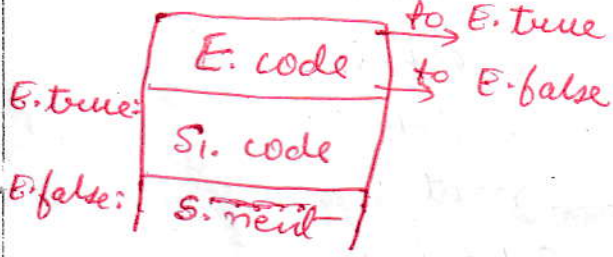
We see the translation of boolean expressions into three-address code in the content of if-then, if-then-else and while-do statements, grammer is

$S \rightarrow$ if E then S_1
| if E then S_1 else S_2
| while E do S_1

here E is the boolean expression

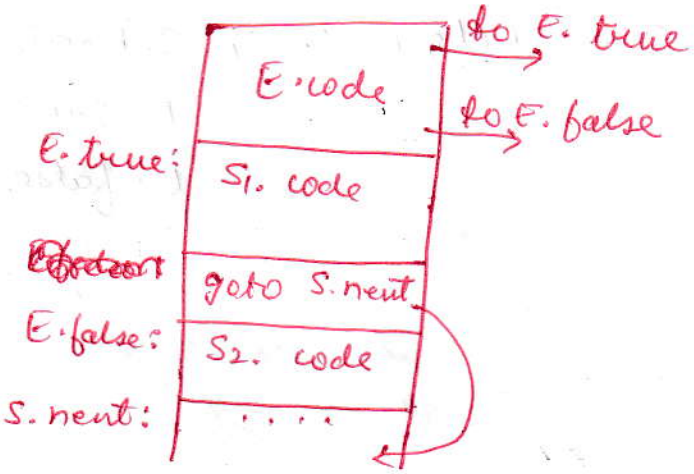
Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Labels : E.true, the label to which control flows if E is true and E.false, the label to which control flows if E is false.



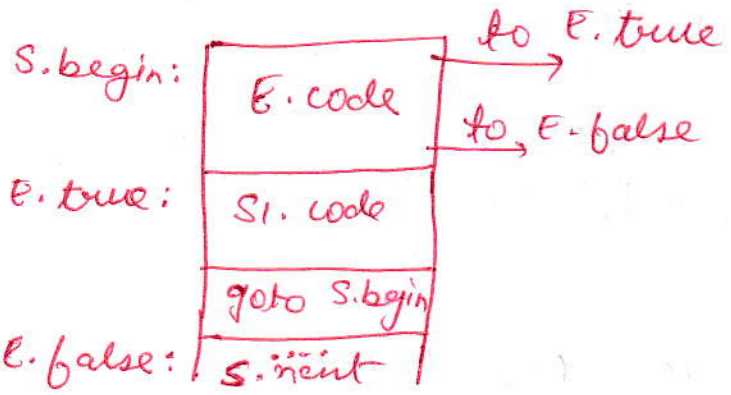
(if - then)

S.next and S1.next equal



(if - then - else)

Production.	Semantic Rules
$S \rightarrow$ if E then S_1	E.true := newlabel; E.false := S.next; S_1.next := S.next; S.code := E.code || gen (E.true ':') || S_1.code
$S \rightarrow$ if E then S_1 else S_2	E.true := newlabel; E.false := newlabel; S_1.next := S.next; S_2.next := S.next; S.code := E.code gen (gen ('goto'



(while-do)

Production	Semantic Rules
$S \rightarrow$ while E do S_1	S.begin := newlabel ; E.true := newlabel ; E.false := S.next ; S_1.next := S.begin ; S.code := gen (S.begin ':') || E.code || gen (E.true ':') || S_1.code || gen ('goto' S.begin)

[Syntax directed definition for flow-of-control statements]

Control flow Translation of Boolean Expression:

Suppose E is of the form E_1 or E_2. And may be any other boolean expression like E_1 and E_2 E not E_1 then Syntax Directed Translation definition to produce three-address code for booleans as shown below:

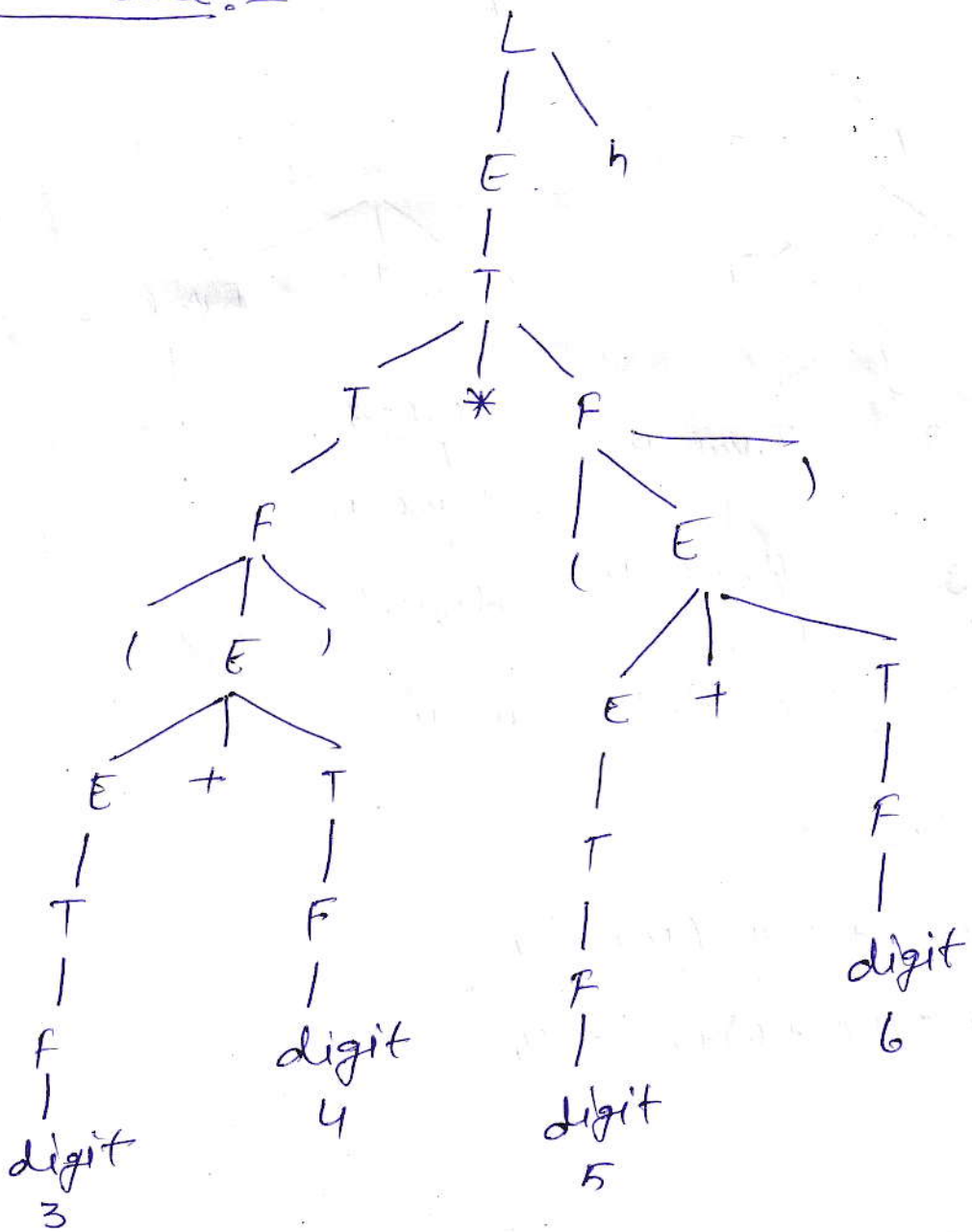
Productions	Semantic Rules
$E \rightarrow E_1$ or E_2	E_1.true := E.true; E_1.false := newlabel; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.false ':') || E_2.code
$E \rightarrow E_1$ and E_2	E_1.true := newlabel; E_1.false := E.false; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.true ':') || E_2.code
$E \rightarrow$ not E_1	E_1.true := E.false; E_1.false := E.true; E.code := E_1.code
$E \rightarrow (E_1)$	E_1.true := E.true; E_1.false := E.false;

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

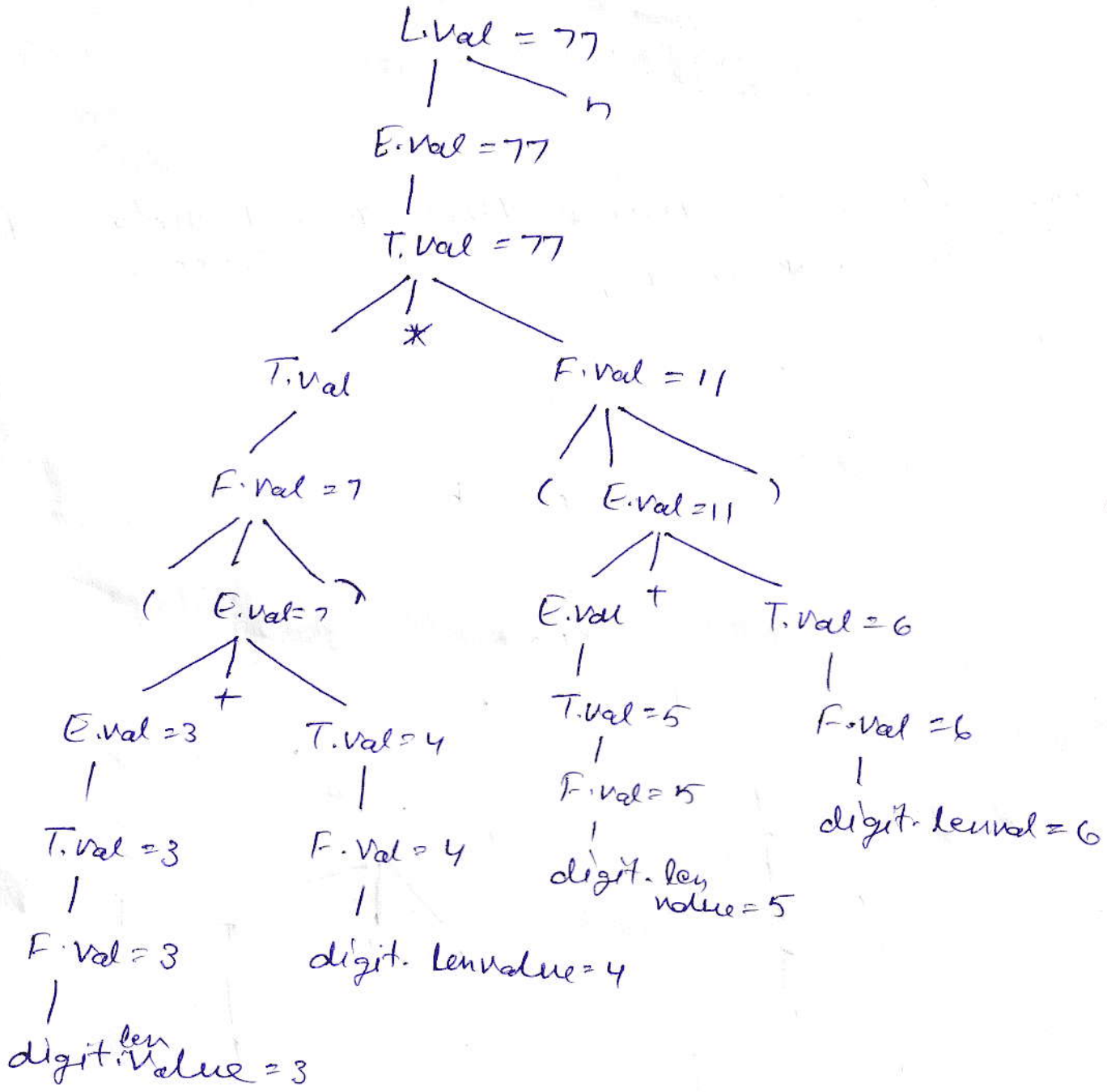
$E \rightarrow id_1$ relop id_2	E.code := gen ('if' id_1.place relop.op id_2.place 'goto' E.true) || gen ('goto' E.false)
E → true	E.code := gen ('goto' E.true)
E → false	E.code := gen ('goto' E.false)

eg:- Draw the parse tree & Annotated Parse Tree of
(3+4) * (5+6)n

Parse tree:-



Annotated Parse tree :-



eg:- $(* 2 * 3 * (4+5)_n$

eg:- $(9+8*(7+6)+5)*4_n$

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA
COLLEGE OF ENGINEERING

LECTURE NOTES

Campus: ..... Course: ..... Class/Section: ..... Date: .....

Name of Faculty: Reena Sharma Name of Subject: Compiler Construction Code: 7CS05

Date (Prep.): ..... Date (Del.): .....Unit No./Topic: IV .....Lect. No: .....

OBJECTIVE: To be written before taking the lecture (Pl. write in bullet points the main topics/concepts etc., which will be taught in this lecture)

Introduction of Storage organization & Data Structures used in Symbol Tables

IMPORTANT & RELEVANT QUESTIONS:

- What is Storage organization, Storage allocation, Activation records
- Explain Parameters passing, Symbol table organization

FEED BACK QUESTIONS (AFTER 20 MINUTES):

- What is Accessing local & non local names in block Structured language

Explain Activation records.

OUTCOME OF THE DELIVERED LECTURE: To be written after taking the lecture (Pl. write in bullet points about students' feedback on this lecture, level of understanding of this lecture by students etc.)

Student understand block Structure language

REFERENCES: Text/Ref. Book with Page No. and relevant Internet Websites:

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Unit - IV

Storage Organisation

Mani's actual

Procedures :-

→ as the declarations that associates an identifier with an statement.

→ When a procedure name appears within an executable statement, then the procedure is called at that point.

→ The basic idea is that the procedure call executed the procedure body.

→ Some of the identifier appear in a procedure definition are special and ~~these~~ these called parameter of the procedure.

→ here identifier is the name of the procedure and statement is the body of the procedure.

** Main program is subdivided into procedures so that its complexity reduces.

Procedures are coded independently and later are combined into single unit.

The procedure are easy to understand debug and test.

for example :-

```
main ( )
{
  . —
  . =
  . =
  procedure 1 ( )
  procedure 2 ( )
  procedure 3 ( )
}
```

} identifier or name of the procedure

```
procedure 1 ( )
{
}
Procedure 2 ( )
{
}
procedure 3 ( )
{
}
}
```

→ Actual and formal parameter

```
main ( )
{
Procedure (a1, a2 ... an);
}
Procedure ( f1, f2 — fn);
{
}
```

actual parameter

called procedure

formal parameter

Procedure Activation :-

→ An execution of a procedure starts at the beginning of the procedure body. When the procedure ends, we can say, completed, it returns control to the point immediately after the place where the procedure is called. Each execution of a procedure is called as its activation.

→ Lifetime of an activation of a procedure is the sequence of the steps b/w the first and last instruction in the execution of that procedure.

If the procedure is recursive, a new activations can begin before an earlier activation of the same procedure has ended.

Activation Tree :-

→ A Tree which is used to show the way by which control enters and leaves activations.

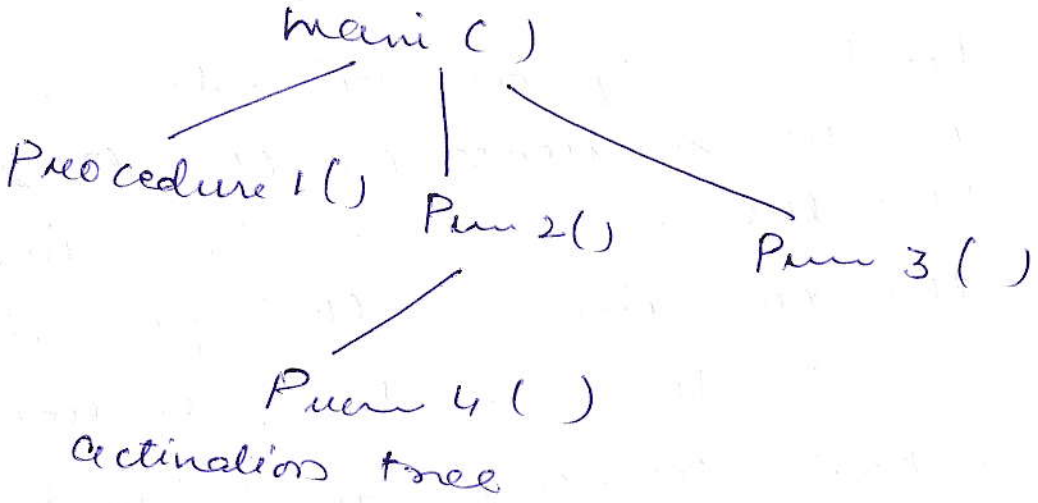
→ Sequence of function calls can be represented as an activation tree.

→ In an activation Tree :-

2) Activation of the main program is represented by the root of the tree.

3) Node A is the parent of node B if A call B.

4) The node A is to the left of the node B if and only if the lifetime of A occurs before the lifetime of B.

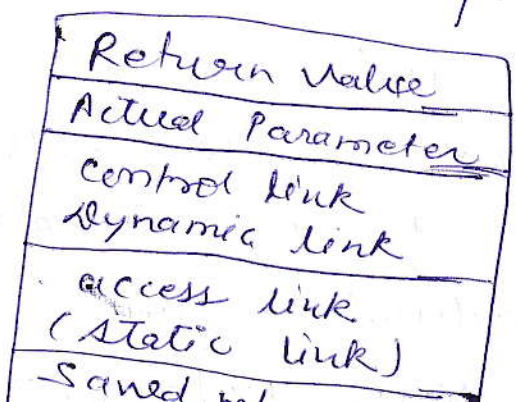


Activation tree

Activation Record :-

→ To manage the single execution of the procedure.

→ Created when calling.



POORNIMA

~~(activation ends) then it will popped.~~

→ Activation record is used to manage the info needed by a single execution of procedure.

→ An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.

Content of Activation Record

Return Value
Actual Parameters
Control Link
Access Link
Saved Machine states
Local Data
Temporaries

Return Value → It is used by called procedure to return a value to calling procedure.

Actual Parameter → It is used by calling procedures to supply parameters to the called procedure.

Control Link → It points to activation

of the caller.

Access Link It is used to refer to non-local data held in other activation records.

Saved Machine Status → It holds the information about status of machine before the procedure is called.

Local Data → It hold the data that is local to the execution of the procedure.

Temporaries → It stores the value that arises in the evaluation of an expression.

for eg:-

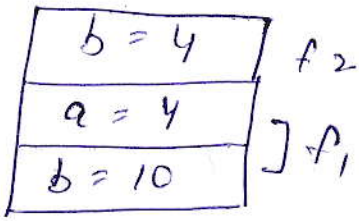
```
f1 ( int a)
  {
   int b = 10;
   return ( a +b);
  }
f2 ( int b)
 {
   return (b + f1 (b));
 }
main ()
{
  f2 (4) ;
}
```

1) f2(4) is called

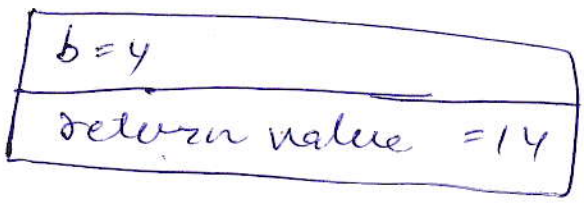
   b = 4

2) Activation record for f1

f1(4)



3) f1 (4) return 14



4) f(4)

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Control Stack :—

→ Keeps track of live procedure activations.

→ The flow of control in program corresponds to a depth first traversal of the activation tree that :

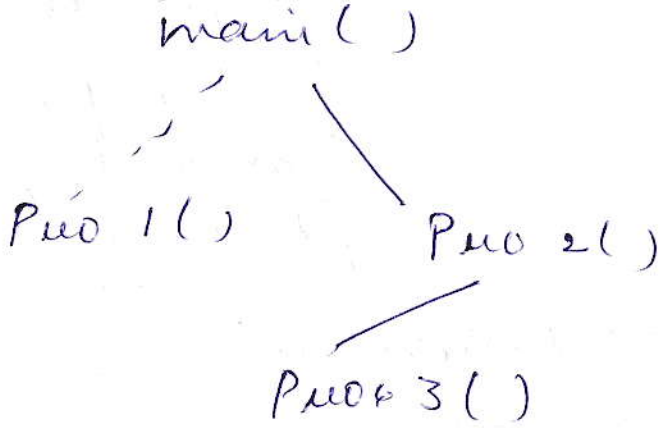
↳ starts at the root

↳ visit a node before its children are visited.

↳ recursively visit children at each node in a left or right order.

↳ an activation record is pushed onto the control stack Activation begins.

↳ Activation record is popped when that activation ends.



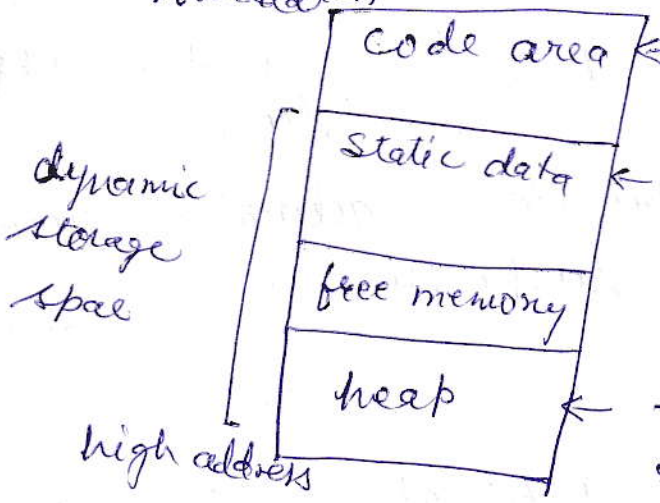
~~Storage allocation strategy~~ :-

Storage Organisation :-

→ organisation of the run-time storage.

→ This is the type of storage provided for different types of data objects.

Subdivision of runtime memory :-



← memory location for code are determined at compile time

← for activation records, local data, Parameters, control info

← for dynamic memory allocated to programs

At compile time, compiler obtains the block of storage from OS, i.e. at the start of the program execution. So the compiled program can run in that available block of storage.

→ This run-time storage might be subdivided to hold.

1) Generated target code
2) Data object

1) Code area
   → Contains target code
   → size of code is fixed
   → static
   → placed at the lower end

2) Static data area :-
   → Contains global data object, const
   → static

3) Stack :- {Value of the program counter and machine registers is saved on the stack.}
   → manages the activation of procedures
   → procedure call - creation of activation record
      ~ return - suspension "

4) Heap :-
   → stores all remaining info about program.
   → may face overhead as compared to static or stack data allocation

```
Void heap ()
{
   int size = get-size ();  ← heap
   int * a = new int [size];
}
# include <stdio.h>
int a[500];  ← static data
main ()
```

⇒ Storage allocation strategies :-

Static Allocation :-

→ for data obj which are known at compile time and static in nature.

→ allocation is done at once, can't be changed.

→ Compiler determines storage required so it is easy to make available the addresses.

Limitations :-

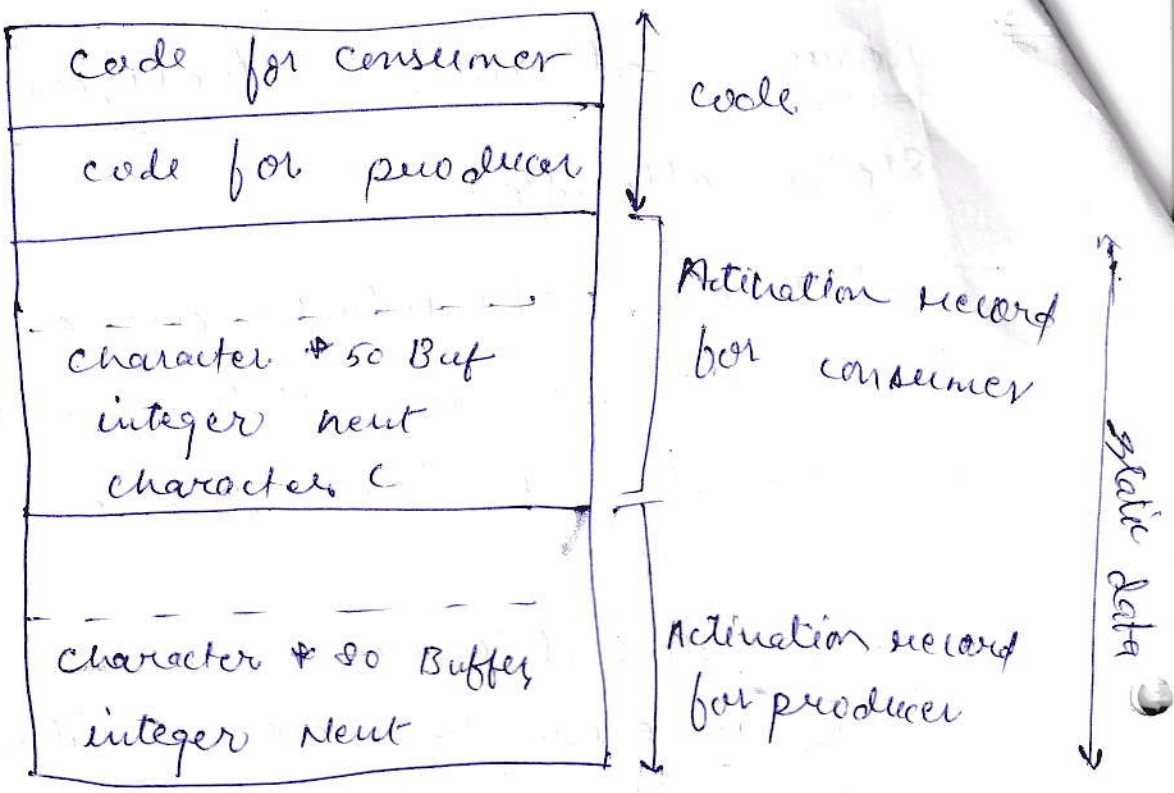
→ Only when size of data is known

→ not possible to allocate data/memory at run-time.

example :-

```
consumer()
{
    char * Buf [50];
    int next;
    char c;
    c = produce();
    ...
}
```

```
char producer()
{
    char * Buf [50];
    int next;
    ...
}
```



2) Stack allocation strategy :-

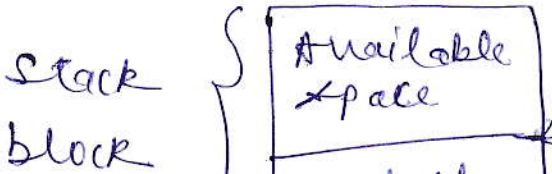
→ manage run time data storage.

→ based on concept of control ~~of~~ stack.

→ A stack is called control stack when it keeps track of live procedure.

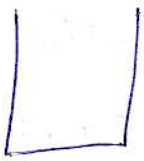
→ Activation records are pushed and popped onto stack as the activation record is popped off storage for locals are disappered.

→ based on LIFO.

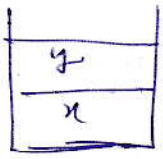


```
main()
{
    int x, y;
    - - -
    proc 1();
    proc 2();
    - - -
}
proc 1()
{
    int a, b;
}
proc 2()
{
    int c, d
    proc 3();
}
proc 3()
{
    int e, f;
}
```

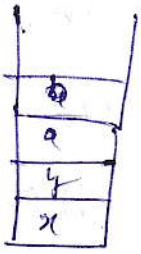
Step 1 :-



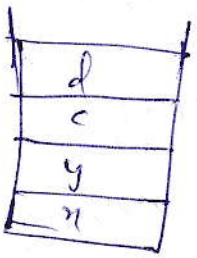
Step 2 :-



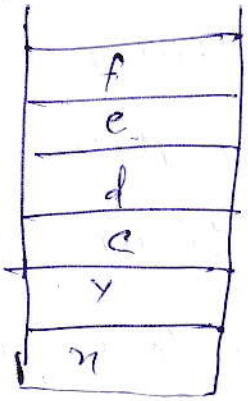
Step 3 :-

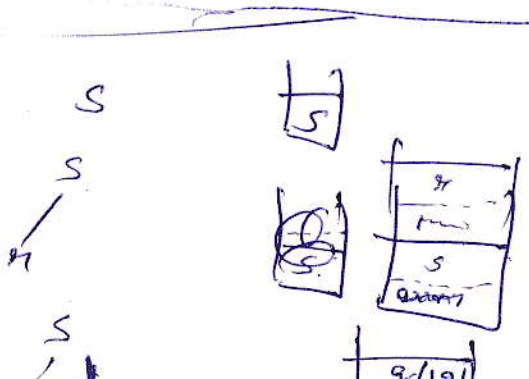


Step 4 :-



Step 5.





Step 6.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Call Sequence:-

Procedure calls are implemented by generating what is called a "call sequence" and "return sequence".

Calling sequence for a procedure are as follows:-

1. When called, an activation record is allocated
2. Actual parameter are loaded
3. m/c state is ~~stored~~ saved
4. Transfer control to the called.

Return Sequence:-

→ a sequence of m/c instruction w/c are executed each time when a procedure returns control to its caller.

→ deactivation of record of called procedure.

→ sets up return value.

→ restore m/c state

Symbol table →

① A compiler uses a symbol table to keep track of scope and binding information about names.

→ ~~After the completion of Syntax phase, Semantic phase development has been started.~~

② A symbol table is searched every time a name is encountered in the source text.

→ But before it, a ~~symbol table organization is designed.~~

③ → A data structure is used by compiler to hold info about source program Constructs.

→ Entries in the symbol table will contain info about each identifier, such as its lexeme, its type, its position in storage and any other info relevant to the translation process.

→ Symbol is searched each time where a symbol or ~~data~~ name is encountered in the source program. If the name exist in the symbol table then the info corresponding to that name is made available otherwise that name is added in symbol table.

→ Also when any new info is encountered about the existing name in the symbol table then changes are made in the info correspondance to that name.

Thus symbol table provides the mechanism to add new entries of names as well as searching or update the existing ones.

Symbol table entries :—

→ name of variable
→ constants
→ function names
→ name of procedure
→ Temporaries generated by compiler
→ Labels used in source lang. etc.

How to ~~store~~ store name in symbol table :—

2 methods by w/c names are stored in symbol table.

→ fixed length Name
→ Variable length Name

1) A fixed space is allocated to the name in the symbol table. But there are some ~~cases~~ cases where the names are too small in size as compared to space allocated for them. In such situations there is a wastage of storage space.

Name								Attribute
d	i	s	p	l	a	g		
a	d	d	r	e	s	s		
x								
y								

2) An amount of storage space is allocated to the name as it requires, with the help of starting index and length of each name, the name can be stored in the symbol table

Name		Attribute
Starting index	length	
0	8	
8	8	
16	2	
18	2	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
d	i	s	p	l	a	y	$	a	d	d	r	e	s	s	$	x	$	y	$

→ $ is a special symbol to show the end of the string and is known as end of string marker.

functions of Symbol table :-

1. Insert
2. Search
3. Delete

Symbol table management :-

→ It has to manage for

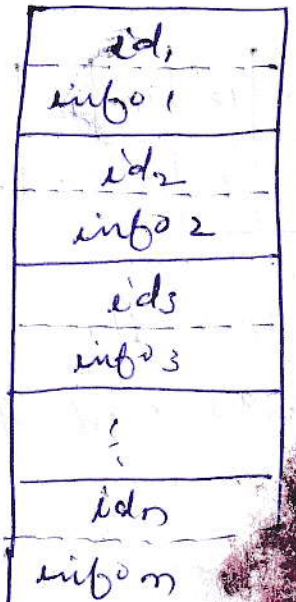
① Quick insertion of names or identifier and their related information.

② Quick searching of identifiers commonly used data structure for symbol table construction are,

List data structure for symbol table :-

→ Simplest method to implement

→ array is used to store name and info associated to them.



Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

→ New names declared are added in the order as they arrived in source program.

→ "Available" is a pointer w/c is maintained at the end of the ~~table~~ stored records. Pointer 'available' points the empty slot where ni new names w/c are assined are inserted.

→ for retrieving info^n about some names, start searching that name from the begining to the array and search it upto the available pointer.

→ If we can't find the value the name upto the pointer, then we get an error as of undeclared name.

→ Also an error occurs if we inserted a name ni the symbol table w/c is already existed ni the table.

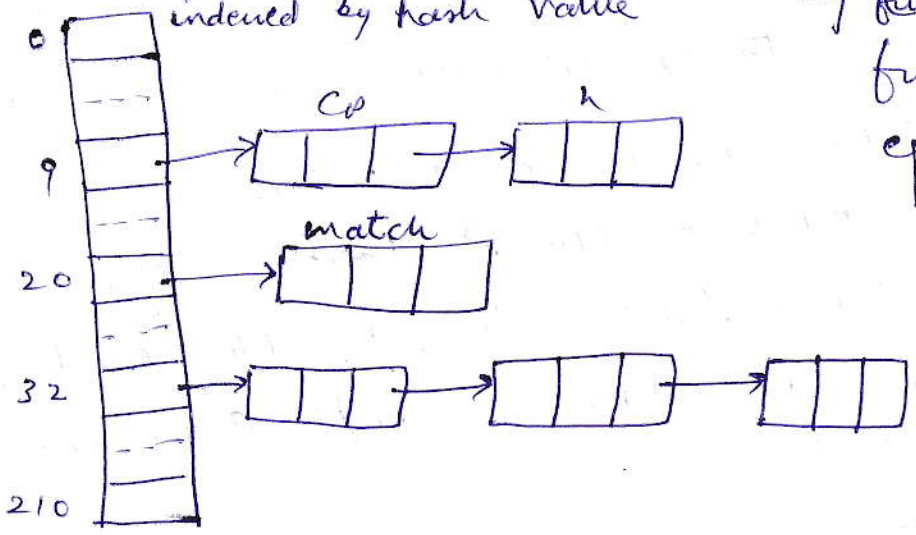
→ error occurs is "multible defined name."

→ List organization takes the minimum amount of storage space.

→ But searching ni the linear list is very time consuming so we introduce the hash table.

Hash table :-

Array of list headers, indexed by hash value



[hash value is calculated by function hash function h(5) eg MD 5 etc.

Technique to complete hash function :-

→ let string S is given to us then from its characters $c_1, c_2, c_3 \ldots c_k$, we have to determine the positive integer h.

→ Conversion of characters into integers can be done by the way used.

→ Now convert integer into the no. of a list entry, i.e., the no. in b/w 0 to P-1

[To determine whether there is an entry for string s in the symbol table, we apply a hash function h to s, such that h(s) returns an integer between 0 and m-1. If s is in the symbol table, it is on the list numbered h(s) if not in the symbol table by creating

POORNIMA
COLLEGE OF ENGINEERING

LECTURE NOTES

Campus: P.C.E. Course: B.Tech. Class/Section: VII B Date: .....

Name of Faculty: Reena Sharma Name of Subject: C.C. Code: 7CS05A

Date (Prep.): ..... Date (Del.): .....Unit No./Topic: IV .....Lect. No: .....

OBJECTIVE: To be written before taking the lecture (Pl. write in bullet points the main topics/concepts etc., which will be taught in this lecture)

Parameter Passing

IMPORTANT & RELEVANT QUESTIONS:

1. Write a short note on:
   a) Parameter Passing (RTU - 2016)

FEED BACK QUESTIONS (AFTER 20 MINUTES):

OUTCOME OF THE DELIVERED LECTURE: To be written after taking the lecture (Pl. write in bullet points about students' feedback on this lecture, level of understanding of this lecture by students etc.)

REFERENCES: Text/Ref. Book with Page No. and relevant Internet Websites:

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

DETAILED LECTURE NOTES

Campus: ..... Course: ..... Class/Section: ..... Date: .....
Name of Faculty: ..... Name of Subject: ..... Code: .....

Parameter Passing :

~~A way in w/s parameters are transferred b/w functions~~ The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

R-value : The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

l-value: The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left hand side of an assignment operator.

Pass by value: In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record. Formal parameters that hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters.

Pass by reference: In this, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. If the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value.

DETAILED LECTURE NOTES

Campus: ..... Course: ..... Class/Section: ..... Date: .....
Name of Faculty: ..... Name of Subject: ..... Code: .....

Pass by copy-restore: it is similar to "pass-by-reference" except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameter if manipulated have no real-time effect on actual parameters (as r-values are passed), but when the called procedure ends, the l-value of formal parameters are copied to the l-values of actual parameter.

eg:-

```
int y;
calling_procedure()
{
  y = 10;
  copy-restore(y); // l-value of y is passed
  printf y; // prints 99
}
```

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

```
copy_restore (int x)
{
    x = 99; // y still has value 10 (unaffected)
    y = 0; // y is now 0
}
```

When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied the the l-value of y making it behave like call by reference.

Pass by name:

Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

POORNIMA
COLLEGE OF ENGINEERING

LECTURE NOTES

Campus: PCE Course: B.Tech Class/Section: VII B Date:
Name of Faculty: Reena Sharma Name of Subject: CC Code: 7CS05A
Date (Prep.): Date (Del.): Unit No./Topic: V Lect. No:

OBJECTIVE: To be written before taking the lecture (Pl. write in bullet points the main topics/concepts etc., which will be taught in this lecture)

Basic block control flow graph, DAG representation of basic block, etc

IMPORTANT & RELEVANT QUESTIONS:

1. Write short note on:
   a) Flow graph
   b) Basic block
   c) DAG

FEED BACK QUESTIONS (AFTER 20 MINUTES):

OUTCOME OF THE DELIVERED LECTURE: To be written after taking the lecture (Pl. write in bullet points about students' feedback on this lecture, level of understanding of this lecture by students etc.)

REFERENCES: Text/Ref. Book with Page No. and relevant Internet Websites:

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

A graph representation of three address statements, called a flow graph. Nodes represent computation and edges represent flow of control.

Basic Blocks and flow Graph :-

Basic Blocks :- A basic block is a sequence of consecutive statement in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

eg :-
$$t_1 = a * a$$
$$t_2 = a * b$$
$$t_3 = 2 * t_2$$
$$t_4 = t_1 + t_3$$
$$t_5 = b * b$$
$$t_6 = t_4 + t_5$$

} *//

A name in 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 point.

Following algo can be used to partition a sequence of three address statement into basic block.

Algo :- Partition into basic blocks.

I/P - A sequence of three address statements

O/P - A list of basic blocks with each three address statement in exactly one block.

Method: 1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are the following.

(i) The first statement is a leader.

(ii) Any statement that is the target of a conditional or unconditional goto is a leader.

(iii) Any statement that immediately follows a goto or conditional goto statement is a leader.

2. for each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

eg Compute the dot product of two vectors a and b of length 20.

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i := i + 1
    end
    while i <= 20
end
```

Three address code :-

1. prod := 0
2. i := 1
3. t_1 := 4 * i
4. t_2 = a [t_1]
5. t_3 = 4 * i
6. t_4 = b [t_3]
7. t_5 = t_2 * t_4
8. t_6 = prod + t_5
9. prod = t_6
10. t_7 = i + 1
11. i = t_7
12. if i <= 20 goto (3)

Transformations on Basic Block :-

A number of transformation can be applied to a basic block without changing the set of expressions computed by the block. Many of these transformations are useful for improving the quality of code that will be ultimately generated from a basic block.

There are two important classes of local transformations that can be applied to basic blocks; these are :-

1. Structure preserving transformations
2. algebraic transformation

Structure preserving transformations :-

The primary structure preserving transformations on basic blocks are :

1. Common subexpression elimination
2. dead code elimination
3. renaming of temporary variables
4. interchange of two independent adjacent statements

[we assume basic blocks have no arrays, pointers, or procedure calls.]

1. **Common subexpression elimination :—**

eg:-
$$a = b + c$$
$$b = a - d$$
$$c = b + c$$
$$d = a - d$$

2 and 4 statements compute the same expression, $b + c - d$, and hence this basic block may be transformed into the equivalent block

$$a = b + c$$
$$b = a - d$$
$$c = b + c$$
$$d = b$$

Dead code elimination :- x is dead, that is, never ~~used~~ subsequently used, at the point where the statement $x = y+2$ appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

3. Renaming temporary variable :- [If we create basic block & normal form block then---

$t = b+c$, t is temporary. If we change this statement, $u = b+c$, where u is a new temporary variable and change all uses of this instance of t to u, then the value of the basic block is not changed.

There are called normal form block.

4. Interchange of statement :- We have a block with the two adjacent statement

$t_1 = b+c$

$t_2 = x+y$

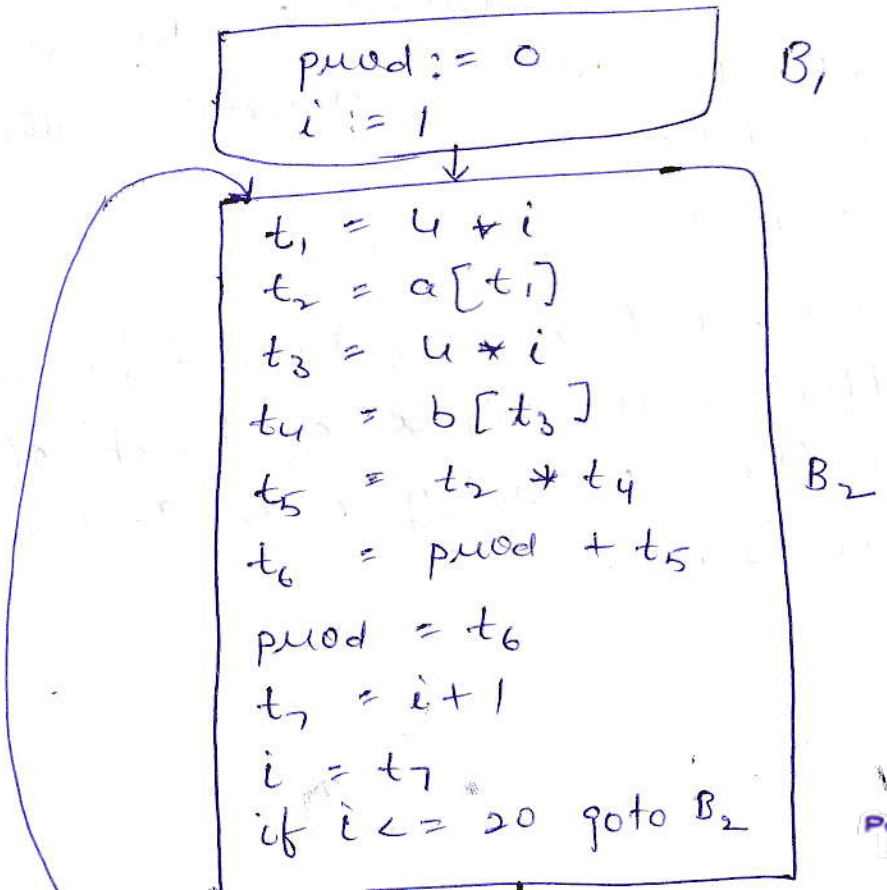
Then we can interchange the two statement without affecting the value of block if and only if neither x nor y is t_1 and neither b nor c is t_2.

Algebraic transformations :-

Flow Graph :-

We can add 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.



POORNIMA

Three address code :-

Three address code is a sequence of statements of the general form

$x := y \ op \ z$

x, y, z are names, constants or compiler generated temporaries.
op stand for operator.

Thus a source lang expression like $x + y * z$ translated into a sequence

$t_1 := y * z$
$t_2 := x + t_1$

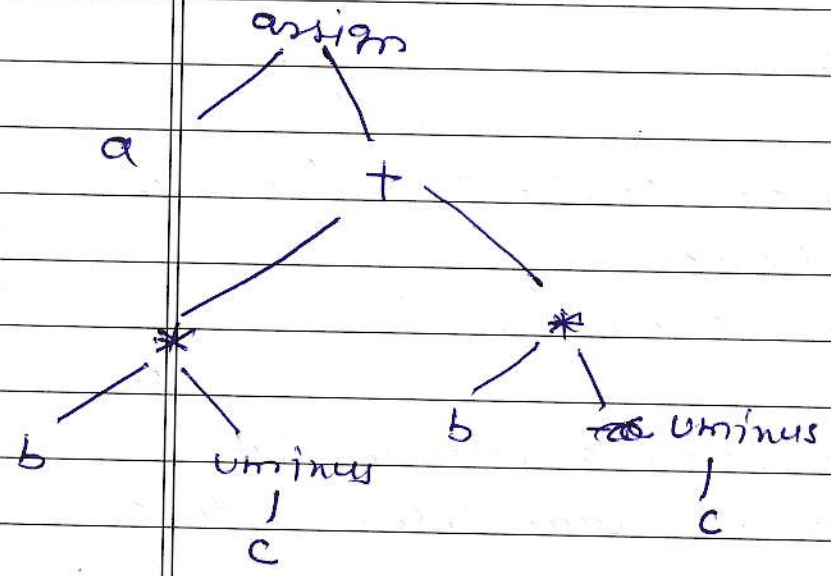
Three address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

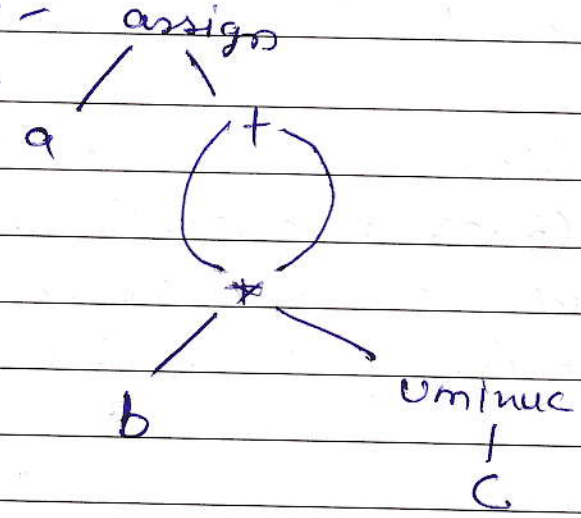
POORNIMA

a := b * -c + b * -c

Syntax tree :-



Dag :-



Postfix notation

POORNIMA

Three address code for Syntax tree :-

$t_1 := -c$
$t_2 := b * t_1$
$t_3 := -c$
$t_4 := b * t_3$
$t_5 := t_2 + t_4$
$a := t_5$

Three address code for dag :-

$t_1 := -c$
$t_2 := b * t_1$
$t_5 := t_2 + t_2$
$a := t_5$

The reason for the term "three address code" is that each statement usually contains three addresses, two for the operands and one for the result.

DAG Representation for basic blocks:-

A DAG for basic block is a directed acyclic graph with the following labels on node:

1. The leaves of graph are labeled by unique identifier and that identifiers can be variable names or constants.

2. Interior nodes of the graph is labeled by an operator symbol.

3. Nodes are also given a sequence of identifiers for labels to store the computed value.

→ DAGs are a type of data structure. It is used to implement transformations on basic blocks.

→ DAG provide a good way to determine the common sub-expression.

→ It gives a picture representation of how the value computed by the statement is used in subsequent

Algorithm for construction of DAG :-

Input : It contains a basic block.
Output : It contains the following information :

→ Each node contains a label. For leaves, the label is an identifier.
→ Each node contains a list of attached identifiers to hold the computed values.

Case (i) $x := y \ op \ z$
Case (ii) $x := op \ y$
Case (iii) $x := y$

Method :

Step 1 :

If y operand is undefined then create node (y).
If z operand is undefined then for case (i) create node (z).

Step 2 :

For case (i) create node (op) whose right child is node (z) and left child is node (y).

For case (iii), node n will be node (y).

output :

For node (x) delete x from the list of identifiers. Append x to attached identifiers list for the node n found in step 2. Finally set node (x) to n.

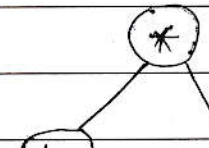
Example :-

Consider the following three address statement :

S1 := 4 * i
S2 := a[S1]
S3 := 4 * i
S4 := b[S3]
S5 := S2 * S4
S6 := prod + S5
Prod := S6
S7 := i + 1
i := S7
if i <= 20 goto (1)

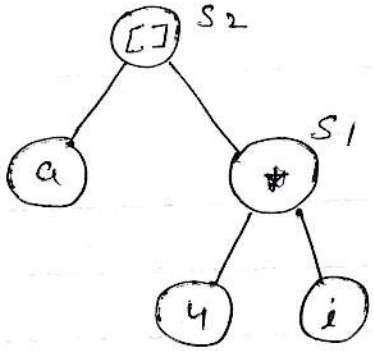
Stages in DAG construction :

a)



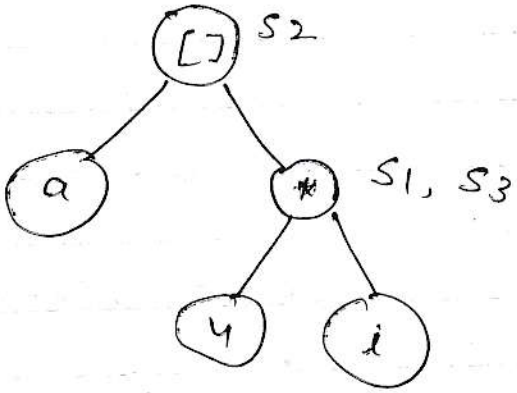
Statement (1)

(b)



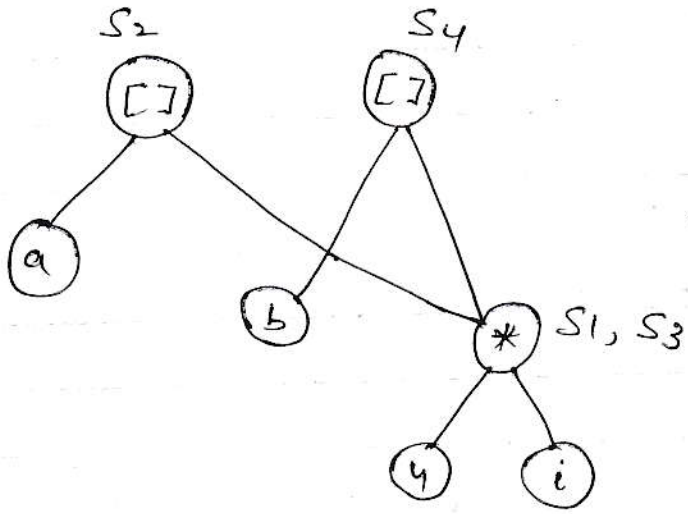
Statement (2)

(c)



4 * i node exist already hence attach identifier S3 to the existing node for statement (3)

(d)



Statement (4)

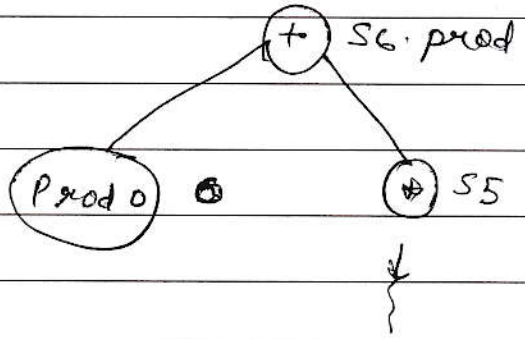
(e)

Statement (5)

(d)

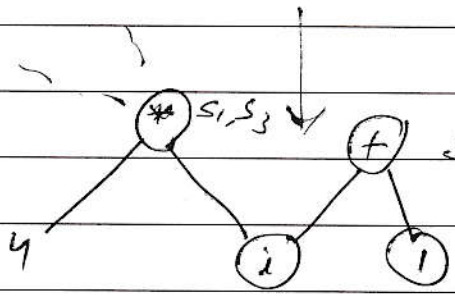
Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

(f)



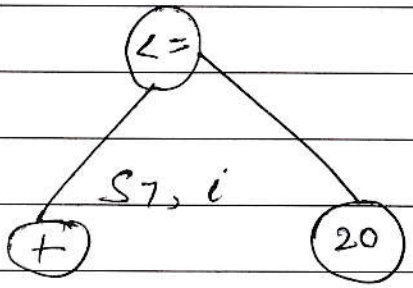
Statement (6), attach identifier prod for statement (7)

(g)



Statement (8), attach identifier i for statement (9)

(h)



Final Dag

F.

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA
COLLEGE OF ENGINEERING

LECTURE NOTES

Campus: PLE Course: B.Tech Class/Section: VII B Date: .....

Name of Faculty: Reena Sharma Name of Subject: CC Code: 7CS05A

Date (Prep.): ..... Date (Del.): .....Unit No./Topic: V .....Lect. No: .....

OBJECTIVE: To be written before taking the lecture (Pl. write in bullet points the main topics/concepts etc., which will be taught in this lecture)

Loop optimization, Loop invariant computation, Issues in design of code generator.

IMPORTANT & RELEVANT QUESTIONS:

1. What are the various issues in design of code generator, loop optimization? (RTU 2013-14)

FEED BACK QUESTIONS (AFTER 20 MINUTES):

OUTCOME OF THE DELIVERED LECTURE: To be written after taking the lecture (Pl. write in bullet points about students' feedback on this lecture, level of understanding of this lecture by students etc.)

REFERENCES: Text/Ref. Book with Page No. and relevant Internet Websites:

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

Issues in the design of code generator:

In the code generation phase, various issues can arrise:

1. Input to the code generator:

→ The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front-end.

→ Intermediate representation has the several choices:

a) Postfix notation
b) Syntax tree
c) Three address code

→ We assume front-end phase produces low-level intermediate representation i.e. values of names in it can directly manipulated by the machine instruct

→ The code generation phase need

2. Target Program: is the o/p of code gene

The o/p can be:

a) Assembly language: It allows subprograms to be seperately compiled.

b) Relocatable Machine language: It makes the process of code generation easier.

c) Absolute machine language: It can be placed in a fixed location in memory.

3. Memory management:

→ During code generation process the symbol table entries have to be mapped to actual p addresses and levels have to be mapped to instruction address.

→ Mapping name in the source program to address of data is co-operating done by the front end and code generator.

→ Local variables are stack allocation in the activation record while global variables are in static area.

4. Instruction selection:

→ Nature of instruction set of the target machine should be complete and uniform.

→ When you consider the [illegible] of target machine then the

→ The quality of the generated code can be determined by its speed and size.

5. Register allocation:

Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.
The following sub problems arise when we use registers:

Register allocation: In this, we select the set of variables that will reside in register,

Register assignment: In this, we pick the register that contains variable.

6. Evaluation Order: The efficiency of the target code can be affected by the order in which the computations are performed. Some computation orders need fewer registers to hold result of intermediate than others.

POORNIMA
COLLEGE OF ENGINEERING

DETAILED LECTURE NOTES

Campus: PCE Course: 7CS05A Class/Section: VII, A Date:
Name of Faculty: Reena Sharma Name of Subject: CC Code: 7CS05A

Loop - Invariant Computation and code Motion :

- Loop invariant computation

  A computation whose value does not change as long as control stays within the loop

- Loop invariant code motion (LICM)

  Move a loop invariant statement within a loop to the pre-header of the loop

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

The principal sources of optimization :-

[Optimization is ~~used~~ for code improving transformations]

→ local xformation if it can be performed by looking only at the statements in a basic block.
otherwise it is global.

→ Function - Preserving Transformations :-

In many ways, a compiler can improve a program without changing the function it computes.

1. Common subexpression elimination
2. Copy propagation
3. dead code elimination
4. Constant folding

1. Common Sub expression elimination :-

An occurrence 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

for example :-

$t_1 = 4 * i$
$t_2 = a[t_1]$
$t_3 = 4 * j$
$t_4 = 4 * i$
$t_5 = n$
$t_6 = b[t_4] + t_5$

the above code can be optimized using the common sub-expression elimination as

$t_1 = 4 * i$
$t_2 = a[t_1]$
$t_3 = 4 * j$
$t_5 = n$
$t_6 = b[t_1] + t_5$

the common sub expression $t_4 = 4 * i$ is eliminated as its computation is already in t_1. And value of i is not been changed from definition to use.

2. Copy propagation :-

→ Assignment of the form $f = g$ called copy statements or copies for short.

→ The idea behind the copy propagation transformation is to use g for f, whenever possible after

→ Copy propagation means use of one variable instead of another.

→ This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

Eg:-

```
x = Pi ;
 .
 .
 .
A = x * x * x ;
```

The optimization using copy propagation can be done as follows:

```
A = Pi * x * x ;
```

Here the variable x is eliminated.

3. dead code eliminations:-

Loop optimization :-

The running time of a program may be improved if we decrease the number of instruction in an inner loop, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization :-

1. Code motion, which moves code outside a loop.
2. Induction - variable elimination, which we apply to replace variables from inner loop.
3. Reduction in length, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

1. Code motion :→ An important modification that decrease the amount of code in a loop is code motion.

→ This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop invariant computation) and places the expression before the loop.

→ Note that the notion "before the loop" assumes the existence of an

→ for example, evaluation of limit - 2 is a loop - invariant computation in the following while - statement:

↳ while (i <= limit - 2) /* statement does not change limit */

↳ code motion will result in the equivalent of t = limit - 2;

↳ while (i <= t) /* statement does not change limit or t */

2. Induction Variables :-

→ Loops are usually processed inside out. For example consider the loop around B_3.

→ Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decrease by 4 because $4 * j$ is assigned to t_4. Such identifiers are called induction variables.

→ When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B_3 in Fig. we cannot get rid of either j or t_4 completely; t_4 is used in B_3 and in B_4.

→ However, we can illustrate reduction in strength and illustrate a

3. Reduction in strength :

→ Strength reduction is used to replace the expensive operation by the cheaper once on the target machine.

→ Addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop.

eg:

```
while (i<10)
  {
  j = 3 * i + 1 ;
  a[j] = a[j] * -2 ;
  i = i + 2;
  }
```

After ~~reduction~~ strength reduction the code will be :

```
S = 3 * i + 1;
  while (i<10)
{
  j = S;
a[j] = a[j] - 2;
```

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

POORNIMA
COLLEGE OF ENGINEERING

LECTURE NOTES

Campus: P.C.E. Course: B.Tech. Class/Section: VII B Date: .....

Name of Faculty: Reena Sharma Name of Subject: C.C. Code: 7CS05A

Date (Prep.): ..... Date (Del.): .....Unit No./Topic: V .....Lect. No: .....

OBJECTIVE: To be written before taking the lecture (Pl. write in bullet points the main topics/concepts etc., which will be taught in this lecture)

peephole optimization

IMPORTANT & RELEVANT QUESTIONS:

1. What is peephole optimization? Explain it. (RTU 2012-13)

FEED BACK QUESTIONS (AFTER 20 MINUTES):

OUTCOME OF THE DELIVERED LECTURE: To be written after taking the lecture (Pl. write in bullet points about students' feedback on this lecture, level of understanding of this lecture by students etc.)

REFERENCES: Text/Ref. Book with Page No. and relevant Internet Websites:

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
181-B, RIICO Institutional Area
Sitapura, JAIPUR

DETAILED LECTURE NOTES

Campus: ..... Course: ..... Class/Section: ..... Date: .....
Name of Faculty: ..... Name of Subject: ..... Code: .....

Peephole optimization: a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence.

The Peephole is a small, moving window on the target program.

A bunch of statements is analyzed and are checked for the following possible optimization.

Redundant instruction eliminations:

At source code level, the following can be done by the user:

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

```
int add_ten (int x)
{
  int y, z;
  y = 10;
  z = x+y;
  return z;
}
```

```
int add_ten (int
  {
  int y;
  y = 10;
  y = x+y;
  return y;
  }
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. for eg:

```
MOV x, R0
MOV R0, R1
```

We can delete the first instruction and re-write the sentence as;

```
MOV x, R1
```

Unreachable code:

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

eg:

DETAILED LECTURE NOTES

Campus: ..... Course: .....
Name of Faculty: .....
Class/Section: .....
Name of Subject: .....
Date: .....
Code: .....


```
void add (int x)
{
  return * x + 10;
  printf ("value of x is %d", x);
}
```

In this code segment, the printf statement will never be executed as the program control returns back before it can execute, hence printf can be removed.

Flow of control optimization :

There are instance in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code :

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

```
MOV R1, R2
GOTO L1
...
L1 : GOTO L2
L2 : INC R1
```

In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
MOV R1, R2
GOTO L2
...
L2 : INC R1
```

Algebraic expression simplification:

the expression a = a+0 can be replaced by a itself and the expression a = a+1 can simply be replaced by INC a.

Strength reduction:

There are operations that consume more time and space. Their strength can be reduced by replacing them with other operations that consume less time and space, but produce the same result.

for eg: a*2 can be replaced by a<<1, which involves only one left shift. Though the o/p of a*a and a^2 is same a*a is much more efficient to implement.

Accessing machine instructions :

The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently. If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient results.

Data flow analysis :—

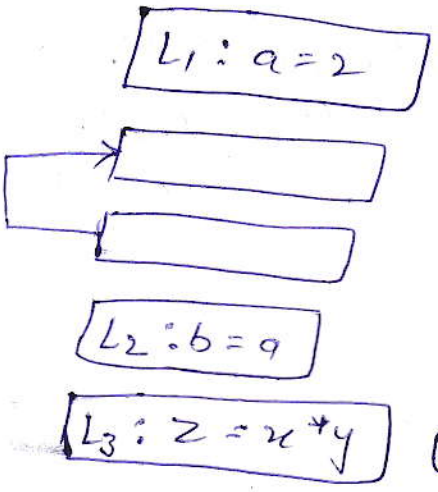
It is the analysis of flow of data in control flow graph, i.e., the analysis that determines the information regarding the definition and use of data in program. With the help of this analysis optimization can be done. In general, its process in which values are computed using

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

data flow analysis. The data flow property represents information which can be used for optimization.

Basic Terminologies:-

- **Definition Point**: a point in a program containing some definition.
- **Reference Point**: a point in a program containing a reference to a data item.
- **Evaluation Point**: a point in a program containing evaluation of expression.



Definition Point

Reference Point

Evaluation Point

Data flow Properties:

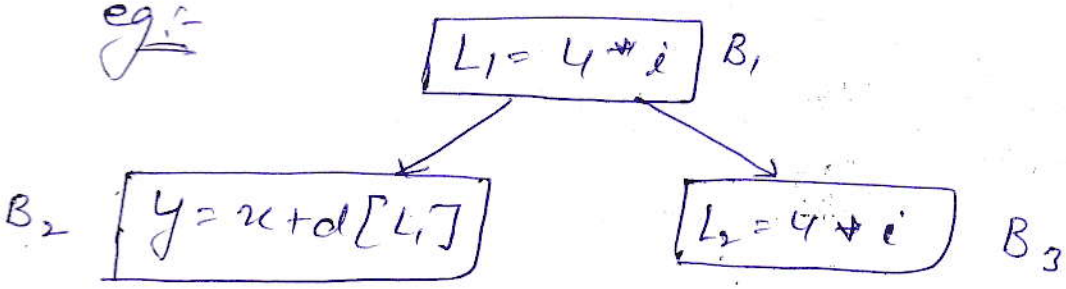
- **Available Expression**: A expression is said to be available at a program point x iff along paths its reaching to x. A expression is available at its evaluation point.

A expression a+b is said to [illegible] if none of the operands gets modified before their use.

DETAILED LECTURE NOTES

Campus: ..... Course: ..... Class/Section: ..... Date: .....
Name of Faculty: ..... Name of Subject: ..... Code: .....

eg:-



Expression 4*i is available for block B_2, B_3

Advantage :

It is used to eliminate common sub expression.

Reaching Definition : A definition D is reaches a point x if there is path from D to x in which D is not killed. i.e., not redefined.

eg:-

$D_1 : x = 4$ B_1

$D_2 : x = x + 2$ B_2

$D_3 : y = 9x + 2$ B_3

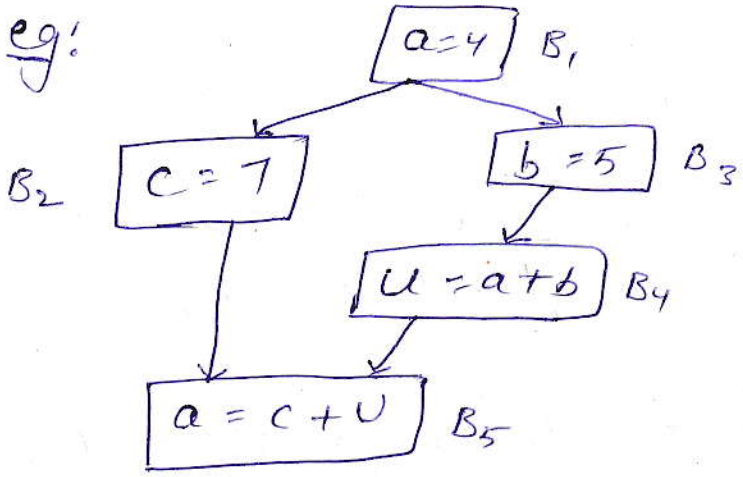
D_1 is reaching definition for B_2 but not for B_3 it is killed by D_2

Advantage :

It is used in constant and variable propa

- Live variable : A variable is said to be live at some point p if from p to end the variable is used before it is redined else it becomes dead.

eg:



a is live at block B_1, B_3, B_4 but killed at B_5

Advantages :

1. It is useful for register allocation
2. It is used in dead code elimination.

- Busy Expression : An expression is busy along a ~~Path~~ path iff its evaluation exists along that path and none of its operand definition exists before its evaluation along the path.

Advantages :

It is used for performing code motion optimization.

POORNIMA
COLLEGE OF ENGINEERING

LECTURE NOTES

Campus: PCE Course: B.Tech Class/Section: VII B Date: .....
Name of Faculty: Reena Sharma Name of Subject: CC Code: 7CS05A
Date (Prep.): ..... Date (Del.): ..... Unit No./Topic: V Lect. No: .....

OBJECTIVE: To be written before taking the lecture (Pl. write in bullet points the main topics/concepts etc., which will be taught in this lecture)

A Simple Code generator, Code generation from DAG

IMPORTANT & RELEVANT QUESTIONS:

Q.1 Explain the steps required for code code generation from DAG. (RTU 2013-14)

FEED BACK QUESTIONS (AFTER 20 MINUTES):

OUTCOME OF THE DELIVERED LECTURE: To be written after taking the lecture (Pl. write in bullet points about students' feedback on this lecture, level of understanding of this lecture by students etc.)

REFERENCES: Text/Ref. Book with Page No. and relevant Internet Websites:

Dr. Mahesh Bundele
B.E., M.E., Ph.D.
Director
Poornima College of Engineering
ISI-6, RIICO Institutional Area
Sitapura, JAIPUR

A Simple Code generator:-

A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

→ Target language: The code generator has to be ~~aware~~ aware of the nature of the target language for which the code is to xformed.

→ IR Type: Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code

→ Selection of instruction: The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set. One representation can have many ways (~~wants~~ instruction) to convert it; so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.

→ Register Allocation: A prog[illegible]

architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.

Ordering of instructions: At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

Descriptors: The code generator has to track both the registers (for availability) and addresses (Location of values) while generating the code. For both of them, the following two descriptors are used:

→ Register descriptor :- Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.

Address descriptor: Values of the names (identifiers) used in the program might be stored at different locations while in execution. AD also used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

Code generator keeps both the descriptor updated in a real-time. For a load statement, LD R1, x, the code generator:

→ Updates the Register Descriptor R1 that has value of x and

→ updates the Address Descriptor (x) to show that one instance of x is in R1.

Code generation :—

Basic blocks comprise of a sequence of three address instructions. code generator takes these sequence of instructions as input.

Note :- If the value of a name is found at more than one place (register, cache or memory), the registers value will preferred over the cache and main

barely any preference.

getReg: Code generator uses getReg function to determine the status of available registers and the location of name values.

getReg works as follows:

1. If variable Y is already in register R, it uses that register.
2. else if some register R is available, it uses that register.
3. else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

For an instruction x = y op z, the code generator may perform the following actions. Let us assume that L is the location (preferably register) where the o/p of y op z is to be saved.

→ Call function getReg, to decide the location of L.

→ Determine the present location (register or memory) of y by consulting the address descriptor of y. If y is not present in register L, then generate the following instruction to copy the value

determine the present location of z using the same method used in step 2 for y and generate the following instruction:

OP z', L

where z' represents the copied value of z.

→ Now L contains the value of y or z, that is intended to be assigned to x. So, if L is a register, update its descriptor to indicate that it contains the value of x. Update the descriptor of x to indicate that it is stored at location L.

→ If y and z has no further use, they can be given back to the system.

Other code constructs like loops and conditional statements are transformed into assembly language in general assembly may