In Pursuit of Excellence	**Course and Faculty Details**	SESSION-2019-2020
		SEM-:6th

Faculty Details

Name of the Faculty: PRIYANKA GOEL

Designation: ASSISTANT PROFESSOR

Department: DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Course Details

Name of the Programme: B.Tech. Batch: 2017-2021

Branch: COMPUTER SCIENCE & ENGINEERING Section: B, C

Name of Subject: COMPILER DESIGN Subject Code: RCS 602

Category of Course: CORE SUBJECT

In Pursuit of Excellence	**Index**	SESSION-2019-2020 SEM- 6th

In Pursuit of Excellence	Vision & Mission of Institute	SESSION-2019-2020
		SEM- 6th

Vision of Institute

To develop industry ready professionals with values and ethics for global needs.

Mission of Institute

- To impart education through outcome based pedagogic principles.
- To provide conducive environment for personality development, training and entrepreneurial skills.
- To induct high professional ethics and accountability towards society in students.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	Vision & Mission Of Department	SESSION-2019-2020
		SEM- 6th

Vision of Department

To develop globally recognized computer science and engineering graduates with ethical values for need of software industries.

Mission of Department

1. To impart knowledge through well defined instructional objectives in the field of computer science and engineering.
2. To provide learning ambiance for skills, innovation, leadership and overall personality development.
3. To inculcate professional ethics, teamwork and responsiveness towards society.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Program Education Objectives**	SESSION-2019-2020 SEM- 6th

Program Education Objectives

PEO 1 : The graduates will have entrepreneurial and employable skills in software industries, by adapting themselves in the corporate world by utilizing the defined instructional objectives learnt in the program.

PEO 2 : The graduates will engage in skill enhancement, that would help to work in their own area of interest, individually or in a team.

PEO 3 : The graduates will demonstrate ownership and responsiveness towards the profession and the society.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	Program Outcomes	SESSION-2019-2020 SEM- 6th

Program Outcomes

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization for the solution of complex engineering problems.

2. Problem analysis: Identify, formulate, research literature, and analyse 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 public health and safety, and 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 information to provide valid conclusions.

5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools, including prediction and modeling to complex engineering activities, with an understanding of the limitations.

6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

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

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

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

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with the 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.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Program Specific Outcomes**	SESSION-2019-2020
		SEM- 6th

After completing their graduation, students of Computer Science and Engineering will be able to -

PSO 1: Comprehend the core subjects of CSE and apply them to resolve domain specific tribulations.

PSO 2: Extrapolate the fundamental concepts in engineering and to apply latest technology with programming language skills to develop, test, implement and maintain software products.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	Academic Calendar	SESSION-2019-2020 SEM- 6th

Moradabad Institute of Technology

Ramganga Vihar Phase – II, Moradabad

Date: 16-01-2020

ACADEMIC CALENDAR

Even Semester **Session: 2019 – 2020**

S. No.	Particulars	Date	Responsibility
1.	**Time Table** (a) Display on Notice Boards (b) Distribution to concerned Teachers	 18 Jan 2020 18 Jan 2020	 O.C. Time Table
2.	Distribution of Students' lists to teachers	18 Jan 2020	Concerned HODs /O.C. Class
3.	Blow up submission to HODs	18 Jan 2020	Concerned Teachers
4.	**Registrations** (a) 2nd and 4th Semester (b) 6th and 8th Semester (b)List of unregistered students to various department (c) Notifying unregistered students for getting registered at the earliest (through class O.Cs, / Faculty)	 20 Jan 2020 21 Jan 2020 27 Jan 2020 29 Jan 2020	 Concerned Teachers OS Academic Concerned HODs
5.	**Commencement of Classes** (a) 2nd and 4th Semester (b) 6th and 8th Semester	 21 Jan 2020 22 Jan 2020	 HODs and Concerned Teachers
6.	Announcement of Test series dates	30 Jan 2020	Dean Academics
7.	Procurement of stationary & materials for Test Series for full semester (a) Requirement (b) Actual Procurement	 10 Feb 2020 15 Feb 2020	 Convener Test Series Committee O.S. Academics
8.	(a)Short attendance compilation before Class Test-1 (b) Information to parents (c)Undertaking form handed over to students (b)Collection of undertaking form	20 Feb 2020 21 Feb 2020 21 Feb 2020 22 Feb 2020	O.C. Class
9.	**1st Test Series**	**24, 25 and 26 Feb 2020**	
	Announcement of Test Series schedule, Invigilation Programme, Seating arrangement etc.	18 Feb 2020	Class Test Committee
	After completion of Test Series (a) Evaluation of test copies & showing of copies to students (b) Report of poor performance of students to class OCs (c)) Submission of test copies in Nodal Centre	 29 Feb 2020 29 Feb 2020 29 Feb 2020	 Concerned Teachers Concerned Teachers Concerned Teachers
10.	(a) Last date for submission of examination forms to office (b) Submission of forms to University	06 March 2020** 07 March 2020**	OS Academic to take timely action as per University directions.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Academic Calendar**	SESSION-2019-2020 SEM- 6th

11.	Mid Semester break	09 March to 11 March 2020	
12.	Announcement of dues list and its last date for clearing dues (Current semester)	25 March 2020	Accounts/ OS Academic
13.	(a)Short attendance compilation before Class Test-2 (b) Information to parents (c)Undertaking form handed over to students (b)Collection of undertaking form	01 April 2020 03 April 2020 03 April 2020 04 April 2020	O.C. Class
14.	**2nd Test Series**	**07, 08 and 09 April 2020**	
	Announcement of Test Series schedule, Invigilation Programme, Seating arrangement etc.	03 April 2020	Class Test Committee
	After completion of Test Series (a) Evaluation of test copies & showing of copies to students (b) Report of poor performance of students to class OCs (c)) Submission of test copies in Nodal Centre	 13 April 2020 13 April 2020 13 April 2020	 Concerned Teachers Concerned Teachers Concerned Teachers
15.	Filling of student feedback forms for current semester	22 April 2020	Concerned HODs
16.	Requirement of additional Faculty (to be conveyed to Director) (for even semester)	30 April 2020	Concerned HODs
17.	(a) Floating the electives for even semester (b) Last date for students choice	22 April 2020 23 April 2020	Concerned HODs
18.	Date up to which final attendance is to be counted	26 April 2020	Concerned teachers
19.	Submission of consolidated list of shortage of attendance to Director and information to Parents	27 April 2020	Class O.Cs
20.	**3rd Test Series**	**28,29,30 April 2020**	
	Announcement of Test Series schedule, Invigilation Programme, Seating arrangement etc.	23 April 2020	Class Test Committee
	After completion of Test Series (a) Evaluation of test copies & showing of copies to students (b) Report of poor performance of students to class OCs (c)) Submission of test copies in Nodal Centre	 04 May 2020 04 May 2020 04 May 2020	Concerned Teacher Concerned Teachers Concerned Teachers
21.	**Submission of sessional marks:** (a) Meeting of Dean Academics, , all HODs and Director regarding attendance and performance of students.	05 May 2020	Dean Academics
	(b) Checking of Teachers' Records by HODs	06 May 2020	Concerned HODs
	(c) Finalization of sessional marks	08 May 2020	Concerned Teachers
	(d) Submission of Award list after final checking and uploading to OS Academics for further necessary action	As per date announced by AKTU	HODs Concerned Teachers
22.	**Theory Examinations:** (a) Collection of Admit Cards / Roll Nos. from University (b) Preparation of Roll lists (c) Collection of stationery such as copies, practical copies drawing sheets, graph paper etc. from University. (c) Procurement of stationery and other materials locally as necessary.	As per AKTU schedule	OS Academics to take appropriate actions as per University directions.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Academic Calendar**	SESSION-2019-2020 SEM- 6th

23.	**Practical Examinations:**	As per AKTU schedule	Concerned HODs
	(a) Appointment of Internal Examiners	3 days before the practical exam schedule	Concerned HODs
	(b) Obtaining list of panel of External Examiners from AKTU & preparation of schedule of practical examination.	As per AKTU schedule	OS Academics
	(d) Dispatch of letters/contacting the external examiners	Within 2 days of list obtained from AKTU	HODs and concerned teachers
24.	**Preparation for Even Semester** (a) Load Distribution by Department (b) Submission to O.C. Time Table	 15 May 2020 16 May 2020	 Concerned Coordinators O.C. Time Table
25.	Registration for odd semester (2020 – 21)	To be announced**	OS Academic

****May be revised as per AKTU Schedule.**

16-01-2020

Dean Academics

Director

Copy to:

1. Chairman
2. Secretary
3. P.A. to Director for Director's folder
4. All HODs
5. DOSW
6. Controller of Examination
7. O.C. Time Table
8. Registrar
9. All Faculty Members through HODs
10. O.S. Academics
11. A.S. Examinations
12. Account Section
13. T & P Cell
14. Librarian
15. Convener Test Series

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

MORADABAD INSTITUTE OF TECHNOLOGY, MORADABAD

Ram Ganga Vihar Phase-II Moradabad (U.P.)

Approved by AICTE and Affiliated to Dr. A.P.J. Abdul Kalam Technical University, Lucknow

Website: http://mitmoradabad.edu.in

Department Academic Calendar, Even Semester, Session (2019 – 2020)

VISION

To develop globally recognized computer science and engineering graduates with ethical values for need of software industries.

MISSION

M1: To impart knowledge through well defined instructional objectives in the field of computer science and engineering.

M2: To provide a learning ambience for skills, innovation, leadership and overall personality development.

M3: To inculcate professional ethics, teamwork and responsiveness towards society.

JANUARY-2020

Su	M	T	W	Th	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

FEBRUARY-2020

Su	M	T	W	Th	F	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

MARCH-2020

Su	M	T	W	Th	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

APRIL-2020

Su	M	T	W	Th	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

MAY-2020

Su	M	T	W	Th	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

JUNE-2020

Su	M	T	W	Th	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

A	Time Table Display on Notice Boards
B	Blow Up Submission to HODs
C	2nd /4th /6th /8h semester registration
D	2nd /4th /6th /8h SEM Commencement of Classes
E	Republic Day
F	Expert Lecture on Machine Learning & IOT by Mr. Abhey Kumar Bains & Mr. Aman Kumar Singh, Scope Telecom, Chandigarh
G	Expert Lecture on Image Classification using Machine Learning by Mr. Rahul Pathak, CETPA Noida
H	Event 'Dosto ki Mehfil' by CSSS
I	Short Attendance Compilation
J	Maha Shivratri
K	1st Test Series
L	Submission of Test copies in Nodal Center
M	Event 'Filmy Bytes' by CSSS
N	Mid Semester Break(Holi and Birthday of Mohd. Hazrat Ali)
O	Classes Suspended due to Lockdown
P	New Time Table for Online Classes
Q	Commencement of Online Classes
R	Information of CTs to parents and students through Counsellors
S	2nd Test Series
T	Information regarding filling of Examination Form to students
U	Submission of CT 2 Marks on MIT ERP
V	Departmental Meeting on Google Meet
W	Submission of Concept Map by Subject Coordinators
X	Onine Conduction of Event 'Lockdown with Family' by CSSS
Y	Online Conduction of Event 'MAA' by CSSS
Z	Webinar on 'Internet Routing' by Dr. Mahesh Kumar organized by CSE Deptt, MIT
AA	Webinar on 'Apache Airflow' by Shivam Saxena organized by CSE Deptt, MIT
AB	Submission of Sessional Marks on AKTU ERP

Month	Dates of Teaching Days (2nd, 3rd & 4th Year)	No. of Teaching Days	No. of Lecture Hours
Jan-2020	22,23,24,25,27,28,29,30,31	09	91 × 6 = 546
Feb-2020	1,3,4,5,6,7,8,10,11,12,13,14,15,17,18,19,20,22,27,28,29	21	
Mar-2020	2,3,4,5,6,7,12,13,14,16,23,24,25,26,27,28,30,31	18	
Apr-2020	1,2,3,4,6,7,8,9,10,11,15,16,17,18,20,21,22,23,24,25,27,28,29,30	24	
May-2020	1,2,4,5,6,7,8,9,11,12,13,14,16	13	
	Total	85	
	Sessional Examinations	06	
	Total Teaching Days	**91**	546

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	Course Evaluation Scheme	SESSION-2019-2020
		SEM- 6th

SIXTH SEMESTER

Sl No.	Subject Code	Subject Name	L-T-P	Theory/ Lab (ESE) Marks	Sessional		Total	Credit
					Test	Assign/Att.		
1	RAS601	INDUSTRIAL MANAGEMENT	3---0---0	70	20	10	100	3
2	RAS602 / RUC601	INDUSTRIAL SOCIOLOGY/ CYBER SECURITY	3---0---0	70	20	10	100	3
3	RCS-601	Computer Networks	3---0---0	70	20	10	100	3
4	RCS-602	Compiler Design	3---1---0	70	20	10	100	4
5	RCS-603	Computer Graphics	3---0---0	70	20	10	100	3
6	CS-Elective-2	DEPTT ELECTIVE COURSE-2	3---1---0	70	20	10	100	4
7	RCS-651	Computer Networks Lab	0---0---2	50	-	50	100	1
8	RCS-652	Compiler Design Lab	0---0---2	50	-	50	100	1
9	RCS-653	Computer Graphics Lab	0---0---2	50	-	50	100	1
10	RCS-654	Data Warehousing & Data Mining Lab	0---0---2	50	-	50	100	1
	TOTAL						**1000**	**24**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	Course Syllabus as per University	SESSION-2019-2020 SEM- 6th

RCS-602: COMPILER DESIGN		3-1-0
Unit	**Topic**	**Proposed Lecture**
I	**Introduction to Compiler**: Phases and passes, Bootstrapping, Finite state machines and regular expressions and their applications to lexical analysis, Optimization of DFA-Based Pattern Matchers implementation of lexical analyzers, lexical-analyzer generator, LEX compiler, Formal grammars and their application to syntax analysis, BNF notation, ambiguity, YACC. The syntactic specification of programming languages: Context free grammars, derivation and parse trees, capabilities of CFG.	08
II	**Basic Parsing Techniques:** Parsers, Shift reduce parsing, operator precedence parsing, top down parsing, predictive parsers Automatic Construction of efficient Parsers: LR parsers, the canonical Collection of LR(0) items, constructing SLR parsing tables, constructing Canonical LR parsing tables, Constructing LALR parsing tables, using ambiguous grammars, an automatic parser generator, implementation of LR parsing tables.	08
III	**Syntax-directed Translation:** Syntax-directed Translation schemes, Implementation of Syntax-directed Translators, Intermediate code, postfix notation, Parse trees & syntax trees, three address code, quadruple & triples, translation of assignment statements, Boolean expressions, statements that alter the flow of control, postfix translation, translation with a top down parser. More about translation: Array references in arithmetic expressions, procedures call, declarations and case statements.	08
IV	**Symbol Tables**: Data structure for symbols tables, representing scope information. Run-Time Administration: Implementation of simple stack allocation scheme, storage allocation in block structured language. Error Detection & Recovery: Lexical Phase errors, syntactic phase errors semantic errors.	08
V	**Code Generation:** Design Issues, the Target Language. Addresses in the Target Code, Basic Blocks and Flow Graphs, Optimization of Basic Blocks, Code Generator. Code optimization: Machine-Independent Optimizations, Loop optimization, DAG representation of basic blocks, value numbers and algebraic laws, Global Data-Flow analysis.	08

REFRENCES:

1. K. Muneeswaran,Compiler Design,First Edition,Oxford University Press.
2. J.P. Bennet, "Introduction to Compiler Techniques", Second Edition, Tata McGraw-Hill,2003.
3. Henk Alblas and Albert Nymeyer, "Practice and Principles of Compiler Building with C", PHI, 2001.
4. Aho, Sethi & Ullman, "Compilers: Principles, Techniques and Tools", Pearson Education
5. V Raghvan, " Principles of Compiler Design", TMH
6. Kenneth Louden," Compiler Construction", Cengage Learning.
7. Charles Fischer and Ricard LeBlanc," Crafting a Compiler with C", Pearson Education

RCS-652: COMPILER DESIGN LAB

1. Implementation of LEXICAL ANALYZER for IF STATEMENT
2. Implementation of LEXICAL ANALYZER for ARITHMETIC EXPRESSION
3. Construction of NFA from REGULAR EXPRESSION
4. Construction of DFA from NFA
5. Implementation of SHIFT REDUCE PARSING ALGORITHM
6. Implementation of OPERATOR PRECEDENCE PARSER
7. Implementation of RECURSIVE DESCENT PARSER
8. Implementation of CODE OPTIMIZATION TECHNIQUES
9. Implementation of CODE GENERATOR

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Syllabus Adopted by the Program**	SESSION-2019-2020 SEM- 6th

Syllabus

Pre-requisites:

The student should have basic knowledge of TAFL and CAO and basics of any programming language .

RCS 602 : Compiler Design

Unit I

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

Beyond: ***Different types of translators, Thompson Construction Algorithm and Subset Construction Algorithm***

Unit II

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

Unit III

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

Unit IV

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

Unit V

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

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	Course Outcomes	SESSION-2019-2020
		SEM- 6th

COURSE OUTCOMES

Once the student has successfully completed this course, he/she will be able to:

Course Code	CO		Course Outcomes(COs)	Cognitive Levels
RCS 602	CO1	RCS 602.1	Understand the concept of translator and various phases involved in the compilation process.	Understand
	CO2	RCS 602.2	Implement the parsing techniques for the given programming construct	Apply
	CO3	RCS 602.3	Understand the different representations of intermediate code.	Understand
	CO4	RCS 602.4	Understand the use of symbol table and apply different error recovery routines to recover the errors seen at different phases of compilation.	Understand
	CO5	RCS 602.5	Describe and Implement techniques for machine code generation and its optimization.	Apply

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Course Delivery Method**	SESSION-2019-2020 SEM- 6th

Name of Subject: Compiler Design Subject Code: RCS 602

Branch: Computer Science and Engineering

Delivery Methods for each Unit :

Unit 1	**Chalk & Talk, Power Point Presentation, Tutorials, solving numericals, Practicals, assignments**
Unit 2	Chalk & Talk, Power Point Presentation, Tutorials, solving Numericals/Design exercises, assignments and Practicals
Unit 3	Power Point Presentation, Tutorials, Video Lectures, solving Numericals/ Design exercises, assignments and Quiz, Self created Videos*
Unit 4	Power Point Presentation, Interactive Sessions, Tutorials
Unit 5	Power Point Presentation, Tutorials, solving Numericals/Design exercises,, assignments, quiz

* https://www.youtube.com/user/priyanka01ful

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	Mapping	SESSION-2019-2020
		SEM- 6th

Mapping of Course Outcomes with POs & PSOs:

Course Code	CO		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
RCS 602	CO1	RCS 602.1	3		2	3	2	2				1		1
	CO2	RCS 602.2	3	3		3	3							2
	CO3	RCS 602.3	3	2		2								1
	CO4	RCS 602.4	2	1										1
	CO5	RCS 602.5	3	2		3	2					1		2
Mapping Strength		RCS 602	2.8	2	2	2.8		2				1		2

Course Code	CO		PSO1	PSO2
RCS 602	CO1	RCS 602.1	3	3
	CO2	RCS 602.2	3	3
	CO3	RCS 602.3	2	2
	CO4	RCS 602.4	2	2
	CO5	RCS 602.5	3	3
Mapping Strength		RCS 602	2.6	2.6

CO 1: Understand the concept of translator and various phases involved in the compilation process.

CO 2: Able to implement the parsing techniques for the given programming construct

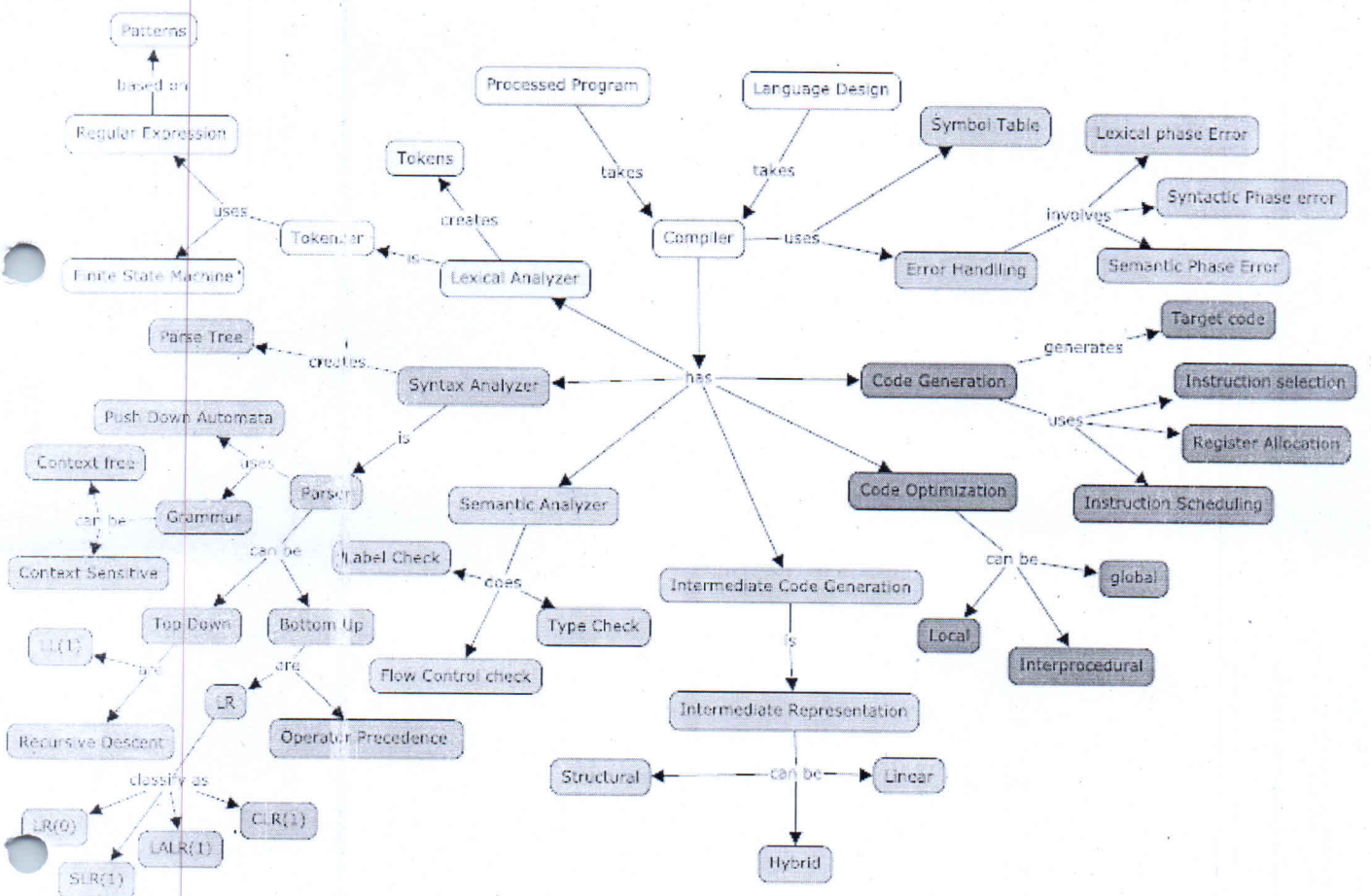
CO 3: Understand the different representations of intermediate code.

CO 4: Understand the use of symbol table and apply different error recovery routines to recover the errors seen at different phases of compilation.

CO 5: Able to describe and implement techniques for machine code generation and its optimization.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology

In Pursuit of Excellence	**Concept Map**	SESSION-2019-2020 SEM- 6th



Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

MORADABAD INSTITUTE OF TECHNOLOGY, MORADABAD
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
FACULTY TIME TABLE -2019-20 (EVEN SEMESTER)

FACULTY NAME – MS. PRIYANKA GOEL (PYG)

L	T	P	TOTAL
6	6	2	14

w.e.f. – 21/01/2020 Revised on:27/01/2020

TIME / DAY	9.00-10.00 am	10.00-11.00am	11.00 - 12.00pm	12.00-01.00pm	01.00-2.00pm	2.00-3.00pm	3.00-4.00pm	4.00-5.00pm
MON	RCS-652 6TH C3 B-113			RCS-602 (T) 6TH B3 B-318		RCS-602 (L) 6TH B B-321		
TUE	RCS-602 (L) 6TH B B-311	RCS-602 (L) 6TH C B-321		RCS-602 (T) 6TH C2 B-321		RCS-602 (T) 6TH C1 B-316		RCS-602 (T) 6TH B2 B-318
WED								
THU	RCS-602 (L) 6TH C B-311					RCS-602 (T) 6TH B1 B-319		RCS-602 (T) 6TH C3 B-321
FRI		RCS-602 (L) 6TH C B-311						
SAT			RCS-602 (L) 6TH B B-311					

LUNCH

SUBJECT CODE	SUBJECT NAME
RCS-602	COMPILER DESIGN
RCS-652	COMPILER DESIGN LAB

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Ms.Kanchan
(Deptt. Coordinator Time-Table)

Mr.Rakesh Kumar Gangwar
(O.C.Time-Table)

MORADABAD INSTITUTE OF TECHNOLOGY, MORADABAD
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
FACULTY ONLINE TIME TABLE 2019-20 (EVEN SEMESTER)

SEMESTER: 6th
w.e.f.: 22/03/2020

TIME / DAY	9.00-9.50 am	9.55-10.45am	10.50 –11.40pm	11.45 -12.35 pm	12.40-1.30 pm
MONDAY		RCS-602(L) B+C			
TUESDAY		RCS-602(L) B+C			
WEDNESDAY		RCS-602(L) B+C			
THURSDAY		RCS-602(L) B+C			
FRIDAY		RCS-602(L) B+C			
SATURDAY		RCS-602(L) B+C			

Subject Code	Subject Name
RCS-602	Compiler Design

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

(Kanchan)
(Deptt. Coordinator Time-Table)

(Mr.Rakesh Kumar Gangwar)
(O.C Time-Table)

In Pursuit of Excellence	**Lecture Plan**	SESSION-2019-2020 SEM- 6th

SECTION B **Total Lectures: 40**

Sr. No.	No. of Lectures	Topics/Sub Topics	Reference Books	CO Covered	Planned Date	Coverage Date	Sign
1.	1	Introduction to Course Educational Objective, Course Outcomes, Scheme, Adopted Syllabus, PEOs, POs,PSOs Pre-requisite, Vision & Mission of Institute and Department			25/01/20	25/01/20	
2.	1	Introduction of Translators, Types of Translators, Need of translators	[1,2]	CO1	25/01/20	25/01/20	
3.	1	Different phases of Compilation process	[1,2]	CO1	27/01/20	27/01/20 28/01/20	
4.	1	Passes and Bootstrapping	[2]	CO1	28/01/20	01/02/20	
5.	1	Finite state machines and regular expressions and their applications to lexical analysis'	[1,2]	CO1	01/02/20	03/02/20	
6.	2	Thompson Construction Algorithm and Subset Construction Algorithm	[2]	CO1	03/02/20 04/02/20	04/02/20 08/02/20	
7.	1	Optimization of DFA-Based Pattern Matchers implementation of lexical analyzers, lexical-analyzer generator, LEX compiler	[1,2]	CO1	04/02/20	10/02/20	
8.	1	Formal grammars and their application to syntax analysis, BNF notation, ambiguity, YACC	[1,2]	CO1	08/02/20	11/02/20	
9.	1	The syntactic specification of programming languages: Context free grammars, derivation and parse trees, capabilities of CFG	[1,2]	CO1	10/02/20	15/02/20	
10.	1	Introduction to Parsers and different types of parsers	[1,2]	CO2	10/02/20	15/02/20	
11.	1	Shift Reduce parsers	[1,2]	CO2	13/02/20	17/02/20	
12.	2	Operator precedence parsing	[1,2]	CO2	15/02/20 17/02/20	18/02/20 21/02/20	
13.	1	Top down parsing	[1,2]	CO2	18/02/20	02/03/20	
14.	2	Predictive parsers	[1,2]	CO2	22/02/20 24/02/20	03/03/20 05/03/20	
15.	1	Automatic Construction of efficient Parsers: LR parsers, the canonical Collection of LR(0) items, constructing SLR parsing tables	[1,2]	CO2	02/03/20	14/03/20	

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

16.	1	Constructing Canonical LR parsing tables	[1,2]	CO2	03/03/20	16/03/20	
17.	1	Constructing LALR parsing tables	[1,2]	CO2	03/03/20	23/03/20	
18.	1	Using ambiguous grammars, an automatic parser generator, implementation of LR parsing tables	[1,2]	CO2	07/03/20	24/03/20	
19	1	Syntax-directed Translation schemes Implementation of Syntax-directed Translators	[2]	CO3	14/03/20	25/03/20	
20	1	Intermediate code, postfix notation, Parse trees & syntax trees	[1,2]	CO3	16/03/20	30/03/20 31/03/20	
21	1	Three address code, quadruple & triples and indirect triples	[2]	CO3	17/03/20	01/04/20	
22	1	Translation of assignment statements, Boolean expressions	[1,2]	CO3	21/03/20	07/04/20 08/04/20	
23	1	Statements that alter the flow of control	[3]	CO3	23/03/20	20/04/20	
24	1	Postfix translation, translation with a top down parser	[1,2]	CO3	24/03/20	21/04/20	
25	1	More about translation: Array references in arithmetic expressions	[1,2]	CO3	28/03/20	22/04/20	
26	1	Procedures call, declarations and case statements	[1,2]	CO3	30/03/20	27/04/20 28/04/20	
27	1	Backpatching	[2]	CO3	30/03/20	29/04/20	
28	1	Data structure for symbols tables	[1,2]	CO4	04/04/20	04/05/20	
29	1	Representing scope information	[1,2]	CO4	04/04/20	05/05/20	
30	1	Run-Time Administration: Implementation of simple stack allocation scheme	[1,2]	CO4	07/04/20	06/05/20	
31	1	Storage allocation in block structured language	[1,2]	CO4	07/04/20	11/05/20	
32	2	Error Detection & Recovery: Lexical Phase errors, syntactic phase errors semantic errors	[1,2]	CO5	11/04/20 18/04/20	12/05/20 13/05/20	
33	1	Design Issues, the Target Language. Addresses in the Target Code	[1]	CO5	20/04/20	18/05/20	
34	1	Basic Blocks and Flow Graphs Optimization of Basic Blocks	[1],[2],[4]	CO5	21/04/20	19/05/20 20/05/20	
35	1	Code Generator. Code optimization: Machine-Independent Optimizations, Loop optimization	[1]	CO5	25/04/20	25/05/20 26/05/20	
36	1	Peephole Optimization ; DAG representation of basic blocks	[1]	CO5	27/04/20	27/05/20 01/06/20	
37	1	Value numbers and algebraic laws ; Global Data Flow Analysis	[1], [4]	CO5	27/04/20	02/06/20 03/06/20	

Reference Books: .1. Aho, Sethi & Ullman, "Compilers: Principles, Techniques and Tools", Pearson Education
2. Aho, Ullman, " Principles of Compiler Design"
3. https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/IntermediateCode.html/node4.html
4. https://nptel.ac.in/content/storage2/courses/106108113/module7/Lecture24.pdf

Name & Sign. of Faculty **Sign. of Reviewer** **Sign. of HOD**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Lecture Plan**	SESSION-2019-2020 SEM- 6th

SECTION C **Total Lectures: 40**

Sr. No.	No. of Lectures	Topics/Sub Topics	Reference Books	CO Covered	Planned Date	Coverage Date	Sign
1.	1	Introduction to Course Educational Objective, Course Outcomes, Scheme, Adopted Syllabus, PEOs, POs,PSOs Pre-requisite, Vision & Mission of Institute and Department			23/01/20	23/01/20	
2.	1	Introduction of Translators, Types of Translators, Need of translators	[1,2]	CO1	23/01/20	23/01/20	
3.	1	Different phases of Compilation process	[1,2]	CO1	24/01/20	24/01/20 28/01/20	
4.	1	Passes and Bootstrapping	[2]	CO1	28/01/20	30/01/20	
5.	1	Finite state machines and regular expressions and their applications to lexical analysis	[1,2]	CO1	30/01/20	31/01/20	
6.	2	Thompson Construction Algorithm and Subset Construction Algorithm	[2]	CO1	31/01/20 04/02/20	04/02/20 06/02/20	
7.	1	Optimization of DFA-Based Pattern Matchers implementation of lexical analyzers, lexical-analyzer generator, LEX compiler	[1,2]	CO1	04/02/20	07/02/20	
8.	1	Formal grammars and their application to syntax analysis, BNF notation, ambiguity, YACC	[1,2]	CO1	06/02/20	11/02/20	
9.	1	The syntactic specification of programming languages: Context free grammars, derivation and parse trees, capabilities of CFG	[1,2]	CO1	07/02/20	13/02/20	
10.	1	Introduction to Parsers and different types of parsers	[1,2]	CO2	07/02/20	14/02/20	
11.	1	Shift Reduce parsers	[1,2]	CO2	11/02/20	18/02/20	
12.	2	Operator precedence parsing	[1,2]	CO2	13/02/20 14/02/20	20/02/20 27/02/20	
13.	1	Top down parsing	[1,2]	CO2	18/02/20	28/02/20	
14.	2	Predictive parsers	[1,2]	CO2	20/02/20 21/02/20	03/03/20 05/03/20	
15.	1	Automatic Construction of efficient Parsers: LR parsers, the canonical Collection of LR(0) items, constructing SLR parsing tables	[1,2]	CO2	28/02/20	06/03/20	

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

16.	1	Constructing Canonical LR parsing tables	[1,2]	CO2	03/03/20	12/03/20	
17.	1	Constructing LALR parsing tables	[1,2]	CO2	03/03/20	12/03/20	
18.	1	Using ambiguous grammars, an automatic parser generator, implementation of LR parsing tables	[1,2]	CO2	05/03/20	23/03/20 13/03/20 24/03/20	
19	1	Syntax-directed Translation schemes Implementation of Syntax-directed Translators	[2]	CO3	06/03/20	25/03/20	
20	1	Intermediate code, postfix notation, Parse trees & syntax trees	[1,2]	CO3	12/03/20	30/03/20 31/03/20	
21	1	Three address code, quadruple & triples and indirect triples	[2]	CO3	13/03/20	01/04/20	
22	1	Translation of assignment statements, Boolean expressions	[1,2]	CO3	17/03/20	07/04/20 08/04/20	
23	1	Statements that alter the flow of control	[3]	CO3	19/03/20	20/04/20	
24	1	Postfix translation, translation with a top down parser	[1,2]	CO3	20/03/20	21/04/20	
25	1	More about translation: Array references in arithmetic expressions	[1,2]	CO3	24/03/20	22/04/20	
26	1	Procedures call, declarations and case statements	[1,2]	CO3	26/03/20	27/04/20 28/04/20	
27	1	Backpatching	[2]	CO3	26/03/20	29/04/20	
28	1	Data structure for symbols tables	[1,2]	CO4	27/03/20	04/05/20	
29	1	Representing scope information	[1,2]	CO4	27/03/20	05/05/20	
30	1	Run-Time Administration: Implementation of simple stack allocation scheme	[1,2]	CO4	31/03/20	06/05/20	
31	1	Storage allocation in block structured language	[1,2]	CO4	03/04/20	11/05/20	
32	2	Error Detection & Recovery: Lexical Phase errors, syntactic phase errors semantic errors	[1,2]	CO5	16/04/20 17/04/20	12/05/20 13/05/20	
33	1	Design Issues, the Target Language. Addresses in the Target Code	[1]	CO5	21/04/20	18/05/20	
34	1	Basic Blocks and Flow Graphs Optimization of Basic Blocks	[1],[2],[4]	CO5	21/04/20	19/05/20 20/05/20	
35	1	Code Generator. Code optimization: Machine-Independent Optimizations, Loop optimization	[1]	CO5	23/04/20	25/05/20 26/05/20	
36	1	Peephole Optimization ; DAG representation of basic blocks	[1]	CO5	29/04/20	27/05/20 01/06/20	
37	1	Value numbers and algebraic laws ; Global Data Flow Analysis	[1], [4]	CO5	29/04/20	02/06/20 03/06/20	

Reference Books: 1. Aho, Sethi & Ullman, "Compilers: Principles, Techniques and Tools", Pearson Education
2. Aho, Ullman, " Principles of Compiler Design"
3. https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/IntermediateCode.html/node4.html
4. https://nptel.ac.in/content/storage2/courses/106108113/module7/Lecture24.pdf

Name & Sign. of Faculty **Sign. of Reviewer** **Sign. of HOD**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Tutorial-1**	SESSION-2019-2020 SEM-6th

Tutorial 1 [CO - 1]

Sr. No.	No. of Periods	Topics/ Sub topics	Coverage Date		Sign
			Section B	Section C	
1.	1	Introduction to Translator, Phases of Compilation	24/01/20; 30/01/20 28/01/20 27/01/20 03/02/20	24/01/20; 28/01/20 28/01/20 30/01/20	

Q1. What is a Translator? Why are Translators needed?

Q2. What do you understand by Pass? Discuss merits and demerits of multi pass and single pass compiler?

Q3. Explain all the necessary phases in a compiler design?

Q4. What do you understand by Preliminary Scanning?

Q5. Explain Loader & Link Editor.

Q6. Write a short note on Context Free Grammar. Give examples of CFG.

Q7. Write a short note on Parse Trees. Give an example of Parse Tree.

Q8. Create a Symbol Table for the following sentence:
IF(5 .EQ. MAX) GOTO 100

Q9. Apply all the phases of compiler to generate Assembly code for the following code of a source program:
Position := initial + rate * 60

Q10. Construct a parse tree for the following statement:
IF(MAX=5) then GOTO 100

Name & Sign. of Faculty **Sign. of Reviewer** **Sign. of HOD**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	Tutorial-2	SESSION-2019-2020 SEM- 6th

Tutorial 2 [CO -1]

Sr. No.	No. of Periods	Topics/Sub Topics	Coverage Date		Sign
			Section B	Section C	
1.	1	Revision of Theory of Automata topics prescribed in Syllabus	06/02/20 04/02/20 03/02/20 10/02/20	04/02/20 04/02/20 06/02/20	

Q1. Let G be a CFG for which the production rules are given as below:

S->aB | bA
A->a |aS|bAA
B->b|bS|aBB
Derive the string aaabbabbba & draw the parse tree for the derivation.

Q2. Describe the model for Finite Automata. What do you understand by Finite State Machines?

Q3. What are Regular Expressions. Write a R.E. for a language containing the strings of length two over $\sum=\{0,1\}$

Q4. Write R.E. to denote a language L over $\sum=\{a,b\}$ such that 3rd character from right end of the string is always "a" .

Q5. Give the algorithm for subset construction with example.

Q6. Write down the Thompson's Construction Algorithm with example.

Q7. Explain Cross Compiler. Suppose you have a working C compiler on machine A. Discuss the steps you would take to create a working compiler for another language C' on machine B.

Q8. What do you understand by Lexical Analyzer Generator & LEX Compiler?

Q9. What are the issues of the Lexical Analyzer?

Q10. Show that the following grammar is ambiguous:

S-> aSbS
S-> bSaS
S-> €

Name & Sign. of Faculty **Sign. of Reviewer** **Sign. of HOD**

Dr. Sanjeev Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Tutorial-3**	SESSION-2019-2020 SEM- 6th

Tutorial 3 [CO -2]

Sr. No.	No. of Periods	Topics/Sub Topics	Coverage Date		Sign
			Section B	Section C	
1.	2	Parsing Techniques	13/02/20; 20/02/20 11/02/20; 18/02/20 17/02/20 02/03/20	11/02/20; 18/02/20 11/02/20; 03/03/20 13/02/20 20/02/20	

Q1. What do you mean by Shift Reduce Parsing? Explain with example the stack implementation of shift-reduce parsing.

Q2. What is Handle-Pruning? What are the various possible actions a shift-reduce parser can make? Discuss with example.

Q3. Write an algorithm for computing Leading & Trailing.

Q4. What do you mean by operator precedence parsing? Give an algorithm for computing operator precedence relations.

Q5. What is the difference between operator precedence parsing & Recursive Descent Parsing.

Q6. Write down the rules for eliminating Left-Recursion & Left-Factoring. Give an example of each.

Q7. What are the rules for computing FIRST & FOLLOW. Give an example.

Q8. How to construct a Predictive Parsing Table?

Q9. What steps are followed for the stack implementation by Predictive Parser.

Q10. Draw & Explain the model of a Predictive Parser.

Name & Sign. of Faculty **Sign. of Reviewer** **Sign. of HOD**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Tutorial-4**	SESSION-2019-2020 SEM- 6th

Tutorial 4 [CO -2]

Sr. No.	No. of Periods	Topics/Sub Topics	Coverage Date		Sign
			Section B	Section C	
1.	2	Numericals on Parsing Techniques	27/02/20; 05/03/20 03/03/20 26/03/20 16/03/20 26/03/20	03/03/20 26/03/20 26/03/20 27/02/20 05/03/20	

Q1. Consider the following grammar & test whether it is LL(1) or not?
S → AaAb | BbBa , A → € , B → €

Q2. Given grammar is LL(1) or not?
S →1AB |€ , A → 1AC |0C , B→ 0S , C→1

Q3. Compute FIRST & FOLLOW for the following grammar:
S→ A , A→aB |Ad , B→ bBC |F , C→ g

Q4. Construct M-table for the grammar given below:
S→ aBDh , B→ cC , C→ bC |€ , D→EF , E→g |€ , F→f |€

Q5. Is the given grammar LL(1)???
E→TE' E'→+E |€ T→FT' T'→T|€ F→PF'

F'→*F'|€ P→(E) |a|b|€

Q6. Discuss the architecture for LR-Parsers.

Q7. Give the algorithm for computing CLOSURE & GOTO function.

Q8. Discuss the steps for constructing SLR Parser.

Q9. Give an algorithm for constructing SLR Parsing Table.

Q10. Write an algorithm for parsing the input using SLR parsing table.

Name & Sign. of Faculty **Sign. of Reviewer** **Sign. of HOD**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Tutorial-5**	SESSION-2019-2020 SEM- 6th

Tutorial 5 [CO -3]

Sr. No.	No. of Periods	Topics/Sub Topics	Coverage Date		Sign
			Section B	Section C	
1.	1	Syntax Directed Translation Scheme	27/03/20 28/03/20	27/03/20 28/03/20	

Q1. Write a short note on:-

- SDT
- Synthesized & Inherited Attributes
- Synthesized & Inherited Translations

Q2. List & Explain the three kinds of Intermediate Representation.

Q3. What is an Abstract or Syntax Tree? Draw the Syntax Tree & DAG for the statement: a:= b*-c+b*-c

Q4. What are the various methods of implementing three address statements. Translate the expression into those methods: -(a+b)*(c+d)+(a+b+c)

Q5. Define three address code. Why "Three address code" is named so. List the types of three address statements.

Q6. Consider the following Syntax Directed Translation Scheme (SDTS), with non-terminals {S, A} and terminals {a, b}}.

S → aA {print 1}
S → a {print 2}
A → Sb {print 3}

Using the above SDTS, what will the output be printed by a bottom-up parser, for the input aab?

Q7. Give the SDD for if-else statement.

Q8. What is meant by short circuit or jumping code? Translate the following statements into three address code:-

(i) a or b and not c

(ii) a<b

Name & Sign. of Faculty **Sign. of Reviewer** **Sign. of HOD**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Tutorial-6**	SESSION-2019-2020 SEM- 6th

Tutorial 6 [CO -3]

Sr. No.	No. of Periods	Topics/Sub Topics	Coverage Date		Sign
			Section B	Section C	
1.	1	Translation of Boolean Expressions, Array References, procedure calls and switch case	2/4/20 04/04/20 10/04/20	3/4/20 04/04/20 10/04/20	

Q1. Let A be a 10X20 array with low1=low2=1. Therefore n1=10 & n2=20.Take w=4.Translate the assignment statement x:=A[y,z] into three address statements.

Q2. Define Boolean Expressions. What are the two primary purpose of Boolean Expressions? What are the two methods to represent the value of a Boolean Expression?

Q3. Define Backpatching. Explain the following three functions:-

makelist(i) merge(p1,p2) backpatch(p,i)

Q4. Write semantic actions for procedure calls for translation into intermediate code.

Q5. Write semantic actions for switch case statements.

Q6. Define: L-attributed grammar in detail.

Name & Sign. of Faculty **Sign. of Reviewer** **Sign. of HOD**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Tutorial-7**	SESSION-2019-2020 SEM- 6th

Tutorial 7 [CO -4]

Sr. No.	No. of Periods	Topics/Sub Topics	Coverage Date		Sign
			Section B	Section C	
1.	1	Symbol Table and Storage Management	07/05/20 08/05/20	07/05/20 08/05/20	

Q1. Write a short note on Symbol Table.

Q2. What are the contents of a symbol table?

Q3. What are the capabilities & requirements of a symbol table?

Q4. List & Explain the various data structures needed for symbol table.

Q5. Describe various ways of representing scope information.

Q6. Explain storage management schemes.

Q7. What do you mean by Activation Records?

Q8. Describe various Parameter Passing Techniques.

Q9. What are Dangling Pointers? Explain its use in Compiler in detail.

Q10. What are the limitations of stack allocation?

Name & Sign. of Faculty **Sign. of Reviewer** **Sign. of HOD**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Tutorial-8**	SESSION-2019-2020 SEM- 6th

Tutorial 8 [CO -4]

Sr. No.	No. of Periods	Topics/Sub Topics	Coverage Date		Sign
			Section B	Section C	
1.	1	Error Detection and Recovery	09/05/20 15/05/20	09/05/20 15/05/20	

Q1. What do you mean by Errors & what are the various sources of errors?

Q2.Classify and Describe Error Classification.

Q3. What are different types of errors detected in different phases of compilation?

Q4. Discuss error recovery techniques used by a compiler.

Q5. Write short note on Panic Recovery Mode .

Name & Sign. of Faculty **Sign. of Reviewer** **Sign. of HOD**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Tutorial-9**	SESSION-2019-2020 SEM- 6th

Tutorial 9 [CO -5]

Sr. No.	No. of Periods	Topics/Sub Topics	Coverage Date		Sign
			Section B	Section C	
1.	1	Introduction to Code Optimization Techniques	21/05/21 22/05/21 23/05/21	21/05/21 22/05/21 23/05/21	

Q1. What are principal sources of optimization?

Q2. What are basic blocks? Write algorithm for constructing Basic Blocks.

Q3. What is code motion?

Q4. What is DAG? Mention its applications.

Q5. Mention the issues to be considered while applying the techniques for code optimization.

Q6. What do you mean by machine dependent and machine independent optimization?

Q7. Write short note on :

Loop Optimization

Constant folding

Loop Unrolling

Q8 . What is peephole optimization?

Name & Sign. of Faculty **Sign. of Reviewer** **Sign. of HOD**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Tutorial-10**	SESSION-2019-2020 SEM- 6th

Tutorial 10 [CO -5]

Sr. No.	No. of Periods	Topics/Sub Topics	Coverage Date		Sign
			Section B	Section C	
1.	1	Numericals on Directed Acyclic Graphs	28/05/20 29/05/20 30/05/20	28/05/20 29/05/20 30/05/20	

Q1.Construct the DAG and identify the value numbers for the subexpressions of the following expressions, assuming ++ associates from the left.

a. a+b+(a+b)a+b+(a+b)
b. a+b+a+ba+b+a+b
c. a+a+((a+a+a+(a+a+a+a))a+a+((a+a+a+(a+a+a+a))

Q2. Construct the DAG for the expression
((x+y)−((x+y)*(x−y)))+((x+y)*(x−y))

Q3. Consider the following block and construct a DAG for it using value numbering method

d.
e. (1) a = b x c
f. (2) d = b
g. (3)'e = d x c
h. (4) b = e
i. (5) f = b + c
j. (6) g = f + d

Q4. What do you mean by dominators?

Q5. Consider the basic block given below.

a = b + c
c = a + d
d = b + c
e = d - b
a = e + b

Construct DAG for the above basic block and find dominators for each node.

Q6. What are the number of non terminal nodes in DAG of a = (b+c)*(b+c) expression?

Name & Sign. of Faculty **Sign. of Reviewer** **Sign. of HOD**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	ASSIGNMENT - 1	SESSION-2019-2020 SEM- 6th

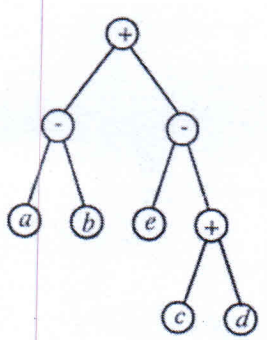
Home Assignments

Unit 1[CO- 1]

Q1 : Find the number of tokens in the following C statement :

```
printf("i=%d, &i=%x", i, &i);
```

Q2: Consider evaluating the following expression tree on a machine with load-store architecture in which memory can be accessed only through load and store instructions. The variables a,b,c,d and e are initially stored in memory. The binary operators used in this expression tree can be evaluated by the machine only when operands are in registers. The instructions produce result only in a register. If no intermediate results can be stored in memory, what is the minimum number of registers needed to evaluate this expression?



Q3 : The number of tokens in the following C code segment is :

```
1. switch(inputvalue)
2. {
3.     case 1 : b =c*d; break;
4.     default : b =b++; break;
5. }
```

Q4 : Considering a two pass compiler, what will be the sequence of the pass numbers for each of the following activities :

i. object code generation
ii. literals added to literal table
iii. listing printed
iv. address resolution of local symbols that occur in a two pass assembler

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**ASSIGNMENT - 2**	SESSION-2019-20 SEM- 6th

Unit 2 [CO- 2]

Q1: Which one of the following grammars is free from left recursion?

A. S→AB
   A→Aa|b
   B→c
B. S→Ab|Bb|c
   A→Bd|ϵ
   B→e
C. S→Aa|B
   A→Bb|Sc|ϵ
   B→d
D. S→Aa|Bb|c
   A→Bd|ϵ
   B→Ae|ϵ

Q2 : The following grammar is clearly not LL(1), how would you transform the grammar to make it LL(1)?
A -> A + B | A - B | B
B -> C * B | C / B | C
C -> (A) | int

Q3 : Consider the following grammar with terminals [,], a, b, c, +, and -:

1 $S \rightarrow [\,S\,X\,]$
2 $\quad\quad | \; a$
3 $X \rightarrow \varepsilon$
4 $\quad\quad | + S\,Y$
5 $\quad\quad | \; Y\,b$
6 $Y \rightarrow \varepsilon$
7 $\quad\quad | - S\,X\,c$

Compute FIRST and FOLLOW sets for the non terminals S,X and Y .

Q4: Considering grammar given in Q3, construct top down parsing table and parse the input string : [a+a-ac]

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	ASSIGNMENT - 3	SESSION-2019-20 SEM- 6th

Unit 3 [CO- 3]

The following context-free grammar describes part of the syntax of a simple programming language. Nonterminals are given in capitals and terminals in lower case. VAR represents a variable name and CONST represents a constant.

1 PROGRAM → Procedure STMT-LIST

2 STMT-LIST → STMT STMT-LIST

3 | STMT

5 STMT → do VAR = CONST to CONST { STMT-LIST }

5 | ASSN-STMT

Q1 : Show the parse tree for the following:

Procedure
do i = 1 to 100 {
ASSN-STMT
ASSN-STMT
}
ASSN-STMT

Q2 : Create attribute(s) and add semantic functions to the above grammar that compute the number of executed statements for a program conforming to this grammar.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**ASSIGNMENT - 4**	SESSION-2019-20 SEM- 6th

Unit 4 [CO- 4]

Consider the following C program :

```
# include <stdio.h>
main ( )
   {
      int x = 10, y = 12;
      char * a;
      a = &x ;
      x = 1xab;
      printf ("%d %d", x, * a);
   }
```

Q1 : Which of the following type of error(earliest phase) is identified during compilation of above program?

Q2 : Write the comparison among Static allocation, Stack allocation and Heap Allocation with their merits and limitations.

Q3: Define activation records. Explain how it is related with runtime storage allocation.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	ASSIGNMENT - 5	SESSION-2019-20 SEM- 6th

Unit 5 [CO- 5]

Q1 : Consider the following code which computes the inner product of 2 vectors:

```
prod := 0;
i := 1;
repeat {
    prod := prod + a[i] * b[i]
    i = i+ 1;
 until i > 20
}
```

i. Construct the IR for above program
ii. Create Basic Blocks and the Control Flow Graph
iii. Show any optimizations if possible
iv. Perform Global Data Flow Analysis on above code

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

List of Topics assigned to students for Presentation

Section B

S.No.	Roll No.	Name of Students	Topic
1.	1708210063	LALIT KUMAR	Representing Scope Information
2.	1708210064	MANAS ARORA	Data Structure for Symbol Table
3.	1708210065	MANU PANWAR	Phases of Compilation process
4.	1708210066	MAYANK BHATNAGAR	Data Structure for Symbol Table
5.	1708210067	MAYANK UPADHYAY	Phases of Compilation process
6.	1708210068	MOHHAMMAD AMAAN	Storage Allocation in Block Structured Languages
7.	1708210069	MOHAMMAD ANAS	Phases of Compilation process
8.	1708210070	MOHD. AFZAL	LEX and YACC
9.	1708210071	MOHD. AKIF	LEX and YACC
10.	1708210072	MOHD. ANAS	Storage Allocation in Block Structured Languages
11.	1708210073	MOHD. ASHIR	Storage Allocation in Block Structured Languages
12.	1708210074	MOHD. FARDEEN	Representing Scope Information
13.	1708210075	MOHD. HARIS	LEX and YACC
14.	1708210076	MOHD. SADIQ	Didn't Submit Topic
15.	1708210077	MOHD. ASIF	Didn't Submit Topic
16.	1708210078	MOHD. SHOAIB	Representing Scope Information
17.	1708210079	MOHD. SUHAIL	Data Structure for Symbol Table
18.	1708210080	MOHIT AGARWAL	Storage Allocation in Block Structured Languages
19.	1708210081	MUDIT KUMAR SHARMA	LEX and YACC
20.	1708210082	MUKESH KUMAR	LEX and YACC
21.	1708210083	MUKUL KUMAR	Error Detection and Recovery
22.	1708210084	MUSKAN MEHROTRA	Representing Scope Information
23.	1708210085	MUSKAN AGARWAL	Backpatching
24.	1708210086	NAMAN AGARWAL	Data Structure for Symbol Table
25.	1708210087	NANDITA GAURI	Error Detection and Recovery
26.	1708210090	NIKITA SINGH	Error Detection and Recovery
27.	1708210091	NIRBHAY PAL	Comparison between parsing techniques
28.	1708210092	NISHITA AGARWAL	Representing Scope Information

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

29.	1708210093	NITESH SAINI	Comparison between parsing techniques
30.	1708210094	NITIKA RASTOGI	Error Detection and Recovery
31.	1708210096	NITIN CHAUHAN	Error Detection and Recovery
32.	1708210097	NITIN VERMA	Representing Scope Information
33.	1708210098	NIVESH KUMAR	Error Detection and Recovery
34.	1708210099	NUPUR GUPTA	Implementation of Simple Stack Allocation Scheme
35.	1708210100	PALAK GOEL	Implementation of Simple Stack Allocation Scheme
36.	1708210101	PALAK RASTOGI	Implementation of Simple Stack Allocation Scheme
37.	1708210102	PARAS VISHNOI	Data Structure for Symbol Table
38.	1708210103	PARTH SHARMA	Data Structure for Symbol Table
39.	1708210104	PIYUSH DHAWAN	Data Structure for Symbol Table
40.	1708210105	PIYUSH SHARMA	Error Detection and Recovery
41.	1708210106	PRADEEP KUMAR	Comparison between parsing techniques
42.	1708210108	PRANVI JAIN	Representing Scope Information
43.	1708210109	PRASHANT KUMAR	Comparison between parsing techniques
44.	1708210110	PRATHAM MAHESHWARI	Implementation of Simple Stack Allocation Scheme
45.	1708210111	PRAYAG VERMA	Error Detection and Recovery
46.	1708210112	PRIYANK RAGHAV	Implementation of Simple Stack Allocation Scheme
47.	1708210113	PUSHKAR SHARMA	Error Detection and Recovery
48.	1708210114	RAGHAV AGARWAL	Error Detection and Recovery
49.	1708210115	RAHUL SUKHIJA	Data Structure for Symbol Table
50.	1708210116	RANOJIT MALIK	Implementation of Simple Stack Allocation Scheme
51.	1708210118	RAVI RANJAN	Representing Scope Information
52.	1708210120	RISHABH CHAUDHARY	Error Detection and Recovery
53.	1708210121	RISHABH KUMAR SHARMA	Error Detection and Recovery
54.	1708210122	RISHI RAJ SINGH	Data Structure for Symbol Table
55.	1708210123	RITIKA SAXENA	Representing Scope Information

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Section C

S.No.	Roll No.	Name of Students	Topic
1.	1708210124	RITVIK DAYAL	Data Structure for Symbol Table
2.	1708210125	RIYANSHI SINGHAL	Implementation of Simple Stack Allocation Scheme
3.	1708210126	ROHIT KUMAR SINGH	Implementation of Simple Stack Allocation Scheme
4.	1708210127	S. ALI ABID ABIDI	Phases of Compilation process
5.	1708210129	SAKIB	Representing Scope Information
6.	1708210130	SALONI BHATNAGAR	Implementation of Simple Stack Allocation Scheme
7.	1708210131	SANCHIT LAMBA	Data Structure for Symbol Table
8.	1708210132	SANPREET KAUR	Error Detection and Recovery
9.	1708210133	SATENDRA SAINI	Phases of Compilation process
10.	1708210134	SATISH SAINI	Data Structure for Symbol Table
11.	1708210135	SAURABH BHATNAGAR	Error Detection and Recovery
12.	1708210136	SAURABH SAINI	Phases of Compilation process
13.	1708210137	SHASHIWALA	Error Detection and Recovery
14.	1708210138	SHIKHAR RASTOGI	Phases of Compilation process
15.	1708210139	SHIVAM KHURANA	Representing Scope Information
16.	1708210140	SHIVANGI ARORA	Error Detection and Recovery
17.	1708210141	SHOBHIT RASTOGI	Implementation of Simple Stack Allocation Scheme
18.	1708210142	SHRUTI ARYA	Error Detection and Recovery
19.	1708210143	SHRUTI GUPTA	Data Structure for Symbol Table
20.	1708210144	SHUBHAM KHANNA	Representing Scope Information
21.	1708210145	SIDDHANT MISHRA	Implementation of Simple Stack Allocation Scheme
22.	1708210146	SIDDHARTH SINGH	Implementation of Simple Stack Allocation Scheme
23.	1708210147	SOHAIL KHAN	Error Detection and Recovery
24.	1708210149	SRASHTI GAUTAM	Representing Scope Information
25.	1708210150	SUMIT KUMAR	Error Detection and Recovery
26.	1708210152	SUMIT DEBNATH	Error Detection and Recovery
27.	1708210153	SUMIT KUMAR S/O Sri Naresh Pal	Storage Allocation in Block Structured Languages
28.	1708210154	SYED ASHRAF HUSSAIN	Implementation of Simple Stack Allocation Scheme
29.	1708210155	TANYA BHASIN	Backpatching

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

30.	1708210156	TANYA DUGGAL	Data Structure for Symbol Table
31.	1708210157	TASHA JOHARI	Data Structure for Symbol Table
32.	1708210158	TUSHAR AGARWAL	Storage Allocation in Block Structured Languages
33.	1708210159	UTKARSH VARSHNEY	Storage Allocation in Block Structured Languages
34.	1708210160	VAIBHAV CHODHARY	Implementation of Simple Stack Allocation Scheme
35.	1708210161	VAIBHAV CHAUHAN	Representing Scope Information
36.	1708210162	VEDIKA KHANNA	Representing Scope Information
37.	1708210163	VIBHOR AGARWAL	Data Structure for Symbol Table
38.	1708210164	VILA ZEHRA	Error Detection and Recovery
39.	1708210165	VINEET JOSHI	Implementation of Simple Stack Allocation Scheme
40.	1708210166	VIPIN KASHYAP	Comparison between parsing techniques
41.	1708210167	VISHAKHA RASTOGI	Error Detection and Recovery
42.	1708210168	VISHAKHA TANDON	Representing Scope Information
43.	1708210169	VISHAL KUMAR S/O Sri Bhudev Sharma	Data Structure for Symbol Table
44.	1708210170	VISHAL KUMAR S/O Sri Manoj Kumar	Implementation of Simple Stack Allocation Scheme
45.	1708210171	VRATIKA GUPTA	Representing Scope Information
46.	1708210172	YASH CHAUDHARY	Implementation of Simple Stack Allocation Scheme
47.	1708210173	ZAINAB AZEEM	Error Detection and Recovery
48.	1508210100	PRANAV BHATNAGAR	Data Structure for Symbol Table
49	1808210903	Manglam Sharma	Error Detection and Recovery
50	1808210902	Aryan Thakur	Error Detection and Recovery
51	1808210901	Aanchal Kumari	Representing Scope Information
52	1808210904	Shivam Kumar	Comparison between parsing techniques

Topics are assigned from Unit -4 and students having same topic can form a group, within their Section.

Those students who didn't submit their choice have been allotted topics from Unit- 1 and 2.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Assignments provided during Online classes

Details of Student

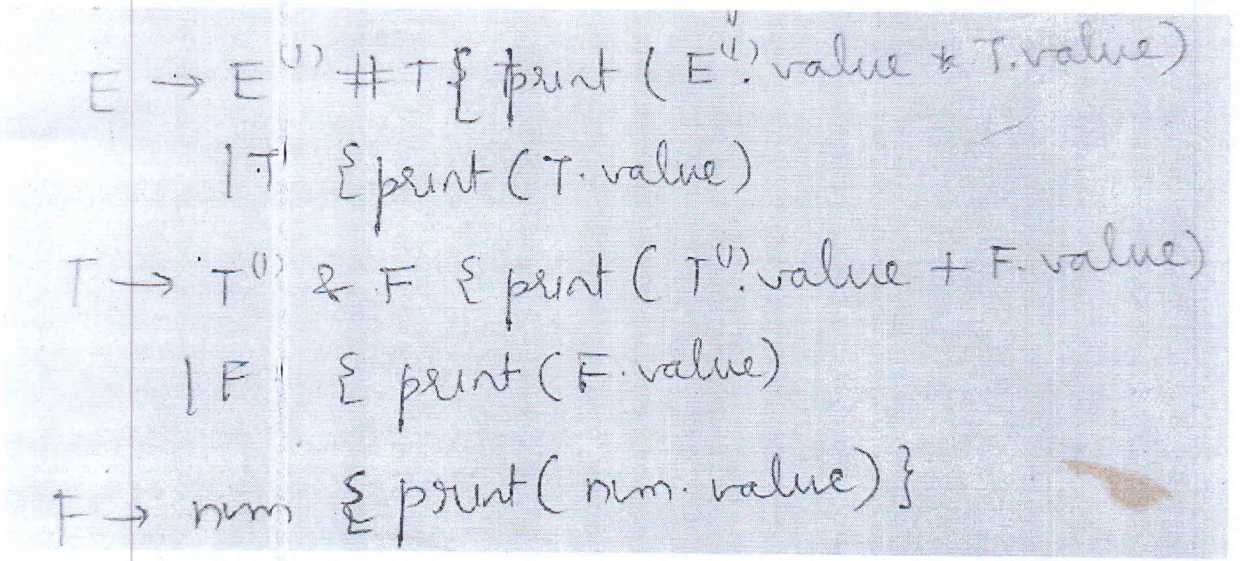
*Required

1. Name of Student *

2. University Roll No. *

Assignment Questions

3. Consider grammar with following translation rules and E as start symbol. Compute expression 2#3&5#3&6

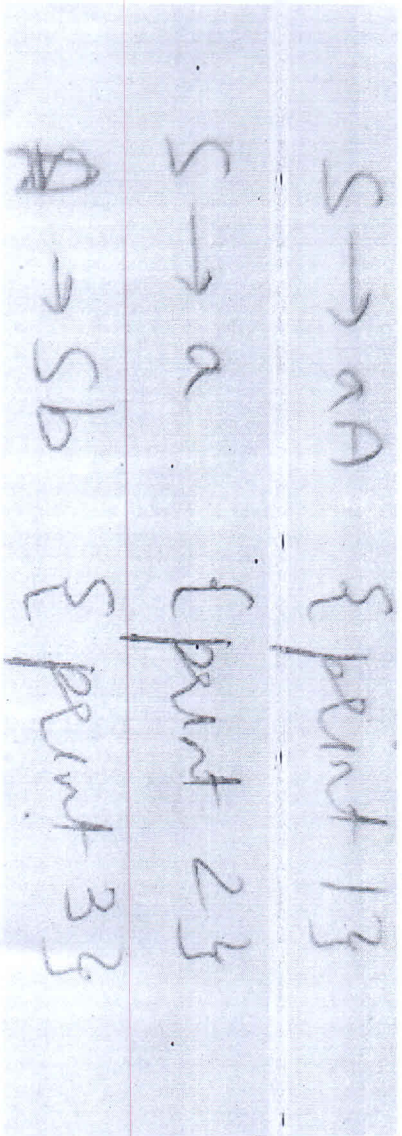


Mark only one oval.

- 98
- 52
- 224
- None of these

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

4. Consider following SDTS and using it, what will be the output printed by bottom up parser for the input aab is:



Mark only one oval.

- 1 3 2
- 2 3 1
- 2 2 3
- Syntax Error

5. Consider the expression : (a+b)*(c+d)-(a+b+c) and find minimum number of temporary variables used in generation of 3-address code:

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

6. In question no. 3, generate quadruple structure and find no. of temporary variables generated in it

7. In question no. 3, find number of pointer addresses created if triples/ indirect triples structure is used:

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Assignment from Unit 5

*Required

1. Email *

2. Name of student *

3. Roll No. *

Questions

Each question is of 5 marks

4. Generate the 3 -address code for following C program fragment assuming that 3 registers are available:

a) x = 1
b) x = y
c) x = x + 1
d) x = a + b * c
e) x = a/(b+c) - d*(e+f)

Files submitted:

5.

```
main()
{
    int i;
    int a[10];
    while (i <= 10)
        a[i] = 0;
```

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

6. Construct DAG for following code:

```
d := b * c
e := a + b
b := b * c
a := e - d
```

Files submitted:

7. The optimization technique which is typically applied on loops is

Mark only one oval.

- Removal of invariant computation
- Peephole Optimization
- Constant folding
- All of these

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Assignment from Unit 5

*Required

1. Email *

2. Name of Student *

3. Roll No. *

Numerical Question

4. Q1 :Consider following code of matrix multiplication, assuming that there are four bytes per word, solve (a) to (f) : a) Generate three address statements for above code; b) Generate target machine code from three address statements c) Construct flow graph from three address statements d) Eliminate common subexpressions from each basic block. e) Find the loops in the flow graph f)Find the induction variables of each loop and eliminate them wherever possible

```
begin
    for i := 1 to n do
        for j := 1 to n do
            c[i,j] := 0;
    for i := 1 to n do
        for j := 1 to n do
            for k := 1 to n do
                c[i,j] := c[i,j] + a[i,k] * b[k,j]
end
```

Files submitted:

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Revision on Parsing

Details of Students

*Required

1. Email *

2. Name of Student *

3. University Roll Number *

Questions

4. 2 points

In SLR parsing to get a shift-reduce conflict for state I on terminal symbol 'a',

a) A →α.β with First(β) containing 'a' should be in I
b) A →δ. be in I with Follow(A) having 'a'
c) A →α.β with First(β) containing 'a' should be in I and A →δ. be in I with Follow(A) having 'a'
d) None of the other options

Mark only one oval.

- a
- b
- c
- d

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

5. 2 points

Between SLR, Canonical LR and LALR, which have same number of states

a) SLR and LALR
b) SLR and canonical LR
c) Canonical LR and LALR
d) All of them

Mark only one oval.

- a
- b
- c
- d

2 points

For the grammar

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC \mid d$

In state 0 of LR(1) parser, an item included is

a) $C \rightarrow .d; c$
b) $C \rightarrow .d; d$
c) $C \rightarrow .d; c,d$
d) $C \rightarrow .d; c, \$$

Mark only one oval.

- a
- b
- c
- d

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

7. 2 points

Amount of look ahead in LALR parser is

a) 1
b) 2
c) 3
d) None

Mark only one oval.

- 1
- 2
- 3
- 4

8. 2 points

Construction of parsing table in which strategies do not need the Follow set?

a) SLR and canonical LR
b) Canonical LR and LALR
c) SLR and LALR
d) None of the given options

Mark only one oval.

- a
- b
- c
- d

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**List of Students**	SESSION-2019-2020 SEM- 6th

SECTION B

S.No.	Student No	Roll No.	Name of Students	
1.	1710256	1708210063	LALIT KUMAR	FW
2.	1710060	1708210064	MANAS ARORA	
3.	1710021	1708210065	MANU PANWAR	
4.	1710276	1708210066	MAYANK BHATNAGAR	
5.	1710085	1708210067	MAYANK UPADHYAY	
6.	1710046	1708210068	MOHHAMMAD AMAAN	
7.	1710302	1708210069	MOHAMMAD ANAS	
8.	1710042	1708210070	MOHD. AFZAL	
9.	1710295	1708210071	MOHD. AKIF	
10.	1710283	1708210072	MOHD. ANAS	
11.	1710008	1708210073	MOHD. ASHIR	
12.	1710139	1708210074	MOHD. FARDEEN	
13.	1710054	1708210075	MOHD. HARIS	FW
14.	1710252	1708210076	MOHD. SADIQ	
15.	1710166	1708210077	MOHD. ASIF	
16.	1710214	1708210078	MOHD. SHOAIB	
17.	1710068	1708210079	MOHD. SUHAIL S/O Sri Zahid Ali	
18.	1710308	1708210080	MOHIT AGARWAL	
19.	1710020	1708210081	MUDIT KUMAR SHARMA	
20.	1710243	1708210082	MUKESH KUMAR	
21.	1710086	1708210083	MUKUL KUMAR	
22.	1710273	1708210084	MUSKAN MEHROTRA	
23.	1710333	1708210085	MUSKAN AGARWAL	
24.	1710248	1708210086	NAMAN AGARWAL	
25.	1710048	1708210087	NANDITA GAURI	
26.	1710186	1708210090	NIKITA SINGH	
27.	1710203	1708210091	NIRBHAY PAL	
28.	1710282	1708210092	NISHITA AGARWAL	
29.	1710320	1708210093	NITESH SAINI	
30.	1710152	1708210094	NITIKA RASTOGI	FW
31.	1710257	1708210096	NITIN CHAUHAN	
32.	1710209	1708210097	NITIN VERMA	
33.	1710195	1708210098	NIVESH KUMAR	
34.	1710210	1708210099	NUPUR GUPTA	
35.	1710144	1708210100	PALAK GOEL	
36.	1710018	1708210101	PALAK RASTOGI	
37.	1710096	1708210102	PARAS VISHNOI	
38.	1710029	1708210103	PARTH SHARMA	
39.	1710075	1708210104	PIYUSH DHAWAN	

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

40.	1710079	1708210105	PIYUSH SHARMA	
41.	1710212	1708210106	PRADEEP KUMAR S/O Sri Naresh Kumar	
42.	1710168	1708210108	PRANVI JAIN	
43.	1710313	1708210109	PRASHANT KUMAR	
44.	1710132	1708210110	PRATHAM MAHESHWARI	
45.	1710255	1708210111	PRAYAG VERMA	
46.	1710324	1708210112	PRIYANK RAGHAV	
47.	1710321	1708210113	PUSHKAR SHARMA	
48.	1710307	1708210114	RAGHAV AGARWAL	
49.	1710114	1708210115	RAHUL SUKHIJA	
50.	1710240	1708210116	RANOJIT MALIK	
51.	1710263	1708210118	RAVI RANJAN	
52.	1710285	1708210120	RISHABH CHAUDHARY	
53.	1710293	1708210121	RISHABH KUMAR SHARMA	
54.	1710134	1708210122	RISHI RAJ SINGH	
55.	1710005	1708210123	RITIKA SAXENA	

SECTION C

S.No.	Student No	Roll No.	Name of Students	
1.	1710222	1708210124	RITVIK DAYAL	
2.	1710043	1708210125	RIYANSHI SINGHAL	
3.	1710116	1708210126	ROHIT KUMAR SINGH	
4.	1710206	1708210127	S. ALI ABID ABIDI	
5.	1710318	1708210129	SAKIB	
6.	1710336	1708210130	SALONI BHATNAGAR	
7.	1710062	1708210131	SANCHIT LAMBA	
8.	1710314	1708210132	SANPREET KAUR	
9.	1710027	1708210133	SATENDRA SAINI	
10.	1710229	1708210134	SATISH SAINI	
11.	1710145	1708210135	SAURABH BHATNAGAR	
12.	1710142	1708210136	SAURABH SAINI	
13.	1710173	1708210137	SHASHIWALA	
14.	1710025	1708210138	SHIKHAR RASTOGI	
15.	1710002	1708210139	SHIVAM KHURANA	
16.	1710227	1708210140	SHIVANGI ARORA	
17.	1710175	1708210141	SHOBHIT RASTOGI	
18.	1710011	1708210142	SHRUTI ARYA	
19.	1710101	1708210143	SHRUTI GUPTA	
20.	1710038	1708210144	SHUBHAM KHANNA	
21.	1710066	1708210145	SIDDHANT MISHRA	
22.	1710036	1708210146	SIDDHARTH SINGH	
23.	1710246	1708210147	SOHAIL KHAN	
24.	1710016	1708210149	SRASHTI GAUTAM	
25.	1710235	1708210150	SUMIT KUMAR S/O Sri Latoor Singh	
26.	1710238	1708210152	SUMIT DEBNATH	

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

27.	1710249	1708210153	SUMIT KUMAR S/O Sri Naresh Pal	
28.	1710326	1708210154	SYED ASHRAF HUSSAIN	
29.	1710130	1708210155	TANYA BHASIN	
30.	1710128	1708210156	TANYA DUGGAL	
31.	1710105	1708210157	TASHA JOHARI	
32.	1710179	1708210158	TUSHAR AGARWAL	
33.	1710309	1708210159	UTKARSH VARSHNEY	
34.	1710323	1708210160	VAIBHAV CHODHARY	
35.	1710120	1708210161	VAIBHAV CHAUHAN	
36.	1710155	1708210162	VEDIKA KHANNA	
37.	1710031	1708210163	VIBHOR AGARWAL	
38.	1710192	1708210164	VILA ZEHRA	
39.	1710069	1708210165	VINEET JOSHI	
40.	1710291	1708210166	VIPIN KASHYAP	
41.	1710041	1708210167	VISHAKHA RASTOGI	
42.	1710241	1708210168	VISHAKHA TANDON	
43.	1710131	1708210169	VISHAL KUMAR S/O Sri Bhudev Sharma	FW
44.	1710055	1708210170	VISHAL KUMAR S/O Sri Manoj Kumar	FW
45.	1710237	1708210171	VRATIKA GUPTA	
46.	1710081	1708210172	YASH CHAUDHARY	
47.	1710102	1708210173	ZAINAB AZEEM	
48.	1510592	1508210100	PRANAV BHATNAGAR	

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Class Test Papers with Solution**	SESSION-2019-2020 SEM- 6th

Class Test 1 Question Paper

Q.NO.	1	2	3	4	5	6
CO	1	1	1	1	1	2

	Attempt all questions [For Sec A, B,C]	
	Section A	**Marks**
1.	Define boot-strapping with the help of an example.	**(2)**
2.	What is left Recursion? How to remove left recursion?	**(2)**
	Section B	
3.	What do you mean by ambiguous grammar? Show that the following grammar is ambiguous. S->aSbS|bSaS|∈	**(3)**
4.	Show the construction of NFA for following regular expression. (a|b)*a(a|b)(a|b)	**(3)**
	Section C	
5.	Explain in detail the process of compilation. Illustrate the output of each phase of compilation the input "a=(b+c)*(b+c)*2".	**(5)**
6	Explain non recursive predictive parsing (LL1). Consider the following grammar and construct the predictive parsing table E --> TE' E' --> +TE' | ∈ T --> FT' T' --> *FT' | ∈ F --> id | (E)	**(5)**

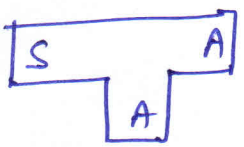
Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Class Test-1 Solution

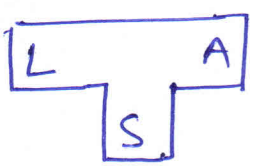
Ans 1 :- Bootstrapping is a technique that is used for creating compiler & to move them from one machine to another by modifying the backend.

e.g. If we want to create a new language L for machine A :-

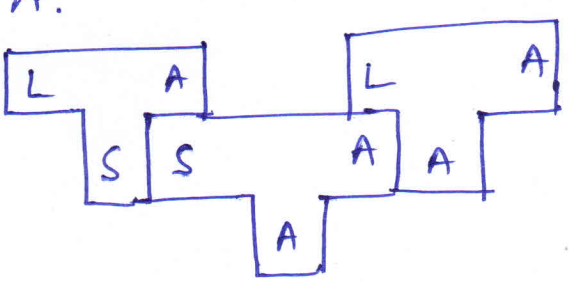
Step 1 :- create S^{A}_{CA} using language A, which runs on machine A :-



Step 2 :- Create L^{A}_{CS}, a compiler for language L, written in a subset of L :-



Step 3 :- Compile L^{A}_{CS} using S^{A}_{CA} to obtain LC^{A}_{A} ; a compiler for language L, which runs on machine A and produces code for machine A.



Ans 2 :- Left Recursion is an issue that occurs with top-down parsing. A grammar is left recursive if it has a non terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some α.

Left Recursion can cause a top down parser to go into infinite loop.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

How to eliminate left recursion

If we have a production $A \longrightarrow A\alpha \mid \beta$ such that β doesnot begin with A; then left recursion can be eliminated by replacing $A \longrightarrow A\alpha \mid \beta$

with —

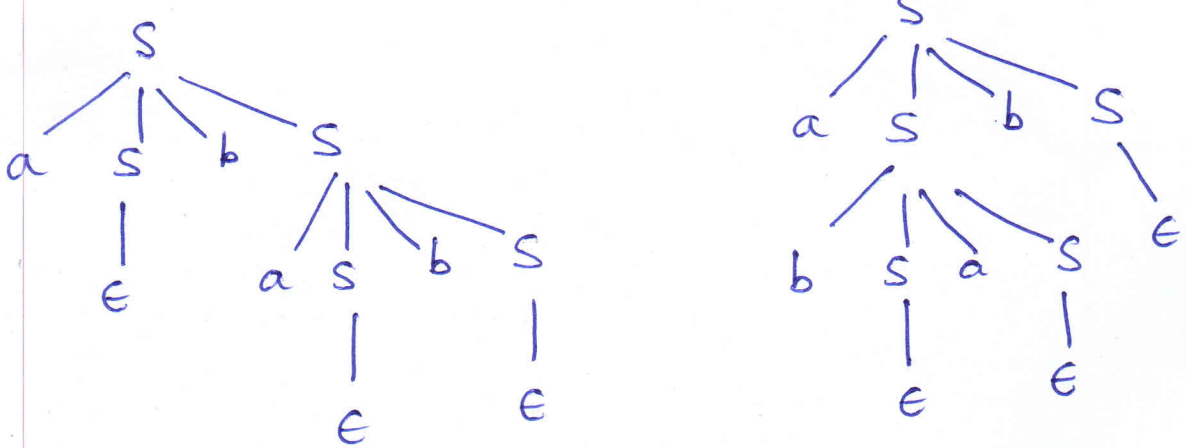
$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

Ans 3 :- An ambiguous grammar is the grammar which has more than one left-most or right most derivation trees.

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

Let $w = abab$

Then, we can have two leftmost derivation/parse trees as—



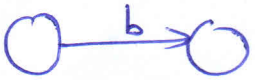
Thus, this grammar is ambiguous.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

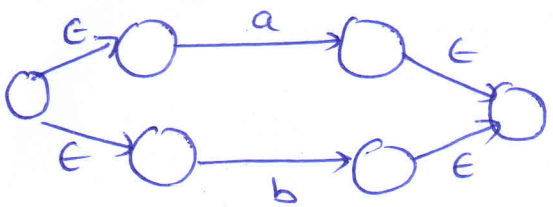
Ans 4 :- $(a|b)^* a(a|b)(a|b)$

Step1 :-

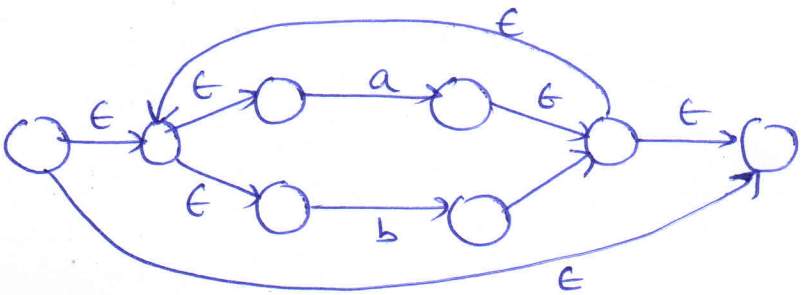
Step2 :-



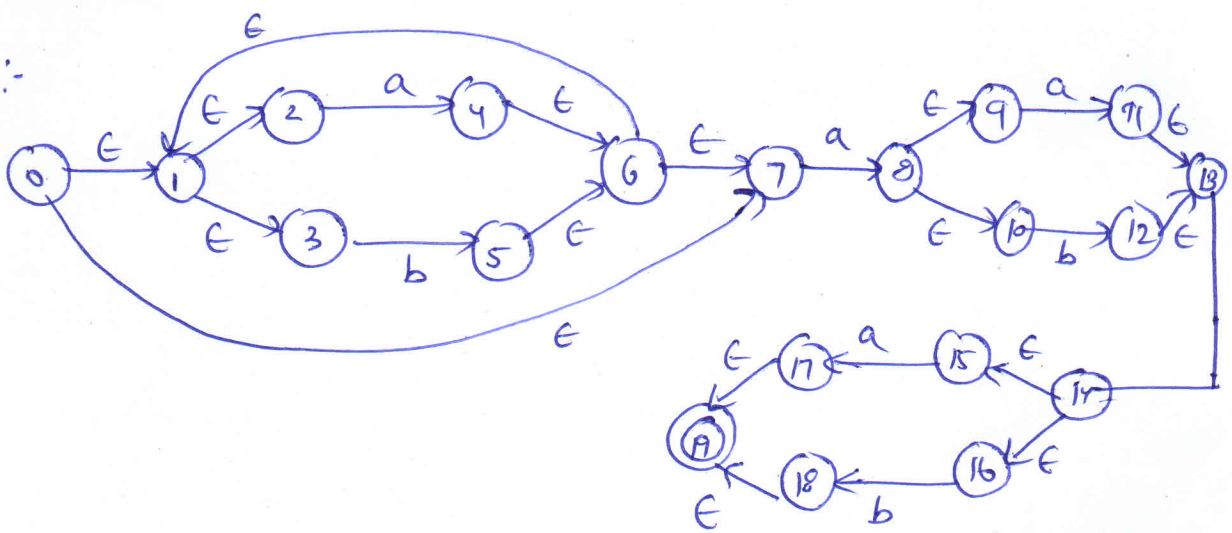
Step3 :-



Step4 :-



Step5 :-



Ans 5 :-

$a = (b+c)*(b+c)*2$

↓

Lexical Analysis

↓ Tokens

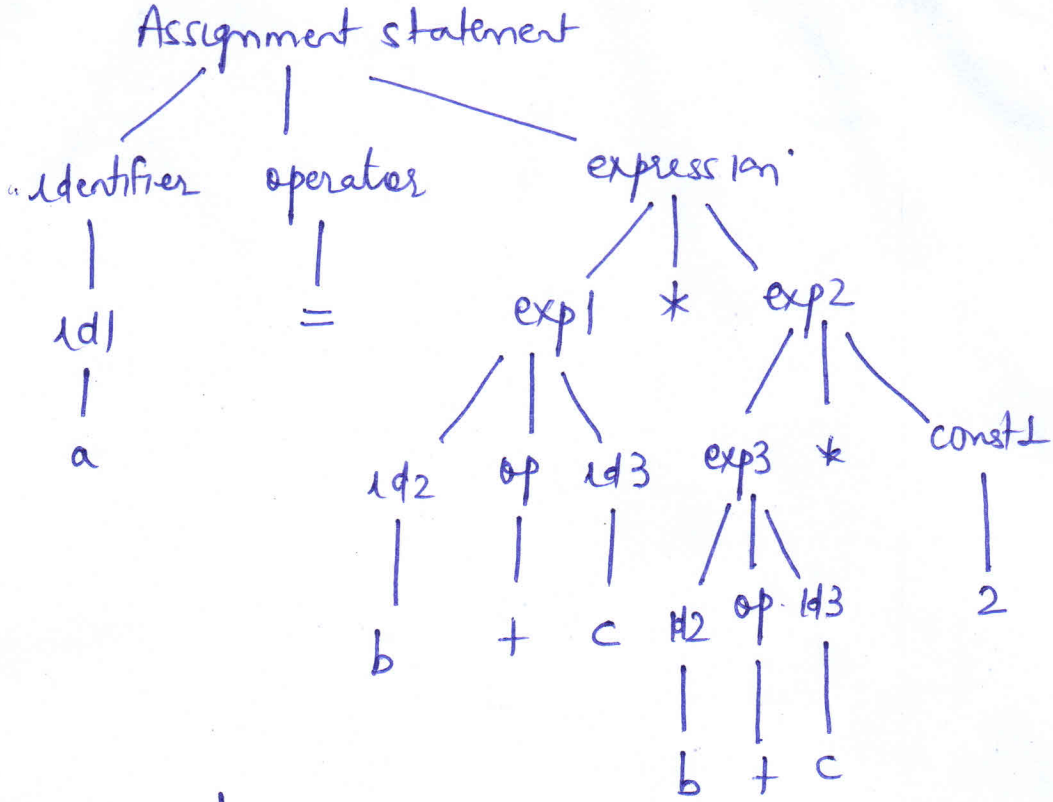
$id1 = (id2 + id3)*(id2+id3) * const1$

↓

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Syntax Analysis

↓ Parse tree



↓

Semantic Analysis

↓ Annotated syntax tree

Here, no change as we are assuming all variables & constants of same data type

↓

Intermediate Code generation

↓ 3-addr code

$t1 = b + c$

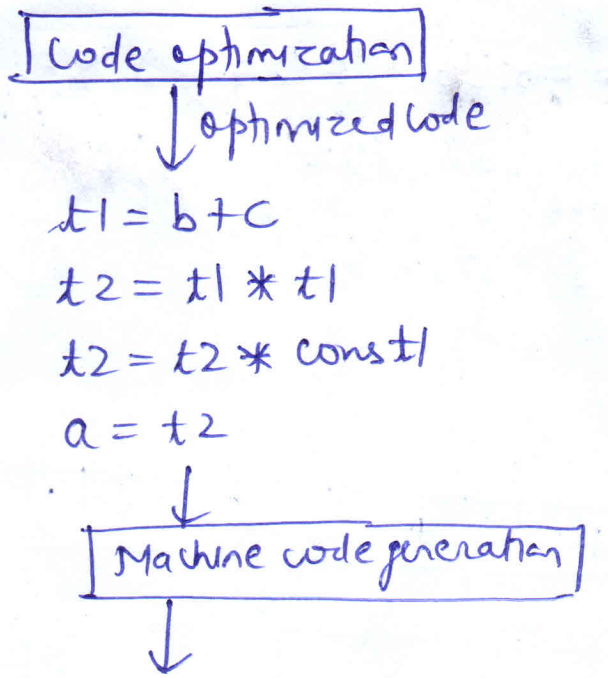
$t2 = t1 * t1$

$t3 = t2 * const1$

$a = t3$

↓

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001



Ans 6 :-

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

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

FIRST(E') = {+, ϵ}

FIRST(T') = {*, ϵ}

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

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

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

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

Parsing table :-

	id	+	*	(	)	$
E	E→TE'	error		E→TE'		
E'		E'→+TE'			E'→ϵ	E'→ϵ
T	T→FT'			T→FT'		
T'		T'→ϵ	T'→*FT'		T'→ϵ	T'→ϵ
F	F→id			F→(E)		

In Pursuit of Excellence	**Class Test Papers with Solution**	SESSION-2019-2020 SEM- 6th

Class Test 2 Question Paper

Q.NO.	1	2	3	4	5	6
CO	2	3	2	3	2	3

Attempt all questions [For Sec A, B,C]

	Section A	**Marks**
1.	Construct LR(0) Items for: S → AaAb | BbBa , A → € , B → €	(2)
2.	Define Backpatching. Explain the following three functions:- makelist(i) merge(p1,p2) backpatch(p,i)	(2)
	Section B	
3.	Consider the following grammar: E--> E+E E -> E* E E--> id Parse the string id+id*id using CLR parsing technique.	(3)
4.	What are the various methods of implementing three address statements. Translate the expression into those methods: -(a+b)*(c+d)+(a+b+c)	(3)
	Section C	
5.	Show the following grammar is LALR(1) but not SLR(1): S→Aa|bAc|dc|bca A→d	(5)
6	Give the Parse Tree and translation for the expression g=a+b-(c*d) according to the syntax directed translation scheme of assignment statement.	(5)

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

CLASS TEST 2 SOLUTION

1 - (2)
2 - (2)
3 - (3)
4 - (3)
5 - (5)
6 - (5)

Ans1 :-

$S' \rightarrow S$
$S \rightarrow AaAb \mid BbBa$
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$

I_0 : CLOSURE $(S' \rightarrow S)$

$S' \rightarrow .S$
$S \rightarrow .AaAb$
$S \rightarrow .BbBa$
$A \rightarrow .\epsilon$
$B \rightarrow .\epsilon$

GOTO(I_0, S)
I_1 : $S' \rightarrow S.$

I_2
GOTO(I_0, A)
$S \rightarrow A.aAb$

I_3:
GOTO(I_0, B)
$S \rightarrow B.bBa$

I_4:
GOTO(I_0, ϵ)
$A \rightarrow \epsilon.$
$B \rightarrow \epsilon.$

I_5:
GOTO(I_2, a)
$S \rightarrow Aa.Ab$
$A \rightarrow .\epsilon$

I_6:
GOTO(I_3, b)
$S \rightarrow Bb.Ba$
$B \rightarrow .\epsilon$

I_7:
GOTO(I_5, A)
$S \rightarrow AaA.b$

I_8:
GOTO(I_5, ϵ)
$A \rightarrow \epsilon.$

I_9:
GOTO(I_6, B)
$S \rightarrow BbB.a$

I_{10}:
GOTO(I_6, ϵ)
$B \rightarrow \epsilon.$

I_{11}:
GOTO(I_7, b)
$S \rightarrow AaAb.$

I_{12}:
GOTO(I_9, a)
$S \rightarrow BbBa.$

Ans3 :-

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

Parse string id+id * id using CLR

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

(0) $E' \to .E$
(1) $E \to .E+E$
(2) $E \to .E*E$
(3) $E \to .id$

I_0 : CLOSURE($E' \to .E$)

$E' \to .E$, $
$E \to .E+E$, $
$E \to .E*E$, $
$E \to .id$, $/+/*

I_1: GOTO(I_0, E)

$E' \to E.$, $
$E \to E.+E$, $/+
$E \to E.*E$, $/*

I_2: GOTO(I_0, id)

$E \to id.$, $/+/*

I_3: GOTO(I_1, +)

$E \to E+.E$, $/+
$E \to .E+E$, $/+
$E \to .E*E$, $/+
$E \to .id$, +

I_4: GOTO(I_1, *)

$E \to E*.E$, $/*
$E \to .E+E$, $/*
$E \to .E*E$, $/*
$E \to .id$, *

I_5: GOTO(I_3, E)

$E \to E+E.$, $/+
$E \to E.+E$, $/+
$E \to E.*E$, $/*

I_6: GOTO(I_4, E)

$E \to E*E.$, $/*
$E \to E.+E$, $/*
$E \to E.*E$, *

I_7: GOTO(I_5, +)

$E \to E+.E$, +
~~$E \to .E+E$, +~~ I_3
$E \to .E*E$, +
$E \to .id$, +

I_8: GOTO(I_4, id)

$E \to id.$, *

GOTO(I_5, *)

$E \to E*.E$, *

GOTO(I_3, id)

I_6: $E \to id.$, +

GOTO(I_5, *)

$E \to E*.E$, *
$E \to .E+E$, *
$E \to .E*E$, *
$E \to .id$, *

~~I_3~~ I_4

GOTO(I_7, +)

$E \to E+.E$, *
$E \to .E+E$, *
$E \to .E*E$, *
$E \to .id$, *

I_9

GOTO(I_9, E)

$E \to E+E.$, *
$E \to E.+E$, *
$E \to E.*E$, *
~~$E \to .id$~~

I_{10}

GOTO(I_9, id)

I_8

GOTO(I_{10}, +)

$E \to E+.E$, *
:

I_9

GOTO(I_{10}, *)

$E \to E*.E$, *
:

I_4

GOTO(I_7, *)

$E \to E*.E$, *
$E \to .E+E$, *
$E \to .E*E$, *
$E \to .id$, *

$\Rightarrow I_4$

Ans4 :- $(a+b)*(c+d)+(a+b+c)$

$t1 = a+b$

$t2 = c+d$

$t3 = t1+t2$

$t4 = t1+c$

$t5 = t3+t4$

Ans 5 :- $S \rightarrow Aa \mid bAc \mid dc \mid bca$

$A \rightarrow d$

(0) $S' \rightarrow S$
(1) $S \rightarrow Aa$
(2) $S \rightarrow bAc$
(3) $S \rightarrow dc$
(4) $S \rightarrow bca$
(5) $A \rightarrow d$

SLR

I_0 : CLOSURE($S' \rightarrow S$)

$S' \rightarrow .S$
$S \rightarrow .Aa$
$S \rightarrow .bAc$
$S \rightarrow .dc$
$S \rightarrow .bca$
$A \rightarrow .d$

I_1 : GOTO(I_0, S)

$S' \rightarrow S.$

I_2 : GOTO(I_0, A)

$S \rightarrow A.a$

I_3 : GOTO(I_0, b)

$S \rightarrow b.Ac$
$A \rightarrow .d$
$S \rightarrow b.ca$

I_4 : GOTO(I_0, d)

$S' \rightarrow d.c$
$A \rightarrow d.$

~~I_5 : GOTO(I_0, d)~~

I_5 : GOTO(I_2, a)

$S \rightarrow Aa.$

I_6 : GOTO(I_3, A)

$S \rightarrow bA.c$

I_7 : GOTO(I_3, d)

$A \rightarrow d.$

I_8 : GOTO(I_3, c)

$S \rightarrow bc.a$

I_9 : GOTO(I_4, c)

$S \rightarrow dc.$

~~I_{10}~~ : GOTO(I_4, d)

I_7

I_{10} : GOTO(I_6, c)

$S \rightarrow bAc.$

I_{11} : GOTO(I_8, a)

$S \rightarrow bca.$

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

States	Inputs				Action GOTO
	+	*	id	$	E
0			s2		1
1	s3	s4		accept	
2	r3	r3		r3	
3			s6		5
4			s8		7
5	r1/s3	s4			
6	r3				
7	s9	s4/r2			
8		r3			
9	r2		s8		10
10	s9	r1/s4			

Stack	Input	Action
$0	id+id*id$	shift
$0id2	+id*id$	reduce by E→id
$0E1	+id*id$	shift
$0E1+3	id*id$	shift
$0E1+3id6	*id$	

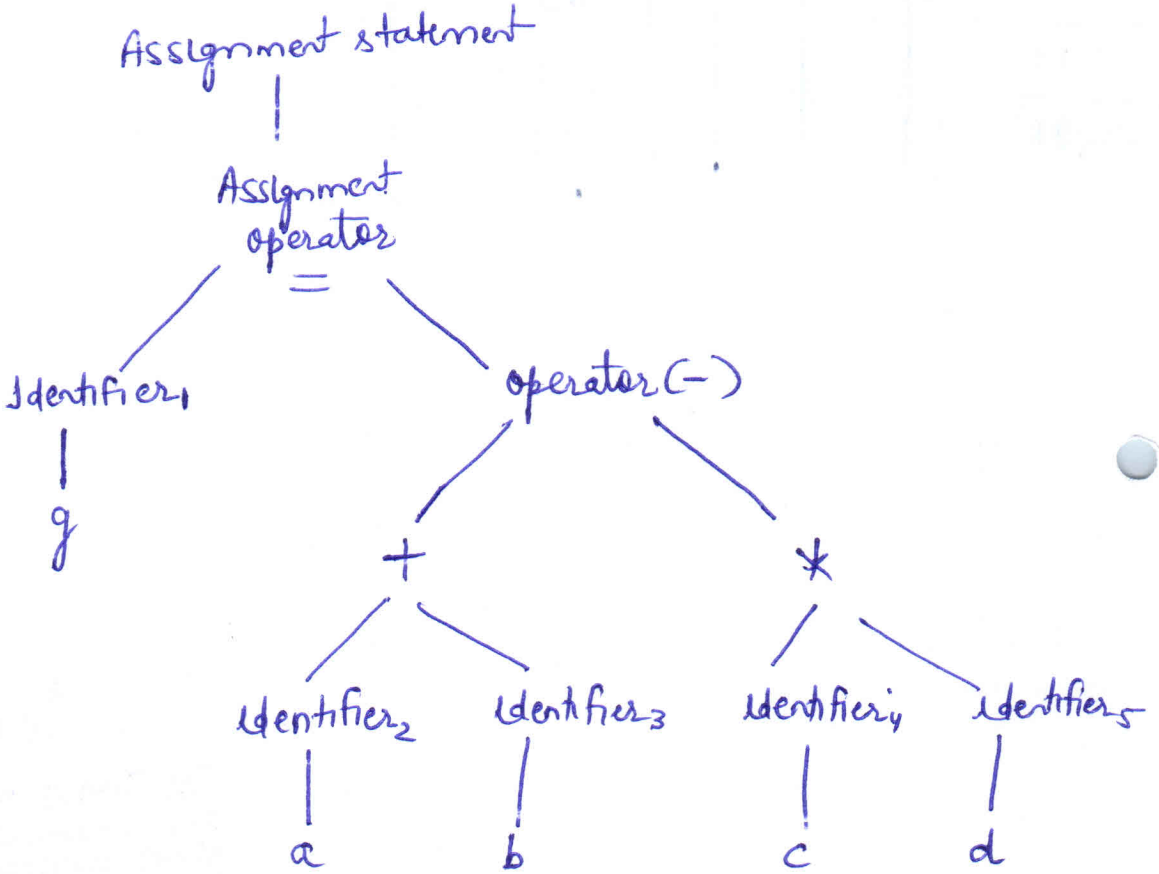
States	Input					Action/GOTO	
	a	b	c	d	$	S	A
0		s3		s4		1	2
1					accept		
2	s5						
3			s8	s7			6
4	r5		r5	s7			
5					r1		
6			s10				
7	r5		r5				
8	s11						
9					r3		
10					r2		
11					r4		

FOLLOW(S) = {$}

FOLLOW(A) = {c, a}

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

A6 :- $g = a + b - (c * d)$


Assignment statement
Assignment operator
=
Identifier1
g
operator (-)
+
*
Identifier2
a
Identifier3
b
Identifier4
c
Identifier5
d

Subject Teacher: Ms. Priyanka

MIT Group of Institutions, Moradabad

ATTENDANCE SHEET

Session: 2019-20

Class Test I / ~~II / III~~

Date: 26.02.20 **Shift:** II

Room No: A-313

Year: III **Semester:** VI

Section/Branch: B/CS

Subject Name: Compiler Design

Subject Code: RCS-602

	S. No	Roll No.	Name of Student		Branch	Signature
A	1.	1708210064	Manas Arora	~~13~~ 10.5	C.S.E.	Manas.
P	2.	1708210065	Manu Panwar	~~03~~ 2.5	CSE	Manwar
A	3.	1708210067	Mayank Upadhyay	13.5	CSE	Mayank
A	4.	1708210068	Mohd Amaan	10	CSE	Mohd
P	5.	1708210072	Mohd Anas	06	CSE	Anas
A	6.	1708210074	Mohd Fardeen	10.5	CSE	Fardeen
	7.	1708210086	Naman Agarwal	16	C.S.E.	Naman
	8.	1708210084	Muskan Menotra	16.5	CSE	Muskan
	9.	1708210083	Mukul Kumar	17.5	"	Mukul
	10	1708210079	Mohd Suhail	~~08~~ 8.5	"	Suhail
	11.	1708210078	Mohd Shoaib	10	"	Shoaib
A	12.	1708210075	Mohd Haris	13.5	CSE	Haris
	13.	1708210087	Nandita Gaur	13	CSE	Gaur
	14.	1708210090	Nikita Singh	14.5	CSE	Nikita
	15.	1708210092	Nishita Agarwal	17	CSE	Nishita
	16.	1708210094	Nitika Rastogi	19.5	CSE	Nitika.
	17.	1708210100	Palak Goel	15	CSE	palak.
	18.	1708210101	Palak Rastogi	12	CSE	Palak.
	19.	1708210121	Rishabh Kumar Sharma	18.5	CSE	Rishabh
	20.	1708210120	Rishabh Choudhary	17.5	CSE	Rishabh Choudhary
	21.	1708210115	Rahul Sukhija	17.5 ~~18~~	CSE	Rahul Sukhija
	22.	1708210111	Prayag Verma	17.5	CSE	Prayag
	23.	1708210106	Pradeep Kumar	04	CSE	Pradeep
	24.	1708210104	Piyush Dhawan.	7	CSE	Piyush
	25.					
	26.					
	27.					
	28.					
	29.					
	30.					

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Total No. of Students allotted in Room: 24 Students Absent: NIL Students Present: 24

Invigilators: 1) Name Dr. Harendra Kumar Sign:

2) Name Sign:

Subject Teacher: Mrs. Priyanka Goyal

MIT Group of Institutions, Moradabad

ATTENDANCE SHEET

Session: 2019-20

Class Test I / II / III

Date: 26-02-2020 Shift: 2nd Room No: B-321

Year: 3rd Semester: 6th Section/Branch: C

Subject Name: Computer Design Subject Code: RCS-602

S. No	Roll No.	Name of Student		Branch	Signature
1.	1708210124	Ritvik Dayal +1	17	CSE	Ritvik
2.	1708210125	Riyanshi Singhal 1015	~~9.5~~	CSE	Riyanshi
3.	1708210129	Sakib	9	"	Sakib
4.	1708210130	Saloni Bhatnagar +1	13.5	"	Saloni
5.	1708210132	Sanpreet Kaur +1	16	"	Sanpreet
6.	1708210135	Saurabh Bhatnagar +1	13.5	"	Bhatnagar
7.	1708210152	Sumit Debnath	10.5	"	Sumit
8.	1708210144	Shubham Khanna	6.5	"	Shubham
9.	1708210142	Shruti Arya +1	9.5	"	Shruti
10.	1708210140	Shivangi Arora +1	16	"	Shivangi
11.	1708210139	Shivam Khurana 08		17	Shivam
12.	1708210156	Tanya Duggal	17	"	Tanya
13.	1708210157	Tasha Johari +1	15.5	"	Tasha
14.	1708210158	Tushar Agarwal	09	"	Tushar
15.	1708210161	Vaibhav Chauhan	14.5	"	Vaibhav
16.	1708210162	Vedika Khanna +1	13.5	"	Vkhanna
17.	1708210166	Vilsin Kashyap	09	"	Vilsin
18.	1708210167	Vishakha Rastogi	~~16~~.5	"	Vishakha
19.	1708210137	Shashi mala +1	13.5	"	Shashi
20.	1708210153	Sumit Kumar	6.5	"	Sumit
21.	1808210904	Shivam Kumar	9.5	"	SK
22.	1708210173	Zainab Azeem	17.5	"	Zainab Azeem
23.	1708210171	Vedika Gupta	10 ~~08~~	"	Vedika
24.	1708210169	Vishal Kumar +1	09	"	Vishal
25.	1708210168	Vishakha Tandon +1	19	91	Vishakha
26.	1708210150	ABSENT Sumit Kumar			
27.					
28.					
29.					
30.					

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Total No. of Students allotted in Room: 26 Students Absent: 01 Students Present: 25

Invigilators: 1) Name Mr. Prashant Singh Sign:

2) Name Sign:

In Pursuit of Excellence	**List of Students having short attendance in CT-1**	SESSION-2019-2020 SEM- 6th

SECTION B

SNo	Name of student	Roll No	Att %
1	Prashant Kumar	1708210109	35/83
2	Parth Sharma	1708210103	40/83
3	Mohd Asif	1708210077	41/85
4	Piyush Sharma	1708210105	43/85
5	Paras Vishnoi	1708210102	44/85
6	Mohd Akif	1708210071	47/86
7	Mayank Bhatnagar	1708210066	48/86
8	Mohd Afzal	1708210070	48/86
9	Ranojit Malik	1708210116	48/85
10	Mohd Sadiq	1708210076	49/86
11	Muskan Agarwal	1708210085	48/83
12	Nirbhay Pal	1708210091	48/83
13	Raghav Agarwal	1708210114	49/84
14	Mudit Kumar Sharma	1708210081	50/85
15	Nitin Chauhan	1708210096	49/83
16	Mohd Anas	1708210069	51/86
17	Rishi Raj Singh	1708210122	50/84
18	Pratham Maheshwari	1708210110	51/85
19	Nivesh Kumar	1708210098	50/83
20	Priyank Raghav	1708210112	52/85
21	Pranvi Jain	1708210108	51/83
22	Mohit Agarwal	1708210080	53/85
23	Ritika Saxena	1708210123	53/84
24	Nitika Rastogi	1708210094	53/83

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

SECTION C

SNo	Name of student	Roll No	Att %
1	Vibhor Agarwal	1708210163	39/88
2	Satendra Saini	1708210133	41/91
3	Shikhar Rastogi	1708210138	43/91
4	Srashti Gautam	1708210149	44/91
5	Sanchit Lamba	1708210131	46/91
6	S Ali Abid Abidi	1708210127	49/92
7	Yash Chaudhary	1708210172	50/89
8	Aryan Thakur	1808210902	50/88
9	Vaibhav Choudhary	1708210160	52/90
10	Shruti Gupta	1708210143	53/91
11	Siddharth Singh	1708210146	55/91
12	Sourabh Saini	1708210136	56/91
13	Vila Zehra	1708210164	55/89
14	Mangalam Sharma	1808210903	56/88
15	Shobhit Rastogi	1708210141	58/91
16	Vineet Joshi	1708210165	57/89
17	Tanya Bhasin	1708210155	58/90
18	Aanchal Kumari	1808210901	57/88
19	Siddhant Mishra	1708210145	59/91

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Class Test 1 Marks** **Section B**	SESSION-2019-2020 SEM- 6th

SNo	Roll No.	Name of student	Q1(2)	Q2(2)	Q3(3)	Q4(3)	Q5(5)	Q6(5)	Total (20)
1.	1708210063	LALIT KUMAR	A	A	A	A	A	A	A
2.	1708210064	MANAS ARORA	1	0.5	3	3	3	0	10.5
3.	1708210065	MANU PANWAR	1	0	1		0.5		2.5
4.	1708210066	MAYANK BHATNAGAR	A	A	A	A	A	A	A
5.	1708210067	MAYANK UPADHYAY	1	1.5	3	1	4	3	13.5
6.	1708210068	MOHHAMMAD AMAAN	2	0	2.5	1.5	4		10
7.	1708210069	MOHAMMAD ANAS	A	A	A	A	A	A	A
8.	1708210070	MOHD. AFZAL	A	A	A	A	A	A	A
9.	1708210071	MOHD. AKIF	A	A	A	A	A	A	A
10.	1708210072	MOHD. ANAS	1	0	0	1	4		6
11.	1708210073	MOHD. ASHIR	A	A	A	A	A	A	A
12.	1708210074	MOHD. FARDEEN	1	1.5	3	1.5	3.5	0	10.5
13.	1708210075	MOHD. HARIS	2	1	3	3	4.5		13.5
14.	1708210076	MOHD. SADIQ	A	A	A	A	A	A	A
15.	1708210077	MOHD. ASIF	A	A	A	A	A	A	A
16.	1708210078	MOHD. SHOAIB	2	0	3	1.5	3.5		10
17.	1708210079	MOHD. SUHAIL	2	1	1	2.5	2		8.5
18.	1708210080	MOHIT AGARWAL	A	A	A	A	A	A	A
19.	1708210081	MUDIT KUMAR SHARMA	A	A	A	A	A	A	A
20.	1708210082	MUKESH KUMAR	A	A	A	A	A	A	A
21.	1708210083	MUKUL KUMAR	2	2	3	3	5	2.5	17.5
22.	1708210084	MUSKAN MEHROTRA	2	2	3	2.5	4	3	16.5
23.	1708210085	MUSKAN AGARWAL	A	A	A	A	A	A	A
24.	1708210086	NAMAN AGARWAL	2	1.5	3	3	5	1.5	16
25.	1708210087	NANDITA GAURI	1.5	1	3	3	5	1.5	15
26.	1708210090	NIKITA SINGH	1.5	1.5	3	2.5	3	3	14.5
27.	1708210091	NIRBHAY PAL	A	A	A	A	A	A	A
28.	1708210092	NISHITA AGARWAL	2	2	3	3	5	2	17
29.	1708210093	NITESH SAINI	A	A	A	A	A	A	A
30.	1708210094	NITIKA RASTOGI	2	2	3	3	5	4.5	19.5
31.	1708210096	NITIN CHAUHAN	A	A	A	A	A	A	A
32.	1708210097	NITIN VERMA	A	A	A	A	A	A	A
33.	1708210098	NIVESH KUMAR	A	A	A	A	A		A
34.	1708210099	NUPUR GUPTA	A	A	A	A	A	A	A

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

SNo	Roll No.	Name of student	Q1(2)	Q2(2)	Q3(3)	Q4(3)	Q5(5)	Q6(5)	Total (20)
35.	1708210100	PALAK GOEL	1.5	2	3	3	4	1.5	15
36.	1708210101	PALAK RASTOGI	1	0	2	3	5	1	12
37.	1708210102	PARAS VISHNOI	A	A	A	A	A	A	A
38.	1708210103	PARTH SHARMA	A	A	A	A	A	A	A
39.	1708210104	PIYUSH DHAWAN	1	0	1	3	2	0	7
40.	1708210105	PIYUSH SHARMA	A	A	A	A	A	A	A
41.	1708210106	PRADEEP KUMAR	0	0		1	2.5	0.5	4
42.	1708210108	PRANVI JAIN	A	A	A	A	A	A	A
43.	1708210109	PRASHANT KUMAR	A	A	A	A	A	A	A
44.	1708210110	PRATHAM MAHESHWARI	A	A	A	A	A	A	A
45.	1708210111	PRAYAG VERMA	2	2	3	2.5	4	4	17.5
46.	1708210112	PRIYANK RAGHAV	A	A	A	A	A	A	A
47.	1708210113	PUSHKAR SHARMA	A	A	A	A	A	A	A
48.	1708210114	RAGHAV AGARWAL	A	A	A	A	A	A	A
49.	1708210115	RAHUL SUKHIJA	2	2	3	2.5	4	4	17.5
50.	1708210116	RANOJIT MALIK	A	A	A	A	A	A	A
51.	1708210118	RAVI RANJAN	A	A	A	A	A	A	A
52.	1708210120	RISHABH CHAUDHARY	2	2	3	3	5	2.5	17.5
53.	1708210121	RISHABH KUMAR SHARMA	2	2	3	2.5	5	4	18.5
54.	1708210122	RISHI RAJ SINGH	A	A	A	A	A	A	A
55.	1708210123	RITIKA SAXENA	A	A	A	A	A	A	A

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Class Test 1 Marks** **Section C**	SESSION-2019-2020 SEM- 6th

SNo	Roll No	Name of student	Q1(2)	Q2(2)	Q3(3)	Q4(3)	Q5(5)	Q6(5)	TOTAL (20)
1.	1708210124	RITVIK DAYAL	2	1	3	3	4.5	3.5	17
2.	1708210125	RIYANSHI SINGHAL	2	.0	3	0	4.5	1	10.5
3.	1708210126	ROHIT KUMAR SINGH	A	A	A	A	A	A	A
4.	1708210127	S. ALI ABID ABIDI	A	A	A	A	A	A	A
5.	1708210129	SAKIB	0.5	0.5	3	1	4		9
6.	1708210130	SALONI BHATNAGAR	2	1	1.5	3	5	1	13.5
7.	1708210131	SANCHIT LAMBA	A	A	A	A	A	A	A
8.	1708210132	SANPREET KAUR	1.5	1.5	3	2.5	5	2.5	16
9.	1708210133	SATENDRA SAINI	A	A	A	A	A	A	A
10	1708210134	SATISH SAINI	A	A	A	A	A	A	A
11	1708210135	SAURABH BHATNAGAR	1.5	1	3	2.5	5	1.5	14.5
12	1708210136	SAURABH SAINI	A	A	A	A	A	A	A
13	1708210137	SHASHIWALA	2	1	3	3	4.5		13.5
14	1708210138	SHIKHAR RASTOGI	A	A	A	A	A	A	A
15	1708210139	SHIVAM KHURANA	1		1.5	0.5	4.5	0.5	8
16	1708210140	SHIVANGI ARORA	1.5	2	3	2.5	5	2	16
17	1708210141	SHOBHIT RASTOGI	A	A	A	A	A	A	A
18	1708210142	SHRUTI ARYA	1.5		3	2	4		10.5
19	1708210143	SHRUTI GUPTA	A	A	A	A	A	A	A
20	1708210144	SHUBHAM KHANNA	1		1	2	2.5		6.5
21	1708210145	SIDDHANT MISHRA	A	A	A	A	A	A	A
22	1708210146	SIDDHARTH SINGH	A	A	A	A	A	A	A
23	1708210147	SOHAIL KHAN	A	A	A	A	A	A	A
24	1708210149	SRASHTI GAUTAM	A	A	A	A	A	A	A
25	1708210150	SUMIT KUMAR	A	A	A	A	A	A	A
26	1708210152	SUMIT DEBNATH	1.5	1	3	0	5	0	10.5
27	1708210153	SUMIT KUMAR	1		0	2.5	3		6.5
28	1708210154	SYED ASHRAF HUSSAIN	A	A	A	A	A	A	A
29	1708210155	TANYA BHASIN	A	A	A	A	A	A	A

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

SNo	Roll No.	Name of student	Q1(2)	Q2(2)	Q3(3)	Q4(3)	Q5(5)	Q6(5)	Total (20)
30	1708210156	TANYA DUGGAL	2	2	3	2.5	5	2.5	17
31	1708210157	TASHA JOHARI	1.5	2	3	2.5	4.5	2	15.5
32	1708210158	TUSHAR AGARWAL	2		1	2.5	3.5		9
33	1708210159	UTKARSH VARSHNEY	A	A	A	A	A	A	A
34	1708210160	VAIBHAV CHODHARY	A	A	A	A	A	A	A
35	1708210161	VAIBHAV CHAUHAN	2	1.5	3	3	4.5	0.5	14.5
36	1708210162	VEDIKA KHANNA	1.5	1.5	3	2.5	5		13.5
37	1708210163	VIBHOR AGARWAL	A	A	A	A	A	A	A
38	1708210164	VILA ZEHRA	A	A	A	A	A	A	A
39	1708210165	VINEET JOSHI	A	A	A	A	A	A	A
40	1708210166	VIPIN KASHYAP		2	0.5	2.5	3	1	9
41	1708210167	VISHAKHA RASTOGI	2	2	3	2.5	5	2	16.5
42	1708210168	VISHAKHA TANDON	2	2	3	2.5	5	4.5	19
43	1708210169	VISHAL KUMAR	1.5	0	1	2.5	4	0	9
44	1708210170	VISHAL KUMAR	A	A	A	A	A	A	A
45	1708210171	VRATIKA GUPTA	1	0	2	2.5	4.5		10
46	1708210172	YASH CHAUDHARY	A	A	A	A	A	A	A
47	1708210173	ZAINAB AZEEM	2	2	3	2.5	5	3	17.5
48	1508210100	PRANAV BHATNAGAR	A	A	A	A	A	A	A
49	1808210901	AANCHAL KUMARI	A	A	A	A	A	A	A
50	1808210902	ARYAN THAKUR	A	A	A	A	A	A	A
51	1808210903	MANGLAM SHARMA	A	A	A	A	A	A	A
52	1808210904	SHIVAM KUMAR	1.5		2	2.5	3.5		9.5

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Paper Code: RCS-602 - (Compiler Design), Course - B.Tech, Sem - 6, Section - B

Session: 2019-2020, **Sessional Name:** CT-02 (Even Sem)

S. No.	Student Id	Roll No.	Name	Max. Marks						Total	Per. (%)
				Q. 1	Q. 2	Q. 3	Q. 4	Q. 5	Q. 6		
				2	2	3	3	5	5	20	
1.	1710256	1708210063	Lalit Kumar	2	2	1.5	3	2	2	12.5	62.5
2.	1710060	1708210064	Manas Arora	1.5	2	1.5	3	5	2	15	75
3.	1710021	1708210065	Manu Panwar	2	2	1.5	3		2	10.5	52.5
4.	1710276	1708210066	Mayank Bhatnagar	2	2	1.5	3	5	2	15.5	77.5
5.	1710085	1708210067	Mayank Upadhyay	2	2	0	3		2	9	45
6.	1710046	1708210068	Mohammad Amaan	1.5	2	1.5	2.5		2	9.5	47.5
7.	1710302	1708210069	Mohammad Anas	2	2	1.5	1			6.5	32.5
8.	1710042	1708210070	Mohd Afzal	2	2	0	3	0	2	9	45
9.	1710295	1708210071	Mohd Akif	2	2		3	2	2	11	55
10.	1710283	1708210072	Mohd Anas	2	2	1.5	3	2	2	12.5	62.5
11.	1710008	1708210073	Mohd Ashir	2	2	1.5	3		2	10.5	52.5
12.	1710139	1708210074	Mohd Fardeen	2	2	0	3		2	9	45
13.	1710054	1708210075	Mohd Haris	2		2		2	3	9	45
14.	1710252	1708210076	Mohd Sadiq	2	2	1.5	3	2	2	12.5	62.5
15.	1710166	1708210077	Mohd Asif	2	2	1.5	3		2	10.5	52.5
16.	1710214	1708210078	Mohd Shoaib	2	2	1.5	3	2	2	12.5	62.5
17.	1710068	1708210079	Mohd Suhail	2	2	1.5	3	2	2	12.5	62.5
18.	1710308	1708210080	Mohit Agarwal	1.5	2		3	2	2	10.5	52.5
19.	1710020	1708210081	Mudit Kumar Sharma	1	2	1.5	1		2	7.5	37.5
20.	1710243	1708210082	Mukesh Kumar	2	2	1.5	3		2	10.5	52.5
21.	1710086	1708210083	Mukul Kumar	2	2		3		2	9	45
22.	1710273	1708210084	Mushkan Mehrotra	2	2	1.5	3	5	2	15.5	77.5
23.	1710333	1708210085	Muskan Agarwal	0	2	1.5	3	3	2	11.5	57.5
24.	1710248	1708210086	Naman Agarwal	2	2	2	3	5	4	18	90
25.	1710048	1708210087	Nandita Gauri	0	2	1.5	3			6.5	32.5
26.	1710186	1708210090	Nikita Singh	2	2	1.5	3	3	2	13.5	67.5
27.	1710203	1708210091	Nirbhay Pal	0	2	1.5	3	5	2	13.5	67.5
28.	1710282	1708210092	Nishita Agarwal	0	2	1.5	3	5	2	13.5	67.5
29.	1710320	1708210093	Nitesh Saini	2	2	0	3	2		9	45
30.	1710152	1708210094	Nitika Rastogi	2	2	2	3	5		14	70
31.	1710257	1708210096	Nitin Chauhan	2	2	0	3	0	2	9	45
32.	1710209	1708210097	Nitin Verma	2	2		3	3	2	12	60
33.	1710195	1708210098	Nivesh Kumar	2	2		3	2	2	11	55
34.	1710210	1708210099	Nupur Gupta	1.5	2		3	3	2	11.5	57.5
35.	1710144	1708210100	Palak Goel	1.5	2		3	2	2	10.5	52.5
36.	1710018	1708210101	Palak Rastogi	1.5	2	1.5	3	5	2	15	75
37.	1710096	1708210102	Paras Vishnoi	2	2	0	3	4	2	13	65
38.	1710029	1708210103	Parth Sharma	1.5	2	1.5	2	3	2	12	60
39.	1710075	1708210104	Piyush Dhawan	2	2	1.5	3	5	2	15.5	77.5
40.	1710079	1708210105	Piyush Sharma	2	2	0	3	3	2	12	60
41.	1710212	1708210106	Pradeep Kumar	2	2	1.5	3	0	2	10.5	52.5
42.	1710168	1708210108	Pranvi Jain	2	2	1.5	3	5	2	15.5	77.5
43.	1710313	1708210109	Prashant Kumar	2	2	0	3	2	2	11	55
44.	1710132	1708210110	Pratham Maheshwari	2	2	0.5	3	3	2	12.5	62.5
45.	1710255	1708210111	Prayag Verma	2	2	1.5	3	4		12.5	62.5
46.	1710324	1708210112	Priyank Raghav	2	2	0	3	3	2	12	60
47.	1710321	1708210113	Pushkar Sharma	2	2		3	5	2	14	70
48.	1710307	1708210114	Raghav Agarwal	2	2		3	5		12	60
49.	1710114	1708210115	Rahul Sukhija	2	2	1.5	3	5	2	15.5	77.5
50.	1710240	1708210116	Ranojit Malik	2	2	1.5	3		2	10.5	52.5

50. 1710240 1708210116 Ranojit Malik 2 2 1.5 3 2 10.5 52.5

56

S. No.	Student Id	Roll No.	Name	Max. Marks						Total	Per. (%)
				Q. 1	Q. 2	Q. 3	Q. 4	Q. 5	Q. 6		
				2	2	3	3	5	5	20	
51.	1710263	1708210118	Ravi Ranjan	2	2	1.5	3	3	2	13.5	67.5
52.	1710285	1708210120	Rishabh Chaudhary	2	2	1.5	3	3	2	13.5	67.5
53.	1710293	1708210121	Rishabh Kumar Sharma	0.5		1.5	3	3	4	12	60
54.	1710134	1708210122	Rishi Raj Singh	2	2	1.5	3	2	2	12.5	62.5
55.	1710005	1708210123	Ritika Saxena	0.5	2	1.5	3	5	2	14	70

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology

Paper Code: RCS-602 - (Compiler Design), Course - B.Tech, Sem - 6, Section - C

Session: 2019-2020, **Sessional Name:** CT-02 (Even Sem)

S. No.	Student Id	Roll No.	Name	Max. Marks						Total	Per. (%)
				Q. 1	Q. 2	Q. 3	Q. 4	Q. 5	Q. 6		
				2	2	3	3	5	5	20	
1.	1510592	1508210100	Pranav Bhatnagar	2	2	1.5	3		2	10.5	52.5
2.	1710222	1708210124	Ritvik Dayal	2	2	1.5	3	5		13.5	67.5
3.	1710043	1708210125	Riyanshi Singhal	2	2	0	3	3	2	12	60
4.	1710116	1708210126	Rohit Kumar Singh	2	2	1.5	3	2	2	12.5	62.5
5.	1710206	1708210127	S Ali Abid Abidi	1.5	2	1	3	2	2	11.5	57.5
6.	1710318	1708210129	Sakib	2	2	0	3		2	9	45
7.	1710336	1708210130	Saloni Bhatnagar	2	2	1.5	3	3	2	13.5	67.5
8.	1710062	1708210131	Sanchit Lamba	2	2	1.5	3	2	2	12.5	62.5
9.	1710314	1708210132	Sanpreet Kaur	2	2	1.5	3	2	2	12.5	62.5
10.	1710027	1708210133	Satendra Saini	2	2	0	3	3	2	12	60
11.	1710229	1708210134	Satish Saini	2	2	1.5	3	2	2	12.5	62.5
12.	1710145	1708210135	Saurabh Bhatnagar	0	2	1.5	3	4		10.5	52.5
13.	1710142	1708210136	Sourabh Saini	2	2	1.5	3	2	2	12.5	62.5
14.	1710173	1708210137	Shashi Wala	2	2	0	3	3	2	12	60
15.	1710025	1708210138	Shikhar Rastogi	2	2	1.5	3	2	2	12.5	62.5
16.	1710002	1708210139	Shivam Khurana	2	2	1.5	3	3	2	13.5	67.5
17.	1710227	1708210140	Shivangi Arora	2	2	1.5	3	4	2	14.5	72.5
18.	1710175	1708210141	Shobhit Rastogi	2	2	1.5	3	3	2	13.5	67.5
19.	1710011	1708210142	Shruti Arya	0	2	1.5	3	3	2	11.5	57.5
20.	1710101	1708210143	Shruti Gupta	2	2	1.5	3	3	2	13.5	67.5
21.	1710038	1708210144	Shubham Khanna	1	2	1.5	3	3	2	12.5	62.5
22.	1710066	1708210145	Siddhant Mishra	0.5	2	1.5	3	3	2	12	60
23.	1710036	1708210146	Siddharth Singh	2	2	1.5	3	0.5	2	11	55
24.	1710246	1708210147	Sohail Khan	2	2	1.5	3	2	2	12.5	62.5
25.	1710016	1708210149	Srashti Gautam	2	2	1.5	3	3	2	13.5	67.5
26.	1710235	1708210150	Sumit Kumar	2	2	0	3	2	2	11	55
27.	1710238	1708210152	Sumit Debnath	2	2	1.5	3	1	2	11.5	57.5
28.	1710249	1708210153	Sumit Kumar	2	2	0	3	2	2	11	55
29.	1710326	1708210154	Syed Ashraf Hussain	2	2	1	3	2	2	12	60
30.	1710130	1708210155	Tanya Bhasin	2	2	1.5	3	3	2	13.5	67.5
31.	1710128	1708210156	Tanya Duggal	0	2	1.5	3	5	2	13.5	67.5
32.	1710105	1708210157	Tasha Johari	2	2	1.5	3	5	2	15.5	77.5
33.	1710179	1708210158	Tushar Agarwal	2	2	1.5	3	2		10.5	52.5
34.	1710309	1708210159	Utkarsh Varshney	2	2	1	3	2	2	12	60
35.	1710323	1708210160	Vaibhav Choudhary	2	2	1.5	3	2	2	12.5	62.5
36.	1710120	1708210161	Vaibhav Chauhan	2	2	1.5	3	3	2	13.5	67.5
37.	1710155	1708210162	Vedika Khanna	2	2	1.5	3	2		10.5	52.5
38.	1710031	1708210163	Vibhor Agarwal	2	2		3	3	2	12	60
39.	1710192	1708210164	Vila Zehra	0	2	2	3	4		11	55
40.	1710069	1708210165	Vineet Joshi	2	2	1	3	3	2	13	65
41.	1710291	1708210166	Vipin Kashyap	2	2	1.5	3	2	1	11.5	57.5
42.	1710041	1708210167	Vishakha Rastogi	0	2	1.5	3	4		10.5	52.5
43.	1710241	1708210168	Vishakha Tandon	2	2	1.5	3	2	4	14.5	72.5
44.	1710131	1708210169	Vishal Kumar	2	2	1.5	3	2	2	12.5	62.5
45.	1710055	1708210170	Vishal Kumar	1	2	1.5	3	3		10.5	52.5
46.	1710237	1708210171	Vratika Gupta		2	1.5	3	3		9.5	47.5
47.	1710081	1708210172	Yash Chaudhary	2	2	0	3	2	2	11	55
48.	1710102	1708210173	Zainab Azeem	0	2	3	3	3	3	14	70
49.	2181008	1808210901	Aanchal Kumari	2	2	1.5	3	2	2	12.5	62.5
50.	2181013	1808210902	Aryan Thakur	0	2	1.5	3	2	2	10.5	52.5

S. No.	Student Id	Roll No.	Name	Max. Marks						Total	Per. (%)
				Q. 1	Q. 2	Q. 3	Q. 4	Q. 5	Q. 6		
				2	2	3	3	5	5	20	
51.	2181001	1808210903	Mangalam Sharma	0	2	1.5	3	2		8.5	42.5
52.	2181005	1808210904	Shivam Kumar	2	2	1.5			2	7.5	37.5

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**List of Weak Students** (Action taken for Improvement)	SESSION-2019-2020 SEM – 6th

SNo	Name of Student	Section	Action taken for improvement
1	Manu Panwar	B	* Focus on these students during Lectures & Tutorials * Questions covering basic & important topics, from Question Bank were given to them for preparation.
2	Pradeep Kumar		
3	Mohd Afzal		
4	Mohd Akif		
5	Mudit Kr Sharma		
6	Prashant Kumar		
7	Sakib	C	
8	Sourabh Saini		
9	Shikhar Rastogi		
10	Syed Ashraf Hussain		
11	Vineet Joshi		
12	Pranav Bhatnagar		

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

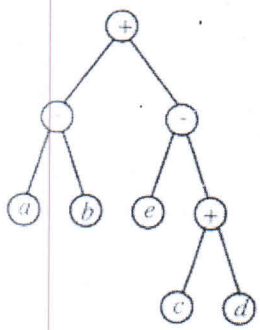
In Pursuit of Excellence	**List of Bright Students** (Action taken for enhancing performance)	SESSION-2019-2020 SEM – 6th

SNo	Name of Student	Section	Action taken for enhancing their performance
1	Naman Agarwal	B	* Tricky question set was provided to understand the concepts in-depth. * Questions from GATE were also provided for practice.
2	Nishita Agarwal		
3	Nitika Agarwal		
4	Rishabh Kumar Sharma		
7	Rityik Dayal	C	
8	Sanpreet Kaur		
9	Tanya Duggal		
10	Vishakha Rastogi		
11	Vishakha Tandon		
12	Zainab Azeem		

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

QUESTION SET FROM COMPETITIVE EXAMS

Q1: Consider evaluating the following expression tree on a machine with load-store architecture in which memory can be accessed only through load and store instructions. The variables a,b,c,d and e are initially stored in memory. The binary operators used in this expression tree can be evaluated by the machine only when operands are in registers. The instructions produce result only in a register. If no intermediate results can be stored in memory, what is the minimum number of registers needed to evaluate this expression? [GATE 2011]



Q2: What is the type of error (earliest phase) identified during the compilation of the following program ?

```
#include<stdio.h>
main( )
{
int x, y, z;
x = y = z = 10.3;
printf("%c", x);
}
```

Q4: Which one of the following grammars is free from left recursion? [GATE 2016]

A. S→AB
A→Aa|b
B→c

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

B. S→Ab|Bb|c
A→Bd|ϵ
B→e

C. S→Aa|B
A→Bb|Sc|ϵ
B→d

D. S→Aa|Bb|c
A→Bd|ϵ
B→Ae|ϵ

Q5 : Which of the following comparisons between static and dynamic type checking is incorrect? [ISRO 2018]

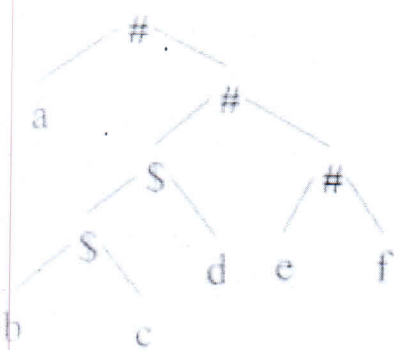
a) Dynamic type checking slows down the execution

b) Dynamic type checking offers more flexibility to the programmers

c) In contrast to Static type checking, dynamic type checking may cause failure in runtime due to type errors

d) Unlike static type checking, dynamic type checking is done during compilation

Q6: Consider the following parse tree for the expression a#bcd#e#fa#bcd#e#f, involving two binary operators $ and #. [GATE 2018]



Which one of the following is correct for the given parse tree?

(A) $ has higher precedence and is left associative; # is right associative

(B) # has higher precedence and is left associative; $ is right associative

(C) $ has higher precedence and is left associative; # is left associative

(D) # has higher precedence and is right associative; $ is left associative

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Q7 : Consider the following grammar: [GATE 2017]

stmt →→ if expr then expr else expr; stmt | 0

expr →→ term relop term | term

term →→ id |number

id →→ a | b | c

number →→ [0-9]

where relop is a rational operator (e.g.,<,>,...), O refers to the empty statement, and if, then, else are terminals. Consider a preogram P following the above grammar containing ten if terminals. The number of control flow paths in P is _____. For example, the program if e1 then e2 else e3 has 2 control flow paths, e1→e2ande1→e3e1→e2ande1→e3.

Q8: For the grammar below, a partial LL(1) parsing table is also presented along with the grammar. Entries that need to be filled are indicated as E1, E2, and E3. ε is the empty string, $ indicates end of input, and, | separates alternate right hand sides of productions. [GATE 2012]

S → a A b B | b A a B | ε

A → S

B → S

	a	b	$
S	E1	E2	S→ε
A	A→S	A→S	error
B	B→S	B→S	E3

The FIRST and FOLLOW sets for the non-terminals A and B are

(A) FIRST(A) = {a, b, ε} = FIRST(B)

FOLLOW(A) = {a, b}

FOLLOW(B) = {a, b, $}

(B) FIRST(A) = {a, b, $}

FIRST(B) = {a, b, ε}

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

FOLLOW(A) = {a, b}

FOLLOW(B) = {$}

(C) FIRST(A) = {a, b, ε} = FIRST(B)

FOLLOW(A) = {a, b}

FOLLOW(B) = □

(D) FIRST(A) = {a, b} = FIRST(B)

FOLLOW(A) = {a, b}

FOLLOW(B) = {a, b}

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Compiler Design (RCS – 602)
Questions based on Parsing (Set 2)

1. Construct Operator Precedence Parsing table for following grammar:

   A→ A $ B | B

   B → B # C | C

   C → C @ D | D

   D → d

2. Eliminate Left Recursion :

   S → S0S1S | 01

   S → (L) | x

   L → L,S | S

3. Compute FIRST and FOLLOW and construct parsing table:

S → aABCD A→b B→c C→d D→e	FR(S) = {a} FR(A) = {b} FR(B) = {c} FR(C) = {d} FR(D) = {e}	FL(S)= {$} FL(A)= {c} FL(B)= {d} FL(C)= {e} FL(D)= {$}
S→ABCD A→b|^ B→ c C→d D→e	FR(S) = {b, c} {b, ϵ} {c} {d} {e}	={$} ={c} ={d} ={e} ={$}
S→BCDE A→a|^ ** B → b|^ C →c D → d|^ E→ e|^	= {b, c} {a, ϵ} {b, ϵ} {c} {d, ϵ} {e, ϵ}	S = {$} A = {-} B = {c} C = {d, e, $}
S → Bb | Cd B → aB |^ C → cC | ^		
S→AB | CbB |Ba A→da | BC B →g |^ C → h |^		
S → aABb A →c |^ B →d | ^		
S→aBDh B→cC C→bC |^ D →EF E →g |^ F →f |^		

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Compiler Design (RCS – 602)
Questions based on Parsing (Set 3)

Find which of the following grammar are LL (1):

S → aSbS | bSaS | ^	
S→ aABb A → c|^ B → d |^	
S →A |a A → a	
S → aB | ^ B → bC | ^ C → cS | ^	
S → AB A→a | ^ B → b |^	
S →A A→ Bb | Cd B → aB |^ C →cC |^	
S →aSA |^ A →c |^	
S → aAa |^ A → abS | ^	
S → iEtSS' |a S' → eS |^ E →b	

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	Previous Year Question Papers	SESSION-2019-2020 SEM -6th

Printed Pages:2 Sub Code:RCS602

Paper Id: 110263 Roll No.

BTECH
(SEM VI) THEORY EXAMINATION 2018-19
COMPILER DESIGN

Time: 3 Hours *Total Marks: 70*

Note: 1. Attempt all Sections. If require any missing data; then choose suitably.

SECTION A

1. **Attempt *all* questions in brief.** **2 x 7 = 14**

a. What are the two parts of a compilation? Explain briefly.
b. What is meant by viable prefixes?
c. What are the classifications of a compiler?
d. List the various error recovery strategies for a lexical analysis.
e. What is dangling else problem?
f. What are the various types of intermediate code representation?
g. Define peephole optimization.

SECTION B

2. **Attempt any *three* of the following:** **7 x 3 = 21**

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

b. What are the problems with top down parsing? Write the algorithm for FIRST and FOLLOW.

c. Perform Shift Reduce Parsing for the given input strings using the grammar
S->(L)|a
L->L,S|S
i) (a,(a,a))
ii) (a,a)

d. What is global data flow analysis? How does it use in code optimization?

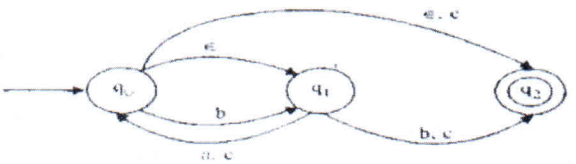
e. Construct LR(0) parsing table for the following grammar
S-> cB | ccA
A-> cA | a
B-> ccB | b

SECTION C

3. **Attempt any *one* part of the following:** **7 x 1 = 7**

(a) Convert following NFA to equivalent DFA and hence minimize the number of states in the DFA.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001



(b) Explain the various parameter passing mechanisms of a high level language.

4. **Attempt any *one* part of the following:** **7 x 1 = 7**

(a) How would you represent the following equation using DAG?

a:=b*-c+b*-c

(b) Distinguish between static scope and dynamic scope. Briefly explain access to non-local names in static scope.

5. **Attempt any *one* part of the following:** **7 x 1 = 7**

(a) Write short notes on the following with the help of example:
- (i) Loop unrolling
- (ii) Loop Jamming
- (iii) Dominators
- (iv) Viable Prefix

(b) Draw the format of Activation Record in stack allocation and explain each field in it.

6. **Attempt any *one* part of the following:** **7 x 1 = 7**

(a) Write down the translation procedure for control statement and switch statement

(b) Define Syntax Directed Translation. Construct an annotated parse tree for the expression (4 * 7 + 1) * 2, using the simple desk calculator grammar.

7. **Attempt any *one* part of the following:** **7 x 1 = 7**

(a) Explain in detail the error recovery process in operator precedence parsing method.

(b) Explain what constitute a loop in flow graph and how will you do loop optimizations in code optimization of a compiler.

Printed Pages:02 | Sub Code: NCS603

Paper Id: 110252 | Roll No. | | | | | | | | | |

B.TECH.
(VI-SEMESTER) THEORY EXAMINATION 2017-18
COMPILER DESIGN

Time: 3 Hours | *Total Marks: 100*

Note: 1. Attempt all Sections. If require any missing data; then choose suitably.
2. Any special paper specific instruction.

SECTION A

1. Attempt *all* questions in brief. **2 x10 = 20**

a. What is Bootstrapping?
b. What is Code Generator?
c. What is YACC & LEX tools?
d. Define Regular Expression using suitable example.
e. Explain Error detection in Symbol Table.
f. Explain Back patching using suitable example.
g. What is DAG?
h. What is the difference between Syntax Analyzer & Symantec Analyzer?
i. What is Data Flow Analysis?
j. Explain the difference between Top Down Parsing & Bottom Up Parsing.

SECTION B

2. Attempt any *three* of the following: **10 x 3 = 30**

a. What are the Phases and Passes of compiler? Explain the function of each Phases briefly.
b. Explain LR(0) parsing Algorithm using suitable example.
c. Define a SDT to generate Three Address Code.
d. What is role of Symbol Table? Discuss Data Structures used for Symbol Table.
e. Construct the DAG for the expression:
a+a*(b-c)+(b-c)*d+ e+e*(f-g)+(f-g)*h

SECTION C

3. Attempt any *one* part of the following: **10 x 1 = 10**

(a) i) Remove left factoring of the following grammar:
S→aAB | aA | aAC
ii) Remove left Recursion of the following grammar:
S→Ab | B, A→Ac | Sb | ε

(b) i) Explain the role of precedence & associativity for the conversion of ambiguous grammar to unambiguous grammar.

Printed Pages : 1 **Roll No.** | | | | | | | | | | | **ECS603**

B.TECH.

THEORY EXAMINATION (SEM–VI) 2016-17

COMPILER DESIGN

Time : 3 Hours ***Max. Marks : 100***

Note : *Be precise in your answer. In case of numerical problem assume data wherever not provided.*

SECTION – A

1. Attempt the following: **10 x 2 = 20**

(a) State any two reasons as to why phases of compiler should be grouped.

(b) Write regular expression to describe a language consist of strings made of even numbers a & b.

(c) Write a CF grammar to represent palindrome.

(d) Why are quadruples preferred over triples in an optimizing compiler?

(e) Give syntax directed translation for case statement.

(f) What is a syntax tree? Draw the syntax tree for the following statement: c b c b a − * + − * =

(g) How to perform register assignment for outer loops?

(h) List out the criteria for code improving transformations.

(i) Represent the following in flow graph i=1;sum=0;while (i<=10){sum+=i;i++}

(j) What is the use of algebraic identities in optimization of basic blocks?

SECTION – B

2. Attempt any five of the following questions: **5 x 10 = 50**

(a) Explain in detail the process of compilation. Illustrate the output of each phase of compilation of the input "a=(b+c)*(b+c)*2"..

(b) Construct the minimized DFA for the regular expression (0+1)*(0+1) 10.

(c) What is an ambiguous grammar? Is the following grammar ambiguous? Prove EE+|E(E)|id.The grammar should be moved to the next line ,centered.

(d) Draw NFA for the regular expression ab*/ab.

(e) How names can be looked up in the symbol table? Discuss.

(f) Write an algorithm to partition a sequence of three address statements into basic blocks.

(g) Discuss in detail the process of optimization of basic blocks. Give an example

(h) How to subdivide a run-time memory into code and data areas. Explain

SECTION – C

Attempt any two of the following questions: **2 x 15 = 30**

3 Consider the following grammar

S-AS|b

A-SA|a.

Construct the SLR parse table for the grammar. Show the actions of the parser for the input string "abab".

4 How would you convert the following into intermediate code? Give a suitable example. i) Assignment Statements. ii) Case Statements

5 Define a directed acyclic graph. Construct a DAG and write the sequence of instructions for the expression a+a*(b-c)+(b-c)*d.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	QUESTION BANK	SESSION-2019-2020 SEM- 6th

UNIT 1 [CO-1]

1. What is Compiler? Design the Analysis and Synthesis Model of Compiler.
2. Write down the five properties of compiler.
3. What is translator? Write down the steps to execute a program.
4. Discuss all the phases of compiler with a with a diagram.
5. Write a short note on:

a. YACC

b. Pass

c. Bootstrapping

d. LEX Compiler

e. Tokens, Patterns and Lexemes

6. Write the steps to convert Non-Deterministic Finite Automata (NDFA) into Deterministic Finite Automata (DFA).
7. Let M=({q0,q1}, {0,1}, δ, q0, {q1}).

Be NFA where δ(q0,0)={q0,q1}, δ(q1,1) = {q1}
δ(q1, 0)=$_{\varphi}$, δ(q1, 1)={q0, q1}
Construct its equivalent DFA.

8. Convert the given NFA to DFA:

Input/State	0	1
→q0	{q0, q1}	q0
q1	q2	q1
q2	q3	q3
q3 (final state)	$_{\varphi}$ (null character)	q2

UNIT 2 [CO-2]

1. Explain the parsing techniques with a hierarchical diagram.

2. What are the problems associated with Top Down Parsing?

3. Write the production rules to eliminate the left recursion and left factoring problems.

4. Consider the following Grammar:

A-> ABd|Aa|a B-> Be|b

Remove left recursion.

5. Do left factoring in the following grammar: A-> aAB|aA|a B-> bB|b
6. Write a short note on:

a. Ambiguity (with example)

b. Recursive Descent Parser

c. Predictive LL(1) parser (working)

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

d. Handle pruning

e. Operator Precedence Parser

7. Write Rules to construct FIRST Function and FOLLOW Function.
8. Consider Grammar:
   E-> E+T|T T-> T*F|F F-> (E)|id
9. Write the algorithm to create Predictive parsing table with the scanning of input string.
10. Show the following Grammar:
    S-> AaAb|BbBa A-> ϵ B-> ϵ
Is LL(1) and parse the input string "ba".
11. Consider the grammar:
    E-> E+E E-> E*E E->id
Perform shift reduce parsing of the input string "id1+id2+id3".
12. Write the properties of LR parser with its structure. Also explain the techniques of LR parser.
13. Write a short note on:
    a. Augmented grammar
    b. Rules of closure operation and goto operation
    c. Rules to construct the LR(0) items
14. Consider the following grammar:
    S-> Aa|bAc|Bc|bBa A-> d B-> d
Compute closure and goto.
15. Write the rules to construct the SLR parsing table.
16. Consider the following grammar:
    E-> E+T|T T-> TF|F F-> F*|a|b
Construct the SLR parsing table and also parse the input "a*b+a"
17. Write the rules to construct the LR(1) items.
18. What is LALR parser? Construct the set of LR(1) items for this grammar:
    S-> CC C-> aC C->d
19. Show the following grammar
    S->Aa|bAc|Bc|bBa A->d B->d
Is LR(1) but not LALR(1).
20. Write the comparison among SLR Parser, LALR parser and Canonical LR Parser.

UNIT 3 [CO3]

1. What is syntax directed translation (SDD)?
2. Write short note on:
    a. Synthesized attributes
    b. Inherited attributes
    c. Dependency graph
    d. Evaluation order
    e. Directed Acyclic Graph (DAG)
3. Draw the syntax tree and DAG for the following expression: (a*b)+(c-d)*(a*b)+b
4. Differentiate between synthesized translation and inherited translation.
5. What is intermediate code and write the two benefits of intermediate code generation.
6. Write the short note on:
    a. Abstract syntax tree
    b. Polish notation
    c. Three address code
    d. Backpatching
7. Construct syntax tree and postfix notation for the following expression:(a+(b*c)^d-e/(f+g)
8. Write quadruples, triples and indirect triples for the expression: -(a*b)+(c+d)-(a+b+c+d)

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

9. Write the three address statement with example for:
    a. Assignment
    b. Unconditional jump (goto)
    c. Array statement (2D and 3D)
    d. Boolean expression
    e. If-then-else statement
    f. While, do-while statement
    g. Switch case statement

UNIT 4 [CO-4]

1. Write the definition of symbol table and procedure to store the names in symbol table.
2. What are the data structures used in symbol table?
3. What are the limitations of stack allocation?
4. Write two important points about heap management.
5. Write the comparison among Static allocation, Stack allocation and Heap Allocation with their merits and limitations.
6. What is activation record? Write the various fields of Activation Record.
7. What are the functions of error handler?
8. Write a short note on Error Detection and Recovery.
9. Classify the errors and discuss the errors in each phase of Compiler.

UNIT 5 [CO-5]

1. What are the properties of code generation phase? Also explain the Design Issues of this phase.
2. What are basic blocks? Write the algorithm for partitioning into Blocks.
3. Write a short note on:
    a. Flow graph (with example)
    b. Dominators
    c. Natural loops
    d. Inner loops
    e. Reducible flow graphs
4. Consider the following program code:

```
Prod=0;
I=1;
Do{
Prod=prod+a[i]*b[i];
I=i+1;
}while (i<=10);
```

    a. Partition in into blocks
    b. Construct the flow graph
5. What is code optimization? Explain machine dependent and independent code optimization.
6. What is common sub-expression and how to eliminate it? Explain with example.
7. Write a short note with example to optimize the code:
    a. Dead code elimination
    b. Variable elimination
    c. Code motion
    d. Reduction in strength
8. What is control and data flow analysis? Explain with example.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

In Pursuit of Excellence	**Final Internal Marks**	SESSION-2019-2020 SEM – 6th

S.No	Roll No.	Name of Students	CT1 (20)	CT2 (20)	Best of CT(20)	Att (5)	Assignment(5)	Total (30)
1.	1708210063	LALIT KUMAR	D	12.5	12.5	5	5	23
2.	1708210064	MANAS ARORA	10.5	15	15	5	4	24
3.	1708210065	MANU PANWAR	2.5	10.5	10.5	5	4	20
4.	1708210066	MAYANK BHATNAGAR	D	15.5	15.5	5	5	26
5.	1708210067	MAYANK UPADHYAY	13.5	9	13.5	5	5	24
6.	1708210068	MOHHAMMAD AMAAN	10	9.5	10	5	5	20
7.	1708210069	MOHAMMAD ANAS	D	12.5	12.5	5	5	23
8.	1708210070	MOHD. AFZAL	D	11	11	4	4	19
9.	1708210071	MOHD. AKIF	D	11	11	4	4	19
10.	1708210072	MOHD. ANAS	6	9	9	5	5	19
11.	1708210073	MOHD. ASHIR	D	10.5	10.5	5	5	21
12.	1708210074	MOHD. FARDEEN	10.5	9	10.5	5	5	21
13.	1708210075	MOHD. HARIS	13.5	9	13.5	5	5	24
14.	1708210076	MOHD. SADIQ	D	12.5	12.5	4	4	21
15.	1708210077	MOHD. ASIF	D	10.5	10.5	4	4	19
16.	1708210078	MOHD. SHOAIB	10	12.5	12.5	5	5	23
17.	1708210079	MOHD. SUHAIL	8.5	12.5	12.5	5	5	23
18.	1708210080	MOHIT AGARWAL	D	10.5	10.5	5	5	21
19.	1708210081	MUDIT KUMAR SHARMA	D	10	10	5	4	19
20.	1708210082	MUKESH KUMAR	D	10.5	10.5	5	4	20
21.	1708210083	MUKUL KUMAR	17.5	9	17.5	5	5	28
22.	1708210084	MUSKAN MEHROTRA	16.5	15.5	16.5	5	5	27
23.	1708210085	MUSKAN AGARWAL	D	11.5	11.5	5	4	21

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

24.	1708210086	NAMAN AGARWAL	16	18	18	5	5	28
25.	1708210087	NANDITA GAURI	15	6.5	15	5	5	25
26.	1708210090	NIKITA SINGH	14.5	13.5	14.5	5	5	25
27.	1708210091	NIRBHAY PAL	D	13.5	13.5	5	4	23
28.	1708210092	NISHITA AGARWAL	17	13.5	17	5	5	27
29.	1708210093	NITESH SAINI	D	10	10	5	4	19
30.	1708210094	NITIKA RASTOGI	19.5	14	19.5	5	5	30
31.	1708210096	NITIN CHAUHAN	D	9	9	5	5	19
32.	1708210097	NITIN VERMA	D	12	12	5	5	22
33.	1708210098	NIVESH KUMAR	D	11	11	5	5	21
34.	1708210099	NUPUR GUPTA	D	11.5	11.5	5	5	22
35.	1708210100	PALAK GOEL	15	10.5	15	5	5	25
36.	1708210101	PALAK RASTOGI	12	15	15	5	5	25
37.	1708210102	PARAS VISHNOI	D	13	13	5	5	23
38.	1708210103	PARTH SHARMA	D	12	12	4	5	21
39.	1708210104	PIYUSH DHAWAN	7	15.5	15.5	5	5	26
40.	1708210105	PIYUSH SHARMA	D	12	12	4	5	21
41.	1708210106	PRADEEP KUMAR	4	10.5	10.5	5	4	20
42.	1708210108	PRANVI JAIN	D	15.5	15.5	5	5	26
43.	1708210109	PRASHANT KUMAR	D	11	11	4	4	19
44.	1708210110	PRATHAM MAHESHWARI	D	12.5	12.5	5	5	23
45.	1708210111	PRAYAG VERMA	17.5	12.5	17.5	5	5	28
46.	1708210112	PRIYANK RAGHAV	D	12	12	5	5	22
47.	1708210113	PUSHKAR SHARMA	D	14	14	5	5	24
48.	1708210114	RAGHAV AGARWAL	D	12	12	5	5	22
49.	1708210115	RAHUL SUKHIJA	17.5	15.5	17.5	5	5	28
50.	1708210116	RANOJIT MALIK	D	10.5	10.5	5	5	21
51.	1708210118	RAVI RANJAN	D	13.5	13.5	5	5	24
52.	1708210120	RISHABH CHAUDHARY	17.5	13.5	17.5	5	5	28

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

53.	1708210121	RISHABH KUMAR SHARMA	18.5	12	18.5	5	5	29
54.	1708210122	RISHI RAJ SINGH	D	12.5	12.5	5	5	23
55.	1708210123	RITIKA SAXENA	D	14	14	5	5	24

S.No.	Roll No.	Name of Students	CT1 (20)	CT2 (20)	Best of CT(20)	Att(5)	Assignment (5)	Total (30)
1.	1708210124	RITVIK DAYAL	17	13.5	17	5	5	27
2.	1708210125	RIYANSHI SINGHAL	10.5	12	12	5	5	22
3.	1708210126	ROHIT KUMAR SINGH	D	12.5	12.5	5	5	23
4.	1708210127	S. ALI ABID ABIDI	D	11.5	11.5	5	4	21
5.	1708210129	SAKIB	9	9	9	5	5	19
6.	1708210130	SALONI BHATNAGAR	13.5	13.5	13.5	5	5	24
7.	1708210131	SANCHIT LAMBA	D	12.5	12.5	5	5	23
8.	1708210132	SANPREET KAUR	16	12.5	16	5	5	26
9.	1708210133	SATENDRA SAINI	D	12	12	5	4	21
10.	1708210134	SATISH SAINI	D	12.5	12.5	5	5	23
11.	1708210135	SAURABH BHATNAGAR	14.5	10.5	14.5	5	5	25
12.	1708210136	SAURABH SAINI	D	12.5	12.5	5	4	22
13.	1708210137	SHASHIWALA	13.5	12	13.5	5	5	24
14.	1708210138	SHIKHAR RASTOGI	D	12.5	12.5	4	5	22
15.	1708210139	SHIVAM KHURANA	8	13.5	13.5	5	5	24
16.	1708210140	SHIVANGI ARORA	16	14.5	16	5	5	26
17.	1708210141	SHOBHIT RASTOGI	D	13.5	13.5	5	5	24
18.	1708210142	SHRUTI ARYA	10.5	11.5	11.5	5	5	22
19.	1708210143	SHRUTI GUPTA	D	13.5	13.5	5	5	24
20.	1708210144	SHUBHAM KHANNA	6.5	11.5	11.5	5	5	22

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

21.	1708210145	SIDDHANT MISHRA	D	12	12	5	5	22
22.	1708210146	SIDDHARTH SINGH	D	11	11	5	5	21
23.	1708210147	SOHAIL KHAN	D	12.5	12.5	5	5	23
24.	1708210149	SRASHTI GAUTAM	D	13.5	13.5	5	5	24
25.	1708210150	SUMIT KUMAR	A	11	11	5	5	21
26.	1708210152	SUMIT DEBNATH	10.5	11.5	11.5	5	5	22
27.	1708210153	SUMIT KUMAR	6.5	11	11	5	5	21
28.	1708210154	SYED ASHRAF HUSSAIN	D	12	12	5	5	22
29.	1708210155	TANYA BHASIN	D	13.5	13.5	5	4	23
30.	1708210156	TANYA DUGGAL	17	13.5	17	5	5	27
31.	1708210157	TASHA JOHARI	15.5	15.5	15.5	5	5	26
32.	1708210158	TUSHAR AGARWAL	9	10.5	10.5	5	5	21
33.	1708210159	UTKARSH VARSHNEY	D	12	12	5	5	22
34.	1708210160	VAIBHAV CHODHARY	D	12.5	12.5	5	5	23
35.	1708210161	VAIBHAV CHAUHAN	14.5	13.5	14.5	5	5	25
36.	1708210162	VEDIKA KHANNA	13.5	10.5	13.5	5	5	24
37.	1708210163	VIBHOR AGARWAL	D	12	12	4	5	21
38.	1708210164	VILA ZEHRA	D	15	15	5	5	25
39.	1708210165	VINEET JOSHI	D	13	13	5	5	23
40.	1708210166	VIPIN KASHYAP	9	11.5	11.5	5	4	21
41.	1708210167	VISHAKHA RASTOGI	16.5	10.5	16.5	5	5	27
42.	1708210168	VISHAKHA TANDON	19	14.5	19	5	5	29
43.	1708210169	VISHAL KUMAR	9	12.5	12.5	5	5	23
44.	1708210170	VISHAL KUMAR	D	10.5	10.5	5	5	21
45.	1708210171	VRATIKA GUPTA	10	9.5	10	5	5	20

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

46.	1708210172	YASH CHAUDHARY	D	11	11	5	5	**21**
47.	1708210173	ZAINAB AZEEM	17.5	14	17.5	5	5	**28**
48.	1508210100	PRANAV BHATNAGAR	D	10.5	10.5	5	5	**21**
49	1808210901	Aanchal Kumari	D	8.5	8.5	5	5	**19**
50	1808210902	Aryan Thakur	D	10.5	10.5	5	5	**21**
51	1808210903	Mangalam Sharma	D	12.5	12.5	5	5	**23**
52	1808210904	Shivam Kumar	9.5	7.5	9.5	5	5	**20**

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Course Name	Compiler Design
Course Code	RCS602
Batch	2017 2021
Semester	6
Session	2019 2020
L:T:P	3.1.0

CO Attainment

Course Code	CO	CO Attainment
RCS602	CO1	69.77
	CO2	68.15
	CO3	70.03
	CO4	70.68
	CO5	70.2

CO-PO Mapping

Course Code	CO		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
RCS602	CO1	RCS602.1	3		2	3	2	2				1		3
	CO2	RCS602.2	3	3		3	3							2
	CO3	RCS602.3	3	2		2								1
	CO4	RCS602.4	2	1										1
	CO5	RCS602.5	3	2		3	2					1		3
Mapping Strength		RCS602	2.8	2	2	2.75	2.3333	2				1		2

PO Attainment

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Relevant COs	CO1 CO2 CO3 CO4 CO5	CO2 CO3 CO4 CO5	CO1	CO1 CO2 CO3 CO5	CO1 CO2 CO5	CO1				CO1 CO5		CO1 CO2 CO3 CO4 CO5
Average of attainments of Relevent COs	69.77	69.77	69.77	69.54	69.37	69.77				69.99		69.77
PO Attainment Attainment of PO = (Actual Mapping strength/Maximum Mapping Strength)* Average of attainments of relevent COs	65.12	46.51	46.51	63.75	53.95	46.51				23.33		46.51
PO Attainment (normalized to 3)	1.95	1.4	1.4	1.91	1.62	1.4				0.7		1.4

CO-PSO Mapping

Course Code	CO		PSO1	PSO2
RCS602	CO1	RCS602.1	3	3
	CO2	RCS602.2	3	3
	CO3	RCS602.3	2	2
	CO4	RCS602.4	2	2
	CO5	RCS602.5	3	3
Mapping Strength		RCS602	2.6	2.6

PSO Attainment

PSO	PSO1	PSO2
Relevant COs	CO1 CO2 CO3 CO4 CO5	CO1 CO2 CO3 CO4 CO5
Average of attainments of Relevent COs	69.77	69.77
PSO Attainment Attainment of PSO = (Actual Mapping strength/Maximum Mapping Strength)*	60.47	60.47
PSO Attainment (normalized to 3)	1.81	1.81

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Course Name	Compiler Design
Course Code	RCS602
Batch	2017 2021
Semester	6
Session	2019 2020
L:T:P	3.1.0

CO Attainment Gap

Course Code	CO	CO Targets	CO Attainment	CO Attainment Gap (Target - Attainment)
RCS602	CO1	57	69.77	-12.77
	CO2	57	68.15	-11.15
	CO3	57	70.03	-13.03
	CO4	57	70.68	-13.68
	CO5	57	70.2	-13.20

If Gap > 0 : Target not attained
If Gap ≤ 0 : Target attained

Closure of Quality Loop

Course Code	CO	CO Targets	CO Attainment Gap	Action proposed to bridge the gap where targets are not achieved	Modification of targets where Achieved
RCS602	CO1	57	-12.77		Target is increased to 58%
	CO2	57	-11.15		Target is increased to 58%
	CO3	57	-13.03		Target is increased to 58%
	CO4	57	-13.68		Target is increased to 58%
	CO5	57	-13.20		Target is increased to 58%

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Tricky Assignment Questions for Practice

1. Compute FIRST and FOLLOW :-

$E \to TA$
$A \to +TA \mid \epsilon$
$T \to FB$
$B \to *FB \mid \epsilon$
$F \to (E) \mid id$

$FIRST(E) = \{(, id\}$
$FIRST(A) = \{+, \epsilon\}$
$FIRST(T) = \{(, id\}$
$FIRST(B) = \{*, \epsilon\}$
$FIRST(F) = \{(, id\}$

$FOLLOW(E) = \{\$,)\}$
$FOLLOW(A) = \{\$,)\}$
$FOLLOW(T) = \{+, \$,)\}$
$FOLLOW(B) = \{+,), \$\}$
$FOLLOW(F) = \{+,), *, \$\}$

2. Consider the grammar :-

$S \to AaAb \mid BbBa$
$A \to \epsilon$
$B \to \epsilon$

Test whether grammar is LL(1) or not.

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

3. Compute FIRST and FOLLOW :-

$S \to A$
$A \to aB \mid Ad$
$B \to bBC \mid f$
$C \to g$

$FIRST(S) = \{a\}$
$FIRST(A) = \{a\}$
$FIRST(B) = \{b, f\}$
$FIRST(C) = \{g\}$

$FOLLOW(S) = \{\$\}$
$FOLLOW(A) = \{d\}$
$FOLLOW(B) = \{g\}$
$FOLLOW(C) = \{g\}$

4. Compute FIRST and FOLLOW :-

$S \to aBDh$
$B \to cC$
$C \to bC \mid \epsilon$
$D \to EF$
$E \to g \mid \epsilon$
$F \to f \mid \epsilon$

$FIRST(S) = \{a\}$
$FIRST(B) = \{c\}$
$FIRST(C) = \{b, \epsilon\}$
$FIRST(D) = \{g, f, \epsilon\}$
$FIRST(E) = \{g, \epsilon\}$
$FIRST(F) = \{f, \epsilon\}$

$FOLLOW(S) = \{\$\}$
$FOLLOW(B) = \{g, f, h\}$
$FOLLOW(C) = \{g, f, h\}$
$FOLLOW(D) = \{h\}$
$FOLLOW(E) = \{f, h\}$
$FOLLOW(F) = \{h\}$

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad 244001

$S \rightarrow CC$

$C \rightarrow cC \mid d$

$C \rightarrow \alpha C \beta$

$First(C) = \{c, d\}$

Construct SLR parsing table.

$Follow(S) = \{\$\}$

$Follow(C) = Follow(S) = \{\$, c, d\}$

1. Find whether the grammar is SLR or not :-

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

State	ACTION			GOTO		
	a	b	$	S	A	B
0	R3/R4	R3/R4		1	2	3
1			accept			
2	S4					
3		S5				
4	R3	R3			6	
5	R4	R4				7
6		S8				
7	S9					
8		R1				
9		R2				

2. Consider the grammar :-

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

and construct LALR parsing table.

State	ACTION			GOTO	
	a	b	$	S	A
0	S_{36}	S_{47}		1	2
1			accept		
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		

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology

3. Design SLR parsing table :-

$S \to AA$
$A \to aA$
$A \to b$

State	ACTION			GOTO	
	a	b	$	S	A
0	S3	S4		1	2
1			accept		
2	S3	S4			5
3	S3	S4			6
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

4. Construct SLR parsing table :-

$A \to aAa \mid bAb \mid ba$

State	ACTION			GOTO
	a	b	$	A
0	S2	S3		1
1			accept	
2	S2	S3		4
3	S6	S3		5
4	S7			
5		S8		
6	S2/R3	S3/R3	R3	4
7	R1	R1	R1	
8	R2	R2	R2	

5. Construct LALR(1) parsing table-

$S \to Ba \mid bBc \mid dc \mid bda$
$B \to d$

State	ACTION					GOTO	
	a	b	c	d	$	S	B
0		S3		S4		1	2
1					acc		
2	S5						
3				S7			2
4	R5		S8				
5					R1		
6	S10		S9				
7			R5				
8					R3		
9					R2		
10					R4		

COMPILER DESIGN
(RCS-602)

Instructors:
Ms.Priyanka Goel
Asst.Prof.
Deptt of CS & E

Text books to refer:

1. "Principles of Compiler Design" by Aho, Ullman
2. "Compilers: Principles , Techniques & Tools" by Aho, Sethi, Ullman
3. "Principles of Compiler Design" by A.A. Puntambekar

Course Objectives

- To introduce the major concept areas of language translation and compiler design.
- To describe the utilization of formal Grammar using Parser representations, especially those on bottom-up and top-down approaches and various algorithms
- To enrich the knowledge in various phases of compiler and use of symbol table.
- To understand, design code generation schemes, optimization of codes and runtime environment
- To provide practical programming skills necessary for constructing a compiler.

UNIT 1: Introduction to Compiler

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

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Types of Translators

- **Interpreter:** It transforms a programming language into some simplified form(called intermediate code) which can be understood by the machine or can be easily executed.
- There are several types of interpreters: the syntax-directed interpreter (i.e., the Abstract Syntax Tree (AST) interpreter), bytecode interpreter, and threaded interpreter Just-in-Time (a kind of hybrid interpreter/compiler), and a few others.

4

Compiler Vs Interpreter

Parameter	Compiler	Interpreter
Input	It takes an entire program at a time.	It takes a single line of code or instruction at a time.
Output	It generates intermediate object code.	It does not produce any intermediate object code.
Working mechanism	The compilation is done before execution.	Compilation and execution take place simultaneously.
Speed	Comparatively faster	Slower
Memory	Memory requirement is more due to the creation of object code.	It requires less memory as it does not create intermediate object code.
Errors	Display all errors after compilation, all at the same time.	Displays error of each line one by one.
Error detection	Difficult	Easier comparatively
Code Length	Generally larger in code	Smaller Code
Programming languages	C, C++, C# etc	Java, PHP, Perl, Python etc

5

Types of Translators

- Assembler: If the source language is assembly language and the target language is machine language, then the translator is called assembler.
- The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory(relocatable).

6

Types of Translators

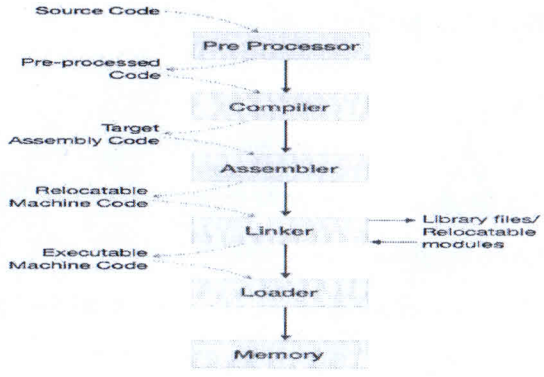
- **Preprocessor:** It takes programs in one high level language and converts them into equivalent programs in another high level language.

It deals with two functions:

a) Macro-processing(macro expansion): Macros are shorthands for longer constructs. For a macro, there are two aspects:
macro definition: e.g. #define PI 3.14
macro use: e.g. Area=PI*r*r

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Context of Compiler



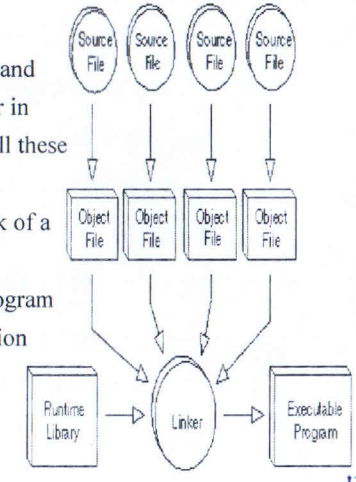
A Language Processing System

12

Context of Compiler

- **Linker**

It is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.



13

Context of Compiler

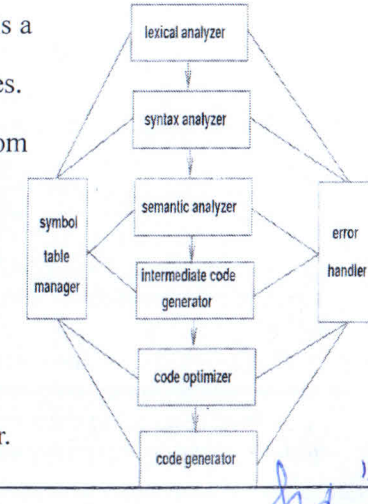
- **Loader**

It is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space(absolute) for it. It initializes various registers to initiate execution.

14

Structure of Compiler

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.



15

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Lexical Analysis (Scanner)

- So, Token stream generated will be:

IF([constant,341) EQ [identifier,750]) GOTO [label,543]

Where symbol table will be having following entries:

341	Constant, integer, value =5
	:
543	Label, value =100
	:
750	Identifier, integer, value =MAX

- Tokens can be generated using regular expressions .
- LEX and FLEX are two such automatic tools for lexical analysis.

20

Syntax Analysis(Hierarchical Analysis/Parsing)

- The syntax analyzer groups tokens together into syntactic structures.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

What is Symbol Table?

- **Symbol table** is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.
- **Symbol table** is used by both the analysis and the synthesis parts of a compiler.

Syntax Analysis(Hierarchical Analysis/Parsing)

- After lexical analysis (scanning), we have a series of tokens.
- In syntax analysis (or parsing), we want to interpret what those tokens mean.
- It is done by checking the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not.
- This is done by building a data structure, called a Parse tree or Syntax tree(compressed form of Parse tree).
- The parse tree is constructed by using the pre-defined Grammar of the language and the input string.
-

Syntax Analysis...

- ***If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax.***

Thus, The Parser has two functions:

- It checks that the token appearing in its input occurs in a pattern permitted by the specification of the source language.
- It make explicitly the hierarchical structure of the incoming token stream.(Parse tree or syntax tree)

Syntax Analysis...

- For construction of parse tree(or syntax tree), context free grammar defined for the particular programming language, is used.
- Parse tree can be constructed using the production rules of that CFG.

e.g. Consider the statement: total=count+rate*10

For such arithmetic expressions, Productions rules defined are:

expr → identifier
expr → number
expr → expr1+expr2
expr → expr1 * expr2
expr → (expr1)

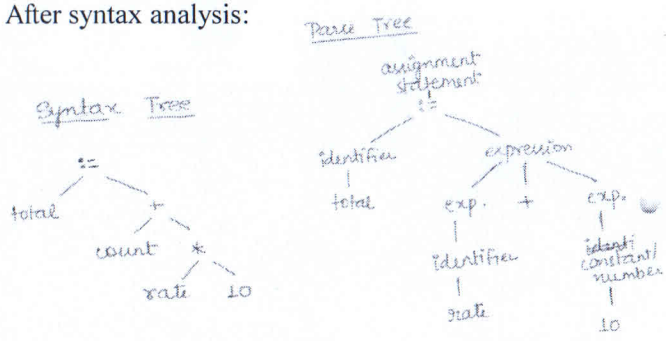
Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Syntax Analysis...

total=count+ rate * 10

After lexical analysis: id1=id2+id3*10

After syntax analysis:



25

Syntax Analysis...

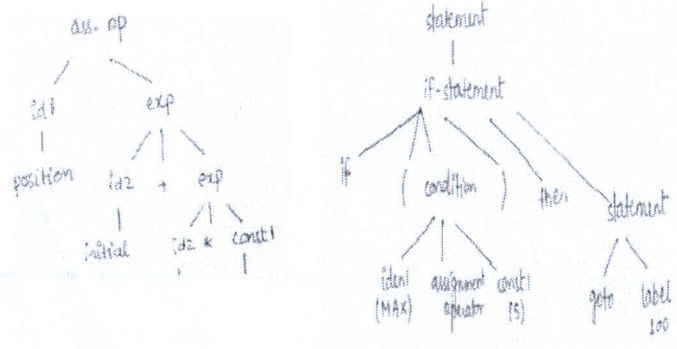
- Parsers are deterministic Push down Automata.
- Parsers cannot handle context sensitive features of programming languages such as:
  - Variable declaration before use
  - Type matching on both sides of assignment operators.
  - Parameter types and number match in definition and use.

26

Syntax Analysis...

Some more examples:

position = initial+ rate * 60 if (MAX = 5) then goto 100



27

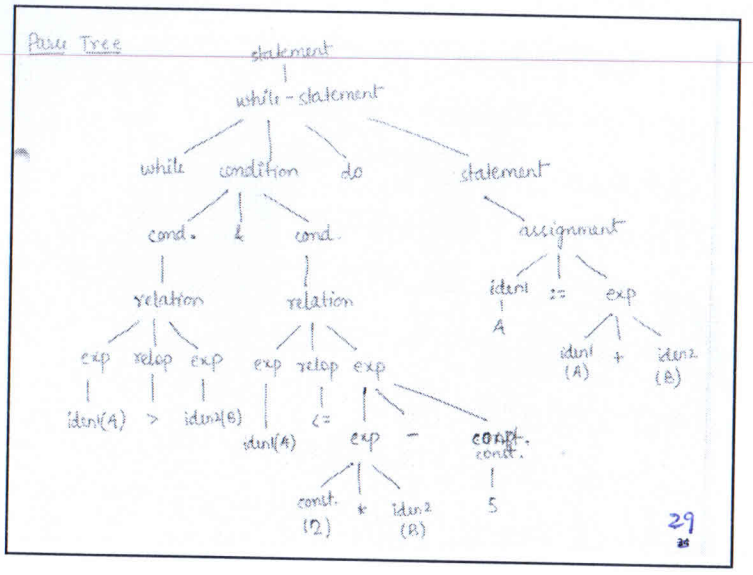
while A>B & A<=2*B-5 do

A=A+B

Token Stream : while [id1, 329] > [id2, 279] & [id1, 329] <= [const1, 492] * [id2, 279] - [const2, 537] do [id1, 329] := [id1, 329] + [id2, 279]

28

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001



Semantic Analysis

- Semantic analysis checks whether the parse tree constructed follows the rules of language.
- For example, assignment of values is between compatible data types, or adding string to an integer.
- Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc.
- The semantic analyzer produces an annotated syntax tree as an output.

30

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Semantic Analysis

- Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e., that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.
- The semantic analyzer produces an annotated syntax tree as an output.
- A parse tree showing the values of attributes at each node is called an annotated parse tree

30

Semantic Analysis...

- Semantic analysis typically involves:
  - Type checking – Data types are used in a manner that is consistent with their definition (i. e., only with compatible data types, only with operations that are defined for them, etc.)
  - Label Checking – Labels references in a program must exist.
  - Flow control checks – control structures must be used in their proper fashion (no breaks outside a loop or switch statement, etc.)

31

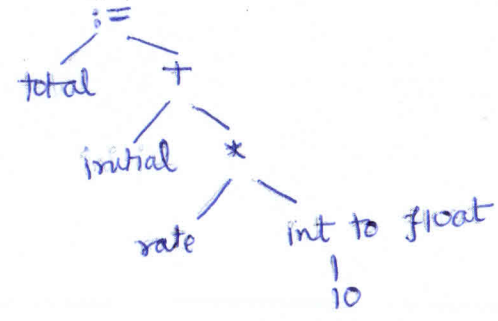
Static vs. Dynamic Semantics

- The static semantics of a language is indirectly related to the meaning of programs during execution. Its names comes from the fact that these specifications can be checked at compile time.
- Dynamic semantics refers to meaning of expressions, statements and other program units. Unlike static semantics, these cannot be checked at compile time and can only be checked at runtime.

32

Semantic Analysis...

e.g. total=count+ rate * 10



33

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Attribute Grammar

- Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.
- Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.
- Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.

34

Attribute Grammar...

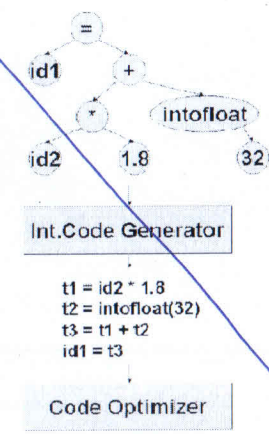
- Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.
- $E \rightarrow E + T$ { E.value = E.value + T.value }
- The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted.
- Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

35

Intermediate Code Generation

- After semantic analysis the compiler generates an intermediate code of the source code for the target machine.
- It act as a glue between front-end and backend (or source and machine codes)
- An intermediate code must be easy to produce and easy to translate to machine code.
- It should not contain any machine-specific parameters(registers, addresses, etc.)
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation.

Translation Overview - Intermediate Code Generation



Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Intermediate Code Generation

- After semantic analysis the compiler generates an intermediate code of the source code for the target machine.
- It act as a glue between front-end and backend (or source and machine codes)
- An intermediate code must be easy to produce and easy to translate to machine code.
- It should not contain any machine-specific parameters(registers, addresses, etc.)
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation.

36

Three Address Codes

•Three address code consists of a sequence of instructions, each of which can have atmost three operands.

e.g. The three-address code for a+b*c-d/(b*c) :

```
t1 = b*c
t2 = a+t1
t3 = d/t1
t5 = t2-t3
```

37

Types of Intermediate Code Representations

•Intermediate representations are usually categorized according to where they fall between a high-level language and machine code.

•IRs that are close to a high level language are called high level IRs.

• IRs that are close to assembly language are called low level IRs.

•High-level IRs usually preserve information such as loop structure and if-then-else statements.

•They tend to reflect the source language they are compiling more than lower-level IRs.

38

Types of Intermediate Code Representations...

•Medium-level IRs often attempt to be independent of both the source language and the target machine.

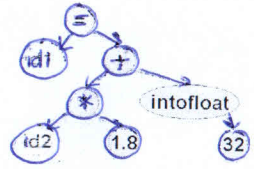
•Low-level IRs tend to reflect the target architecture very closely, and as such are often machine dependent.

Original	*High IR*	*Mid IR*	*Low IR*
float a[10][20]; a[i][j+2];	t1 = a[i, j+2]	t1 = j + 2 t2 = i * 20 t3 = t1 + t2 t4 = 4 * t3 t5 = addr a t6 = t5 + t4 t7 = *t6	r1 = [fp - 4] r2 = [r1 + 2] r3 = [fp - 8] r4 = r3 * 20 r5 = r4 + r2 r6 = 4 * r5 r7 = fp - 216 f1 = [r7 + r6]

39

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Translation Overview - Intermediate Code Generation



Int.Code Generator

```
t1 = id2 * 1.8
t2 = intofloat(32)
t3 = t1 + t2
id1 = t3
```

Code Optimizer

40

Code Optimization

- It is an optional phase designed to improve the intermediate code so that the final object program runs faster and/or takes less space.
- Its output is another intermediate code program that does the same job as the original, but in a way that saves time and/or space.
- There are two types of code optimization:

a) Local Optimization: where local transformations are applied in the program.

e.g. A=B+C+D
E=B+C+E can be written as: T1=B+C
A=T1+D
E=T1+E

41

Code Optimization...

b) Loop Optimization: Loops are good targets of optimization because programs spend most of their time in loops.

One approach is to move a computation out of the loop, which produces same result each time around the loop. Such computations are called loop invariants.

e.g. for(i=1;i<=10;i++) can be optimized as:

```
{                          j=k;
:                          for(i=1;i<=10;i++)
 j=k;                      {
:                            :
}                            :
                           }
```

42

Code Optimization(One more example)

```
t1 = id2 * 1.8
t2 = intofloat(32)
t3 = t1 + t2
id1 = t3
```

↓

Code Optimizer

↓

```
t1 = id2 * 1.8
id1 = t1 + 32.0
```

↓

Code Generator

43

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Machine Independent Code Optimization

•In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations.

```
do
{item=10;
value = value + item;
} while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
item = 10;
do
{ value = value + item;
} while(value<100);
```

44

Machine-dependent Code Optimization

•Machine-dependent optimization is done after the target code has been generated and then the code is transformed according to the target machine architecture.

•It involves CPU registers and may have absolute memory references rather than relative references.

•Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

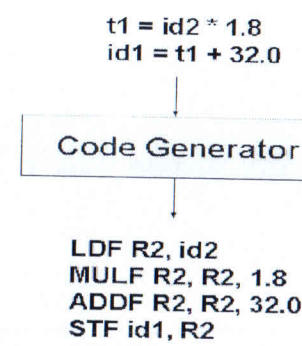
45

Code Generation

- In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.
- The object code is produced by deciding the memory locations for data, selecting code to access each datum and selecting the registers in which each computation is to be done.
- The low-level machine code should have following properties:
  - It should carry the exact meaning of the source code.
  - It should be efficient in terms of CPU usage and memory management.

46

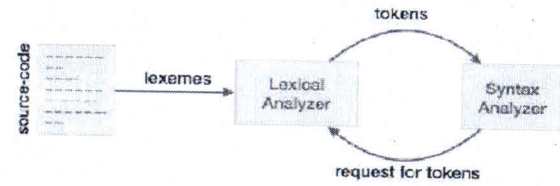
Code Generation- An Example



47

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

LEXICAL ANALYSIS
(Linear Analysis)



1. Lexical analysis is the first phase of a compiler.
2. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.
3. If the lexical analyzer finds a token invalid, it generates an error.

How tokens are recognized?

- All possible lexemes that can appear in code written in a programming language, are described in the **specification of that programming language** as a set of rules called **lexical grammar**.
- The rules in the lexical grammar are often expressed with a set of regular definitions.
- A regular definition is of the form

<element_name> = <production_rule>

where <element_name> is the name given to a symbol or a lexeme that can be encountered in the programming language and <production_rule> is a **regular expression** describing that symbol or lexeme.

How tokens are recognized?

- For example, the following regular definition defines a letter as any lowercase or uppercase alphabet character:

letter = [a-z | A-Z]

- Rules in the lexical grammar are often transformed into automata called **finite state machines** (*FSM*).
- The scanner then simulates the finite state machines to recognize the tokens.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

How tokens are recognized?

•In the following regular definitions , the definition *identifier* reuses the definitions *letter* and *digit*, in its production rule as if *letter* and *digit* were symbols, to define an identifier as *any string starting with a letter or an underscore,* followed by *zero or more occurrences of a letter, a digit or an underscore:*

```
letter      = [a-zA-Z]
digit       = [0-9]
identifier  = (letter | _) (letter | digit | _)*
```

Language:

- Languages are considered as finite sets, and mathematically set operations can be performed on them.
- Finite languages can be described by means of regular expressions.

Regular Expressions :

- Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols.
- The grammar defined by regular expressions is known as **regular grammar**.
- The language defined by regular grammar is known as **regular language.**

Operations on Language.....

The various operations on languages are:

- **Union** of two languages L and M is written as
  L U M = {s | s is in L or s is in M}
- **Concatenation** of two languages L and M is written as
  LM = {st | s is in L and t is in M}
- The **Kleene Closure** of a language L is written as
  L* = Zero or more occurrence of language L.

Operations on Regular Expressions.....

If r and s are regular expressions denoting the languages L(r) and L(s), then

- Union : (r)|(s) is a regular expression denoting L(r) U L(s)
- Concatenation : (r)(s) is a regular expression denoting L(r)L(s)
- Kleene closure : (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r)

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Precedence and Associativity of RE operations

- *, concatenation (.), and | (pipe sign) are **left associative**
- * has the highest precedence
- Concatenation (.) has the second highest precedence.
- | (pipe sign) has the lowest precedence of all.

If x is a regular expression, then:

- **x*** means zero or more occurrence of x.
  i.e., it can generate { €, x, xx, xxx, xxxx, … }
- **x^+** means one or more occurrence of x.
  i.e., it can generate { x, xx, xxx, xxxx … } or x.x*
- **$x^?$** means at most one occurrence of x
  i.e., it can generate either {x} or {€}.

Finite State Machines

•A Finite State Machine or FSM is an abstract machine that is in one and only one state at any point in time.

•The FSM can change from one state to another as a result of an event.

•Changing from a state to another is called a transition.

• FSM has a finite number of states comprising an initial state and a set of accepting states.

•The FSM moves from one state to another by consuming one of the characters or elements in the regular expression.

•Following the transitions from the initial state to one of the accepting states yields a valid string described by the regular expression.

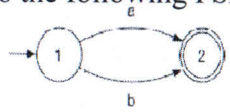
Finite State Machines...

- The mathematical model of finite automata consists of:
  - ✓Finite set of states (Q)
  - ✓Finite set of input symbols (Σ)
  - ✓One Start state (q0)
  - ✓Set of final states (qf)
  - ✓Transition function (δ)
- The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), $Q \times \Sigma \rightarrow Q$

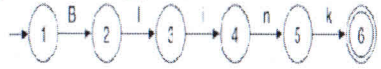
Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Finite State Machines...

•For example, the regular expression a | b can be converted into the following FSM:

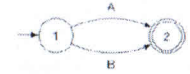


•Following the transitions from the initial state 1 to the accepting state 6 on the following FSM can yield only one string, Blink.



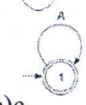
More Examples...

A | B

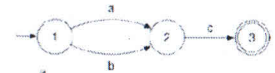


AB

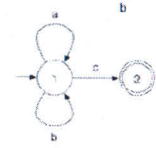
A*



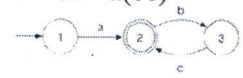
R = (a|b)c



R = (a|b)*c

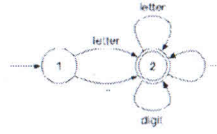


R = a(bc)*



More Examples...

FSM for identifier

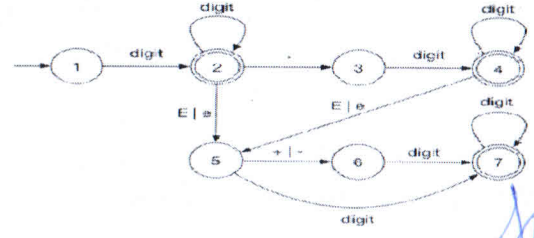


Transition table :

	letter	digit	_
1	2	-	2
2	2	2	2

Regular Expression for a number

```
digit     = [0-9]
digits    = digit digit*
fraction  = . digits | ε
exponent  = ((E | e) (+ | - | ε)) digits) | ε
number    = digits fraction exponent
```



Longest Match Rule

•When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

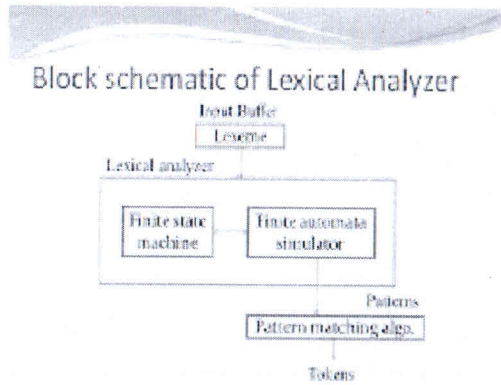
e.g. int intvalue;

•While scanning both lexemes till 'int', the lexical analyzer cannot determine whether it is a keyword int or the initials of identifier int value.

•The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

- The lexical analyzer also follows rule priority where a reserved word, e.g., a keyword, of a language is given priority over user input.
- That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

Lexical Analysis



How to construct Finite Automata from given RE?

Step 1: Convert RE to NFA (Thompson Construction Algorithm)

•Thompson's construction is an algorithm for transforming a regular expression into an equivalent nondeterministic finite automaton (NFA). This NFA can be used to match strings against the regular expression.

Step 2: Convert NFA to DFA (Subset Construction Algorithm)

•A major step of this algorithm is to find Є-closure for each state of NFA.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Formal Definition of an NDFA

•An NDFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where :

Q is a finite set of states.

∑ is a finite set of symbols called the alphabets.

δ is the transition function where δ: Q × ∑ → ~~2^Q~~ 2^{Q}

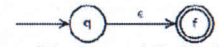
(Here the power set of Q (~~2^Q~~ 2^{Q}) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)

q0 is the initial state from where any input is processed (q0 ∈ Q).

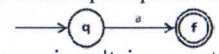
F is a set of final state/states of Q (F ⊆ Q).

Thompson's Construction Algorithm

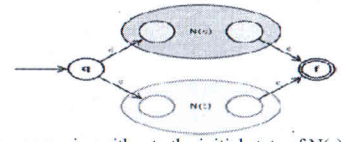
- Following steps are followed:
  - The empty-expression ε is converted to



  - A symbol **a** of the input alphabet is converted to



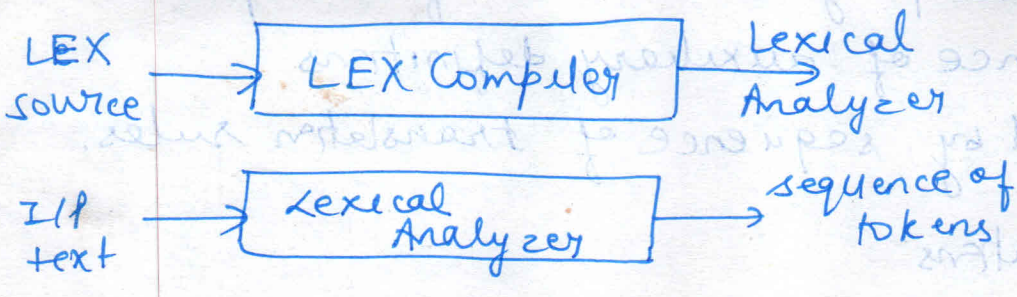
  - The union expression **s|t** is converted to



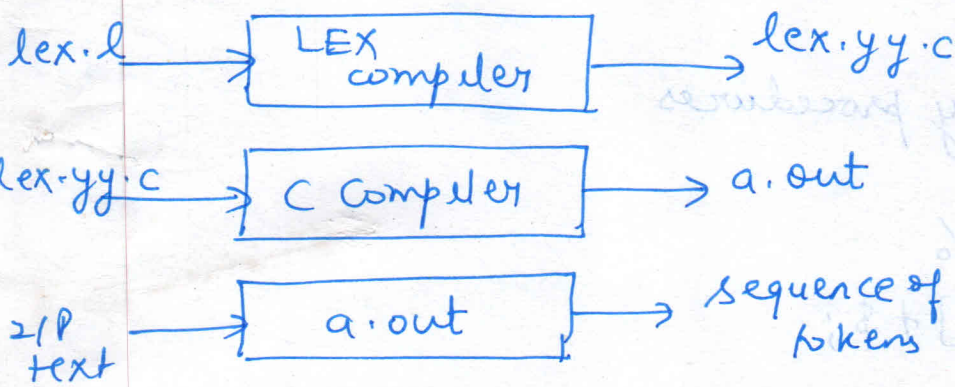
where state q goes via ε either to the initial state of N(s) or N(t). Their final states become intermediate states of the whole NFA and merge via two ε-transitions into the final state of the NFA.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

LEX (A language for specifying Lexical Analyzer)



Its other representation is —



→ LEX is a tool for automatically generating lexical analyzers.

→ A LEX source program is a specification of lexical analyzer consisting of a set of RE together with an action for each RE.

→ This action is a piece of code which is to be executed whenever a token specified by corresponding RE is recognized.

→ Typically an action will pass an indication of token found to the parser.

→ Perhaps with side effects such as making an entry in the symbol table.

→ The output of LEX is a lexical analyser program constructed from LEX source specification.

LEX Specification

→ A LEX source program consists of two parts —
a) a sequence of auxiliary definitions
b) followed by sequence of translation rules.

declarations
%%
translation rules
%%
auxiliary procedures

e.g:
```
%%
[ \t]+$;
%%
```

Auxiliary definitions

→ The Auxiliary definitions are statements of the form —

$D_1 = R_1$
$D_2 = R_2$
:
$D_n = R_n$

Where each D_i is a distinct name and each R_i is RE whose symbols are chosen from $\Sigma \cup (D_1, D_2 \ldots D_{n-1})$
i.e. characters of previously defined names.

→ The D_i's are shorthand names for REs, Σ is our input symbol alphabet.

→ To avoid confusion, lowercase strings are used for D_i's.

e.g. we can define the class of identifier for a typical program language with the following sequence of auxiliary definitions —

letter = A | B | . . . | Z

digit = 0 | 1 | . . . | 9

identifier = letter (letter + digit)*

Translation rules

→ The translation rules of LEX program are statement of the form —

P_1 {A_1}

P_2 {A_2}

.

.

P_n {A_n}

where each P_i is a RE called pattern over the Σ, consisting of alphaset set and auxiliary definition names. The pattern describe the form of tokens.

→ Each A_i is a program fragment describing what action the lexical analyzer should take when token P_i is found.

e.g. Auxiliary definition

letter = A | B | . . . | Z

digit = 0 | 1 | . . . | 9

translation rules

BEGIN	{return 1}
END	{return 2}
IF	{return 3}
THEN	{return 4}
ELSE	{return 5}

letter(letter + digit)*	{ LEXVAL = INSTALL, return 6 }
digit+	{ LEXVAL = INSTALL, return 7 }
<	{ LEXVAL = 1, return 8 }
<=	{ LEXVAL = 2, return 9 }
=	{ LEXVAL = 3, return 10 }
<>	{ LEXVAL = 4, return 11 }
>=	{ LEXVAL = 5, return 12 }

YACC

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma `","` is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and

communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name  :  'J' 'a' 'n'   ;
month_name  :  'F' 'e' 'b'   ;

        . . .

month_name  :  'D' 'e' 'c'   ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and month_name would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a month_name was seen; in this case, month_name would be a token.

Literal characters such as ``,'' must also be passed through the lexical analyzer, and are also considered tokens.Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date  :  month '/' day '/' year  ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be ``slipped in'' to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to

include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%" marks. (The percent ``%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also;

thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A  :  BODY  ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ``.", underscore ``_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes ``'". As in C, the backslash ``\" is an escape character within literals, and all the C escapes are recognized. Thus

```
'\n'    newline
'\r'    return
'\''    single quote  `'''
'\\'    backslash  `\''
'\t'    tab
'\b'    backspace
'\f'    form feed
'\xxx'  `xxx'' in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar ``|" can be used to avoid rewriting the left hand side

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :   ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token   name1  name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start   symbol
```

The end of the input to the parser is signaled by a special token, called the endmarker. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate

2: Actions

With each grammar rule, the user may associate actions to beperformed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces ``{" and ``}". For example,

```
A       :       '('  B  ')'
                     {      hello( 1, "abc" );  }
```

and

```
XXX     :       YYY  ZZZ
                     {      printf("a message\n");
                            flag = 25;    }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol ``dollar sign" ``$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudovariable ``$$" to some value. For example, an action that does nothing but return the value 1 is

```
{  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks ``%{" and ``%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{  int variable = 0;  %}
```

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in ``yy"; the user should avoid such names.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The function returns an integer, the token number, representing

the kind of token read. If there is a value associated with that token, it should be assigned to the external variable yylval.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the ``# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                . . .
        case '0':
        case '1':
          . . .
        case '9':
                yylval = c-'0';
                return( DIGIT );
                . . .
                }
        . . .
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

5: Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr    :       expr  '-'  expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together

with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured.

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.

2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

6: Precedence

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.

2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

7: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called y.tab.c on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called yyparse; it is an integer valued function. When it is called, it in turn repeatedly calls yylex, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) yyparse returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, yyparse returns the value 0.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of main and yyerror. The name of this library is system dependent; on many systems the library is accessed by a -ly argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
        return( yyparse() );
        }
```

and

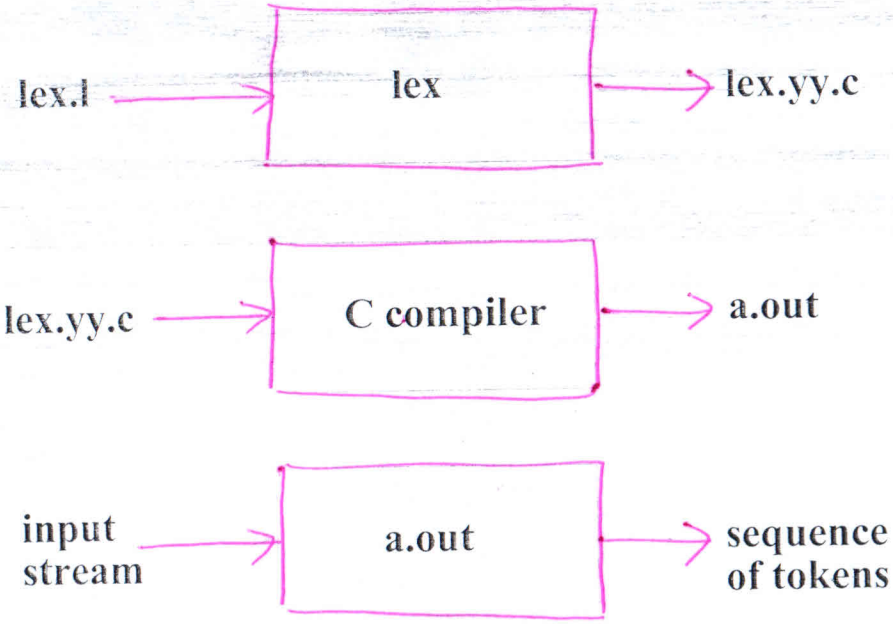
```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to yyerror is a string containing an error message, usually the string ``syntax error''. The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable yychar contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the main program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable yydebug is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

LEX

Creating a Lexical Analyzer with lex




```
lex foo.l; cc lex.yy.c -ll; a.out < test.c;
```

lex specification

A lex program consists of three parts:

```
declarations
%%
translation rules
%%
auxiliary procedures
```

Example:

```
%%
[ \t]+$ ;
%%
```

Three Parts

- declarations: variables, constants, regular definitions
- translation rules: sequence of p_i $\{action_i\}$, where p_i is a regular expression and $action_i$ is a C program fragment describing the action that the lexer takes when the token p_i is found
- auxiliary procedures: functions needed by the actions

Cooperation between Lexer and Parser

- When activated by the parser, the lexer matches the longest lexeme and perform an action
- Typically, the action gives control back to parser via `return(Token_Type)`.
- Otherwise, lexer finds more lexemes until an action returns.
- Lexer returns token to the parser but can pass an attribute via a global variable `yylval`.
- Two reserved variables `yytext` (pointer to the first char of lexeme) and `yyleng` (length of the string)

An Example

```
%{
   /* whatever is included here will be included in lex.yy.c */
   # include subc.h
   # include y.tab.h
int commentdepth = 0;
%}
Letter          [a-zA-a]
Digit           [0-9]
Id              {Letter}({Letter}|{Digit})*
%%
{Id}            {yylval = install_id(); return(ID);
%%
install_id() { /* function to include the id in the symbol table */
```

Operator Characters in `lex`

- " " : take as text characters
- \ : make operators as texts
- ^ : complemented character set (Ex: [^ abc])
- . : arbitrary character (Ex: . `printf(''bad input'');`)
- Context sensitivity: ^ and $: if the first character of an expression is ^ , it is matched only at the beginning of a line; if the last character is $, the expression is matched only at the end of a line

Syntactic specification of Programming Languages

→ for the syntactic specification of a programming language a notation called context free grammar (also called BNF, Backus Naur form) is used.

→ This notation has following advantages to be used as a method of specification for syntax of language —

a) It gives a precise, yet easy to understand syntactic specification for the programs of a particular programming language.

b) An efficient parser can be constructed automatically from a properly designed grammar.

c) A grammar imparts a structure to a program that is useful for its translation into object code and for the detection of errors.

Context free grammars

→ Certain programming language constructs can be defined recursively.

→ A CFG involves four quantities — terminals, nonterminals, a start symbol and productions.

Terminals :- are the basic symbols of which strings in the language are composed. Tokens generated in lexical analysis can be considered as terminals in CFG.

Nonterminals :- are special symbols that denote set of strings e.g. the syntactic categories such as statement, expression & statement-list are nonterminals.

* One nonterminal is selected as **start symbol** and it denotes the language in which we are interested.
* Other nonterminals define other set of strings defining language
* also help to provide a hierarchical structure of language.

Productions (Rewriting rules) :- define the ways in which the syntactic categories may be built up from one another & from the terminals.

Each production consists of a nonterminal, followed by an arrow, followed by a string of nonterminals & terminals

e.g. statement → if expression then statement else statement
expression → expression + expression
statement → begin statement-list end
statement-list → statement
statement-list → statement ; statement list

Note :- We require that each rewriting rule have a known no. of symbols with no ellipses (. . . .) permitted.

Notational conventions

(1) following symbols are usually nonterminals—
* lower-case names such as expression, statement, operator etc
* italic capital letters near the beginning of the alphabet
* letter S, which, when it appears, is usually the start symbol

(2) following symbols are usually terminals—
* single lowercase letters a, b, c . . .
* operator symbols such as +, −, . . .
* punctuation symbols such as parentheses, comma etc. .
* digits 0, 1, . . . 9
* boldface ~~dog~~ strings such as id or if.

(3) Capital symbols near the end of alphabet represent grammar symbols i.e. either nonterminals or terminals

(4) Small letters near the end of alphabet represent strings of terminals.

(5) Lower case Greek letters α, β, γ represent

strings of grammar symbols.

(6) If $A \to \alpha_1, A \to \alpha_2, \ldots A \to \alpha_k$ are all productions with A on the left (these are called A-productions) we may write $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_k$ where $\alpha_1, \alpha_2, \ldots \alpha_k$ are called alternates for A.

(7) Unless otherwise stated, left side of first production is the start symbol.

Derivations and parse trees

→ The central idea to define a language is that productions may be applied repeatedly to expand nonterminals in a string of nonterminals and terminals.

e.g. Consider the following grammar for arithmetic expressions:

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

If we replace a single E by $-E$ then this action can be described as —

$$E \Rightarrow -E \quad (\text{i.e. E derives } -E)$$

Such a sequence of replacements is called derivation of a terminal from a non terminal.

e.g. $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$

→ In general,

$\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \to \gamma$ is a production and α and β are arbitrary strings of grammar symbols.

→ If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ then we say α_1 derives α_n.
↳ means derivation in one step

$\alpha \overset{*}{\Rightarrow} \alpha$ for any string α then
↳ derivation in zero or more steps.

→ If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$ then $\alpha \overset{*}{\Rightarrow} \gamma$

→ $\alpha \overset{+}{\Rightarrow} \beta$
↳ means derivation in one or more steps.

→ Given a context free grammar G with start symbol S, we can use $\overset{+}{\Rightarrow}$ relation to define L(G), the language generated by G.

→ Strings in L(G) may contain only terminal symbols of G.

→ A string of terminals w is in L(G) if and only if $S \overset{+}{\Rightarrow} w$ where w is called sentence of G.

→ If $S \overset{*}{\Rightarrow} \alpha$ where α may contain nonterminals then we say α is a sentential form of G.

→ There is often a degree of arbitrariness in the choice of replacement made in a derivation.

→ At any step of the derivation we may choose which nonterminal we wish to replace.

→ The derivations in which only the leftmost nonterminal in a sentential form is replaced at each step is called <u>leftmost derivations</u>.

→ <u>Rightmost derivations</u> are those in which the rightmost nonterminal is replaced at each step.

<u>e.g</u> Consider following grammar—

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

The sentence id + id * id has following derivations—

$E \Rightarrow E+E$
$\Rightarrow id + E$
$\Rightarrow id + E*E$
$\Rightarrow id + id*E$
$\Rightarrow id + id * id$
(Leftmost derivation)

$E \Rightarrow E+E$
$\Rightarrow E+E*E$
$\Rightarrow E+E*id$
$\Rightarrow E+id*id$
$\Rightarrow id+id*id$
(Rightmost derivation)

Parse trees

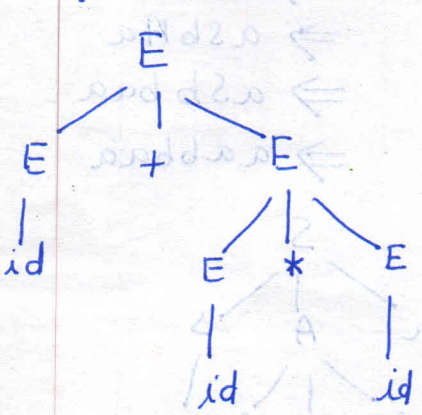
→ Parse trees are the graphical representation for derivations that filters out the choice regarding replacement order

→ It has the important purpose of making explicit the hierarchical syntactic structure of sentences that is implied by the grammar.

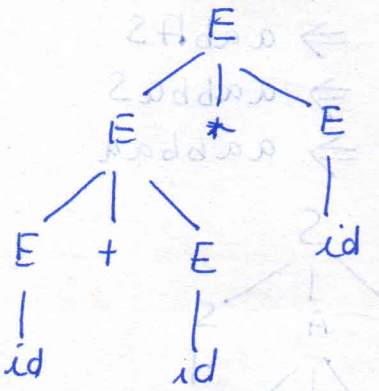
→ Each interior node of the parse tree is labeled by some nonterminal and the leaves of parse tree are labeled by nonterminals or terminals.

→ These constitute the sentential form called the yield or frontier of tree.

e.g $E \Rightarrow E+E$
$\Rightarrow id+E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$



$E \Rightarrow E*E$
$\Rightarrow E+E*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$



Mathematically, a derivation tree for a context free grammar $G_1 = (V_N, \Sigma, P, S)$ is tree satisfying following —

* Each vertex has a variable or terminal of $\in$.
* Root has label S.
* The label of an internal vertex is a variable
* Leaves must be reachable by means of production of grammar G_1.

Q Draw derivation tree for the following grammar —

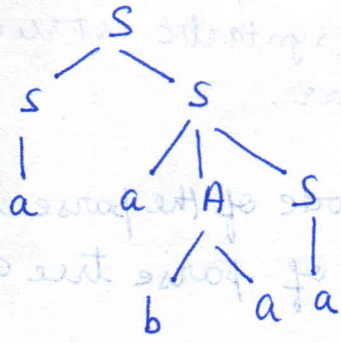
$S \rightarrow aAS \mid a \mid SS$

$A \rightarrow SbA \mid ba$

over the string $w = aabaa$.

Using leftmost derivation —

$S \Rightarrow SS$
$\Rightarrow aS$
$\Rightarrow aaAS$
$\Rightarrow aababS$
$\Rightarrow aababa$

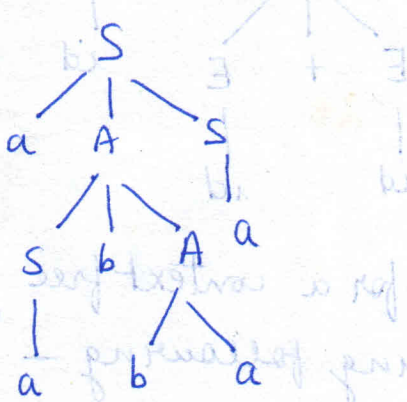


Q Construct leftmost and rightmost derivation for the string $w = aabbaa$ from following grammar —

$S \rightarrow aAS \mid a$
$A \rightarrow SbA \mid SS \mid ba$

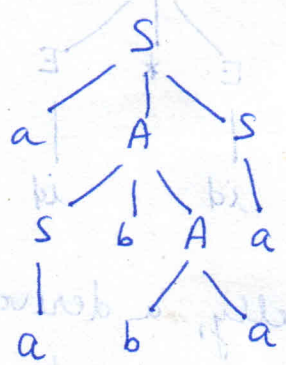
Leftmost derivation

$S \Rightarrow aAS$
$\Rightarrow aSbAS$
$\Rightarrow aabAS$
$\Rightarrow aabbaS$
$\Rightarrow aabbaa$



Rightmost derivation

$S \Rightarrow aAS$
$\Rightarrow aAa$
$\Rightarrow aSbAa$
$\Rightarrow aSbbaa$
$\Rightarrow aabbaa$



Note :- Derivation tree remains same whether it is left most or rightmost i.e. parse tree ignores the variations in the order in which symbols are replaced.

Ambiguity

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

→ for certain types of parsers, it is desirable that the

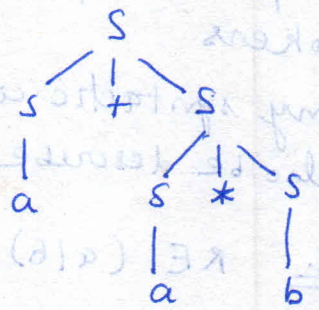
grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence.

Q :- Verify the following G is ~~un~~ambiguous :-

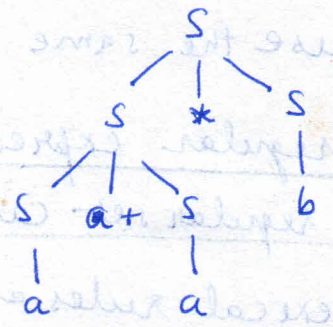
$S \rightarrow S+S \mid S*S \mid a \mid b$ for string $\omega = a+a*b$

Leftmost derivation

$S \Rightarrow S+S \quad \{S \rightarrow a\}$
$\Rightarrow a+S \quad \{S \rightarrow S*S\}$
$\Rightarrow a+S*S \quad \{S \rightarrow a\}$
$\Rightarrow a+a*S \quad \{S \rightarrow a\}$
$\Rightarrow a+a*b \quad \{S \rightarrow b\}$



$S \Rightarrow S*S \quad \{S \rightarrow S+S\}$
$\Rightarrow S+S*S \quad \{S \rightarrow a\}$
$\Rightarrow a+S*S \quad \{S \rightarrow a\}$
$\Rightarrow a+a*S \quad \{S \rightarrow b\}$
$\Rightarrow a+a*b$



Since there are two different leftmost derivation trees so above grammar G is ambiguous.

→ we can disambiguate an ambiguous grammar by specifying the associative & precedence of the arithmetic operators.

e.g. we wish to give the operators the following precedences in decreasing order :-

$-$ (unary minus)
$\uparrow$
$*$ /
$+$ $-$

Also, we wish $\uparrow$ to be right associative (i.e. $a \uparrow b \uparrow c$ is considered $a \uparrow (b \uparrow c)$) and other binary operators to be left associative (e.g. $a-b-c$ to be mean $(a-b)-c$)

Capabilities of Context free Grammars

This section indicates what programming language constructs can and cannot be described by CFG.

Regular Expressions Vs Context free Grammars

* Regular expressions are capable of describing the syntax of tokens
* Any syntactic construct that can be described by a RE can also be described by CFG.

e.g. RE $(a|b)(a|b|0|1)^*$ and the CFG $S \rightarrow aA|bA$, $A \rightarrow aA|bA|0A|1A|\epsilon$ describe the same language.

Why regular expressions are used inspite the fact that every regular set can be described by a CFG?

(1) The lexical rules are usually quite simple and there is no need of using a notation as powerful as CFG.

(2) With the RE notation, it is bit easier to understand what set of strings is being defined that it is to understand a language defined by ~~set of~~ collection of productions.

(3) It is easier to construct efficient recognizers from RE than from CFG.

(4) Separating the syntactic structure of a language into lexical and non lexical parts provides a convenient way of modularizing the front end of compiler into two manageable sized components.

→ Regular expressions are most useful for describing the structure of lexical constructs such as identifiers, constants, keywords etc.

→ On the other hand, CFG are useful in describing ~~nested~~ structures such as balanced parentheses, matching begin-end's etc.

Non Context free Language Constructs

→ There are certain languages that are not context free and that represent constructs of real programming languages.

e.g. $L_1 = \{ w c w \mid w \text{ is in } (a|b)^* \}$ is not ~~CFG~~ CFL.
(it represents the problem of checking that identifiers are declared before use).

$L_2 = \{ a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1 \}$ is not CFL.
(It represents that the procedures be declared with same number of formal parameters as there are actual parameters in their use).

BNF Notation

→ It is also called context free grammar (Backus Naur form).
→ In this grammar there is no left or right context restriction.
→ Mathematically, CFG is defined as follows —
A grammar $G = (V_N, V_T, P, S)$ is said to be context free when all productions in P have the form $\alpha \rightarrow \beta$ where $\alpha \in V_N$ and $\beta \in (V_T \cup V_N)^*$
→ Here V_N is a finite set of nonterminals (generally represented by capital letters)
→ V_T is a finite set of terminals (generally represented by small letters)
→ S is starting nonterminal called start symbol of the grammar and $S \in V_N$.
→ P is set of production rules in CFG, of the form $\alpha \rightarrow \beta$ where
α is single nonterminal and
β is terminal or nonterminal or set of both.

COMPILER DESIGN
(RCS-602)

Instructor:

Ms.Priyanka Goel
Asst.Prof.
Deptt of CS & E

12/02/2019

UNIT 2: Basic Parsing Techniques

- Parsers
- Shift reduce parsing
- Operator precedence parsing
- Top down parsing
- Predictive parsers
- Automatic Construction of efficient Parsers: LR parsers, the canonical Collection of LR(0) items
- Constructing SLR,CLR & LALR parsing tables, using ambiguous grammars
- An automatic parser generator
- Implementation of LR parsing tables

2

Parsers

- In compiler model, the parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language.
- A parser reports any syntax errors if the token string is invalid syntactically.
- Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

3

Types of Parsers

- There are three general types of parsers for grammars: universal, top-down and bottom-up.
- Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar.
- But these general methods are, however, too inefficient to use in production of compilers.

4

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Types of Parsers....

- As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root.
- In either case, the input to the parser is scanned from left to right, one symbol at a time.

5

Representation of Parse Tree

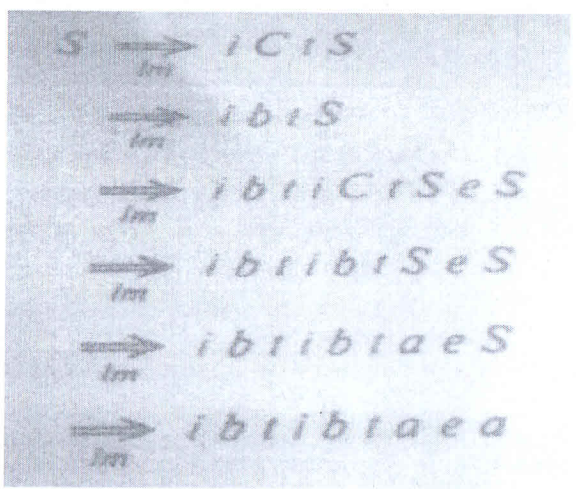
- Recall that derivations may be leftmost or rightmost.
- Rightmost derivations are also called canonical derivations.
- Also, if we traverse the parse tree in preorder way, then it corresponds the order in which leftmost derivations are applied.

6

Consider the grammar:
S → i C t S
S → i C t S e S
S → a
C → b
where i, t, e represents if, then and else;
C and S represents condition and statement.

Following steps will be followed to construct leftmost derivation for the sentence:
w= i b t i b t a e a

7



Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

- Corresponding Parse tree:

9

Shift-Reduce Parsers

- It is a general style of bottom-up parsing.
- It is the process of "reducing" a string *w to* the start symbol of the grammar.
- At each *reduction step, a specific substring* matching the body of a production is replaced by the non terminal at the head of that production.
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

10

Consider the grammar:

S→aAcBe

A→Ab | b

B→ d and the string "abbcde".

abbcde => a**Ab**cde

=> aA**cd**e

=> **aAcBe**

=> S

- Each replacement of the right side of the production by its left side is called **reduction**; b,Ab,d,aAcBe are called **handles**.

11

- Sometimes, the leftmost substring which matches the right side of some production A→ β is not a handle because a reduction by this production may yield a string which cannot be reduced to the start symbol.
- In previous example:

abbcde => a**Ab**cde

=> aA**Ad**e

=> aAABe

=> trap

- Bottom-up parsing during a left-to-right scan of the input constructs a **rightmost derivation in reverse**.

12

Handles in Shift Reduce Parsers

- Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

e.g. abbcde and production is A→b

right sentential form — handle

- If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

13

Handle Pruning

Consider the following grammar:

E→E+E | E*E | (E) | id and the input string id1+id2 * id3

Right Sentential Form	Handle	Reducing Production
id1 + id2 * id3	id1	E→id
E + id2 *id3	id2	E→id
E + E * id3	id3	E→id
E + E * E	E * E	E → E * E
E + E	E + E	E → E + E
E		

- It should be noted that here, the sequence of right sentential forms is just the reverse of the sequence of rightmost derivation.
- This process is called handle pruning.

14

Stack Implementation of Shift Reduce Parsers

- A convenient way to implement a shift-reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string **w** to be parsed.
- We use $ to mark the bottom of the stack and also the right end of the input.
- Initially, the stack is empty, and the string w is on the input, as follows:

Stack	Input
$	w$

15

- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string *β of grammar* symbols on top of the stack.
- It then reduces *β to the head of the appropriate* production.
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

16

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Consider the following grammar:
E→E+E | E*E | (E) | id and the input string id1+id2 * id3

Stack	Input	Action
$	id1+id2 * id3$	shift
$ id1	+id2 * id3$	reduce by E→ id
$ E	+id2 * id3$	shift
$ E+	id2 * id3$	shift
$ E + id2	* id3$	reduce by E→ id
$ E + E	* id3$	shift
$ E + E*	id3$	shift
$ E + E * id3	$	reduce by E→ id
$ E + E * E	$	reduce E → E * E
$ E + E	$	reduce E → E + E
$ E	$	accept

Operations in Shift Reduce Parsers

- While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make:

1. Shift. Shift the next input symbol onto the top of the stack.
2. Reduce. The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what non terminal to replace the string.
3. Accept. Announce successful completion of parsing.
4. Error. Discover a syntax error and call an error recovery routine.

18

Why stack is used?

- The use of a stack in shift-reduce parsing is justified by an important fact that the handle will always eventually appear on top of the stack, never inside.

19

Conflicts in Shift Reduce Parsing

- There are context-free grammars for which shift-reduce parsing cannot be used.
- Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce **(a shift/reduce conflict)**, or cannot decide which of several reductions to make **(a reduce/reduce conflict)**.

20

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Conflicts in Shift Reduce Parsing...

- Following are some examples of syntactic constructs that give rise to such grammars.
- Technically, these grammars are not in the LR(k) class of grammars; such grammars are referred as non-LR grammars.
- Consider the dangling-else grammar :

stmt → *if expr* ***then stmt***
| *if expr then stmt else stmt*
| *other*

21

- If we have a shift-reduce parser in configuration:

STACK	INPUT
.....*if expr then stmt*	*else*....$

- we cannot tell whether ***if expr then stmt*** is the handle, no matter what appears below it on the stack. Here there is a shift/reduce conflict.
- If we resolve the shift/reduce conflict on **else** in favor of shifting, the parser will behave as we expect, associating each else with the previous unmatched then.

22

- Another common conflict occurs when we know we have a handle, but the stack contents and the next input symbol are insufficient to determine which production should be used in a reduction.
- A statement beginning with p (i , j) would appear as the token stream id(id, id) to the parser. After shifting the first three tokens onto the stack, a shift-reduce parser would be in configuration.

e.g.

STACK	INPUT
• • • id (id	, id) • • •

23

- It is evident that the id on top of the stack must be reduced, but by which production?
- The correct choice can vary if p is a procedure or if p is an array.
- The stack does not tell which; information in the symbol table obtained from the declaration of p must be used.
- One solution is to change the token **id in production (1) to procid and to** use a more sophisticated lexical analyzer that returns the token name **procid** when it recognizes a lexeme that is the name of a procedure.

24

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Operator Precedence Parsing

- **Operator Grammar:** A grammar in which no production's right side is Є or has two adjacent non-terminals.

  e.g. E →E A E | (E) | -E | id

  A → + | - | * | / | ^

is not an operator grammar because the right side EAE has two consecutive non terminals.

25

- But if we substitute A with its alternates, we can obtain operator grammar as:

E → E + E | E – E | E * E | E / E | E ^ E | (E) | -E | id

- In operator-precedence parsing, we define three disjoint precedence relations <·, ·> and ≐ between certain pairs of terminals.
- These precedence relations guide the selection of handles and have the following meanings:

26

Operator Precedence Parsing....

a ⋖ b : This means a "yields precedence to" b.

a ⋗ b : This means a "takes precedence over" b.

a ≐ b This means a "has precedence as" b.

- Let G be an Є-free operator grammar. For each two terminals a and b:
  - **Rule 1: a ≐ b** if there is a right side of production of the form αaβbγ where β is either Є or single non terminal, i.e. if a appears immediately to the left of b in right side or if they appear separated by one non terminal.

    e.g. S→ iCtSeS implies i **≐t and t ≐ e**

27

Operator Precedence Parsing....

- **Rule 2 :a ⋖ b** if for some non terminal A there is a right side of the form αaAβ and A ⇒ γbδ where γ is either Є or a single non terminal i.e. if a non terminal A appears immediately to the right of a and derives a string in which b is the first terminal.

  e.g. S→ iCtS and C→b so i ⋖ b.

  Also, define $ ⋖ b if there is a derivation S ⇒ γbδ and γ is Є or a single non terminal.

28

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Operator Precedence Parsing....

– **Rule 3: a ⋗ b** if for some non terminal A there is a right side of the form αAbβ and A ⇒ γaδ where δ is either Є or a single non terminal.

i.e. a ⋗ b if a non terminal appearing immediately to the left of b derives a string whose last terminal is a.

e.g. S→ iCtS and C ⇒b implies b ⋗t.

Also, define a ⋗ $ if S ⇒ γaδ and δ is either Є or a single non terminal.

29

Operator Precedence Parsing....

- An operator precedence grammar is an Є- free operator grammar in which precedence relations ⋖, ⋗ and ≐ are disjoint i.e. For any pair of terminals a and b, never more than one of the relations a ⋖ b, a ⋗ b and a ≐ b is true.

30

Operator Precedence Parsing....

e.g. E → E+E | E * E | (E) | id

- This grammar is not operator precedence because :
  - From rule 3, we have αAbβ and A ⇒ γaδ where δ is either Є or a single non terminal.

    So, considering E+E, α= Є, A=E, b=+, β=E and

    E ⇒ E+E so, + ⋗ +

    γ a δ
  - From rule 2, we have αaAβ and A ⇒ γbδ where γ is either Є or a single non terminal. Again,considering E+E, α= E, a=+, A=E, β= Є and E ⇒ E+E

    so, + ⋖ +

    γ b δ

    + ⋗ + **and** + ⋖ + cant exist at a time.

31

Example continued....

- So, the given grammar can be transformed into operator precedence and unambiguous as:

E → E+T | T
T → T * F | F
F → (E) | id

32

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Algorithm to construct precedence relations

- For each non terminal A, we compute following two sets:
  - **LEADING(A)** consists of all those terminals that can be the first terminal in a string derived from that non terminal. This set helps in computing ⋖ relations.
  - **TRAILING(A)** consists of all those terminals that can be the last terminal in a string derived from that non terminal. This set helps in computing ⋗ relations.

33

e.g. E → E+T | T;T → T * F | F; F → (E) | id

Step 1: Computing LEADING and TRAILING for all non terminals

Non Terminal	Leading	Trailing
E	*,+,(,id	*,+,),id
T	*,(,id	*,),id
F	(,id	),id

34

Example Continued….

Step 2: Computing ≐ relation

There is only one production satisfying rule 1: F→(E) thus (≐)

Step 3: Computing ⋖ relation

+ ⋖ LEADING(T)	* ⋖ LEADING (F)	(⋖ LEADING (E)
i.e.	i.e.	i.e.
+ ⋖ *	* ⋖ (	(⋖ *
+ ⋖ (	* ⋖ id	(⋖ +
+ ⋖ id		(⋖ (
		(⋖ id

Example Continued….

Step 4: Computing ⋗ relation

TRAILING (E) ⋗ +	TRAILING (T) ⋗ *	TRAILING(E) ⋗)
i.e.	i.e.	i.e.
+ ⋗ *	* ⋗ (	(⋗ *
+ ⋗ (	* ⋗ id	(⋗ +
+ ⋗ id		(⋗ (
		(⋗ id

36

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Example Continued….

Step 5: Computing relations for $

$ < LEADING(E)	TRAILING(E) > $
• $ < * • $ < + • $ < (• $ < id	• * > $ • + > $ •) > $ • id > $

37

Example Continued….

Step 6: Constructing precedence table

	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
id	>	>		>		>
$	<	<	<		<	

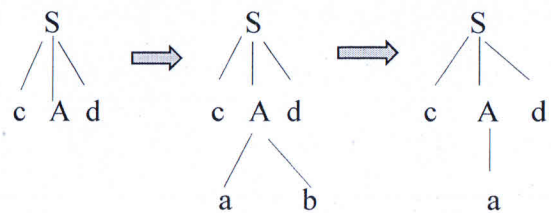
Blanks indicate error entries.

38

Top Down Parsing

- Top down parsing can be viewed as an attempt to find a leftmost derivation for an input string.
- It starts constructing parse tree recursively for the given input, starting from the root.

e.g. S → cAd and A → ab | a ; input w = cad



39

Difficulties with Top Down Parsing

- **Left Recursion:** A grammar is said to be left recursive if it has a non terminal A such that there is a derivation A => Aα for some α.

  Left recursion can cause a top down parser to go into infinite loop.
- **Backtracking:** If we make a sequence of erroneous expansions and then discover a mismatch, we may have to undo the semantic effects of making these erroneous expansions which requires substantial overhead.

40

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Difficulties with Top Down Parsing...

- **Order of choosing alternate:** The order in which alternates are tried can affect the language accepted. e.g. with parse tree : S→ cAd => cad and input string "cabd"

With above parse tree and 'ca' matched, the failure of next input symbol b. to match would imply that the alternate cAd for S was wrong.

- **Error Localization:** When failure is reported, it is difficult to locate the error.

41

Elimination of Left Recursion

- Consider the grammar with left recursion:
  A→ Aα | β where β doesn't begin with A
- Then we can eliminate left recursion by replacing the above productions as:
  A → βA'
  A' → α A' | ∈

e.g. E → E+T |T
T → T* F |F
F → (E) | id

42

Eliminating left recursion, we obtain:
E → TE'
E' →+TE' | ∈
T → F T'
T' → *FT' | ∈
F → (E) |id

- However this process can eliminate all immediate left recursion but it will not eliminate left recursion involving derivations of two or more steps.

43

e.g. S→ Aa | b ; A→ Ac | Sd | e

Here, S is left recursive because S => Aa => Sda
Now, substituting S- Productions in A → Sd:
A → Ac | Aad | bd | e
Eliminating left recursion :
S → Aa | b
A → bdA' | eA'
A'→ cA' | adA' | ∈

44

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Recursive Descent Parsing

- A parser that uses recursive procedures to recognize its input with no backtracking is called a recursive descent parser.
- When the implementation of recursive descent parser is in tabular form, it is called predictive parsing.
- Recursive descent parsers work using the concept of **left factoring** (the process of factoring out the common prefixes of alternates).

45

Left Factoring

- If $A \rightarrow \alpha\beta \mid \alpha\gamma$ are two A-productions and the input begins with non empty string derived from α, we do not know whether to expand $\alpha\beta$ or to $\alpha\gamma$.
- This decision can be deferred by expanding A to $\alpha A'$.
- i.e. After applying left factoring, the original productions become:

  $A \rightarrow \alpha A'$

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

46

e.g. $S \rightarrow iCtS \mid iCtSeS \mid a$

$C \rightarrow b$

- After left factoring, the grammar becomes:

  $S \rightarrow iCtSS' \mid a$

  $S' \rightarrow eS \mid \in$

  $C \rightarrow b$

47

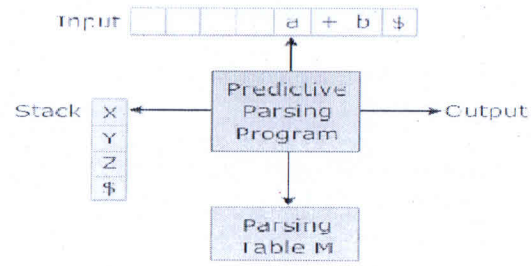
Predictive Parsers

- A predictive parser is an efficient way of implementing recursive descent parsing.
- It has following components:
  - Input : contains the string to be parsed, followed by $ (right end marker).
  - Stack : contains a sequence of grammar symbols, preceded by $(bottom-of-stack marker). Initially the stack contains the start symbol of the grammar preceded by $.
  - Parsing table : a 2D array M[A,a] where A is non terminal and a is a terminal or the symbol $.
  - Output

48

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Predictive Parsers



49

Construction of Predictive Parsing Table

- Two functions FIRST and FOLLOW are computed.
- These functions indicate the proper entries in the table for the grammar G, if such a parsing table for G exists.
- Consider the grammar:

E → TE'

E'→+TE' | ∈

T → FT'

T'→ *FT' | ∈

F → (E) | id

50

Computing FIRST…

- To compute FIRST(X) for any grammar symbol X, apply the following rules until no more terminals or ∈ can be added to any FIRST set:
- For X → ∈, FIRST(X) = { ∈ }
- For any terminal symbol 'a', FIRST(a) = { a }
- For X→ aα, FIRST(X) ={a }
- For $X \rightarrow Y_1Y_2Y_3$:
  - If $\in \notin First(Y_1)$, then First(X) = $First(Y_1)$
  - If $\in \in First(Y_1)$, then First(X) = { $First(Y_1) - \in$ } ∪ $First(Y_2Y_3)$
  - This rule can be expanded for any $X \rightarrow Y_1Y_2Y_3.....Y_n$

51

Considering the given grammar:

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

FIRST (E') = {+, ∈}

FIRST (T') = {*, ∈}

52

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Computing FOLLOW....

- To compute FOLLOW(A) for all non terminals A, apply the following rules until nothing can be added to any FOLLOW set:
- For the start symbol S, place $ in FOLLOW(S).
- For $A \rightarrow \alpha B$ or $A \rightarrow \alpha B\beta$ where FIRST(β) contains $\in$, FOLLOW(B) = FOLLOW(A)
- For any production rule $A \rightarrow \alpha B\beta$ and $\beta \neq \in$ then Follow(B) = First(β) except $\in$

53

Considering the given grammar:

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

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

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

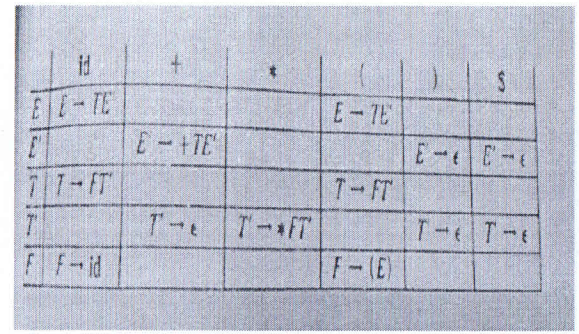
54

Constructing Parsing Table...

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

55

Parsing Table for given grammar:



	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

56

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Example 2

S→ iCtSS' |a

S'→eS | ∈

C→b

FIRST (C) = {b}

FIRST (S) ={i,a}

FIRST (S')= {e, ∈}

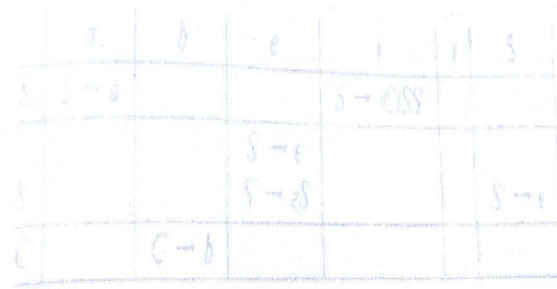
FOLLOW(C) = {t}

FOLLOW(S) = {e, $}

FOLLOW (S') = {e, $}

57

Example 2 continued….



58

Algorithm to parse input using Predictive Parsing

- The parser is controlled by a program that determines X, symbol on top of the stack and a, the current input symbol. There are three possibilities:
  - **Rule 1:** If X=a=$, parser halts and announces successful completion of parsing.
  - **Rule 2:** If X= a ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.

59

Algorithm continued….

- **Rule 3:** If X is a non terminal, pop X and consult the parsing table entry M[X,a] to see which production applies. Push RHS of the production on the stack e.g. if M[X,a] has entry {X → UVW} , the parser replaces X on top of the stack by WVU(U on the top of stack).
- **Rule 4:** If none of the above cases apply or if the entry M[X,a] is empty, the parser calls an error recovery routine.

60

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Example

- Consider the grammar:

  $E \rightarrow TE'$

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

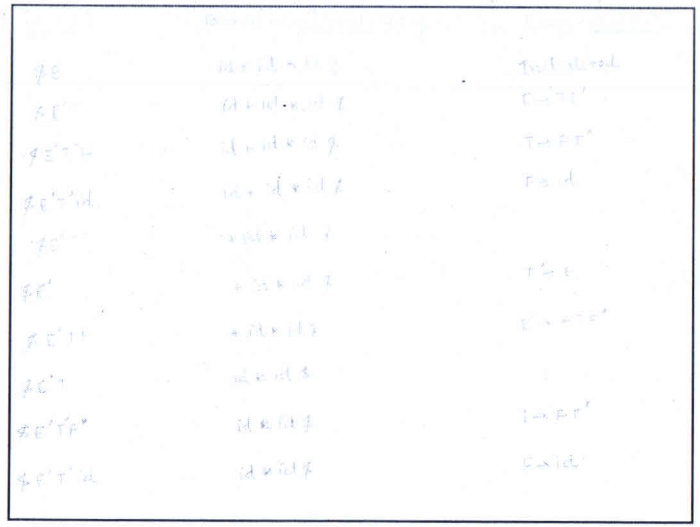
  $T \rightarrow FT'$

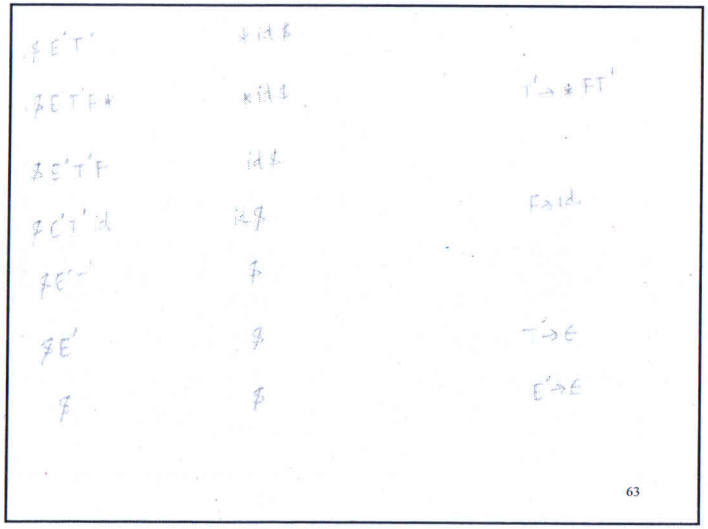
  $T' \rightarrow *FT' \mid \in$

  $F \rightarrow (E) \mid id$

- Parse the input id+id*id using predictive parsing

61





63

LL(1) Grammar

- For some grammars, parsing table may have some entries that are multiply-defined.
- If the grammar is left recursive or ambiguous, then parsing table will have atleast one multiply-defined entry.
- Thus to avoid multiple entries, we should eliminate all left recursion and perform left factoring before creating parsing table for the given grammar.

64

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

LL(1) Grammar....

- A grammar whose parsing table has no multiple defined entries is called LL(1) grammar.
- Formally, a grammar is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions, following conditions hold:
  - For no terminal a, do α and β derive strings beginning with a.
  - Atmost one of α and β can derive the empty string.
  - If $\beta \Rightarrow \in$ then α does not derive any strings beginning with a terminal in FOLLOW(A).

65

LR Parsers

- LR parsers comes under the category of efficient bottom up parsers which can parse large class of context free grammars.
- These are called LR parsers because they scan input from left to right and construct rightmost derivation in reverse.
- There are three different techniques for producing LR parsing tables:

66

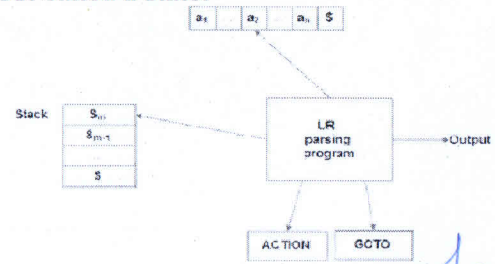
LR Parsers....

- SLR (Simple LR) : easiest to implement but it may fail to produce a table for certain grammar on which the other methods succeed.
- CLR(Canonical LR): most powerful and work on very large class of grammars. But it is very expensive to implement.
- LALR(Lookahead LR) : immediate in power between SLR and CLR methods.
- LR(k) means left to right read, rightmost derivation, taking k lookahead input characters to make parsing decision.

67

LR Parsers....

- The parser has an input, a stack and a parsing table.
- The stack contains a string of the form $s_0X_1s_1X_2s_2..X_ms_m$ where s_m is on the top.
- Each X_i is a grammar symbol and each s_i is a symbol called a state.



68

LR Parsers....

- A parsing table consists of two parts:
  - An **ACTION** table that gives a grammar rule to apply, given the current state and the current terminal symbol in the input stream.
  - A **GOTO** table that prescribes to which new state (item set) it should move.

69

SLR Parsing

- The construction of SLR parsing table is based on the construction of LR(0) items.
- An LR(0) item is defined as a grammar rule with a special dot added somewhere in the RHS of the production rule.
- Thus the production rule E→E+T has four corresponding items:
  - E → .E+T : parser expects to see a string derivable from E+T
  - E → E.+T : parser has recognized string derivable from E and now expects to read + followed by string derivable from T
  - E → E + .T : parser has recognized a string derivable from E+ and now expects to read a string derivable from T
  - E → E + T. : parser has recognized a string derivable from E+ T and now expects to see nothing

70

Construction of LR(0) items

- To construct LR(0) collection for a grammar, we need to define an augmented grammar and two functions CLOSURE and GOTO.
- If G is a grammar with start symbol S , then G' is the augmented grammar with start symbol S' and a new production S'→ S.
- Its purpose is to indicate the parser when it should indicate parsing and announce acceptance of the input.
- This would occur when the parser is about to reduce by S' → S .

71

CLOSURE

- If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules:
  1. Initially, add every item in I to CLOSURE(I).
  2. If A → α .Bβ is in CLOSURE(I) and B→.γ is a production, then add the item B→.γ *to* CLOSURE(I), if it is not already there.
  3. Apply this rule until no more new items can be added to CLOSURE(I).

72

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

GOTO

- The second useful function is GOTO (I, X) where I is a set of items and X is a grammar symbol.
- GOTO (I , X) is defined to be the closure of the set of all items [A → αX. β] such that [A → α . Xβ] is in I.

e.g. If I is the set of items {[E'→E.],[E→E.+T]} then GOTO(I,+) consists of: E → E + .T

T → .T * F

T → .F

F → .(E)

F → .(id)

73

Algorithm for Construction of Canonical Collection of LR(0) items

Input: A augmented grammar G' corresponding to a grammar G

Output: The canonical collection of LR(0) items.

Method: The method consists of following procedure

I_0 = CLOSURE({S'→.S})

Repeat

for each set of items I_i & each grammar symbol X

if GOTO(I_i,X) ≠ ∅ (not empty) and is not already

then add GOTO(I_i,X)

until no more sets of items can be added

Example....

Consider the grammar:

E→E+T | T

T→T*F | F

F→(E) |id

a) Find the canonical collection of LR(0) items.

b) Draw DFA showing transitions between item sets.

75

Example Continued....

Step 1: Construction of Augmented Grammar

E'→E

E→E+T

E→T

T→T*F

T→F

F→(E)

F→id

76

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Example Continued....

Step 2: Computing CLOSURE Io of initial rule :

Io : CLOSURE([E'→E])

Io:
E'→.E
E→ .E+T
E→ .T
T→.T*F
T→.F
F→.(E)
F→.id

Step 3:Computing GOTO(Io, X) for each grammar symbol X:

77

Example Continued....

I_1: GOTO (Io, E)

E'→ E.
E → E.+T

I_2: GOTO (Io, T)

E → T.
T → T.* F

I_3: GOTO (Io, F)

T → F.

I_4 : GOTO (Io, ()

F → (.E)
E→ .E+T
E→ .T
T→.T*F
T→.F
F→.(E)
F→.id

I_5 : GOTO (Io, id)

F → id.

78

Example Continued....

Step 4: Computing GOTO(Ii , X) for each grammar symbol X and item set I_i:

I_6: GOTO (I_1, +)

E → E+.T
T→.T*F
T→.F
F→.(E)
F→.id

I_7: GOTO (I_2, *)

T → T*.F
F→.(E)
F→.id

GOTO (I_4, F) => I_3
GOTO (I_4, () => I_4
GOTO (I_4, id) => I_5

I_8: GOTO (I_4, E)

F →(E.)
E →E.+T

79

Example Continued....

I_9: GOTO (I_6, T)

E → E+T.
T → T.* F

GOTO (I_6, F) => I_3
GOTO (I_6, () => I_4
GOTO (I_6, id) => I_5

I_{10}: GOTO (I_7, F)

T → T* F.

GOTO (I_7, () => I_4
GOTO (I_7, id) => I_5

I_{11}: GOTO (I_8,))

F → (E).

GOTO (I_8, +) => I_6
GOTO (I_9, *) => I_7

80

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Example Continued....

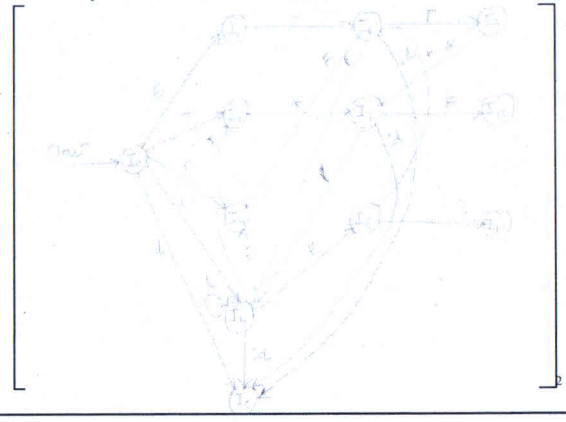
Step 5: Construction of transition table

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
I_0	I_5			I_4			I_1	I_2	I_3
I_1		I_6							
I_2			I_7						
I_3									
I_4	I_5			I_4			I_8	I_2	I_3
I_5									
I_6	I_5			I_4				I_9	I_3
I_7	I_5			I_4					I_{10}
I_8		I_6			I_{11}				
I_9			I_7						
I_{10}									
I_{11}									

81

Example Continued....

Step 6: Construction of DFA



Construction of SLR Parsing Table

- Let C={I_0 , I_1 , I_2 ,……, I_n } and states of the parser be 0,1,2,….,n(state i being constructed from I_i). The **parsing actions** for state i are constructed as:
- **Rule 1:** If [A→ α.aβ] is in I_i and GOTO(I_i ,a) = I_j then set ACTION[i,a] to "shift j".
- **Rule 2:** If [A→α.] is in I_i , then set ACTION[i,a] to "reduce A→ α" for all a in FOLLOW(A).
- **Rule 3:** If [S' → S] is in I_i , then set ACTION[i,$] to "accept".

83

Construction of SLR Parsing Table...

- The **GOTO transitions** for state i are constructed as follows:
- **Rule 4:** If GOTO(I_i , A) = I_j then GOTO[i,A]=j
- **Rule 5:** All entries not defined by rules 1 through 4 are made "error".
- **Rule 6:** The initial state of the parser is the one constructed from the set of items containing [S' → S].

84

Example Continued....

- Computing FOLLOW for construction of SLR parsing table:

Non terminal	FIRST	FOLLOW
E	(, id	+ ,) , $
T	(, id	+ , *,) , $
F	(, id	+ , *,) , $

85

Example Continued....

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

No conflict in SLR parsing table => grammar is SLR(1)

86

SLR Parsing Algorithm

- A configuration of an LR parser is a pair whose first component is the stack component and whose second component is the remaining input.

i.e. Stack contents input buffer

$s_0X_1s_1X_2s_2X_3....X_ms_m$ $a_iX_{i+1}.....a_n\$$

symbol on top of stack current input symbol

- Depending upon the current input symbol and the state on the top of stack, LR parser consults an action table entry ACTION[s_m , a_i] until the string is accepted or a syntax error is reported.

87

SLR Parsing Algorithm....

- The entry ACTION[s_m , a_i] can have any of four values:
- **a shift s_n in which :**
  - the current input terminal is removed from the input stream.
  - the state n is pushed onto the stack and becomes the current state.
- **a reduce r_m in which r_m means reduce A→β :**
  - Calculate r, the length of β, the right side of production.
  - First pop 2r symbols of the stack, exposing state s_{m-r} .
  - Then push both A(left side of production) and s, the entry for GOTO[s_{m-r} , A] onto stack. The current input symbol is not changed in a reduce move.

88

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

SLR Parsing Algorithm....

- **an accept,** meaning the string is accepted.
- **an error** indicated by blank entries meaning that a syntax error is reported.

Here, si means "shift and stack state i"

and rj means "reduce by production numbered j.

89

Example Continued....

Show the moves made by the parser on input id*id+id :

S.No.	Stack	Input	Action
1	0	id * id + id $	shift
2	0 id 5	* id + id $	reduce by F→ id
3	0 F 3	* id + id $	reduce by T → F
4	0 T 2	* id + id $	shift
5	0 T 2 * 7	id + id $	shift
6	0 T 2 * 7 id 5	+ id $	reduce by F→ id
7	0 T 2 * 7 F 10	+ id $	reduce by T → T*F
8	O T 2	+ id $	reduce by E → T
9	O E 1	+ id $	shift
10	O E 1 + 6	id $	shift
11	O E 1 + 6 id 5	$	reduce F → id
12	O E 1 + 6 F 3	$	reduce by T → F
13	O E 1 + 6 T 9	$	reduce by E → E+ T
14	0 E 1	$	accept

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Constructing CLR parsing table

→ The method for constructing the collection of sets of valid LR(1) items is essentially the same as the way we build the canonical collection of LR(0) items.

→ For the CLOSURE operation, consider an item of the form $[A \to \alpha . B\beta, a]$ in the set of items valid for some viable prefix γ.

Algorithm :- Construction of sets of LR(1) items for grammar G

Input :- A grammar G

Output :- The sets of LR(1) items which are the sets of items valid for one or more viable prefixes of G.

Method :- The procedures CLOSURE and GOTO and the main routine for construction the sets of items.

Q :- Consider the following augmented grammar—

$S' \to S$

$S \to CC$

$C \to cC \mid d$

design the CLR parsing table.

Soln :- Computing the CLOSURE of $\{[S' \to .S, \$]\}$—

I_0 : $S' \to .S, \$$

$S \to .CC, \$$

$C \to .cC, c/d$

$C \to .d, c/d$

$\{FIRST(C) = c, d\}$

$A \to \alpha . B\beta$

GOTO(I_0, S)

I_1 : $S' \to S.$, \$

GOTO(I_0, C)

I_2 : $S \to C.C$, \$

$C \to .cC$, \$

$C \to .d$, \$

GOTO(I_0, c)

I_3 : $C \to c.C$, c/d

$C \to .cC$, c/d

$C \to .d$, c/d

GOTO(I_0, d)

I_4 : $C \to d.$ c/d

GOTO(I_2, C)

I_5 : $S \to CC.$, \$

GOTO(I_2, c)

I_6 : $C \to c.C$, \$

$C \to .cC$, \$

$C \to .d$, \$

GOTO(I_2, d)

I_7 : $C \to d.$, \$

GOTO(I_3, C)

I_8 : $C \to cC.$, c/d

GOTO(I_3, c)

$C \to c.C$, c/d

$C \to .cC$, c/d

$C \to .d$, c/d

} $\equiv I_3$

GOTO(I_3, d)

$C \to d.$, c/d } $\equiv I_4$

GOTO(I_6, C)

I_9 : $C \to cC.$, \$ }

GOTO(I_6, c)

$C \to c.C$, \$

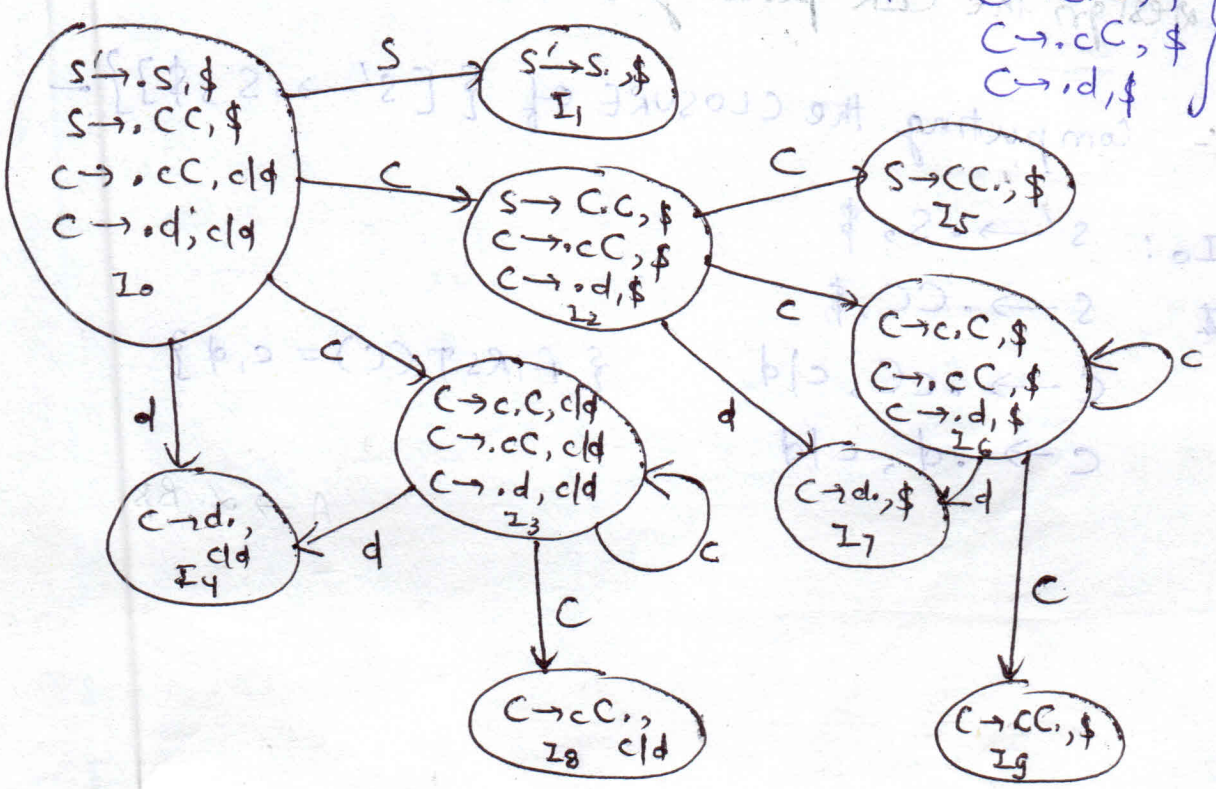
$C \to .cC$, \$

$C \to .d$, \$

} $\equiv I_6$

GOTO(I_6, d)

$C \to d.$, \$ } $\equiv I_7$



Construction of CLR parsing table

Input :- A grammar G augmented by production $S' \rightarrow S$

Output :- If possible, CLR parsing action function ACTION and goto function GOTO.

Method :-

1) Construct $C = \{I_0, I_1, \ldots I_n\}$ the collection of sets of LR(1) items for G.
2) State i of the parser is constructed from I_i, the parsing actions for state i are determined as—
   a) If $[A \rightarrow \alpha . a\beta, b]$ is in I_i and GOTO$(I_i, a) = I_j$, set ACTION$[i, a]$ to "shift j".
   b) If $[A \rightarrow \alpha ., a]$ is in I_i, then set ACTION$[i, a]$ to "reduce $A \rightarrow \alpha$."
   c) If $[S' \rightarrow S, \$]$ is in I_i, set ACTION$[i, \$]$ to "accept".

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

3) The goto transitions for state i are determined as follows:
   If GOTO$(I_i, A) = I_j$ then GOTO$[i, A] = j$
4) All entries not defined are made "error"
5) The initial state of the parser is the one constructed from set containing item $[S' \rightarrow . S, \$]$.

→ The table formed from parsing action and goto functions produced is called canonical LR(1) parsing table.

→ If the parsing action function has no multiple-defined entries, then the given grammar is called LR(1) grammar.

Note :- Every SLR(1) grammar is an LR(1) grammar, but for an SLR(1) grammar the canonical LR parser may have more states than the SLR parser from same grammar.

Q :- Consider the grammar –

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC \mid d$

design its LR(1) parsing table.

Soln :-

0 $S' \rightarrow S$

1 $S \rightarrow CC$

2 $C \rightarrow cC$

3 $C \rightarrow d$

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

COMPILER DESIGN
(RCS-602)

Instructor:

Ms.Priyanka Goel
Asst.Prof.
Deptt of CS & E

Unit 3: Syntax-directed Translation

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

Syntax-directed Translation schemes

- It is a notational framework for intermediate code generation that is an extension of context free grammars.
- This framework allows subroutines or "semantic actions" to be attached to the productions of the CFG.
- These subroutines generate intermediate code when called at appropriate times by a parser for that grammar.

Syntax-directed Translation schemes...

- These schemes are useful because these schemes enable the compiler designer to express the generation of intermediate code directly in terms of the syntactic structure of the source language.
- In designing the intermediate code generator, there are two basic issues:
  - Decide what intermediate code we should generate for each programming language construct.
  - Implement an algorithm to generate this code.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Semantic Actions

- Semantic Actions are the output actions associated with each production in a syntax directed translation scheme.
- The semantic action is enclosed in braces and it appears after the production.

e.g. $E \rightarrow E^{(1)} + E^{(2)}$ { E.VAL := $E^{(1)}$ .VAL + $E^{(2)}$ .VAL }

- Here, the semantic action is a formula which states that the translation E.VAL associated with the E on the left side of production is determined by adding together the translations associated with the E's on the right side of production.

Semantic Actions...

- The output action may involve the computation of values , generation of intermediate code, printing of an error diagnostic, or the placement of some value in a table.
- A value associated with a grammar symbol is called translation of that symbol.
- The translation fields of a grammar symbol X are usually denoted with names such as X.VAL, X.TRUE.
- If we have a production with several instances of the same symbol on the right, we can distinguish the symbols with superscripts.

Semantic Actions...

- The translations can be of two types:
  - Synthesized translation: where the value of the translation of the non terminal on the left side of the production is a function of the translations of the non terminals on the right side.
  - Inherited translation: where the translation of a non terminal on the right side of production is defined in terms of the translation of the non terminal on the left.

e.g. A → XYZ { Y.VAL := 2* A.VAL}

Implementation of Syntax Directed Translators

- One way to implement a syntax directed translator is to use extra fields in the parser stack entries corresponding to the grammar symbols.
- These extra fields hold the values of the corresponding translations.

	STATE	VAL
TOP →	Z	Z.VAL
	Y	Y.VAL
	X	X.VAL

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Ans 1 :- $S' \rightarrow S$

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

I_0: CLOSURE($S' \rightarrow S$)

$S' \rightarrow .S$

$S \rightarrow .AaAb$

$S \rightarrow .BbBa$

$A \rightarrow .$

$B \rightarrow .$

GoTo(I_0, S)

$S' \rightarrow S.$

I_1

GoTo(I_0, A)

$S \rightarrow A.aAb$

I_2

GoTo(I_0, B)

$S \rightarrow B.bBa$

I_3

GoTo(I_2, a)

$S \rightarrow Aa.Ab$

$A \rightarrow .$

I_4

GoTo(I_3, b)

$S \rightarrow Bb.Ba$

$B \rightarrow .$

I_5

GoTo(I_4, A)

$S \rightarrow AaA.b$

I_6

GoTo(I_5, B)

$S \rightarrow BbB.a$

I_7

GoTo(I_6, b)

$S \rightarrow AaAb.$

I_8

GoTo(I_7, a)

$S \rightarrow BbBa.$

I_9

Ans 2 :- Backpatching & its term can be seen from notes.

Ans 3 :- $E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id$

We have to parse string id + id * id using CLR parsing.

Step 1 :- Constructing Augmented grammar

(0) $E' \rightarrow E$

(1) $E \rightarrow E+E$

(2) $E \rightarrow E*E$

(3) $E \rightarrow id$

Step 2 :- Computing CLOSURE I_0

I_0: CLOSURE($E' \rightarrow E$)

$E' \rightarrow .E$, $

$E \rightarrow .E+E$, $ | + | *

$E \rightarrow .E*E$, $ | + | *

$E \rightarrow .id$, $ | + | *

e.g. Implementation of Desk Calculator

- Let us assume that the desk calculator evaluate the arithmetic expressions involving the integer operands and the operators + and *.
- We assume that the input expression is terminated by $.
- The output is the numerical value for the input expression.
- To design such a translator, we need to write a grammar to describe the inputs.

Example Continued....

- So, the productions are:

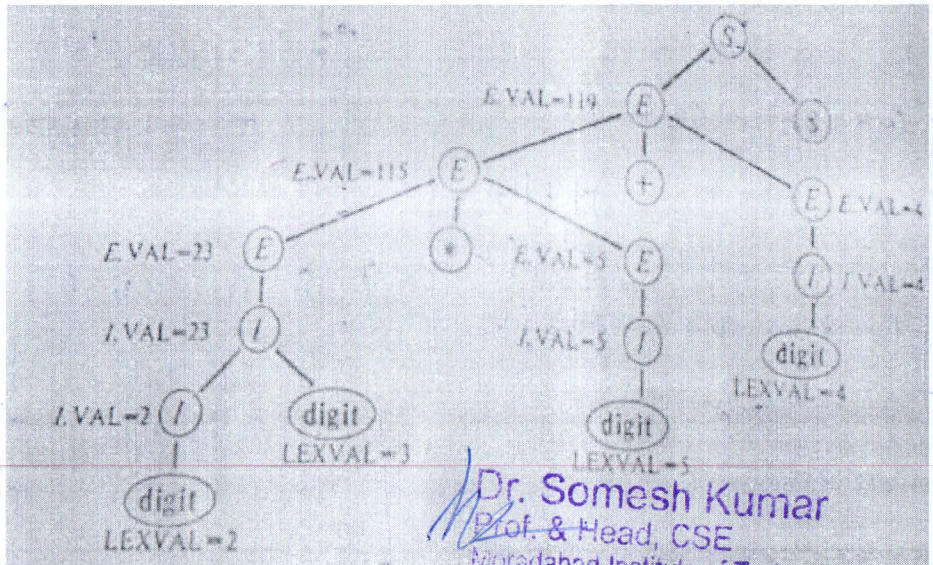
S → E $
E → E + E
E → E * E
E → (E)
E → I
I → I digit
I → digit

Example Continued....

Production	Semantic Action
1. S → E $	{print E.VAL}
2.E → $E^{(1)}$ + $E^{(2)}$	{ E.VAL := $E^{(1)}$ .VAL + $E^{(2)}$ .VAL }
3. E → $E^{(1)}$ * $E^{(2)}$	{ E.VAL := $E^{(1)}$ .VAL * $E^{(2)}$ .VAL }
4. E → ($E^{(1)}$)	{ E.VAL := $E^{(1)}$ .VAL}
5. E → I	{ E.VAL := I.VAL}
6. I → $I^{(1)}$ digit	{ I.VAL := 10* $I^{(1)}$ .VAL + LEXVAL }
7. I → digit	{ I.VAL := LEXVAL}

Syntax Directed Translation Scheme for Desk Calculator

Example Continued....



Parse tree with translations on input 23 * 5 + 4 $

Step3 :- Implementing GOTO()

GOTO(I0, E)

E' → E. , $
E → E.+E , $|+|*
E → E.*E , $|+|*

I_1

GOTO(I0, id)

E → id. , $|+|*

I_2

GOTO(I1, +)

E → E+.E , $|+|*
E → .E+E , $|+|*
E → .E*E , $|+|*
E → .id , $|+|*

I_3

GOTO(I1, *)

E → E*.E , $|+|*
E → .E+E , $|+|*
E → .E*E , $|+|*
E → .id , $|+|*

I_4

GOTO(I3, E)

E → E+E. , $|+|*
E → E.+E , $|+|*
E → E.*E , $|+|*

I_5

GOTO(I3, id)

E → id. , $|+|*

I_2

GOTO(I4, id)

E → id. , $|+|*

I_2

GOTO(I4, E)

E → E*E. , $|+|*
E → E.+E , $|+|*
E → E.*E , $|+|*

I_6

GOTO(I5, +)

E → E+.E , $|+|*
E → .E+E , $|+|*
E → .E*E , $|+|*
E → .id , $|+|*

I_3

GOTO(I5, *)

E → E*.E , $|+|*
E → .E+E , $|+|*
E → .E*E , $|+|*
E → .id , $|+|*

I_4

GOTO(I6, +)

E → E+.E , $|+|*
E → .E+E , $|+|*
E → .E*E , $|+|*
E → .id , $|+|*

I_3

GOTO(I6, *)

E → E*.E , $|+|*
E → .E+E , $|+|*
E → .E*E , $|+|*
E → .id , $|+|*

I_4

Step4 :- Constructing Parsing table

states	+	*	id	$	GOTO E
0			s2		1
1	s3	s4		accept	
2	r3	r3		r3	
3			s2		5
4			s2		6
5	r1/s3	r1/s4		r1	
6	r2/s3	s4/r2		r2	

Example Continued....

Production	Program Fragment
1. S → E $	Print VAL[TOP]
2.E → E+ E	VAL[TOP] := VAL[TOP] + VAL[TOP-2]
3. E → E* E	VAL[TOP] := VAL[TOP] * VAL[TOP-2]
4. E → (E)	VAL[TOP] := VAL[TOP-1]
5. E → I	NONE
6. I → I digit	VAL[TOP] := 10 * VAL[TOP] + LEXVAL
7. I → digit	VAL[TOP] := LEXVAL

Implementation of Desk Calculator

Sequence of Moves made by parser on Input : 23 * 5 + 4 $

Input	STATE	VAL	Production Used
1. 23*5+4$	-	-	
2. 3*5+4$	2	-	
3. 3*5+4$	I	2	I → digit
4. *5+4$	I 3	2_	
5. *5+4$	I	(23)	I → I digit
6. *5+4$	E	(23)	E → I
7. 5+4$	E*	(23)_	
8. +4$	E * 5	(23)_ _	
9. + 4$	E * I	(23)_5	I → digit
10. +4$	E*E	(23)_5	E → I
11. +4$	E	(115)	E → E*E
12. 4$	E +	(115)_	
13. $	E + 4	(115)_ _	

Sequence of Moves made by parser on Input : 23 * 5 + 4 $....

Input	STATE	VAL	Production Used
14. $	E + I	(115)_4	I → digit
15. $	E+ E	(115)_4	E → I
16. $	E	(119)	E → E+E
17. -	E $	(119)_	
18.-	$	-	S → E $

Intermediate Code

- Four types of intermediate code are often used in compilers:
  - Postfix Notation
  - Syntax Trees
  - Quadruples
  - Triples

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

id+id*id

Stack	Input	Action
0	id+id*id$	shift
0 id 2	+id*id$	Reduce by E→id
0 E 1	+id*id$	shift
0 E 1 + 3	id*id$	Shift
0 E 1 + 3 id 2	*id$	Reduce by E→id
0 E 1 + 3 E 5	*id$	Shift
0 E 1 + 3 E 5 * 4	id$	~~Redu~~ Shift
0 E 1 + 3 E 5 * 4 id 2	$	Reduce by E→id
0 E 1 + 3 E $ * 4 E 6	$	Reduce by E→E+E
0 E 1 + 3 E $	$	Reduce by E→E+E
0 E 1	$	accept

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

t1 = a+b
t2 = c+d
t3 = t1*t2
t4 = t1+c
t5 = t3+t4

Quadruple Representation

#	OP	ARG1	ARG2	RESULT
(0)	+	a	b	t1
(1)	+	c	d	t2
(2)	*	t1	t2	t3
(3)	+	t1	c	t4
(4)	+	t3	t4	t5

Triple Representation

#	OP	ARG1	ARG2
(0)	+	a	b
(1)	+	c	d
(2)	*	(0)	(1)
(3)	+	(0)	c
(4)	+	(2)	(3)

Postfix Notation

- In general, if e1 and e2 are any postfix expressions and θ is any binary operator, the result in postfix notation is represented by e1e2θ.
- The postfix code can be easily evaluated using a stack.
- The input is scanned left to right; each operand encountered is pushed onto stack and whenever a k-ary operator is found, top k elements are popped and result of applying k-ary operator is pushed onto stack.

Syntax Directed Translation to Postfix Code

S.N.	Production	Semantic Action
1	$E \rightarrow E^{(1)}$ op $E^{(2)}$	E.CODE := $E^{(1)}$ .CODE || $E^{(2)}$ .CODE || 'op'
2	$E \rightarrow (E^{(1)})$	E.CODE := $E^{(1)}$ .CODE
3	$E \rightarrow$ id	E.CODE := id

Here E.CODE is a string valued translation. Such a translation scheme is called simple postfix and it can be implemented without a translation stack just by emitting the output string after each reduction.

Implementation of Infix-Postfix Translation

Production	Program Fragment
$E \rightarrow E^{(1)}$ op $E^{(2)}$	{ print op }
$E \rightarrow (E^{(1)})$	{ }
$E \rightarrow$ id	{ print id }

- When we reduce by the production E → id, we emit the identifier.
- On reduction by E → (E), we emit nothing.
- When we reduce by E → E op E, we emit the operator op.

By following these 3 steps, we generate the postfix equivalent of the infix expression.

Example showing sequence of moves for the input a+b*c

1. Shift a
2. Reduce by E → id and print a
3. Shift +
4. Shift b
5. Reduce by E → id and print b
6. Shift *
7. Shift c
8. Reduce by E → id and print c
9. Reduce by E → E op E and print *
10. Reduce by E → E op E and print +

Output : abc * +

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Indirect triple

#	OP	ARG1	ARG2
(0)	+	a	b
(1)	+	c	d
(2)	*	(1)	(2)
(3)	+	(0)	c
(4)	+	(2)	(3)

List of pointers

#	Statement
14	(0)
15	(1)
16	(2)
17	(3)
18	(4)

Ans 5 :- $S \to Aa \mid bAc \mid dc \mid bca$

$A \to d$

(0) $S' \to S$
(1) $S \to Aa$
(2) $S \to bAc$
(3) $S \to dc$
(4) $S \to bca$
(5) $A \to d$

SLR parsing

I_0: CLOSURE($S' \to S$)

$S' \to .S$
$S \to .Aa$
$S \to .bAc$
$S \to .dc$
$S \to .bca$
$A \to .d$

GOTO(I_0, S)
$S' \to S.$ I_1

GOTO(I_0, A)
$S \to A.a$ I_2

GOTO(I_0, b)
$S \to b.Ac$
$A \to .d$
$S \to b.ca$ I_3

GOTO(I_0, d)
$S \to d.c$
$A \to d.$ I_4

GOTO(I_2, a)
$S \to Aa.$ I_5

GOTO(I_3, A)
$S \to bA.c$ I_6

GOTO(I_3, d)
$A \to d.$ I_7

GOTO(I_3, c)
$S \to bc.a$ I_8

GOTO(I_4, c)
$S \to dc.$ I_9

GOTO(I_6, c)
$S \to bAc.$ I_{10}

GOTO(I_8, a)
$S \to bca.$ I_{11}

FOLLOW(S) = { $ }
FOLLOW(A) = { a, c }

States	a	b	c	d	$	GOTO A	GOTO S
0		s3		s4		2	1
1					accept		
2	s5						
3			s8	s7		6	
4	r5		s9/r5				
5					r1		
6			s10				
7	r5		r5				
8	s11						
9					r3		
10					r2		
11					r4		

conflict so it is not SLR(1) grammar

Parse Trees and Syntax Trees

- A parse tree often contains redundant information which can be eliminated, thus producing more economical representation of the source program.
- Such variant of parse tree is called an (abstract) syntax tree, in which each leaf represents an operand and each interior node an operator.

Syntax Directed Translation of Syntax Trees

S.No.	Production	Semantic Action
1	$E \rightarrow E^{(1)}$ op $E^{(2)}$	{ E .VAL := NODE (op, $E^{(1)}$ .VAL, $E^{(2)}$ .VAL)}
2	$E \rightarrow (E^{(1)})$	{ E .VAL := $E^{(1)}$ .VAL}
3	$E \rightarrow - E^{(1)}$	{ E .VAL := UNARY (-, $E^{(1)}$ .VAL)}
4	E → id	{ E .VAL := LEAF(id)}

- The function **NODE(op, left, right)** takes three arguments: name of operator, left and right are pointers to root of subtrees. This function creates a new node labeled by the first argument and makes 2nd and 3rd arguments the left and right children of the new node, returning a pointer to it.
- The function **UNARY(op.child)** creates a new node OP and makes CHILD its child. Is also returns a pointer to the created node.
- The function **LEAF (id)** creates a new node labeled by ID and returns a pointer to that node. This node receives no children.

Three address codes

- This type of intermediate code is preferred in many compilers especially those doing extensive code optimization since it allows the intermediate code to be re-arranged in a convenient manner.
- Three address code is a sequence of statements, of the form **A:=B op C** where **A, B** and **C** are either programmer defined names, constants or compiler generated temporary names; **op** can be any fixed or floating point arithmetic operator, or logical operator

Three address codes

- The reason for the name "three address code" is that each statement usually contains three addresses, two for the operands and one for the result.

e.g. expression X + Y * Z would be represented as:

T1 := Y * Z

T2 := X + T1

where T1 and T2 are compiler generated temporary names.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

LALR parsing

(0) $S' \to S$
(1) $S \to Aa$
(2) $S \to bAc$
(3) $S \to dc$
(4) $S \to bca$
(5) $A \to d$

I_0: CLOSURE ($S' \to S$)

$S' \to .S$, $
$S \to .Aa$, $
$S \to .bAc$, $
$S \to .dc$, $
$S \to .bca$, $
$A \to .d$, a

GOTO(I_0, S)
$S' \to S.$, $
I_1

GOTO(I_0, A)
$S \to A.a$, $
I_2

GOTO(I_0, b)
$S \to b.Ac$, $
$A \to .d$, c
$S \to b.ca$, $ I_3

GOTO(I_0, d)
$S \to d.c$, $
$A \to d.$, a
I_4

GOTO(I_2, a)
$S \to Aa.$, $
I_5

GOTO(I_3, A)
$S \to bA.c$, $
I_6

GOTO(I_3, d)
$A \to d.$, c
I_7

GOTO(I_3, c)
$S \to bc.a$, $
I_8

GOTO(I_4, c)
$S \to dc.$, $
I_9

GOTO(I_6, c)
$S \to bAc.$, $
I_{10}

GOTO(I_8, a)
$S \to bca.$, $
I_{11}

States	a	b	c	d	$	GOTO S	GOTO A
0		s3		s4		1	2
1					accept		
2	s5						
3			s8	s7			6
4	r5		s9				
5					r1		
6			s10				
7			r5				
8	s11						
9					r3		
10					r2		
11					r4		

Since there is no conflict in parsing table using LALR technique so this grammar is LALR(1).

Additional Three Address Statements

1. Assignment statement of the form x = y op z and x = op y
2. Copy statement of the form x = y
3. Conditional jump of the form : If x relop y goto X
4. Unconditional jump of the form : goto X
5. Procedure call of the form: param A call P,n which is implemented as:

   param A1
   param A2
   .
   .
   param An
   call P, n

Additional Three Address Statements...

6. Indexed statements of the form A:= B[I] and A[I]:= B
7. Address and pointer assignments of the form A:=addr B, A = *B and *A = B.

- The three address statement is an abstract form of intermediate code and in an actual compiler, these statements can be implemented as:
  - Quadruples
  - Triples
  - Indirect Triples

Quadruples

- It is structure with consist of 4 fields namely **OP, ARG1, ARG2 AND RESULT. OP** denotes the operator and **ARG1** and **ARG2** denotes the two operands and **RESULT** is used to store the result of the expression.
- The contents of fields ARG1, ARG2 and RESULT are normally pointers to the symbol table entries for the names represented by these fields.

Consider expression a = b * – c + b * – c :

The three address code is:

t1 = uminus c
t2 = t1 * b
t3 = uminus c
t4 = t3 * b
t5 = t2 + t4
a = t5

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Ans 6 :-

$g = a+b-(c*d)$

$S \rightarrow id := E$

$E \rightarrow E\$$

$E \rightarrow E+E$

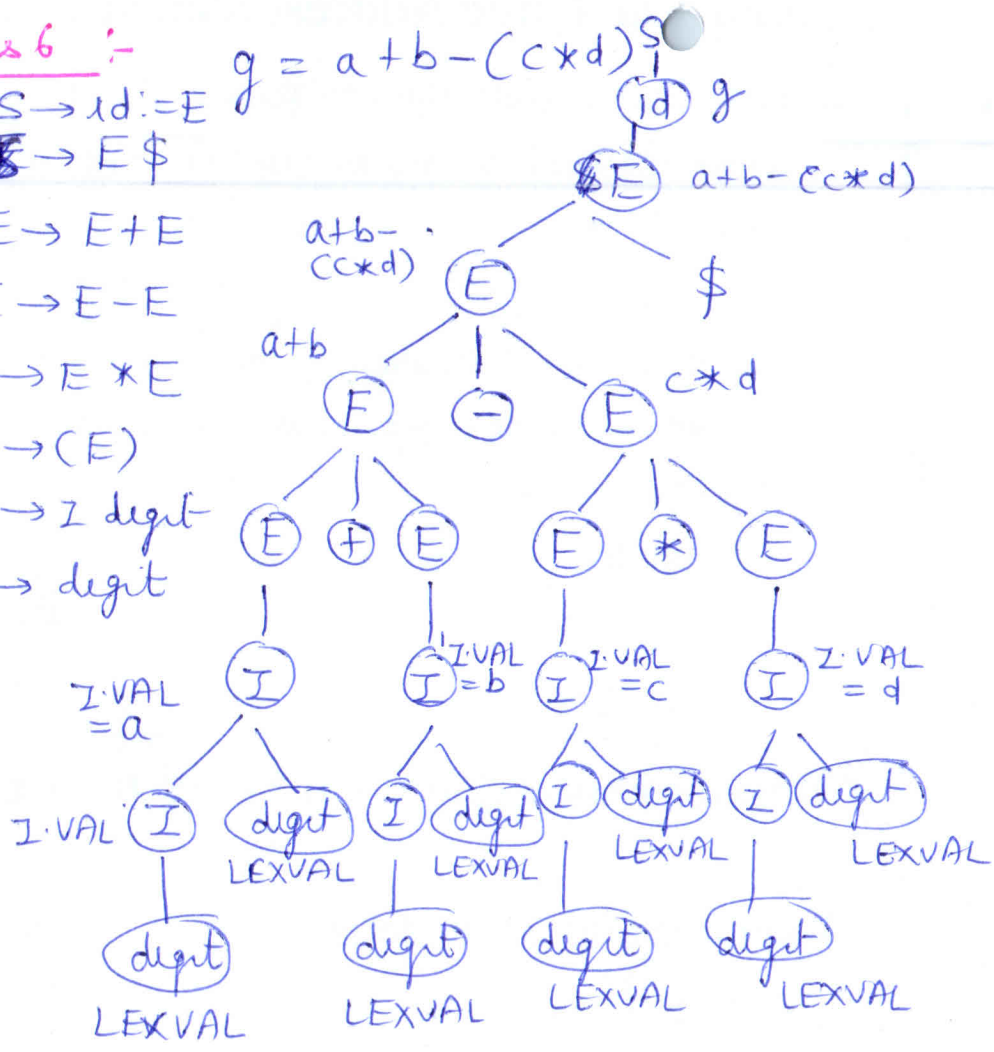
$E \rightarrow E-E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$I \rightarrow I$ digit

$I \rightarrow$ digit



Assuming that a, b, c, d and g are two-digit integer numbers

Example Continued….

- These statements can be represented by quadruples as:

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t5
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruples

Advantages:

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantages:

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

Triples

- To avoid entering temporary names into the symbol table, one can allow the statement computing a temporary value to represent that value.
- Now the three address statements are represented with only three fields OP, ARG1 and ARG2.
- Since three fields are used, this intermediate code format is known as triples.

Consider expression a = b * – c + b * – c :

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples

Disadvantages:

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Indirect Triples

- In this representation, listing of triples is considered rather than listing the triples themselves.

List of pointers to table

#	Op	Arg1	Arg2
(14)	um nus	c	
(15)	*	(14)	b
(16)	um nus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples

Advantages:

- Its similar in utility as compared to quadruple representation but requires less space than it.
- Temporaries are implicit and easier to rearrange code.

Write quadruple, triples and indirect triples for following expression : (x + y) * (y + z) + (x + y + z)

The three address code is:

t1 = x + y

t2 = y + z

t3 = t1 * t2

t4 = t1 + z

t5 = t3 + t4

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation

List of pointers to table

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

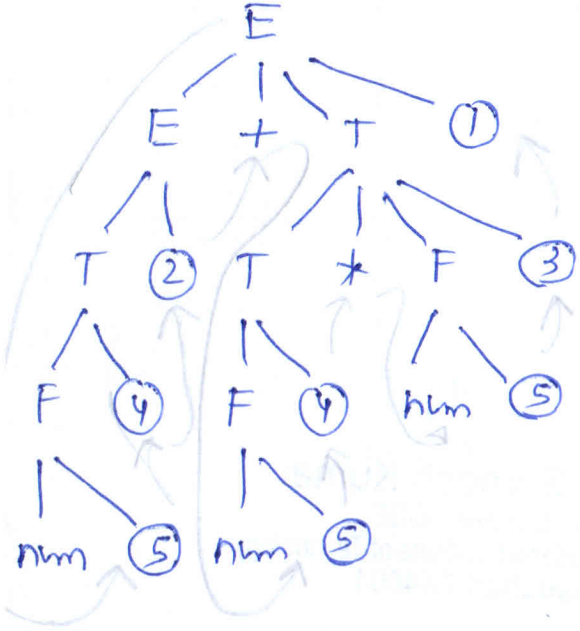
Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

E → E+T {printf("+");} ①
  | T {} ②

T → T*F {printf("*");} ③
  | F {} ④

F → num {printf(num.val);} ⑤

2+3*4



234*+

① Approaches.

A[i,j]

② 1-D array — Pg 2

2-D array — Pg 3

③ Grammar (for 1D or 2D array)

A → L := E

L → id [elist] | id

elist → elist, E | E

E → E+E | (E) | L

④ 1 byte 10×20

A[i,j] row major

$a[i,j] = BA + w*[N*(i-L_r)+(j-L_c)]$

$= a + 1*[20*(i-0)+(j-0)]$

$= a + [20i - 20 + j - 1]$

$= (a-21) + 20i + j$

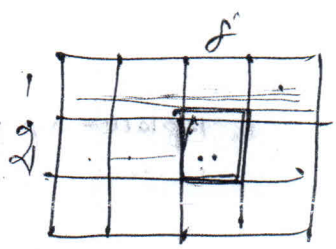
$i \Rightarrow t_1$

$j \Rightarrow t_3$

$t_2 = 20 * t_1$

$t_4 = t_2 + t_3$

$T = (a-21)[t_4]$



⑤ Multi dimensional array

A → L := E

L → ~~id[elist~~ elist] | id → closing dimension; no dimensional variable

elist → elist, E | id [E

E → E+E | (E) | L → for multi dimensional

Pointers :- elist.ARRAY

elist.NDIM

LIMIT(array, i)

elist.PLACE ⇒ computed value

L :- PLACE

OFFSET

1-D array

Address of A[i] = Base Address + w * (i - LB) ✓

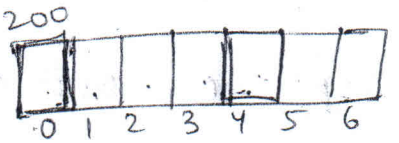
```
int i;
int a[10]
i = 1
while(i<10)
{
  a[i] = 0
  i = i+1
}
```

Assume int takes 4 bytes

3-address code

```
1  i = 1
2  if i<10 goto (4)
3  goto (8)
4  t1 = 4 * i
5  a[t1] = 0
6  i = i+1
7  goto (2)
8  end
```

w = 4



A[4] = BA + 4(4
= 200 + 16
= 216

Calculation of i

Addr = B +

S → L := E

E → E + E | (E)

E → L

L → Elist]

L → id

Elist → Elist, E

Elist → id [E

(Pascal)

~~$e_m = e_{m-1} + i_m$~~ ↳ dimension

Elist.ndim :- no of dimensions in Elist

limit(array, j) :- nj no of elements are retained in j^{th} dimension of array

Elist.place :- temporarily hold value of indexed exp.

array [1..2, 1..3]

Addr of A[i, j] = $base_A + ((i_1 - Low_1) * n_2 + i_2 - Low_2) * w$

$= ((i_1 * n_2 + i_2) * w + C$

$C = base_A - ((low_1 * n_2) + low_2) * w$

→ L := E (L → array variable; E → can also produce array)

→ E + E | (E)

E → L

L → Elist]

L → id

E → Elist , E

elist → id [E

	0	1	2	3
0	8	6	3	2
1	4	5	9	1
2	6	3	2	4

BA = 10000

w = 4

M × N

2D array

Row major

$$A[i][j] = \text{Base address} + w \times [N * (i - L_r) + (j - L_c)]$$

$$A[1][2] = 10000 + 4 \times [4 * (1-0) + (2-0)]$$

$$= 10000 + 4 * [6]$$

$$= 10024$$

Column major

$$A[i][j] = \text{Base Address} + w * [(i - L_r) + M(j - L_c)]$$

$$A[1][2] = 10000 + 4 * [(1-0) + 3(2-0)]$$

$$= 10000 + 4 * [1+6]$$

$$= 10028$$

$$a[i][j] = c + 4 * [10 * (i-0) + (j-0)]$$

$$= c + 4[10i + j]$$

$$= c + (40i + 4j) = c + 44i$$

```
int i;
int a[10][10]
i=0;
while(i<10)
{
a[i][i]=1;
i++;
}
```

```
1  i = 0
2  if (i<10) goto (4)
3  goto (8)
4  t1 = 44 * i
5  a[t1] = 1
6  i = i+1
7  goto (2)
8  end
```

L → elist] L.offset = elist.PLACE * bpw

↳ offset calculation of last dimension

Example from book (4)

(6) A[10x20] bpw = 4

X = A[y, z]

$A(y,z) = a + 4[20*(i-1)+(j-1)]$

$= a + 4[20i + j]$

$= a + 80i + 4j$

$= a + 4(2i)$

$= a + 4[20i - 20 + j - 1]$

$= a + 4[-21 + 20i + j]$

$= (a - 84) + (20i + j)$

$t_1 = y * 20$ (14)

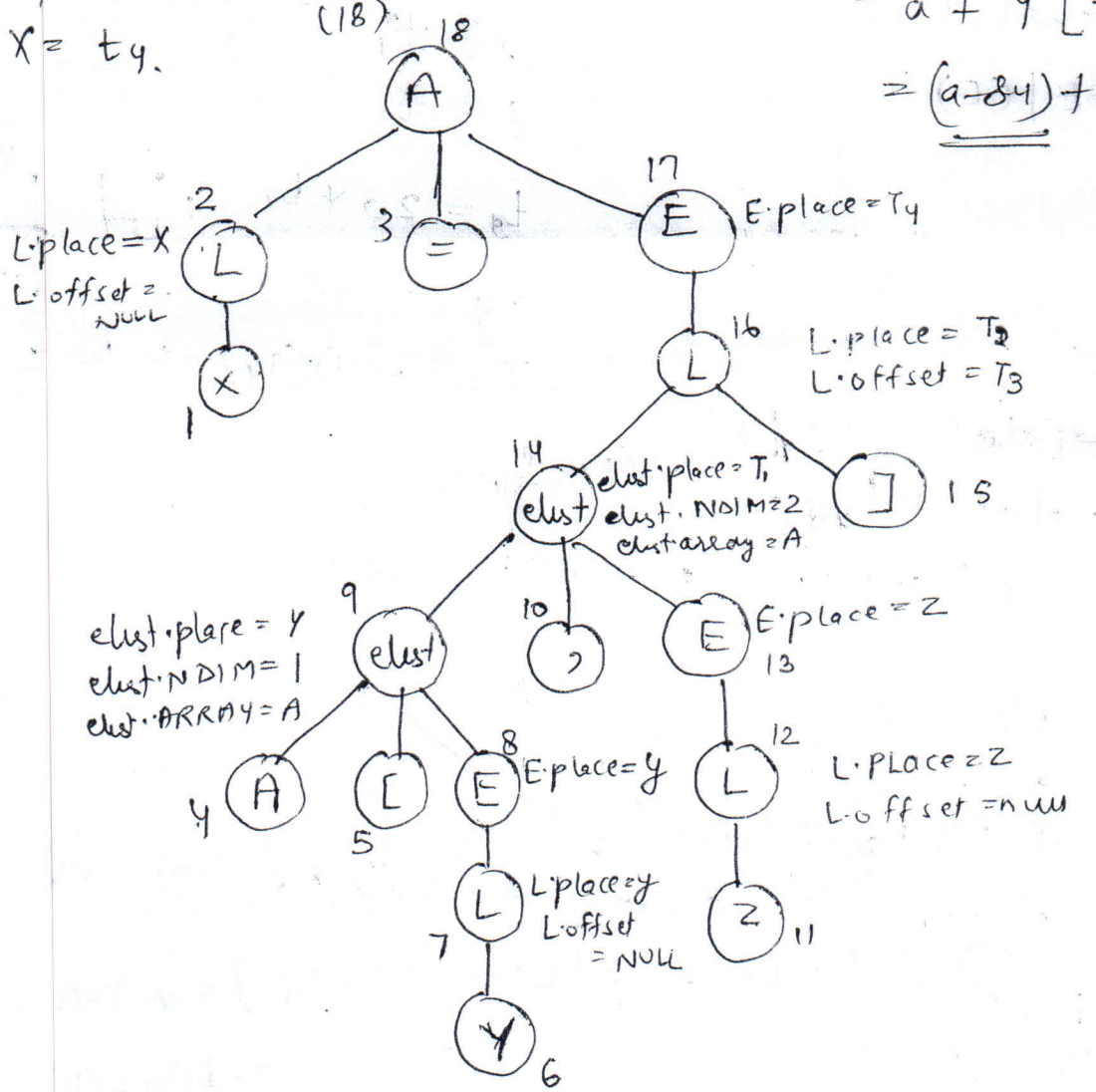
$t_1 = t_1 + z$ (14)

$t_2 = A - 84$ (16)

$T_3 = 4 * t_1$ (16)

$t_4 = t_2[t_3]$ (17)

$X = t_4$. (18)



Translation of Assignment statements

* $x = y$ ⟶ binary operator
* $x = y \text{ op } z$
* $x = \text{op } z$
  ↳ unary (+ or −)
* $x = (y)$

Task :- 3-address code corresponding to given assignment statements

Predefined functions :- look_up()
↳ searches particular variable in symbol table

Emit()
↳ generates 3-addr code

new_temp()
↳ generates/creates a temporary variable

① $S \rightarrow id := E$

```
p = look_up(id.name);
if p ≠ nil then
emit(p = E.place)
else
  error;
```

② $E \rightarrow E_1 + E_2$

```
E.place = new_temp();
emit(E.place = E1.place '+' E2.place)
```

③ $E \rightarrow E_1 * E_2$

E.place = new_temp();
emit(E.place = E_1.place '*' E_2.place

④ $E = -E_1$

E.place = new_temp()
emit (E.place = '-' E_1.place)

⑤ $E \rightarrow (E_1)$

E.place = E_1.place

⑥ $E \rightarrow id$

p = look_up (id.name)
if p ≠ nil then
emit (E.place = p)
else
error ;
}

Boolean Expressions

Unit-4

→ A compiler needs to collect and use information about the names appearing in the source program.
→ This information is entered into a data structure called symbol table.
→ The information collected about a name includes —
* the string of characters by which it is denoted
* its type (e.g. integer, real, string)
* its form (e.g. a simple variable, a structure)
* its location in memory
* other attributes depending on the language.
→ Each entry in the symbol table is a pair of the form (name, information).
→ Each time a name is encountered, the symbol table is searched to see whether that name has been seen previously.
→ If the name is new, it is entered into the table.
→ Information about that name is entered into the table during lexical and syntactic analysis.
→ The information collected in the symbol table is used in semantic analysis (in checking that uses of names are consistent with their implicit or explicit declaration) in code generation also.
→ Symbol tables are also used in error detection & correction.
→ Spaces in the symbol table can be used for code optimization purposes.

Issues in Symbol table Design

(1) Format of the entries
(2) Method of access
(3) Place where they are stored (primary or secondary)

(4) Block structured languages impose another problem in that the same identifier can be used to represent distinct names with nested scopes.

(In such cases, the symbol table mechanism must make sure that the innermost occurence of an identifier is always found first & names are removed from the active portion of symbol table when they are no longer active)

Data structures for Symbol Tables

→ In designing a symbol table mechanism, we would like a scheme that allows us to add new entries and find existing entries in a table efficiently.

→ There are three symbol table mechanisms —

(1) Linear lists
(2) Trees
(3) Hash tables

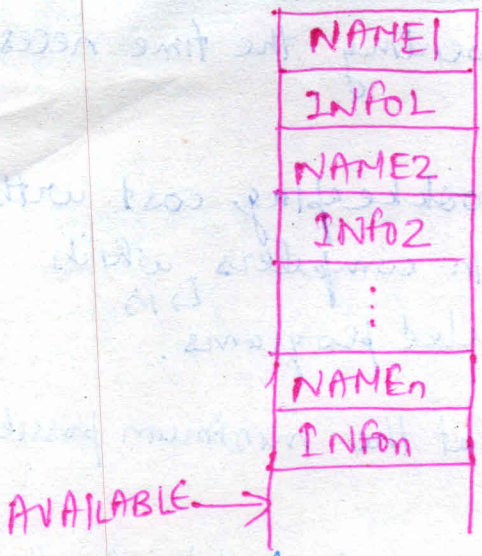
→ ~~Each schem~~ A linear list is the simplest scheme to implement but its performance becomes poor when no. of entries (n) and no. of inquiries (m) become large.

→ A binary search tree gives better performance at some increase in implementation difficulty

→ Hashing schemes provide the best performance for greater programming effort and some extra space.

Lists

→ The conceptually simplest and easiest to implement data structure for the symbol table is the linear list of records.



→ we use single array or equivalently several arrays to store names and their associated information.

→ New names are added to the list in the order in which they are encountered.

→ To retrieve information about a name, we search from the beginning of the array up to the position marked by pointer AVAILABLE which indicates the beginning of the empty portion of the array.

→ When the name is located, the associated information can be found in the words following next.

→ If we reach AVAILABLE without finding the name, we have a fault — the use of an undefined name.

→ To insert a new name, we must scan down the list to be sure if it not already there.

→ If it is, we have another fault — a multiply defined name.

→ If not, we store the new name in the words immediately following AVAILABLE and increase that pointer by the width of symbol table record.

→ If the symbol table contains n names, the work necessary to insert a new name is proportional to n.

→ To find data about a name, on the average, we have to search n/2 names, so the cost of an inquiry is also proportional to n.

→ To insert n names and m inequiries, the total work is $cn(n+m)$ where c is a constant representing the time necessary for a few machine operations.

→ Despite the quadratic growth of bookkeeping cost with program size, linear list are used in compilers which is small jobs dominate the mix of compiled programs.

→ Advantage of the list organization is that the minimum possible space is taken.

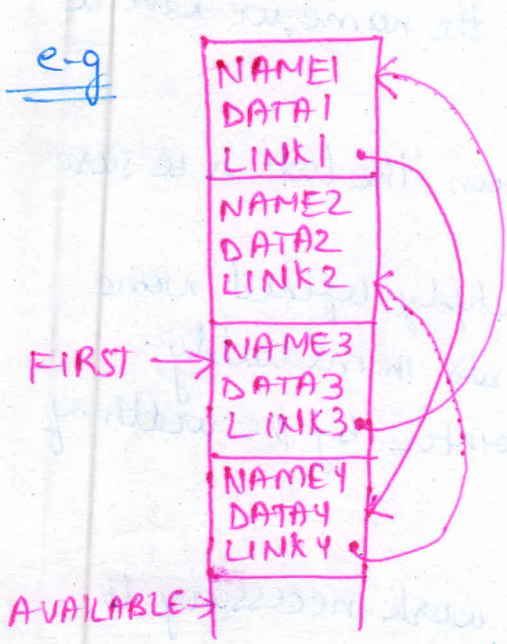
→ In a simple (non optimizing) compiler, the space taken by the symbol table may consume most of the space used for the compiler's data.

Self organizing lists

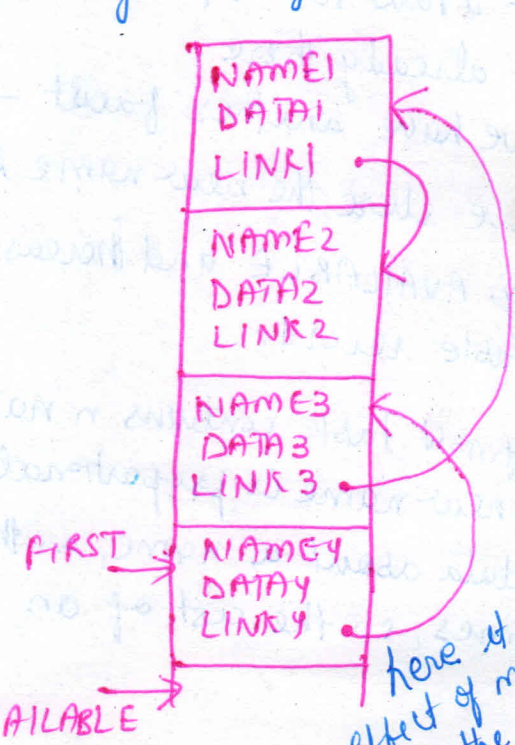
→ In order to save small amount of time spent in searching the symbol table (with the expense of little extra space), we add a LINK field to each record and we search the list in the order indicated by LINK's.

When the name is referenced or its record is first created, we move the record for that name to the front of list by moving pointers—

e.g



here FIRST gives the position of the FIRST RECORD on the linked list and each LINK field indicates next record in list.



here it shows the effect of moving NAME4 to the front of list

Assignment: Representing Scope Information

Submitted by:

Mohd Fardeen (1708210074)

Lalit Kumar (1708210063)

Mohd Shoaib (1708210078)

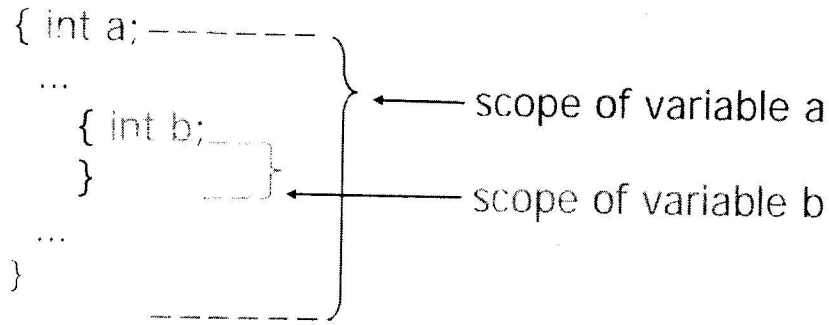
Submitted to:

Mrs. Priyanka Goel

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Representing Scope Information

In the source program, every name possesses a region of validity, called the scope of that name. This scope information characterizes the declaration of identifiers/names and the portions of the program where use of each identifier/name is allowed. For example:



Semantic rules for scope(s) in a block-structured language(s) are as follows:

- If a name is declared within block A then it will be valid only within A.
- If B block is nested within block A then the name that is valid for block A is also valid for B unless the name's identifier is re-declared in B.
- Use a name only if it is defined in enclosing scope.
- Do not declare identifiers of the same kind with identical names more than once in the same scope.

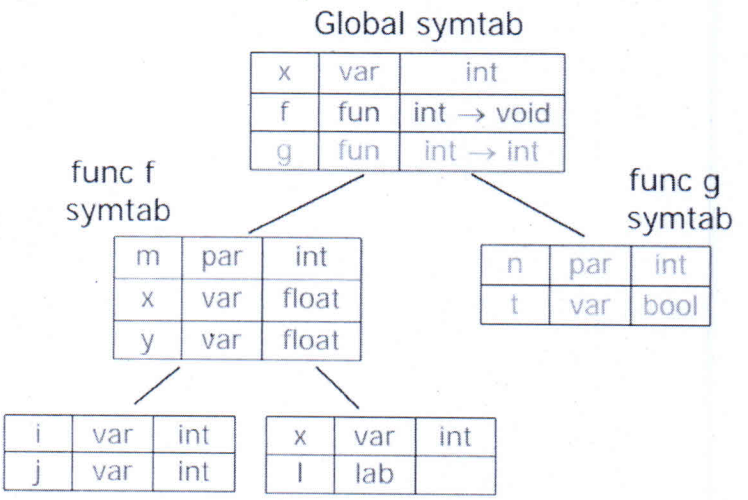
These scope rules need a more complicated organization of symbol table than a list of associations between names and attributes. There is a hierarchy of scopes in the program, so a similar hierarchy of symbol tables is used; one symbol table for each scope, each symbol table contains the symbols declared in that lexical scope (block).

Rules for creating a hierarchical symbol table:

- Tables are organized into tree/stack structures and each table contains the list of names and their associated attributes.
- Whenever a new block is entered then a new table is entered onto the stack/tree. The new table holds the name that is declared as local to this block.
- When the declaration is compiled then the table is searched for a name.
- If the name is not found in the table then the new name is inserted.
- When the name's reference is translated then each table is searched, starting from each table on the stack.

Example

```
int x;

void f(int m) {
   float x, y;
   ...
  { int i, j; ...; }
  { int x; l: ...; }
}

int g(int n) {
   bool t;
   ...;
}
```



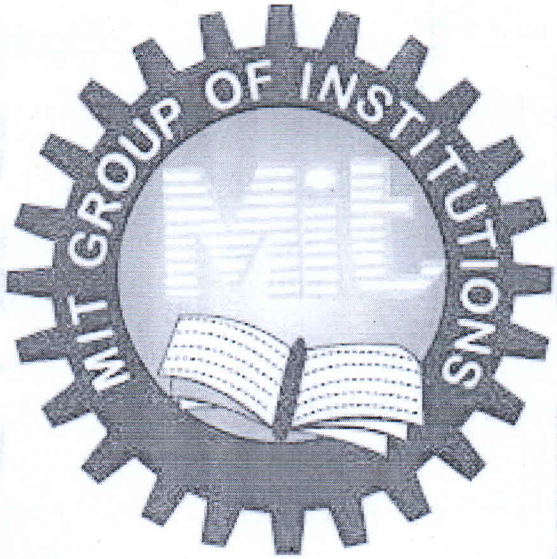
Benefits of using hierarchical structure :

- The hierarchical structure of symbol tables automatically solves the problem of resolving name collisions (identifiers with the same name and overlapping scopes).
- To find the declaration of an identifier that is active at a program point (i.e being executed). Start from the current scope, go up in the hierarchy until you find an identifier with the same name, or fail (when no reference is found going up the hierarchy till the end of global symbol table).

References:

- Javapoint(website).
- Introduction to compilers(pdf).

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001



Topic: Implementation Of Simple Stack Allocation Scheme

Submitted By :

Pratham Maheshwari (1708210110)

Priyank Raghav (1708210112)

Ranojit Malik (1708210116)

Submitted To :

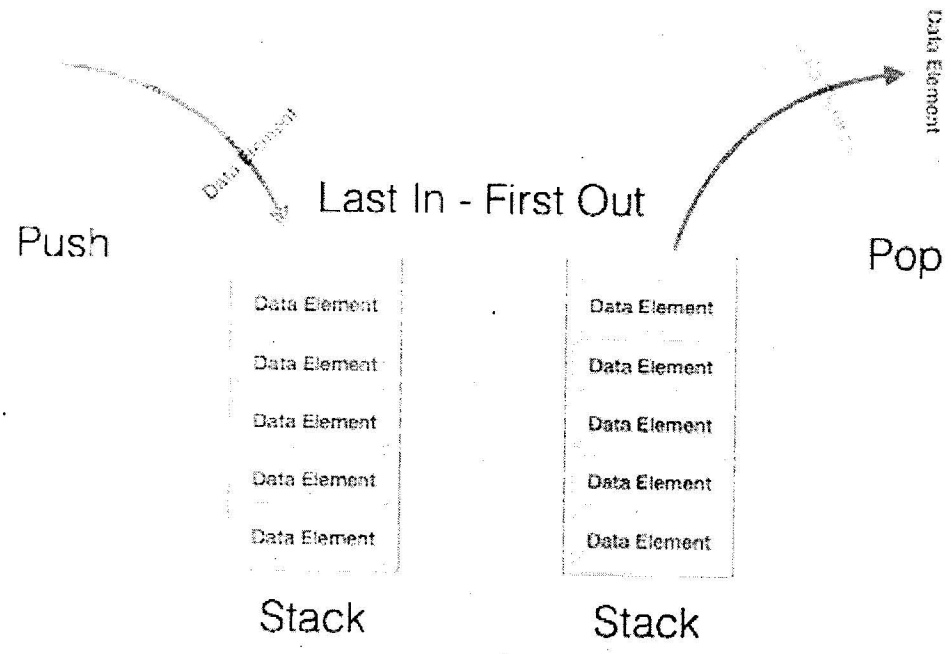
Mrs. Priyanka Agarwal

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Implementation Of Simple Stack Allocation Scheme

We are going to consider an implementation of the UNIX programming language **C**, which allows somewhat simpler implementations that some other stack-oriented languages like **ALGOL**.

Data in **C** can be global, meaning it is allocated static storage and available to any procedure, or local, meaning it can be accessed only by the procedure in which it is declared. A program consists of a list of global data declarations and procedures; there is no block structure.



Stack allocation :

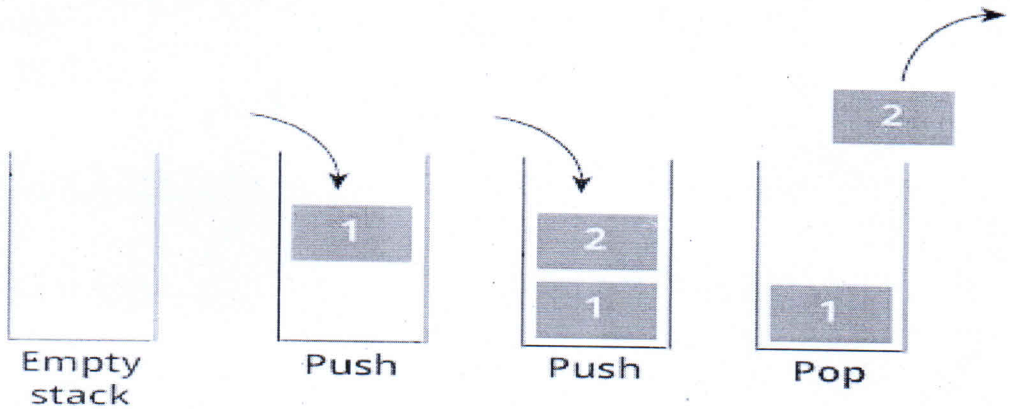
Stack Allocation Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (**LIFO**)method and this allocation strategy is very useful for recursive procedure calls.

- Information needed by a single execution of a routine (a procedure, function or method) is managed using an activation record or frame.
- An activation records contains local variables, parameters, return address, etc.
- Activation records are pushed and popped as functions calls begin and end.

Stack Operations :

The bottom of a stack is a sealed end. Stack may have a capacity which is a limitation on the number of elements in a stack. The operations on stack are:

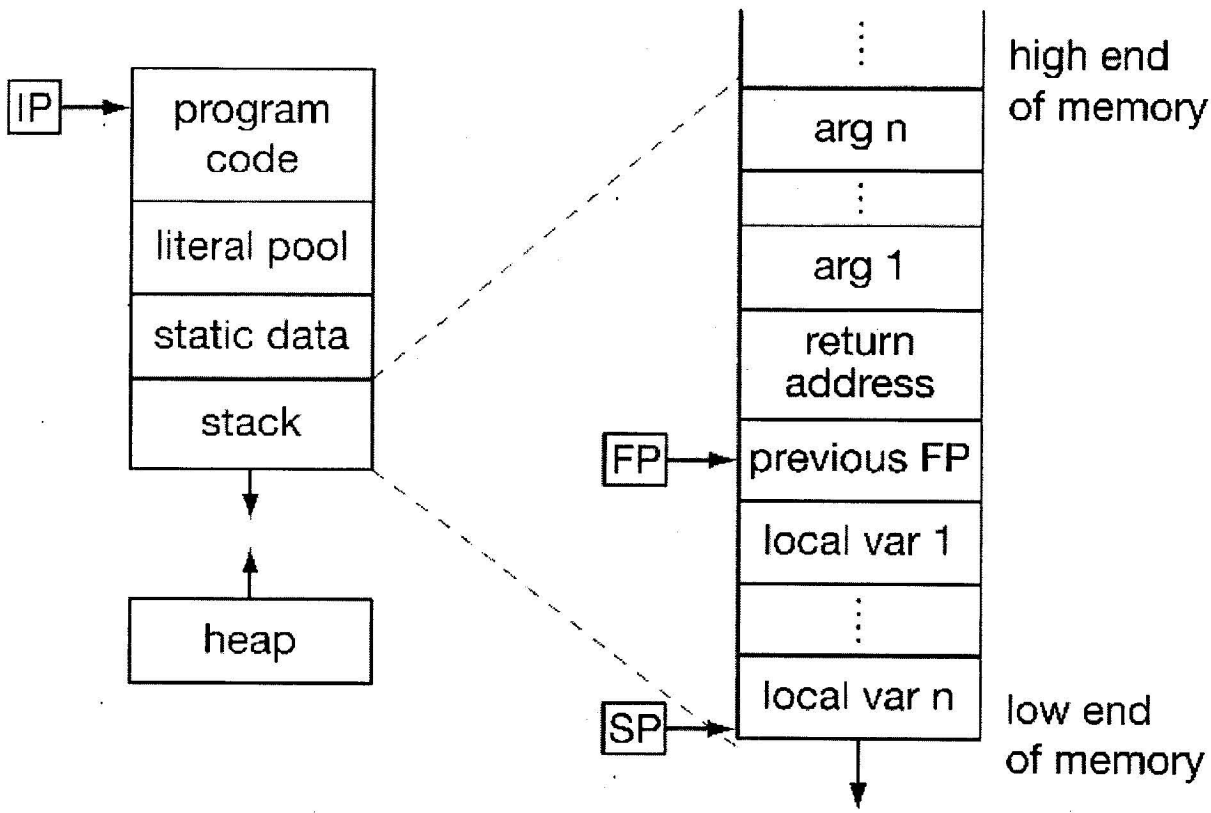
- **Push:** Places an object on the *top* of the stack.
- **Pop:** Removes an object from the *top* of the stack.
- **IsEmpty:** Reports whether the stack is empty or not.
- **IsFull:** Reports whether the stack exceeds limit or not.



Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Stack Layout :

- When calling a subroutine, push a new entry on the stack-stack frame(activation record)
- When retuning from a subroutine, pop its frame from the stack



- Stack pointer register (sp) - address of the first unused location at top of stack.
- Frame pointer register (fp) – address within current stack frame.

Stack Memory :

- Local variables for functions, whose size can be determined at call time.
- Information saved at function call and restored at function return:

- Values of callee arguments
- Register values:
  - Return address (value of PC)
  - Frame pointer (value of FP)
  - Other registers
- Static link (to be discussed)

Stack Terminology:

- **Stack Underflow:** This is the situation when the stack contains no element. At this point the top of stack is present at the bottom of the stack.
- **Stack Overflow:** This is the situation when the stack becomes full, and no more elements can be pushed onto the stack. At this point the stack top is present at the highest location of the stack.

Stacks :

- Real life examples.
- Shipment in a Cargo.
- Plates on a tray.
- Stack of coins.
- Stack of Drawers.
- Shunting of trains in railway yard.
- Follows the Last-In First-Out (LIFO) strategy.

Limitations :

- The size required must be known at compile time.
- Recursive procedures cannot be implemented statically.
- No data structure can be created dynamically as all data is static.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Advantages :

- It supports recursion as memory is always allocated on block entry.
- It allows to create data structures dynamically.
- It allows an array declaration like A(I, J), since actual allocation is made only at execution time. The dimension bounds need not be known at compile time.

Disadvantages :

- Memory addressing has to be effected through pointers and index registers which may be store them, static allocation especially in case of array reference.

Conclusion :

- Using a Dynamic array to implement a stack meets the ADT specification requirements for a stack.
- Doing so does NOT limit the stack size (like a static array).
- Amortization Analysis is required to see how it is also an efficient way to implement a stack (Intuitively it is not necessarily obvious).

Prepared By –
Tanya Duggal

STORAGE ALLOCATION IN BLOCK STRUCTURE LANGUAGES:-

The block structured storage allocation can be done using static scope

Scope storage allocation includes-

->Definition of procedure.

->Declaration of a name or variable.

->Scope of declaration.

- **Scope rules:**

-> A data declaration using a name creates a variable name var (name,var).

->Variable var is visible at a place in the program if some binding (name,var) us available at that place.

->It is possible for data declaration in many blocks of a program to use a same name.

->Scope rules determine which of these bindings is effective at a specific place in the program.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

- **Scope of a variable:**

->If a variable var is created with a name nam in a block b,

-var can be accessed in any statement situated in block b.

-var can be accessed in any statement situated in block b which is enclosed in b unless b contains a declaration using the same name.

->A variable declared in block b is called local variable of block b.

->A variable of enclosing block that is accessible wihin block b is called a non local variable.

- Scope of a variable
  - Example

```
x, y, z : integer;
    g : real;
B   C   h, z : real;
A
        i, j : integer;
    D
```

Block	Accessible variables	
	local	nonlocal
A	x_A, y_A, z_A	---
B	g_B	x_A, y_A, z_A
C	h_C, z_C	x_A, y_A, g_B
D	i_D, j_D	x_A, y_A, z_A

- **Memory allocation and access:**

->Automatic memory allocation can be implemented using the extended stack model.

->Each record in a stack has two reserved pointers.

->Each stack record accomodates the variables for one activation of a block, hence called an activation record.

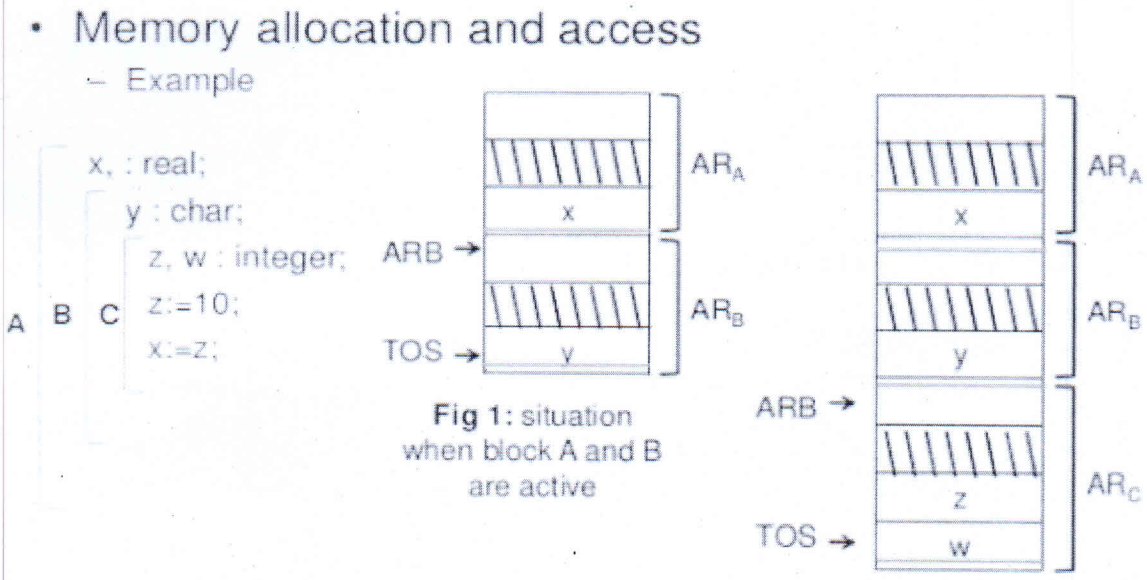


Fig 1: situation when block A and B are active

Fig 2:situation after entry to c.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

– Action at entry of block C

Sr. No.	Statement
1	TOS := TOS + 1;
2	TOS* := ARB; {set the dynamic pointer}
3	ARB := TOS;
4	TOS := TOS + 1;
5	TOS* := ...; {set the dynamic pointer 2}
6	TOS := TOS + n;

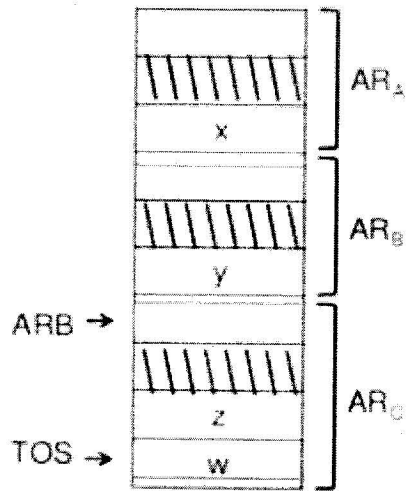


Fig 2

– Action at exit of block C

Sr. No.	Statement
1	TOS := ARB - 1;
2	ARB := ARB*;

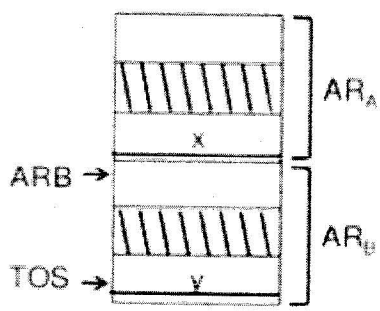


Fig 1: situation when block A and B are active

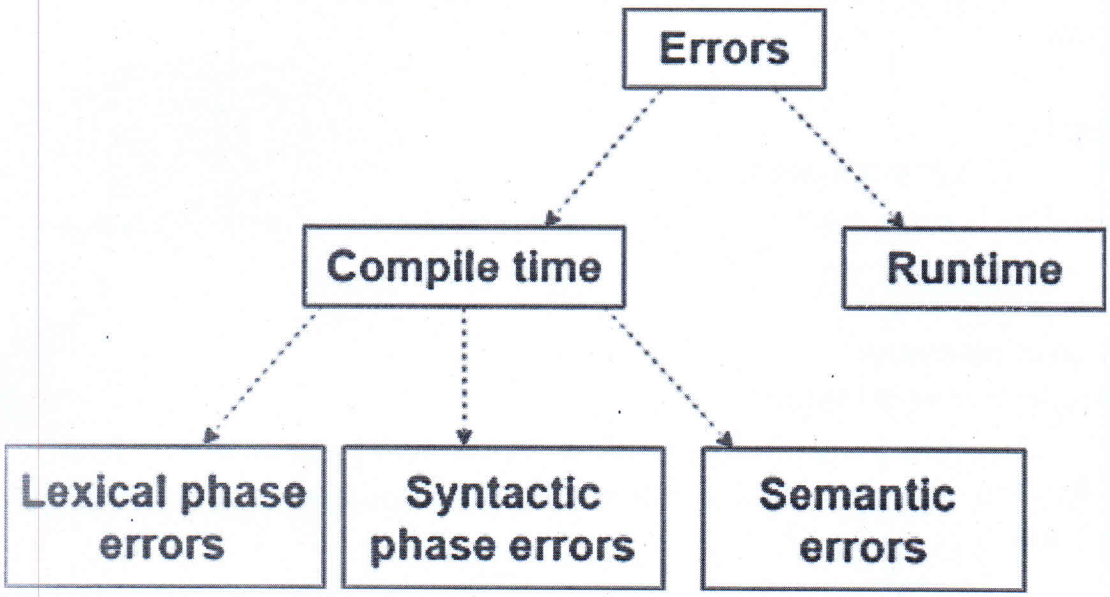
ERROR DETECTION AND RECOVERY

In this phase of compilation, all possible errors made by the user are detected and reported to the user in form of error messages. This process of locating errors and reporting it to user is called **Error Handling process**.

Functions of Error handler

- Detection
- Reporting
- Recovery

Classification of Errors



1. A **run-time error** is an error which takes place during the execution of a program, and usually happens because of adverse system parameters or invalid input data. The lack of sufficient memory to run an application or a memory conflict with another program and logical error are example of this. Logic errors, occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.

2. **Compile-time errors** rises at compile time, before execution of the program. Syntax error or missing file reference that prevents the program from successfully compiling is the example of this.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Compile time errors are of three types:-

1) **Lexical phase errors**

These errors are detected during the lexical analysis phase. Typical lexical errors are

- Exceeding length of identifier or numeric constants.
- Appearance of illegal characters
- Unmatched string

Example 1 :

printf("hello");$

This is a lexical error since an illegal character $ appears at the end of statement.

Example 2 :

This is a comment */

This is an lexical error since end of comment is present but beginning is not present.

Lexical error recovery:

Panic Mode Recovery Method –

It identifies the token and if token is not matched then successive characters from token are removed one at a time until a designated set of synchronizing tokens is found i.e , or ; or }

Advantage is that it is easy to implement and guarantees not to go to infinite loop

Disadvantage is that a considerable amount of input is skipped without checking it for additional errors

2) **Syntactic phase errors**

A syntactic error is an error in the syntax of a sequence of characters or tokens that is intended to be written in a particular programming language. For compiled languages, syntax errors are detected at compile-time. A program will not compile until all syntax errors are corrected. Thus, the error detectable by the lexical or syntactic phase of the compiler is defined as Syntactic Error.

Examples of Syntactic Errors

1.Missing right parenthesis:

ex- printf("Hello" ; {Deletion Error}

2.Missing Operator:

ex- a+b c; {Deletion Error}

3.Extraneous Comma:

ex- int a, , b; {Insertion Error}

4.Colon in place of semicolon:

ex- i=1: {Replacement Error}

5. Misspelled keyword:

ex- swicth(ch) {Transposition Error}

6. Extra Blank:

ex- /*comtment * / {Insertion Error}

In above examples, it is easy for a human being to determine the type and position of error.

But, often it is not easy to say exactly how many errors have occurred and where they have occurred.

Ex- a:=b-c*d+e)

Clearly, we can't figure out whether an extra right parenthesis is added or a left parenthesis is missing, if so, what is its position?

Minimum Distance Correction of Syntactic Errors

One theoretical way of defining errors and their location is the minimum Hamming distance method. We define a collection of error transformations. We then say that a program P has K errors if the shortest sequence of error transformations that will map any valid program into P has length k.

Although minimum distance error correction is a convenient theoretical yardstick, it is not generally used in practice because it is too costly to implement.

Syntactic error recovery:

1. **Panic Mode Recovery**
   - In this method, successive characters from input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are deli-meters such as ; or }

- Advantage is that its easy to implement and guarantees not to go to infinite loop
- Disadvantage is that a considerable amount of input is skipped without checking it for additional errors

2. **Statement Mode Recovery / Phrase Level Recovery**
   - In this method, when a parser encounters an error, it performs necessary correction on remaining input so that the rest of input statement allow the parser to parse ahead.
   - The correction can be deletion of extra semicolons, replacing comma by semicolon or inserting missing semicolon.
   - While performing correction, atmost care should be taken for not going in infinite loop.
   - Disadvantage is that it finds difficult to handle situations where actual error occured before point of detection.
3. **Error production**
   - If user has knowledge of common errors that can be encountered then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.
   - If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.
   - Disadvantage is that its difficult to maintain.
4. **Global Correction**
   - The parser examines the whole program and tries to find out the closest match for it which is error free.
   - The closest match program has less number of insertions, deletions and changes of tokens to recover from erroneous input.
   - Due to high time and space complexity, this method is not implemented practically.

Error handling in LL parsers

We have two major concerns in syntactic error recovery: to avoid infinite loops and to avoid producing corrupt syntax trees.

A third possibility is to discard tokens from the input until a matching token is found.

A fourth possibility is inserting a non-terminal at the front of the prediction, to force a match, but this would again lead to a corrupt syntax tree.

The acceptable-set method

The three steps performed here are-

Step 1: construct the acceptable set A from the state of the parser, using some suitable algorithm C; it is required that A contain the end-of-file token;

- Step 2: discard tokens from the input stream until a token tA from the set A is found;

- Step 3: resynchronize the parser by advancing it until it arrives in a state in which it consumes the token tA from the input, using some suitable algorithm R; this prevents looping.

Error handling in LR parsers

An LR parser will detect an error when it consults the parsing action table and find a blank or error entry.

=) LR shift-reduce conflicts can be resolved by always preferring shift over reduce;

=) LR reduce-reduce conflicts can be resolved by accepting the longest sequence of tokens for the reduce action. The precedence of operators can also help.

=) Generalized LR (GLR) solves the non-determinism left in a non-deterministic LR parser by making multiple copies of the stack, and applying the required actions to the individual stacks. Stacks that are found to lead to an error are abandoned. The stacks can be combined at their heads and at their tails for efficiency; reductions may require this combining to be undone partially.

=) Ambiguous grammars can sometimes be made unambiguous by developing the rule that causes the ambiguity until it becomes explicit; then all rules causing the ambiguity except are removed, and the developing action is rolled back partially.

=) When an error occurs, states are removed from the stack until a state is uncovered that allows a shift on an error recovering non-terminal R; next, a

dummy node R is inserted; finally, input tokens are skipped until one is found that is acceptable in the new state. This attempts to remove all traces of the production of R and replaces it with a dummy R.

3) **Semantic phase error**

During the semantic analysis phase, this type of error appears. These types of error are detected at compile time.

Most of the compile time errors are scope and declaration error. **For example:** undeclared or multiple declared identifiers. Type mismatched is another compile time error.

The semantic error can arises using the wrong variable or using wrong operator or doing operation in wrong order.

Some semantic error can be:

- Incompatible types of operands
- Undeclared variable
- Not matching of actual argument with formal argument

Example 1: Use of a non-initialized variable:

```
int i;
void f (int m)
{
    m=t;
}
```

In this code, t is undeclared that's why it shows the semantic error.

Example 2: Type incompatibility:

```
int a = "hello";     // the types String and int are not compatible
```

Example 3: Errors in expressions:

```
String s = "...";
int a = 5 - s;    // the - operator does not support arguments of type String
```

Semantic error recovery:

- A symbol table entry for correlating identifier is created to recover in case of occurrence of "Undeclared Identifier" error.

- If data types of two operands are not compatible with each other then, the compiler performs automatic type conversion.

Zainab Azeem
Rishakha Rastogi
Rila Zehra
Aryan Thakur
Manglam Sharma

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Unit 5 CODE GENERATION

Design Issues

The final phase in our compiler model is the code generator It takes as input an intermediate representation of the source program and produces as output an equivalent target program

Requirements for code generator

* The output code must be correct & of high quality ie it should make effective use of the resources of target machine
* The code generator itself should run efficiently.

Issues

(1) Input to the Code Generator :- The input consists of intermediate representation of source program produced by front end together with i/f in the symbol table that is used to determine the run time addresses of the data objects denoted by names in intermediate representation.

It is assumed that the front end has scanned, parsed and translated the source program into detailed intermediate code. It is also assumed that necessary type checking has taken place and semantic/syntax errors have already been detected.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

(2) Target program : The output of code generator is the target program which may take variety of forms — absolute machine language, relocatable m/c language or assembly language.

* Producing absolute m/c language program as o/p has the advantage that it can be placed in fixed location in m/m & immediately executed.

* Producing relocatable m/c language program as o/p allows subproblems to be compiled separately. A set of relocatable object modules can be linked together & loaded for execution by linking loader. The advantage is getting flexibility in being able to compile subroutines separately & call other previously compiled programs from an object module. The disadvantage is extra overhead of linking & loading.

* Producing assembly language program as o/p makes the p/c of code generation easier. We can generate symbolic instructions and use macro facilities of assembler to help generate code.

(3) Memory Management : Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by front end and code generator.

If machine code is being generated, labels in 3-address instr have to be converted to addresses of instructions.
This process is analogous to the 'backpatching' technique.

(4) Instruction Selection : The nature of the instruction set of the target m/c determines the difficulty of instruction selection. If the target machine doesnot support each data type in a uniform manner, then each exception to the general rule requires special handling.

Consider the following statements :-

```
a := b + c
d = a + e
```

would be translated as —

```
MOV R0, b
ADD c, R0
MOV a, R0
MOV R0, a
ADD e, R0
MOV d, R0
```

] → these 2 statements are redundant here

The quality of generated code is determined by its speed and size.
e.g if 3-address statement is a := a + 1 and target m/c has increment instruction INC then it will be implemented efficiently. Otherwise following code need to be used :-

```
MOV R0, a
ADD R0, #1
MOV a, R0
```

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

(5) Register Allocation : Instructions involving register operands are usually shorter and faster than those involving operands in memory.
The use of registers is often subdivided into two subproblems—

a) During register allocation, we select the set of variables that will reside in registers at a point in program.

b) During a subsequent register assignment phase, we pick specific register that a variable will reside in.

(6) Choice of Evaluation Order : The order in which computations are performed can affect the efficiency of target code. Some computation orders require fewer registers to hold intermediate results than others.

(7) Approaches to Code generation : The most important criterion for a code generator is that it produce correct code. One such optimization technique is peephole optimization.

The Target Machine

* Familiarity with target machine and its instruction is a pre-requisite for designing a good code generator.
* Here, we are assuming that our target computer is a byte-addressable machine with four bytes to a word and 'n' general purpose registers R0, R1, ..., Rn-1.
* It has two-address instructions of the form :-

op source, destination

op-code — data field — data field

* Assuming that it has the following op-codes (among others) :-

MOV (move source to destination)
ADD (add source to destination)
SUB (subtract source from destination)

* We are considering following address modes together with their assembly language forms & associated costs :-

MODE	FORM	ADDRESS	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	c(R)	c + contents (R)	1
indirect register	*R	contents (R)	0
indirect indexed	*c(R)	contents(c + contents (R))	1
literal	#c	constant c	1

e.g. MOV R0, M

stores contents of register R0 into memory location M

MOV 4(R0), M

stores the value of contents(4+ contents(R0)) into memory location M

MOV *4(R0), M

stores the value of contents (contents (4+ contents (R0))) into memory location M.

MOV #1, R0

loads the constant 1 into register R0.

Instruction Costs

* Cost of instruction = 1 + costs associated with the source and destination address modes
* This cost corresponds to the length (in words) of the instruction.
* Address modes having registers have cost ~~one~~ zero while those with m/m location or literal have cost ~~coz~~ ~~such~~ one coz such operands have to be stored with instruction.

Unit-5

Introduction to Code Optimization

* The term 'code optimization' refers to techniques a compiler can employ in an attempt to produce a better object code for a given source program.

Following criteria can be considered in the selection of optimizing code :—

(a) Does the optimization capture most of the potential improvement w/o an unreasonable amount of effort?

(b) Does the optimization preserve the meaning of the source program?

(c) Does the optimization, atleast on the average, reduce the time or space taken by object program?

Principal Sources of Code Optimization

(1) Efficient utilization of the registers :— this may take place during code generation phase

(2) Inner loops :— governed by 90-10 rule which states that 90% of the time is spent in 10% of code.

Typical loop optimizations are—

(a) Removal of loop invariant computations :— this step is known as code motion

(b) Elimination of induction variables

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

(3) Identification of common sub-expressions

(4) Replacement of runtime computations by compile-time computations (this step is called constant folding) i.e. substitution of values for names whose values are constant.

e.g. J := I + 1
A[J] := B[J]
can be re-written as —
A[I+1] := B[I+1]

A[I+1] := B[I+1]
can be re-written as —
J := I + 1
A[J] := B[J]

(5) Algorithm optimization

e.g. instead of choosing any algorithm of $O(n^2)$ for sorting, we can opt for algorithm of $O(n \log n)$ for the same.

[redacted]

V. Imp

Basic Blocks

* are the sequence of consecutive statements which may be entered only at the beginning & when entered are executed in sequence w/o halt or possibility of branch.

Algorithm to partition a sequence of 3-addr code into basic blocks

Input :- Sequence of 3-addr statements

M/d :-

Step1 :- Determine the set of leaders as —

(a) The first statement is a leader.

(b) Any statement which is the target of conditional or unconditional goto is a leader.

(c) Any statement which immediately follows a conditional goto is a leader.

Step2 :- For each leader, construct its basic block. The basic block consists of leader & all statements upto but not including next leader or, the end of the program.

Note :- Any statements not placed in a block can never be executed & may be removed.

Flow Graphs

A directed graph which is used to represent basic blocks & their successor relationships.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Nodes of flow graph ⇒ basic blocks

↳ one node is entral node (block whose leader is the first statement)

e.g

```
begin
  PROD := 0
  I := 1
  do
    begin
      PROD := PROD + A[I] * B[I]
      I := I + 1
    end
  while I ≤ 20
end.
```

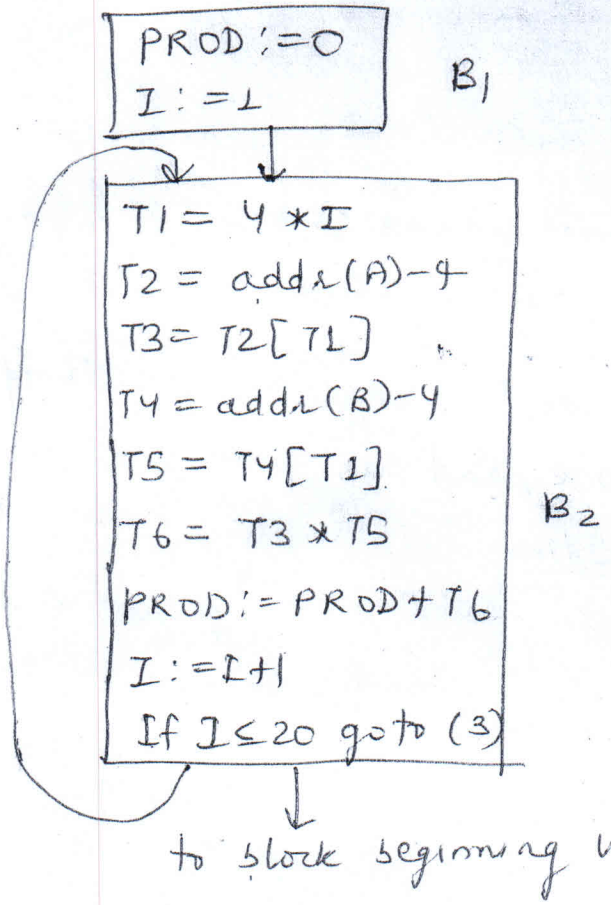
Its three add^r code is — (assuming that m/c has size of four bytes/word)

(1) PROD := 0
(2) I := 1
(3) T1 = 4 * I
(4) T2 = addr(A) - 4
(5) T3 = T2[T1]
(6) T4 = addr(B) - 4
(7) T5 = T4[T1]
(8) T6 := T3 * T5
(9) PROD := PROD + T6
(10) I := I + 1
(11) If I ≤ 20 goto (3)

$$A[I] = BA + w * (I - LB)$$
$$= BA + 4 * (I - 1)$$
$$= BA + 4I - 4$$
$$= (BA - 4) + 4I$$

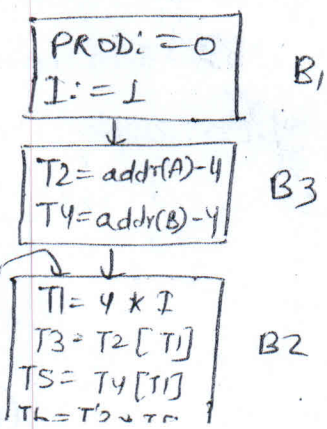
Finding basic blocks

Statement (1) is leader and statement (3) is leader.
So, basic blocks & its corresponding flow graph is —



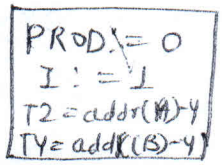
In above code, statements $T_2 = add(A) - 4$ and $T_4 = addr(B) - 4$ are loop-invariant computations (assuming that array A & B are statically allocated). So, these statements can be removed from block B_2. ⇒ This p/c is known as CODE MOTION

So,



⇒

Blocks B1 and B3 can also be combined as —



Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Induction Variables

In the previous example, we can see that value of I is varying from 1 to 20 in the loop. Value of T_1 is also progressing 4 bytes at a time.

i.e. $I = 1, 2, 3 \ldots, 20$ and

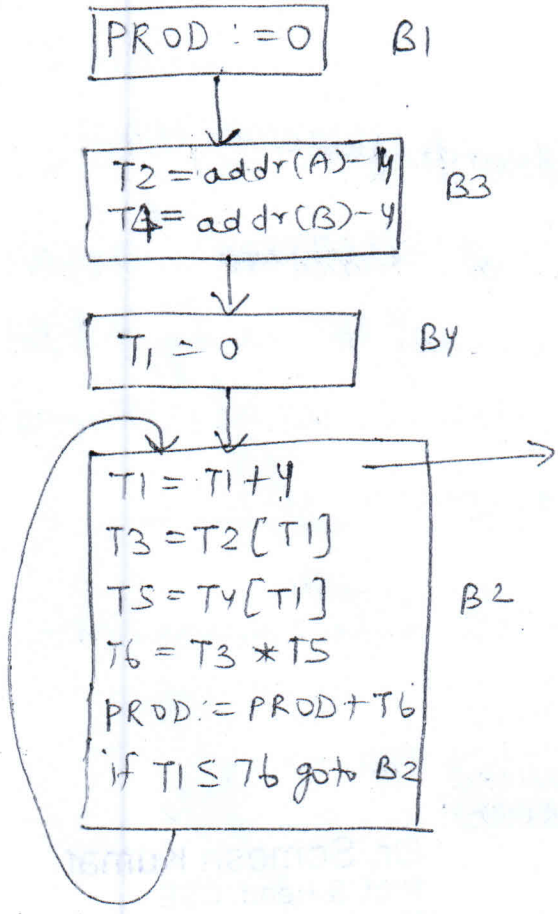
$T_1 = 4, 8, 12 \ldots, 80$

or, both I and T_1 form Arithmetic progressions.

Such variables are called induction variables.

* When there are two or more induction variables in a loop, we can rid all of them except one, and this process is called induction variable elimination.

Thus,



→ the previous multiplication step $T_1 := 4 * I$ is now replaced with $T_1 = T_1 + 4$. This step will speed up object code if addition takes less time than multiplication. Such replacement of an expensive oper^n by a cheaper one is called reduction in strength.

* A three address statement $x = y + z$ is said to define x and to use (or reference) y and z.
* A name in a basic block is said to be live at a given point if its value is used after that point in the program, perhaps in another basic block.

Transformations on basic blocks

* Many transformations can be applied to a basic block w/o changing the set of expressions computed by the block.
* Some transformations re-arrange the computations in a program in an effort to reduce the overall running time or space requirement of final target program.
* There are two important types of local transformations that can be applied to basic blocks —
  a) Structure preserving transformations
  b) Algebraic transformations

STRUCTURE PRESERVING TRANSFORMATIONS

These are —

(i) Common Subexpression Elimination :-

Consider the basic block —

$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$

Here, 2nd & 4th statements are same as these compute same expression 'b+c-d', so this basic block can be transformed as —

$a = b + c$
$b = a - d$
$c = b + c$
$d = b$

these two statements also look same but are different as in 3rd statement value of b used is the one computed in 2nd statement.

(ii) Dead Code Elimination :-
Suppose x is dead (dead means that the name is never used subsequently, at the point where $x = y + z$ (for example) appears in basic block.
Thus, this statement can be safely removed w/o changing the value of basic block.

(iii) Renaming temporary variables :-
If we have statement $t := b + c$ where t is any temporary variable, and if we change this statement as - $u := b + c$ where u is another new temporary variable. Then by doing so, value of basic block does not change.
↳ such basic block is called normal BB
Thus we can create an equivalent basic block to a basic block by just defining a new temporary.

(iv) Interchange of statements :- If we have a block with two adjacent statements $t_1 = b + c$, $t_2 = x + y$

Then we can interchange the two statements w/o affecting the value of block if and only if neither x nor y is t_1 and neither b nor c is t_2.

Note :- A normal basic block permits all statement interchanges that are possible.

ALGEBRAIC TRANSFORMATIONS

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

These transformations are useful when they simplify any given expression or replace any expensive operation by a cheaper one.

e.g. $x = x + 0$ or $x = x * 1$
can be eliminated from a basic block as these statements donot change value of x.

Similarly, $x = y ** 2$
→ exponential operation
is expensive one and can be replaced by $x = y * y$

Loops in flow graph

A loop is a collection of nodes in the flow graph such that :—

a) All nodes in the collection are strongly connected ie from any node in the loop to any other, there is a path of length one or more wholly within the loop and,

b) The collection of nodes has a unique entry i.e. a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.

A loop that contains no other loop is called inner loop.

Register and Address Descriptor

* The code generation algorithm uses descriptors to keep track of register contents & addresses for names.

Register Descriptor :- keeps track of what is currently in each register. It is consulted whenever a new register is needed.
We assume that initially the register descriptor shows that all registers are empty.

Address Descriptor :- keeps track of the location where the current value of the name can be found at run time. This location can be a register, a stack location, a memory address or some set of these.

Code generation Algorithm

The code generation algorithm takes as input a sequence of 3-address statements constituting a basic block.
For each 3-addr statement of the form $x := y \text{ op } z$, following actions are performed :-

(1) Invoke a function getreg to determine the location L where the result of computation 'y op z' should be stored. L could be register or memory location.

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

(2) Consult the address descriptor for y to determine y' (current location of y; we prefer the register value for 'y' if the value of y is currently both in memory and register).
If the value of y is not already in 'L', generate the instruction mov y', L to place a copy of 'y' in 'L'.

(3) Generate the instruction OP z', L where z' is current location of z. Again we prefer a register over memory location if z is present in both.
If L is a register, update its descriptor to indicate that it contains value of x & remove x from all other register descriptors.

(4) If the current values of y or z have no next uses, are not live on exit from block and are in registers then, alter the register descriptor to indicate that, after execution of x = y op z, those registers will not contain y ~~&~~ or z.

Next-Use Information

* This information is collected about names in basic blocks.
* If the name in a register is no longer needed, then the register can be assigned to some other name.

How next use information is computed ?

Suppose we have :-

Statement i : $x = y$ op z

⋮

Statement j : uses x in same form

and there is control flow from statement i to j, then statement j is said to use the value of x computed at i.

Algorithm :-

This algorithm makes a backward scan over each block. Having found the end of basic block, we scan backward to the beginning, recording (in symbol table) for each name x whether x has next use in block and if not, whether it is live on exit from that block.

Suppose we reach 3-addr statement i : $x = y$ op z in our backward scan. Then—

(i) Attach to statement i, the info currently found in symbol table regarding the next use & liveness of x, y & z.

(ii) In the symbol table, set x to 'not live' & 'no next use'.

(iii) Set y and z to 'live' and next uses of y & z to i.

Note that steps (ii) & (iii) cannot be interchanged.

Symbol table :

Names	Liveliness	Next use
x	Not live	no next use
y	live	i
z	live	i

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

More examples

Example1 :- The assignment statement d := (a-b) + (a-c) + (a-c)

Its a 3-addr code is —

t := a-b
u = a-c
v = t + u
d = v + u

Implementation of code generation Algorithm :-

Statements	Code generated	Register Descriptor	Addr descriptor
		Registers empty	
t = a-b	MOV R0, a SUB R0, b	R0 contains t	t in R0
u = a-c	MOV R1, a SUB R1, c	R0 contains t R1 contains u	t in R0 u in R1
v = t+u	ADD R1, R0	R0 contains v R1 contains u	v in R0 u in R1
d = v+u	ADD R0, R1 MOV d, R0	R0 contains d	d in R0 & memory

Example 2 :- Indexed assignment statements a := b[i] and a[i] = b

Statement	if i in Register Ri	if i in Memory Mi
a = b[i]	MOV R, b(Ri) ↳ store value 'a'	MOV R, Mi MOV R, b(R)
a[i] = b	MOV a(Ri), b	MOV R, Mi MOV a(R), b

Example 3 :- Pointer assignment statements a := *p and *p = a

Statement	if p in Register Rp	if p is memory Mp
a = *p	MOV a, *Rp	MOV R1, Mp MOV R, *R1
*p = a	MOV *Rp, a	MOV R1, Mp MOV *R1, a

Example 4 :- Conditional statements

Method 1 :- Branching is done if the value of designated register meets one of following six conditions — negative, zero, positive, non-negative, non-zero & non-positive.

Method 2 :- Machine uses a set of condition codes to indicate whether the last quantity computed or loaded into a register is -ve, zero or +ve.

One such code is Compare Instruction (CMP) which sets the condition code w/o actually computing a value.

A conditional jump (CJ) machine instr makes the jump if a designated condition <, =, >, <=, ≠, >= is met.

e.g. CJ <= z means jump to z if condition code is -ve or zero

if x < y goto z can be implemented as—

CMP x, y

CJ< z

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

The DAG Representation of Basic Blocks

* Directed Acyclic Graphs (DAG) are useful data structures for implementing transformation on basic blocks.
* A DAG gives a picture of how the value computed by each statement in a basic block is used in subsequent statements of the block.
* Constructing a DAG from 3-addr statements is a good way of determining common subexpressions within a block, determining which names are used inside the block but evaluated outside the block, and determining which statements of the block could have their computed value used outside the block.

A DAG for a basic block is a graph with following labels on nodes :-

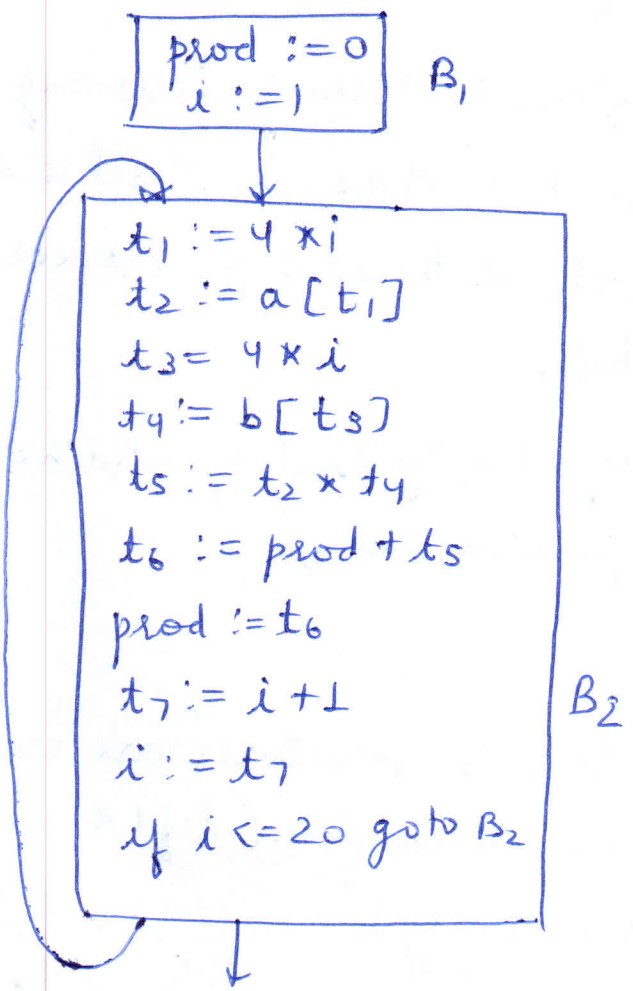
(1) Leaves are labeled by unique identifiers, either variable names or constants.
There are two values associated with a name — l-value or r-value so most leaves represent r-values.
The leaves represent initial value of names & are subscripted with 0 to avoid confusion with a label denoting current values of names.

(2) Interior nodes are labeled by an operator symbol.

(3) Nodes are also optionally given a sequence of identifiers for labels. Interior nodes represent computed values and the identifiers labeling a node are deemed to have that value.

Note that DAGs and flow graphs are different things as each node of a flow graph can be represented using DAGs. Node of a flow graph represents a basic block.

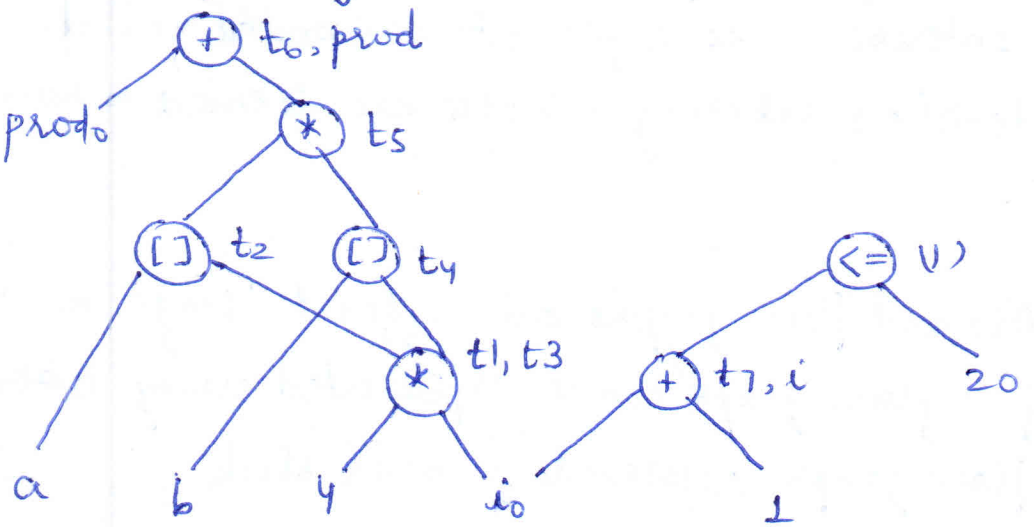
e.g. Consider the 3-addr code of basic block B_2 of following flow graph :-



3-addr code of B2 ↓

(1) $t_1 := 4 * i$
(2) $t_2 := a[t_1]$
(3) $t_3 := 4 * i$
(4) $t_4 := b[t_3]$
(5) $t_5 := t_2 * t_4$
(6) $t_6 := prod + t_5$
(7) $prod := t_6$
(8) $t_7 := i + 1$
(9) $i := t_7$
(10) if $i <= 20$ goto (1)

Its corresponding DAG is –



Algorithm to construct DAG

Input :- A basic block

Output :- A DAG for basic block containing following if :-

(1) A label for each node :- For leaves, the label is an identifier (constants can also be there) & for interior nodes, an operator symbol.

(2) For each node a (possibly empty) list of attached identifiers (constants not permitted here).

Assumptions & Requirements :-

(1) We assume the appropriate data structures are available to create nodes with one or two children (left & right child should be identified uniquely).

(2) A linked list would be required to attach identifiers for each node.

(3) There is a function node(identifier) which as we build DAG, returns the most recently created node associated with identifier.

(4) Initially, there are no nodes and node() is undefined for all arguments.

(5) Let the current 3-address statement is either —

(i) $x := y \text{ op } z$

(ii) $x := \text{op } y$

(iii) $x := y$

Any relational operator expr will be considered as case(i) in this example only.

Algorithm :—

(1) If node(y) is undefined, create a leaf labelled y & let node(y) be this node.
In case(i), if node(z) is also undefined, create a leaf labelled z and let node(z) be this node.

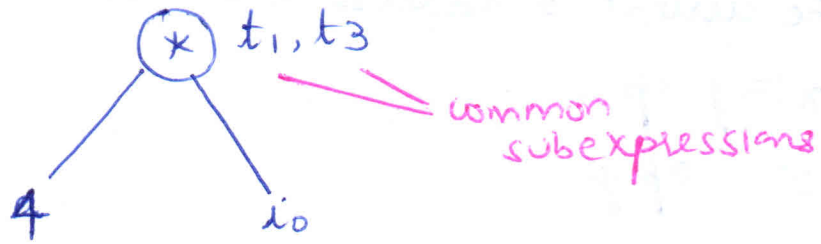
(2) In case(i), find if there is a node labelled op, whose left child is node(y) & right child is node(z). If not, create such node. Let n be the node created or found.
In case(ii), determine whether there is a node labelled op, whose lone child is node(y). If not, create such node & left it be n.
In case(iii), let n be node(y).

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

(3) Delete x from the list of attached identifiers for node(x). Append x to the list of attached identifiers for node n found in (2) & set node(x) to n.

Applications of DAG

(1) We can automatically detect common subexpressions.



(2) We can determine which identifiers have their values used in the block, — they are those for which a leaf is created in step(1) of algorithm.

(3) We can determine which statements compute values that could be used outside the block — these are those statements whose node n constructed/found in step(2) still has node(x) = n at the end of DAG construction.

(4) DAG can be used to re-construct a simplified list of quadruples taking advantage of common subexpressions and not performing assignments of the form x := y unless absolutely necessary.

So, reducing the steps from given basic block, after ~~consider~~ constructing DAG will be —

Earlier one

$t_1 := 4 * i$
$t_2 := a[t_1]$
$t_3 := 4 * i$
$t_4 := b[t_3]$
$t_5 := t_2 * t_4$
$t_6 = prod + t_5$
$prod := t_6$
$t_7 := i + 1$
$i = t_7$
if $i <= 20$ goto (1)

Common SE

→

$t_1 := 4 * i$
$t_2 := a[t_1]$
$t_4 := b[t_1]$
$t_5 := t_2 * t_4$
$prod := prod + t_5$
$i = i + 1$
if $i <= 20$ goto (1)

Peephole optimization

* A statement-by-statement code generation strategy produces target code that contains redundant instructions and suboptimal constructs.
* The quality of such target code can be improved by applying 'optimizing' transformations to the target program.
* One such technique is 'Peephole optimization'.
* In this technique, short sequence of target instructions (called peephole) is examined and these instructions are replaced by shorter or faster instructions, whenever possible.
* The peephole is a small, moving window on the target program.
* The code in the peephole need not be contiguous but in some cases, it is required.
* This technique requires repeated passes over the target code to get maximum benefit.
* Following are characteristics of peephole optimization :-
  a) Redundant Instruction Elimination
  b) Flow-of-control optimizations
  c) Algebraic Simplifications
  d) Use of machine Idioms

Redundant Loads & Stores

If we have the instruction sequence as —

(1) MOV R0, a

(2) MOV a, R0

we can delete instruction (2) because whenever (2) is executed, (1) will ensure that the value of a is already in register R0.

Unreachable Code

Peephole optimization also removes unreachable instructions. Like an unlabeled instruction immediately following an unconditional jump may be removed.

e.g.
```
#define debug 0
  :
if (debug) {
    print debugging info
}
```

This code can be represented in intermediate code as :-

```
if debug = 1 goto L1
goto L2
L1: print debugging info
L2:
```

⇒ This code contains jumps over jumps

It can also be ~~used~~ replaced as —

```
if debug ≠ 1 goto L2
print debugging information
```

But it should be noted that in the above code, debug is set to 0 at the beginning of program; so constant propagation technique* can be implemented as —

```
if 0 ≠ 1 goto L2
print debugging information
L2:
```

* Constant propagation is the process of substituting the values of known constants in expressions.
This technique eliminates cases in which values are copied from one location ~~or~~ to another, in order to simply assign their value to another variable.

e.g. $x = 14$
$y = 7 - x/2$

can be replaced as-

$x = 14$
$y = (7 - 14)/2$

Flow-of-control optimizations

The intermediate code generation may produce jumps-to-jumps, jumps to conditional jumps, or conditional jumps to jumps.
These unnecessary jumps can be eliminated in either of intermediate code or the target code by following types of peephole optimizations :-

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

e.g.1 goto L1 ... L1: goto L2 —replaced by→ goto L2 ... L1: goto L2

if there are no jumps to L1, then it can be possible to eliminate statement L1: goto L2

e.g.2 if a<b goto L1 ... L1: goto L2 —replaced by→ if a<b goto L2 ... L1: goto L2

Algebraic Simplification

Statements such as: $x := x + 0$ or, $x := x * 1$ can be eliminated easily using peephole optimization.

Reduction in strength

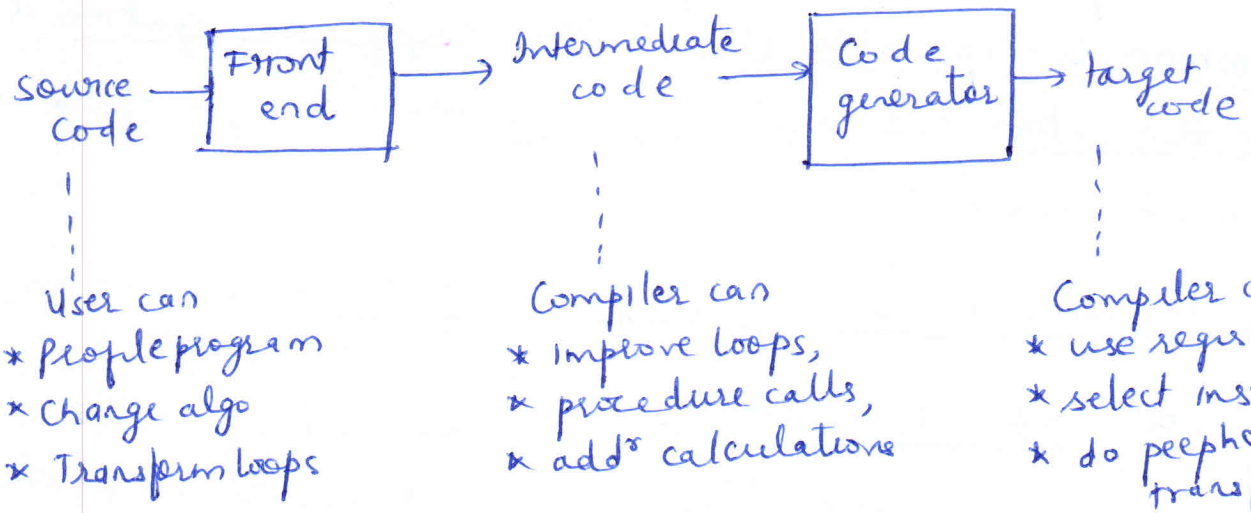
It replaces expensive operations by equivalent cheaper ones on the target machine.

Use of Machine Idioms

The target machine may have h/w instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly.

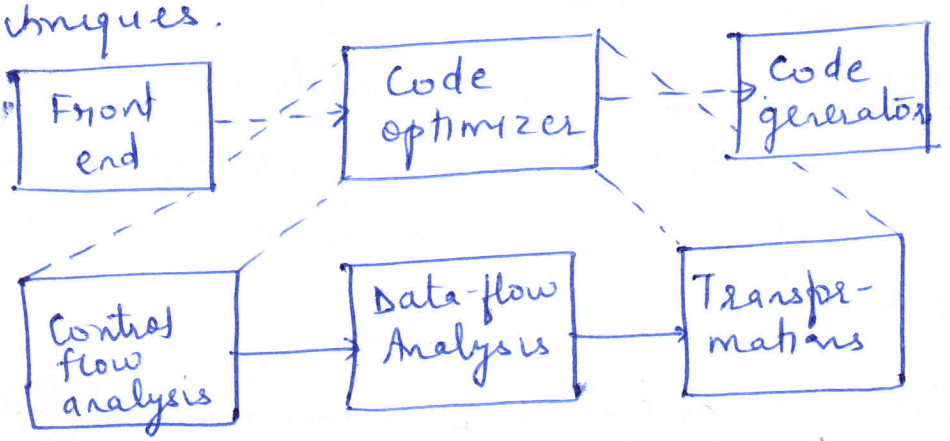
e.g. some m/cs have auto-increment & auto-decrement modes which can be used directly to add or subtract one from an operand respectively.

CODE OPTIMIZATION



Possibilities of optimization by user & the compiler

* In previous topics, we have discussed machine-dependent optimization techniques.
* Now we will discuss about m/c-independent optimization techniques.



Organization of code optimizer

Principle sources of Optimization

* A transformation of a program is called local if it can be performed by looking only at the statements in basic block, otherwise it is called global.

(1) Function Preserving Transformations :- ways in which a program can be improved w/o changing the function it computes. Common subexpression elimination, copy propagation, dead code elimination and constant folding are example of such transformations.

(a) Common subexpression elimination :- discussed on page 13.

(b) Copy propagation :- It means that if we have $f := g$ somewhere, then we should use g for f whenever possible after the copy statement $f := g$

e.g.

$x := t_3$
$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2

can be re-written as

$x := t_3$
$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2

(c) Dead code elimination :- already discussed on page 14

(d) Constant folding :- If at compile time, compiler comes to know that the value of an expr is constant & instead of expr, constant can be used, it is called constant folding.

(2) Loop optimizations :- Three techniques are used for loop optimization — code motion (which moves code outside the loop), induction variable elimination & reduction in strength (which replaces an expensive operation by cheaper one)

↳ Already discussed on page no. 11 & 12.

Optimization of Basic Blocks

(1) Optimization of Basic blocks can be done by constructing DAG of basic block, by which it is easier to find common subexpressions, dead code etc.

(2) Another way is the use of Algebraic Identities.

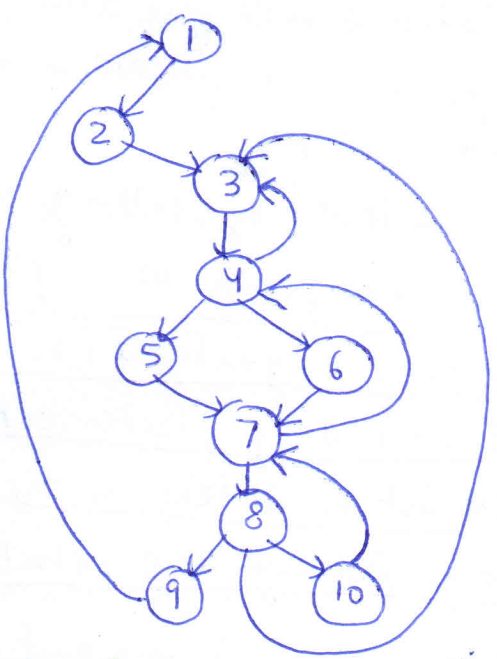
↳ discussed on page no. 15

Loops in Flow Graphs

Dominators :- Node 'd' in a flow graph is said to dominates node 'n', written as d dom n, if every path from initial node of the flow graph to n goes through d.

Thus, every node dominates itself & the entry of a loop dominates all nodes in the loop.

e.g

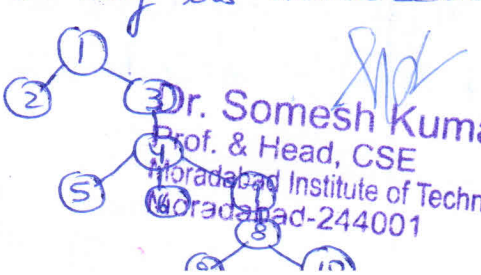


Here,

* Initial node dominates every node
* Node 2 dominates only itself
* Node 3 dominates all except 1,2
* Node 4 dominates all except 1,2,3

The information related to dominators can be represented using tree called dominator tree, in which initial node is the root & each node d dominates only its descendants in tree.

e.g. dominator tree of above graph :-



Natural loops :- One important application of dominator i/f can be to determine the loops of a flow graph which can be improved.

There are two essential properties of such loops —

(a) A loop must have single entry point called header. This entry point dominates all nodes in loop or, it would not be the sole entry to the loop.

(b) There must be atleast one way to iterate the loop i.e. atleast one ~~point~~ path to go back to the header.

So, A good way to find all loops in flow graph is to search for edges in the graph whose heads dominate their tails.

e.g. if $a \to b$ is an edge (such edges are called back edges)

a: tail, b: head

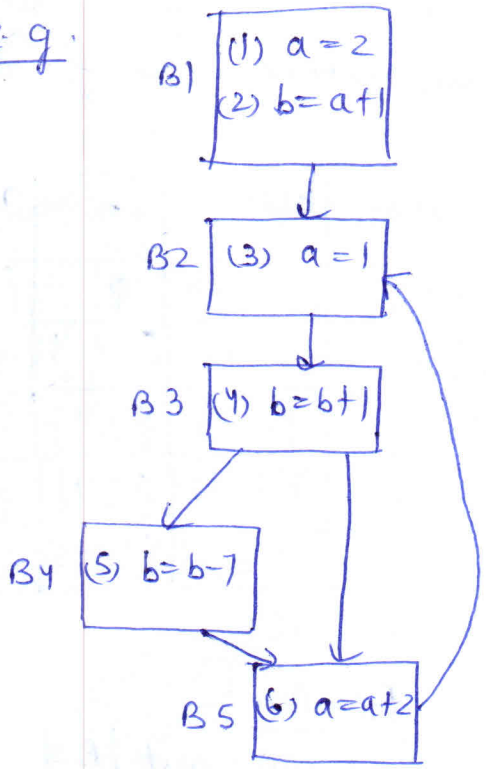
GLOBAL DATA FLOW ANALYSIS

* In order to do code optimization & code generation, a compiler needs to collect i/f about the program as a whole & to distribute this i/f to each block in flow graph.
* Data flow information can be collected by setting up & solving s/s of eqn that relate i/f at various points in a program.
* The eqn is of the form :- $out[s] = gen[s] \cup (in[s] - kill[s])$

and it means — 'the i/f at the end of statement is either generated within the statement or enters at the beginning & is not killed as control flows through the statement.'

out[s] :- definitions that reach block B's exit
gen[s] :- definitions within Block B that reach end of B
in[s] :- definitions that reach B's entry
kill[s] :- definitions that never reaches the end of B.

e.g.



ITERATION 1 :-

Block	Generator	Kill	In	Out
B1	{1, 2}	{3, 4, 5, 6}	ϕ	{1, 2}
B2	{3}	{1, 6}	ϕ	{3}
B3	{4}	{2, 5}	ϕ	{4}
B4	{5}	{2, 4}	ϕ	{5}
B5	{6}	{1, 3}	ϕ	{6}

Kill :- means statements where value used in that particular block, is changed in which other blocks.

Initially, no block is taking any i/p so In(s) for all block is ϕ

ITERATION 2 :-

We will use the data flow analysis eqn to update the values –

$$out(s) = gen(s) \cup \{ in(s) - kill(s) \}$$

For B1 :-

$$out(B1) = gen(B1) \cup \{ in(B1) - kill(B1) \}$$
$$= \{1, 2\} \cup \{ \phi - \{3, 4, 5, 6\} \}$$
$$= \{1, 2\}$$

For B2 :-

$$in(B2) = out(B1) \cup out(B5)$$
$$= \{1, 2\} \cup \{6\} = \{1, 2, 6\}$$
$$out(B2) = gen(B2) \cup \{ in(B2) - kill(B2) \}$$
$$= \{3\} \cup \{ \{1, 2, 6\} - \{1, 6\} \}$$
$$= \{2, 3\}$$

For B3 :-

$$in(B3) = out(B2) = \{3\}$$
$$out(B3) = gen(B3) \cup \{ in(B3) - kill(B3) \}$$
$$= \{4\} \cup \{ \{3\} - \{2, 5\} \}$$
$$= \{3, 4\}$$

Block	Generator	Kill	In	Out
B1	{1, 2}	{3, 4, 5, 6}	{ϕ}	{1, 2}
B2	{3}	{1, 6}	{1, 2, 6}	{2, 3}
B3	{4}	{2, 5}	{3}	{3, 4}
B4	{5}	{2, 4}	{4}	{5}
B5	{6}	{1, 3}	{4, 5}	{4, 5, 6}

For B4 :-

$$in(B4) = out(B3) = \{4\}$$
$$out(B4) = gen(B4) \cup \{ in(B4) - kill(B4) \}$$
$$= \{5\} \cup \{ \{4\} - \{2, 4\} \}$$
$$= \{5\}$$

Dr. Somesh Kumar
Prof. & Head, CSE
Moradabad Institute of Technology
Moradabad-244001

Input and output values in Iteration 1 & 2 are different so we need to repeat this process until same settlement is found.

ITERATION 3 :-

Block	Generator	Kill	In	Out
B1	{1,2}	{3,4,5,6}	ϕ	{1,2}
B2	{3}	{1,6}	{1,2,4,5,6}	{2,3,4,5}
B3	{4}	{2,5}	{2,3}	{3,4}
B4	{5}	{2,4}	{3,4}	{3,5}
B5	{6}	{1,3}	{3,4,5}	{4,5,6}

For B1 :- $in|B1| = \phi$

$out|B1| = gen|B1| \cup \{in|B1| - kill|B1|\}$

$= \{1,2\} \cup \{\phi - \{3,4,5,6\}$

$= \{1,2\}$

For B2 :- $in|B2| = out|B1| \cup out|B5|$

$= \{\{,2\} \cup \{4,5,6\}$

$= \{1,2,4,5,6\}$

$out|B2| = gen|B2| \cup \{in|B2| - kill|B2|\}$

$= \{3\} \cup \{\{1,2,4,5,6\} - \{1,6\}\}$

$= \{3\} \cup \{2,4,5\}$

$= \{2,3,4,5\}$

For B3 :-

$in|B3| = out|B2| = \{2,3\}$

$out|B3| = gen|B3| \cup \{in|B3| - kill|B3|\}$

$= \{4\} \cup \{\{2,3\} - \{2,5\}\}$

$= \{3,4\}$

For B4 :-

$in|B4| = out|B3|$

$= \{3,4\}$

$out|B4| = gen|B4| \cup \{in|B4| - kill|B4|\}$

$= \{5\} \cup \{\{3,4\} - \{2,4\}\}$

$= \{3,5\}$

For B5 :-

$in|B5| = out|B3| \cup out|B4|$

$= \{3,4,5\}$

$out|B5| = \{6\} \cup \{\{3,4,5\} - \{1,3\}\}$

$= \{4,5,6\}$

ITERATION 4 :-

Block	Generator	Kill	In	Out
B1	{1,2}	{3,4,5,6}	ϕ	{1,2}
B2	{3}	{1,6}	{1,2,4,5,6}	{2,3,4,5}
B3	{4}	{2,5}	{2,3}	{3,4}
B4	{5}	{2,4}	{3,4}	{3,5}
B5	{6}	{1,3}	{3,4,5}	{4,5,6}

Since in & out values in iteration 3 & 4 are same, we will stop here.