DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE FILE

COMPILER DESIGN

(ESTD - 1999)

Vidya Jyothi Institute of Technology

(An Autonomous Institution)

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to

J.N.T. University, Hyderabad)

(Aziz Nagar, C.B.Post, Hyderabad -500075)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

REGULATION: R15

BATCH: 2017-2021

ACADEMIC YEAR: 2019-20

PROGRAM: B.TECH (COMPUTER SCIENCE AND ENGINEERING)

YEAR/SEM: III/II

COURSE NAME: COMPILER DESIGN

COURSE CODE: A26524

PRE REQUISITE: FORMAL LANGUAGES AND AUTOMATA THEORY

COURSE COORDINATOR: Dr.D.Aruna Kumari

COURSE INSTRUCTORS:

1. Dr.K.Ramesh Babu
2. M.Vijaya
3. Zaheer

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

S.NO.	DESCRIPTION
1	Syllabus
2	Text Book & Other References
3	Time Table
4	Program Outcomes (PO's) ,Program Specific Outcomes (PSO's) &PEO's
5	Mapping Of Course Outcomes (CO's) With Program Outcomes (PO's) & Program Specific Outcomes (PSO's)
6	Academic Calendar
7	Course Schedule
8	Lesson Plan
9	Assignment Questions
10	Mid I & Mid II Question Papers
11	Unit Wise Questions
12	Minutes of Course Review Meeting
13	Lecture Notes
14	Power Point Presentation
15	Semester End Question Papers
16	Extra Topics Delivered (if any)
17	Innovations In Teaching and Learning
18	Assessment Sheet – Co Wise (Direct Attainment)
19	Course End Survey Form

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Syllabus

COMPILER DESIGN

COMPILER DESIGN

B. Tech. III Year II Semester

L	T	P	C
3	1	0	3

Course Outcomes:

At the end of the course student would be able to

1. Formulate tokens for various programming languages.
2. Apply principles of parsing techniques to do syntax analysis.
3. Formulate semantic rules to do semantic analysis.
4. Apply optimization techniques on the intermediate code.
5. Generate the target code.

UNIT – I

Overview of Compilation: Phases of Compilation – Lexical Analysis, Regular Grammar and regular expression for common programming language features, pass and Phases of translation, interpretation, bootstrapping, data structures in compilation – LEX lexical analyzer generator.

Top down Parsing: Context free grammars, Top down parsing – Backtracking, LL (1), recursive descent parsing, Predictive parsing, Preprocessing steps required for predictive parsing.

UNIT – II

Bottom up parsing: Shift Reduce parsing, LR and LALR parsing, Error recovery in parsing, handling ambiguous grammar, YACC – automatic parser generator.

UNIT – III

Semantic analysis: Intermediate forms of source Programs – abstract syntax tree, polish notation and three address codes. Attributed grammars, Syntax directed translation, Type checker.

Symbol Tables: Symbol table format, organization for block structures languages, hashing, and tree structures representation of scope information. Block structures and non block structure storage allocation: static, Runtime stack and heap storage allocation, storage allocation for arrays, strings and records.

UNIT – IV

Code optimization: Consideration for Optimization, Scope of Optimization, local optimization, loop optimization, frequency reduction, folding, DAG representation.

Data flow analysis: Flow graph, data flow equation, global optimization, redundant sub expression elimination, Induction variable elements, Live variable analysis, Copy propagation.

UNIT – V

Object code generation: Object code forms, machine dependent code optimization, register allocation and assignment generic code generation algorithms, DAG for register allocation.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Text Books & Reference Books

COMPILER DESIGN

Text Book:

1. Alfred V Aho, Jeffrey D Ullman, Principles of Compiler Design, Pearson Education, 2001.

Reference Books:

1. J P Trembly and P G Sorenson, The Theory and practice of Compiler Writing, McGraw Hill, 2005.
2. Alfred V Aho, Ravi sethi, Jeffrey D Ullman, Compilers-Principles, Techniques and Tools, Pearson Education, second edition.
3. Dick Grone, Henri E Bal, Ceriel J H Jacobs, Modern Compiler Design, Wiley Dreamtech, 2006.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Time Table

COMPILER DESIGN

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

Department of Computer Science & Engineering

Sec: **CSE-B**

Year/Sem: **III - II** W.E.F: **09/12/2019** ROOM NO: **N-203**

DAY	9.00 – 9.55	9.55 – 10.50	10.50 – 11.45	11.45 – 12.30	12.30 – 1.25	1.25 – 2.20	2.20 – 3.15	3.15-4.10
MON	CD	MEFA	PDBS	LUNCH	WT/CT LAB			Value Added
TUE	DWDM	OE			OOAD	CD	WT(T)	NPTEL
WED	OOAD	OE			CD/DM LAB			HACKER RANK
THU	MEFA	OOAD	CD		DWDM	WT	PDBS	CISCO
FRI	DWDM	WT	MEFA		CD	OOAD(T)	DWDM(T)	CISCO
SAT	WT	DWDM	OOAD		DWDM	MEFA(T)	CD(T)	Value Added

Subject		Name of the Faculty
WT	Web Technologies	Mr.M.Tarakeshwar Rao
CD	Compiler Design	Dr. D.Aruna Kumari
OOAD	Object Oriented Analysis and Design	Ms.Sailaja
DWDM	Data Warehousing & Data Mining	Mr. Abdul Majeed
MEFA	Managerial Economics & Financial Analysis	Ms. G.Rajitha
PD&BS	Personality Development & Behavioral Skills	Ms.Padma Venkat
WT&CT LAB	Web Technologies & Case Tools Lab	Mr.M.Tarakeshwar Rao / Ms. Sailaja/G.Surekha
CD&DM LAB	Compiler Design & Data Mining Lab	Mr.B.Thikkana / Mr. Abdul Majeed/ Mr.Y.Praveen Kumar
OE	Database Management Systems	Mr.B.Srinivasulu

Class Incharge Mr.M.Tarakeswar Rao

III YEAR Coordinator Ms. M.Vijaya

Time Table I/C

H.O.D

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

Program Outcomes (PO's), Program Specific Outcomes (PSO's) & Program Educational Objectives (PEO's)

Programme Outcomes (PO's)

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 analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for 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 the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools, including prediction and 10odeling 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.

Program Specific Ooutcomes (PSO):

PSO 1: The ability to design and develop Algorithms to provide optimized solutions for societal needs
PSO 2: Apply standard approaches and practices in Software Project Development through trending technologies

Program Educational Objectives (PEO's)

PEO1: Enhance the employability of the graduate in software industries/Public sector/Research organizations

PEO2: Acquire analytical and computational abilities to pursue higher studies for professional growth

PEO3: Work in multidisciplinary project teams with effective communication skills and leadership qualities

PEO4: Develop professional ethics among the students and promote entrepreneural abilities

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Mapping of Course Outcomes (CO's) With Program Outcomes (PO's) & Program Specific Outcomes (PSO's)

COMPILER DESIGN

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY
Department of Computer Science & Engineering

Year &Sem: III year II Sem

Course name: Compiler Design

Course Code:A16523

Regulation: R15

COURSE OUTCOMES:

After completing this course the student must demonstrate the knowledge and ability to	
CO1	Differentiate the phases in compilation & parsing.
CO2	Identify the process in parsing and semantic analysis.
CO3	Explain about symbol tables and code optimization methods.
CO4	Explain about code optimization methods.
CO5	Analyze data flow and generate object code.

]

CO –PO MAPPING:

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12
CO 1	3	3	3	3	1	3	2	2	1	1	-	3
CO 2	3	3	3	3	3	3	2	2	1	1	-	3
CO 3	3	3	3	3	2	3	-	2	2	2	2	3
CO 4	3	3	3	3	2	3	-	2	2	2	2	3
CO 5	3	3	3	3	2	3	2	2	2	2	2	3
Avg	**3**	**3**	**3**	**3**	**2**	**3**	**2**	**2**	**1.6**	**1.6**	**2**	**3**

CO - PSO MAPPING:

	PSO1	PSO2
CO1	3	3
CO2	3	3
CO3	2	2
CO4	3	2
CO5	3	2
Avg	**2.8**	**2.4**

Signature of the Course Coordinator.

Sign. of HOD

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Academic Calendar

COMPILER DESIGN

Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)
(Aziz Nagar, C.B.Post, Hyderabad -500075)

II B.Tech I & II Semester Academic Calendar for the Academic Year 2019-20

II YEAR I SEMESTER		**Commencement of Class Work 17.06.2019**	
	From	**To**	**Duration**
I Spell of Instruction	17.06.2019	10.08.2019	8 WEEKS
I Mid Examinations	13.08.2019	17.08.2019	4 DAYS
II Spell of Instruction	19.08.2019	05.10.2019	7 WEEKS
Dussehra Holidays	07.10.2019	12.10.2019	1 WEEK
II Spell of Instruction Continuation	14.10.2019	19.10.2019	1 WEEK
II Mid Examinations	21.10.2019	24.10.2019	4 DAYS
Practical Examinations	25.10.2019	29.10.2019	4 DAYS
Betterment Examinations	30.10.2019	01.11.2019	3 DAYS
End Semester Examinations	02.11.2019	18.11.2019	2 WEEKS
Supplementary Examinations	19.11.2019	04.12.2019	2 WEEKS
Mid-II Examinations (For Lateral Entry)	18.11.2019	21.11.2019	4 DAYS
Practical Examinations (For Lateral Entry)	22.11.2019	26.11.2019	4 DAYS
Betterment Examinations (For Lateral Entry)	27.11.2019	30.11.2019	4 DAYS
II YEAR II SEMESTER		**Commencement of Class Work 02.12.2019**	
I Spell of Instruction	02.12.2019	10.01.2020	6 WEEKS
Pongal Holidays	11.01.2020	15.01.2020	5 DAYS
Technical/Sports fest	16.01.2020	18.01.2020	3 DAYS
I Spell of Instruction Continuation	20.01.2020	01.02.2020	2 WEEKS
I Mid Examinations	03.02.2020	08.02.2020	1 WEEK
II Spell of Instruction	10.02.2020	04.04.2020	8 WEEKS
II Mid Examinations	06.04.2020	09.04.2020	4 DAYS
Practical Examinations	13.04.2020	17.04.2020	4 DAYS
Betterment Examinations	18.04.2020	22.04.2020	4 DAYS
End Semester Examinations	23.04.2020	08.05.2020	2 WEEKS
Supplementary Examinations	11.05.2020	23.05.2020	2 WEEKS
Commencement of classes will be from	**15.06.2020**		

DIRECTOR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Course Schedule

COMPILER DESIGN

Course Schedule

Distribution of Contact Hours

Unit	Topic	Book1	Total No. of Hours
I	**Introduction to Compilers:**	Ch1-1.1-1.6 Ch2-2.1-2.9 Ch3-3.1-3.8	10
II	**Syntactic Specification: Basic Parsing Techniques:**	Ch4-4.1,4.4	13
III	**Construction of efficient Parsers**	Ch4-4.5,4.9	11
IV	**Syntax Directed Translation Symbol table**	Ch5-5.1-5.6 Ch7-7.1-7.6	10
V	**Code Optimization: Code Generation**	Ch9-9.1-9.9 Ch10-10.1-10.2	10
Total contact classes for syllabus coverage			54

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Lesson Plan

COMPILER DESIGN

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY (AUTONOMOUS)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SUBJECT: COMPILER DESIGN **ACADEMIC YEAR: 2019-20**

NAME:Dr.D.Aruna Kumari **YEAR/SEM/SECTION: III B.TECH/I SEM/B**

Lecture No	Topic	Teaching Learning Process
	Unit-I	
1	Introduction to compilers	Chalk & Board
2	Structure of compiler-Phases of Compiler	Chalk & Board
3	Symbol table management, Grouping of phases in to passes	Chalk & Board
4	Compiler vs Interpreter	Power Point Presentation
5	Lexical- Analysis: role and need of LA	Power Point Presentation
6	Input- buffering, Regular expressions for identifiers, Signed numbers etc.	Power Point Presentation
7	A Language for specifying lexical Analyzer, lexical phase errors	Power Point Presentation
8	Top down parsing – Backtracking	Chalk & Board, Interactive learning
9	LL(1).	Chalk & Board
10	Recursive descent parsing.	Chalk & Board
11	Predictive Parsing	Chalk & Board
12	Preprocessing steps required for Predictive Parsing	Chalk & Board
	Unit-II	
1	Bottom up parsing	Chalk & Board
2	Shiftreduce parsing	Power Point Presentation

3	LR	Chalk & Board, Role play
4	LALR parsing	Power Point Presentation
5	Error Recovery in Parsing	Chalk & Board
6	Handling Ambiguous Grammer	Chalk & Board
7	YACC.	Power Point Presentation
8	YACC – automatic parser generator.	Chalk & Board
Unit-III		
1	Semantic analysis	Power Point Presentation
2	Intermediate forms os source programs	Power Point Presentation
3	Abstraact Systax tree ,Postfix notationand Three Address Code	Power Point Presentation
4	Attributed grammars, Syntax directed translation, Type checker.	Power Point Presentation
5	Symbol tables:Symbol table format	Power Point Presentation
6	Organization for block structure languages,	Power Point Presentation, Quiz
7	Hashing and tree structure representation of scope information	Chalk & Board
8	Block structure and non block structure storage allocation: static	Chalk & Board
9	Runtime stack and heap storage allocation	Power Point Presentation
10	storage allocation for arrays, strings and records.	Chalk & Board
Unit-IV		
1	Code optimization:	Demonstrated by PPT

2	Consideration for optization	Demonstrated by PPT
3	Scope of Optimization	Demonstrated by PPT
4	Local Optimization	Demonstrated by PPT
5	Frequency reduction folding DAG representation	Demonstrated by PPT
6	Data flow analysis	Demonstrated by PPT
7	Flow graph,data flow equeation	Demonstrated by PPT
8	Global optimizatiopn	Demonstrated by PPT
9	Redundent sub expression elimination	Demonstrated by PPT
10	Induction variable elements, live variable analysis, Copy propagation	Demonstrated by PPT
Unit-V		
1	Object code generation	Power Point Presentation
2	Object code forms	Chalk & Board
3	Machine dependent code optimization	Chalk & Board
4	Machine in dependent code optimization	Power Point Presentation
5	Register Allocation and Assignment	Chalk & Board
6	Generic code generation	Power Point Presentation
7	Generic code generation algorithms	Power Point Presentation, PBL
8	DAG for register allocation.	Power Point Presentation

Total No of classes: 54

Course Coordinator

CSE-HOD

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Assignment Questions

COMPILER DESIGN

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

(Accredited by NAAC & Approved by A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)

(Aziz Nagar, C.B.Post, Hyderabad- 500075)
(AUTONOMOUS)

COMPUTER SCIENCE & ENGINEERING

ASSIGNMENT 1

Branch: CSE

Year&Sem: III-I

SUB: Compiler Design

Academic Year: 2019-20

Faculty Name: Dr.D.Aruna Kumari

Marks: 25M

S.No	Question	Marks	CO	BL	PO's
1	Explain what are the different phases of Compiler?	5	1	L2	1-10,12
2	What is pass & phases? Explain the differences between them.	5	1	L1,L2	1-10,12
3	What is Predictive parsing? Write the steps of predictive parser.	5	2	L1	1-10,12
4	What is Recursive descent parsing? Identify the limitations of recursive descent parser?	5	2	L1	1-10,12
5	What is Shift Reduce parsing? Explain with an Example.	5	3	L1,L2	1-6,8-12

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

(Accredited by NAAC & Approved by A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)

(Aziz Nagar, C.B.Post, Hyderabad- 500075)

(AUTONOMOUS)

COMPUTER SCIENCE & ENGINEERING

ASSIGNMENT 2

Branch: CSE **Year&Sem:** III-I

SUB: Compiler Design **Academic Year:** 2019-20

Faculty Name: Dr.D.Aruna Kumari **Marks:** 25M

S.No	Question	Marks	CO	BL	PO's
1	What is ordered and unordered symbol table? What is the function of symbol table in the compilation process? Explain.	5	3	L1	1-6,8-12
2	Discuss about stack allocation, heap allocation. Explain. Discuss various symbol table organization techniques.	5	3	L2	1-6,8-12
3	Explain different principle sources of optimization techniques with suitable examples.	5	4	L2	1-6,8-12
4	Explain various loop optimization techniques? Explain dominators with n Examples.	5	4	L2	1-6,8-12
5	Describe reducible & non reducible flow graphs with a example.	5	5	L2	1-12

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

QUESTION PAPERS
MID-I &II

COMPILER DESIGN

Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)
(Aziz Nagar, C.B.Post, Hyderabad -500075)

III Year B.Tech II Semester I Mid Examination

Branch: CSE
Sub: Compiler Design
Date: 18.02.2020

Duration: 90Min
Marks: 20
Session: FN

Bloom's Level:

Remember	L1
Understand	L2
Apply	L3
Analyze	L4
Evaluate	L5
Create	L6

	PART-A (3Q×2M =6 Marks)	Outcomes		BL	Marks
	ANSWER ALL THE QUESTIONS	CO	PO		
1.i)	List out the difference between Compiler and interpreter.	1	1-10,12	IV	2
	[OR]				
ii)	Write the new grammar after eliminating left recursion for E→E+E|E*E|id	1	1-10,12	II	2
2.i)	What is YACC? Explain in detail.	2	1-10,12	II	2
	[OR]				
ii)	What is an augmented grammar? Describe with an example.	2	1-10,12	II	2
3.i)	What are the benefits of intermediate code generation?	3	1-6,8-12	I	2
	[OR]				
ii)	Define Polish Notation.	3	1-6,8-12	I	2
	PART-B (5+5+4= 14 Marks)	Outcomes		BL	Marks

	ANSWER ALL THE QUESTIONS	CO	PO		
4.i)	Explain the different phases of a compiler, showing the output of each phase using the following statement.Position:=Initial+rate*60.	1	1-10,12	II	5
	[OR]				
ii)	Construct the predictive parser table for the following grammar and parse the input string id+id*id. E →TE' E'→+TE' | ϵ T→FT' T'→*FT' | ϵ F→(E) | id	1	1-10,12	IV	5
5.i)	Construct the CLR parsing table for the following grammar and parse. S → CC C→ aC C → d	2	1-10,12	IV	5
	[OR]				
ii.a)	With neat sketch explain the structure of LR parser and the rules to compute LR item.	2	1-10,12	II	3
b)	Consider the following grammar: E → E+E E → E*E E → (E) E → id Show the shift-reduce parser action for the string: id+id*id	2	1-10,12	III	2
6.i)	Explain Quadruples, triples and indirect triples and translate A: = -B*(C/D)	3	1-6,8-12	II	4
	[OR]				
ii)	Explain Intermediate code forms in detail.	3	1-6,8-12	II	4

Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)
(Aziz Nagar, C.B.Post, Hyderabad -500075)

III B.Tech II Semester II Mid Examination, October/November-2020

Subject : Compiler Design
Time: 1 Hour
Bloom's Level:

BRANCH: CSE
Max Marks:20

Remember	L1
Understand	L2
Apply	L3
Analyze	L4
Evaluate	L5
Create	L6

PART-A ANSWER ALL THE QUESTIONS		Course Outcomes CO	 PO	Bloom's Level	Marks
1.a)	What is symbol table? List the various data structures that can be used to organize a symbol table.	CO3	1-10,12	L1	6
	[OR]				
b)	Explain Heap storage allocation with an example.	CO5	1-10,12	L2	6
PART-B **ANSWER ANY TWO QUESTIONS**		**Course Outcomes** **CO**	 **PO**	**Bloom's Level**	**Marks**
2)	What are the different loop optimization methods? Explain them with examples.	CO3	1-10,12	L2	7
3)	What are the different local optimization methods? Explain them with examples.	CO4	1-10,12	L2	7
4)	Explain in detail about DAG for Register Allocation.	CO4	1-10,12	L1	7

5)	Generate code for the following expression: x = (a+b)-((c- d)-e).	CO5	1-10,12	L4	7

*****VJIT(A)*****

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Unit Wise Questions

COMPILER DESIGN

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY
(Autonomous)

Aziz Nagar Gate, Hyderabad-75

III YEAR B.TECH CSE- II SEM

COMPILER DESIGN

UNIT : I

Long Answers Type Questions: (5 questions):

1. Analyze the different phases of a compiler, showing the output of each phase using the following statement. Position=initial +rate*60.
2. Explain about Language Processing System
3. Explain about role and need of lexical analyzer
4. Explain lexical errors with an example
5. Write the regular expression for the following
   a) For Identifiers b) For unsigned numbers

Short Answer Questions:

1. What is Compiler?
2. Define the role of Loader
3. List out the differences between Compiler and interpreter.
4. Identify the tokens from the statement E=M+C*60
5. Explain symbol table management.
6. What is error handler?
7. Analyze the need of Cross compile.
8. What is Pass ?
9. Explain about passes.
10. What is the role of preprocessor?
11. Give the complete specification of LEX tool and describe various section of it.
12. Write a LEX program to perform lexical analysis.
13. Explain role of lexical analyzer
14. What are Front end Phases and Back End Phases
15. Define Lexeme.
16. Define pattern.
17. What is Regular Expression?
18. Explain regular expressions for identifiers.
19. Write the Regular Expression for numbers?
20. Define the role of Linker

UNIT : II

Long Answers Type Questions: (5 questions):

1. What is context free grammar? With example.
2. Explain about
   a. Parse tree b. semantic error c. syntactic error
3. Write are the preprocessing steps required for predictive parse table construction.
4. Construct the predictive parser for the following grammar.
   S → (L) / a
   L→ L,S/S
5. Construct the predictive parser table for the following grammar and parse the input string id+id*id

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

Short Answer Questions:

1. What is Context Free Grammar?
2. What is parse tree?
3. Explain Capabilities of Context Free Grammar.
4. What are syntactic phase errors?
5. What is top down parser?
6. What is Recursive descent parser?
7. What is Non Recursive descent parser?
8. What is Back-tracking?
9. What are semantic errors?
10. What do you mean by LL(1) grammar? Give example.
11. What are the problems in top down parsing?
12. Write the algorithm for computing FIRST function.
13. Write the algorithm for computing FOLLOW function.
14. Compute the FIRST and FOLLOW functions for the following grammar.
    E →TE' E'→+TE' | ϵ T→FT' T'→*FT' | ϵ F→(E) | id
15. Draw and explain the block diagram for predictive parser.
16. Consider the grammar s → (L) | a , L → L,S | S
    a. What are the terminals, non – terminals and start symbol?
    b. Find the parse tree for (a, a).
17. Consider the grammar s → (L) | a , L → L,S | S
    a. Construct left most derivation for (a, (a, a)).
    b. Construct right most derivation for (a, (a, a)).
18. Derive the string "aaabbabbba" using the following grammer.
    S→aB|bA, A→a|aS|bAA, B→b/bS/aBB
19. What is LL(1) grammar?
20. What are basic parsing techniques?

UNIT : III

Long Answers Type Questions: (5 questions):

1. With neat sketch explain the structure of LR parser and the rules to compute LR item.
2. Consider the following grammar:

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

   Show the shift-reduce parser action for the string: **id*(id+id).**
3. Construct the SLR parsing table for the following grammar and parse the input string "a*b+a"

   $E \rightarrow E+T|T$ $T \rightarrow TF|F$ $F \rightarrow F*|a|b$

4. a) Write the steps for the efficient construction of LALR parsing table.
   b) Construct the LALR parsing table for the following grammar.

   $S \rightarrow L=R|R$ $L \rightarrow *R|id$ $R \rightarrow L$

5. List out the differences between SLR, CLR and LALR.

Short Answer Questions:

1. What is bottom Up Parser?
2. What Shift Reduce Parser?
3. Explain about LR Parsers?
4. Write the steps of canonical collection of LR(0) items.
5. Differentiate between Top-down and Bottom-up parsers.
6. List out the differences between SLR, CLR and LALR.
7. What is Handle pruning?
8. Explain the various actions performed by shift-reduce parsers with an example.
9. Write down the advantages and disadvantages of LR parsers.
10. Draw and explain the block diagram of LR parser.
11. Write an algorithm to find LR(0) items.
12. List out the steps for construction of SLR parsing table.
13. List out the steps for construction of CLR parsing table.
14. List out the steps for construction of LALR parsing table.
15. Perform the shift reduce parsing of the input string "id-id*id" using the following grammar.
    $E \rightarrow E-E|E*E|id$
16. Perform the shift reduce parsing of the input string "**intid,id;**" using the following grammar.
    $S \rightarrow TL;$ $T \rightarrow int|float$ $L \rightarrow L,id|id$
17. Differentiate between LL and LR parsers.
18. Construct LR(1) parsing table for the following grammer.
    $S \rightarrow CC$ $C \rightarrow aC|d$
19. What are the various conflicts that occur during shift reduce parsers?
20. Consider the grammar
    S -> S + S
    S -> S * S
    S -> id
    Perform Shift Reduce parsing for input string "id + id + id".

UNIT : IV

Long Answers Type Questions: (5 questions):

Long Answers Type Questions: (5 questions):

1. a) What is an Abstract syntax tree? How to construct it? Explain by writing syntax directed definition.
2. Explain about Quadruples, triples and indirect triples and translate (a+b)*(c+d)+(a*b/c)*b+60. And list advantages and disadvantages.
3. a) Can we reuse the symbol table space? Explain through an example.
   b) Write short notes on functions of semantic analysis.
4. What is symbol table? List the various data structures that can be used to organize a symbol table? Compare the performance.
5. What is runtime stack? Explain storage allocation strategies used for recursive procedure calls.

Short Answer Questions:

1. What Syntax Directed Translation?
2. What is semantic rule?
3. Write the semantic rules for DTL, T int|real, L L,id|id.
4. Write about order of evaluation of semantic rules in syntax directed translation.
5. Write about dependency graphs in syntax directed translations.
6. What are the benefits of intermediate code generation?
7. How to generate polish notation using translation schemes?
8. Write the quadruple, triple, indirect triple for the expression -(a*b) + (c+d)-(a+b+c+d).
9. Write an algorithm for constructing the dependency graph for a given parse tree.
10. Analyse the different types of three address statements.
11. Explain L-attributed syntax directed translation.
12. Explain S-attributed syntax directed translation.
13. Construct the syntax tree for the expression (a*b) + (c-d) * (a*b) + b.
14. Differentiate synthesis and inherited translation.
15. Explain any two storage allocation strategies.
16. What is an intermediate code? Explain different types of intermediate codes forms and represent the following statement in different forms: W = (A + B) – (C + D) + (A + B + C).
17. What is an activation record? Explain how it is related with run time storage organization?
18. Describe about type expressions.
19. What are the various operations performed on the symbol table?
20. Explain about Representing Scope information.

UNIT : V

Long Answers Type Questions: (5 questions):

1. Explain different principal sources of optimization technique with suitable examples?
2. What are the different loop optimization methods? Explain them with examples.
3. What are the issues to be considered during code generation?
4. Explain about Peephole optimization techniques.
5. Explain in detail about Register Allocation and Assignment?

Short Answer Questions:

1. Mention the issues to be considered while applying the techniques for code optimization.
2. List the properties of optimizing compilers.
3. Explain how code motion and frequency reduction used for loop optimizations?
4. Write about the techniques in local and global transformations.

5. What is copy propagation and dead code elimination?
6. Explain Redundant Sub Expression Elimination?
7. Discuss the role of strength reduction during code optimization of a compiler.
8. Give the criteria for achieving machine independent code optimization.
9. Compare local optimization with global optimization.
10. Explain the following a) Copy propagation b) constant folding.
11. Explain about various levels and types of optimizations.
12. Give the criteria for achieving machine dependent code optimization.
13. What is code motion?
14. What is induction variable?
15. What is global optimization?
16. Explain a Simple Code generator?
17. What is Register Allocation and Assignment?
18. Short note on Peephole optimization.
19. Explain about Object Programs.
20. What are the Problems in Code Generation?

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Minutes of Course Review Meeting

COMPILER DESIGN

Meeting 1

Date: 21/1/2020

Details of Meeting No – 1	
Date of Meeting	**17/5/2021**
Member's Present	**1. Dr.Aruna kumari** **2. Dr.Ramesh babu** **3. M.Vijaya** **4.Zaheer**
Details	Points discussed in the meeting: • Preparation of Unit wise questions and give assignment to students • Teaching Learning Practices • Status of Syllabus coverage of Mid I.
Signatures	1. 2. 3. 4.

Course Coordinator

CSE-HOD

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

Meeting 2

Date: 08.10.2020

Details of Meeting No – 2 (online meeting)	
Date of Meeting	24/7/2021
Member's Present	**1. Dr.Aruna kumari** **2. Dr.Ramesh babu** **3. M.Vijaya** **4.Zaheer**
Details	Discussion on Teaching Learning resource Preparation for Mid 2
Signatures	1. 2. 3. 4.

Course Coordinator

CSE-HOD

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

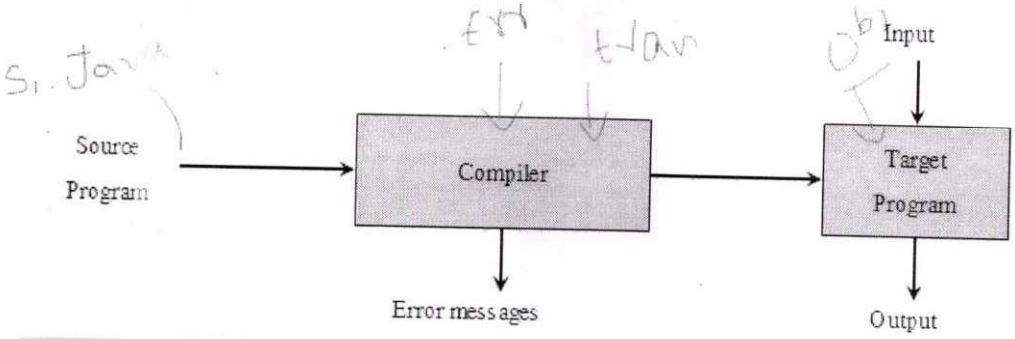
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Power Point Presentation

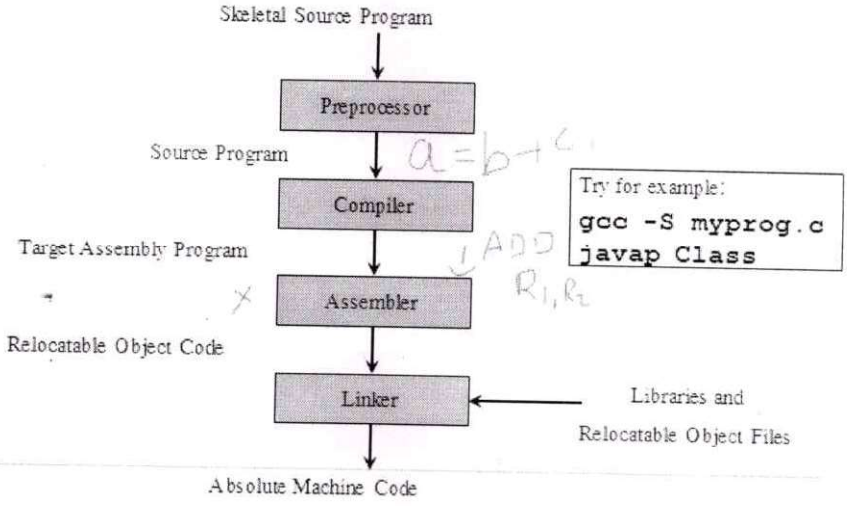
COMPILER DESIGN

Compilers

- *"Compilation"*
  - Translation of a program written in a source language into a semantically equivalent program written in a target language



Preprocessors, Compilers, Assemblers, and Linkers--.. Language Processing System



Phases

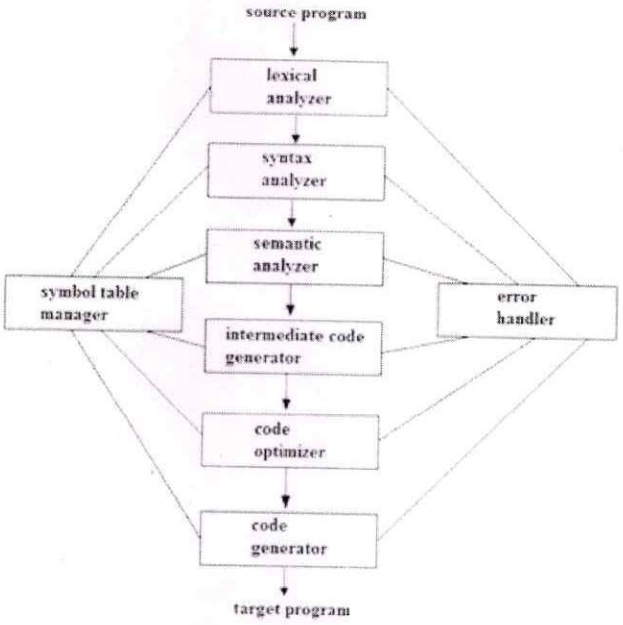


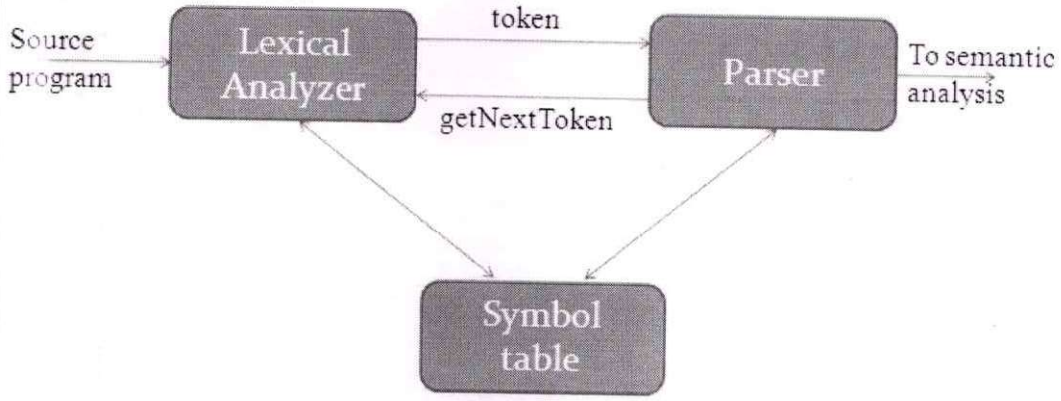
Fig 1.5 Phases of a compiler

Exercises

1. Identify the Differences Between Compiler and Interpreter
2. Identify the token from the following statements

   sum= c+icc/ 79
3. List the components of the Language Processing systems

The role of lexical analyzer



Example

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but " sorrounded by "	"core dumped"

printf("total = %d\n", score);

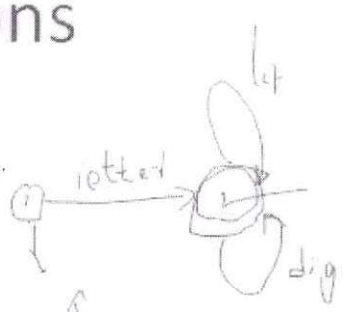
Regular definitions

d1 -> r1

d2 -> r2

...

dn -> rn



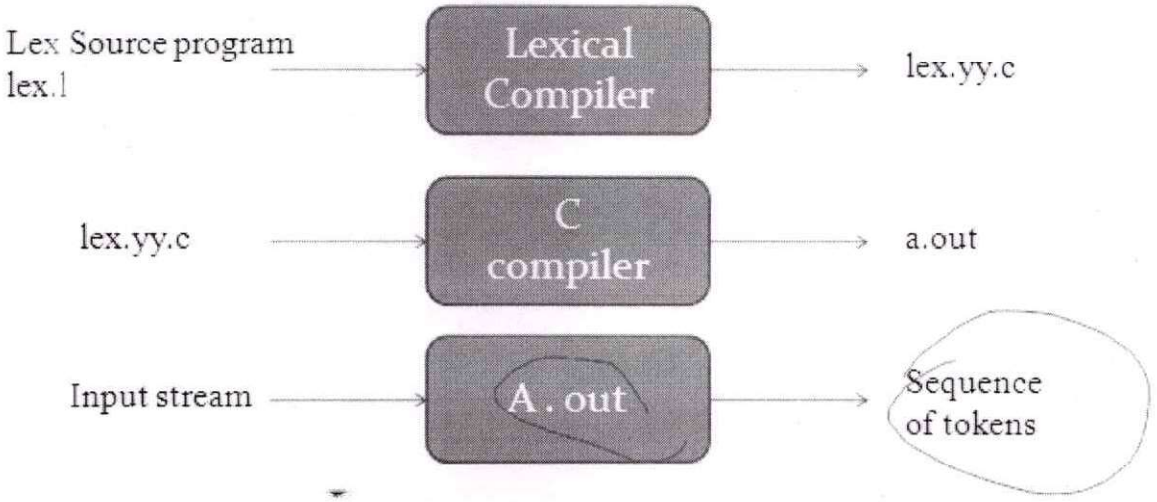
Example:

letter_ -> A | B | ... | Z | a | b | ... | Z | _

digit -> 0 | 1 | ... | 9

id -> letter_ (letter_ | digit)*

Lexical Analyzer Generator - Lex



Ambiguous grammar

- For some input string, if there exists more than one LMD or RMD , Grammar is called as Ambiguous

LMD 1

- E-> E+E -> r3
  - Id + E =->r2
  - Id + E*E => r3
  - Id + id * E => r3
- Id + id * id

LMD 2

$E \rightarrow E * E$

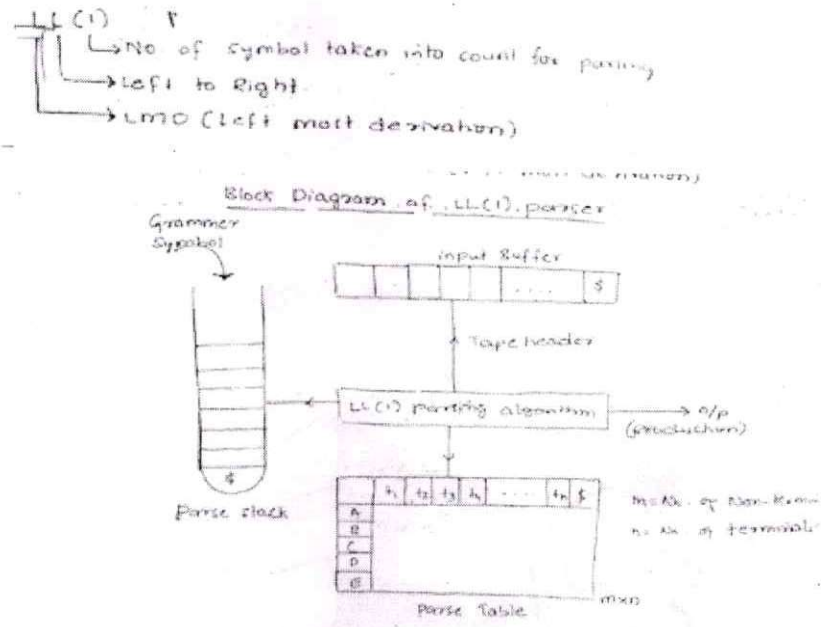
$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$

LL (1) Parser or Table Driven Parser



Example 4.8. Consider the following grammar for arithmetic expressions.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id} \qquad (4.10)$$

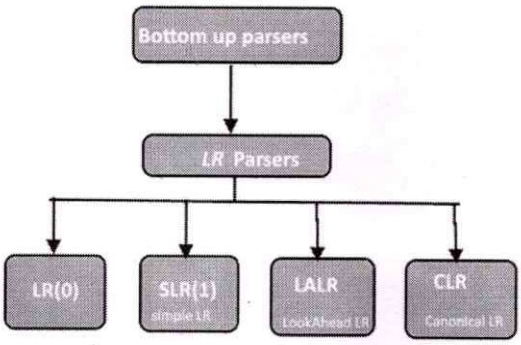
$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow (E) \mid \mathbf{id}$$

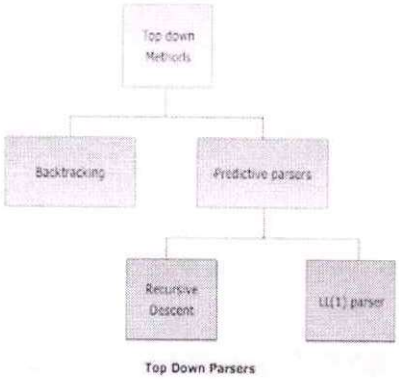
First(α) is a set of terminal symbols that begin in strings derived from α.

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}
FIRST(E') = {+, ε}
FIRST(T') = {*, ε}
FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(T) = FOLLOW(T') = {+,), $}
FOLLOW(F) = {+, *,), $}

Classification





LL Parser	LR Parser
First L of LL is for left to right and second L is for leftmost derivation.	L of LR is for left to right and R is for rightmost derivation.
It follows the left most derivation.	It follows reverse of right most derivation.
Using LL parser parser tree is constructed in top down manner.	Parser tree is constructed in bottom up manner.
In LL parser, non-terminals are expanded.	In LR parser, terminals are compressed.
Starts with the start symbol(S).	Ends with start symbol(S).
Ends when stack used becomes empty.	Starts with an empty stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parser tree.
Terminal is read after popping out of stack.	Terminal is read before pushing into the stack.
It may use backtracking or dynamic programming.	It uses dynamic programming.
LL is easier to write.	LR is difficult to write.
Example: LL(0), LL(1)	**Example:** LR(0), SLR(1), LALR(1), CLR(1)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Lecture Notes

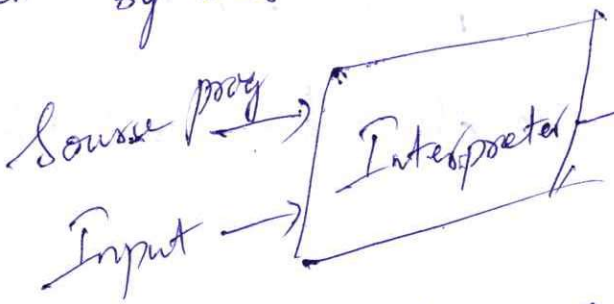
COMPILER DESIGN

Unit - 1

→ Compiler is a program that can read a program in one language - the source lang. - and translate it into an equivalent program in another language - the target language.

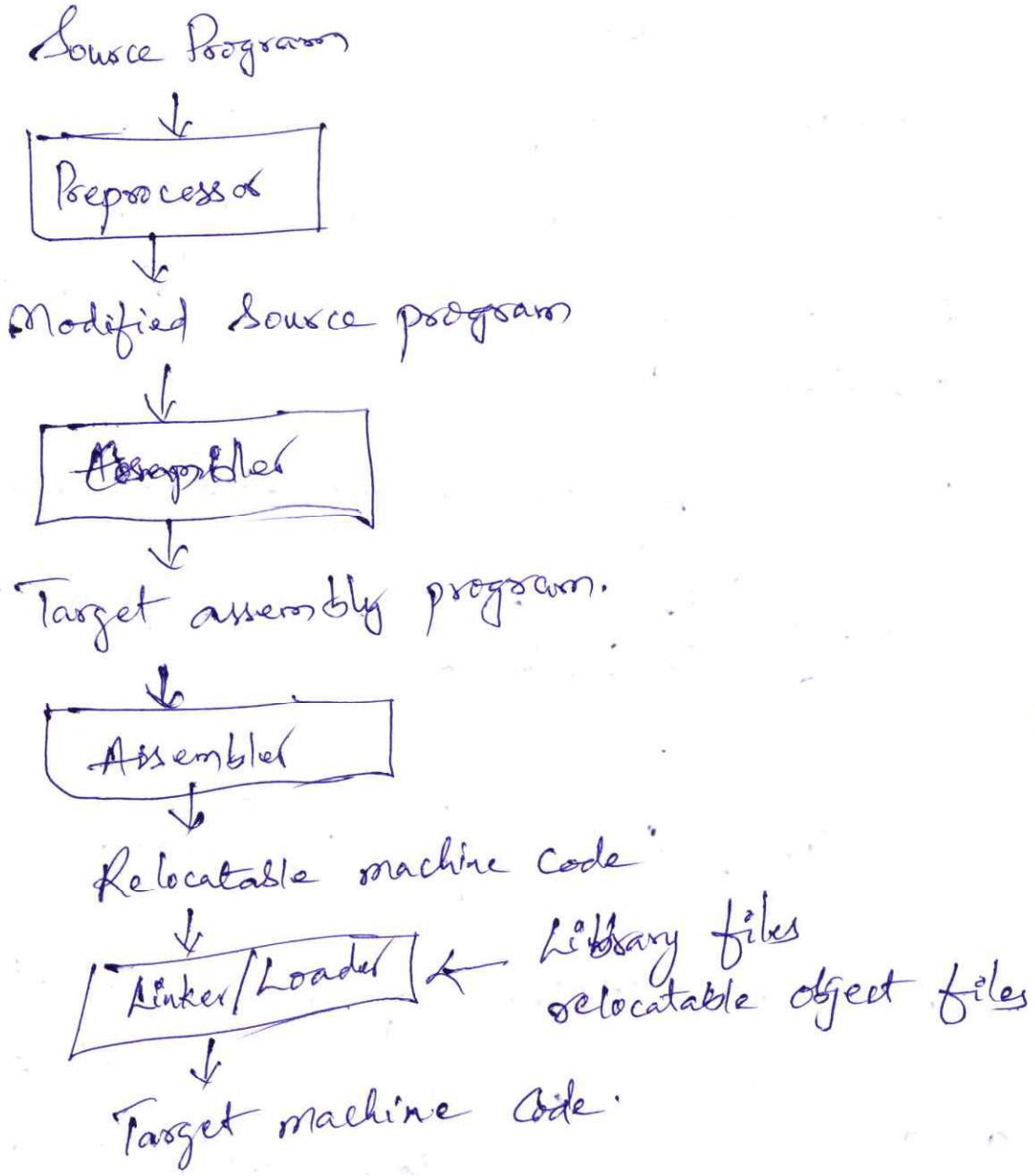
→ To report any errors if any detected during translation.

→ Interpreter: Instead of producing a target program as a translation; an interpreter directly executes the operations specified in source program on inputs supplied by user.



→ Compiler is faster than interpreter at mapping i/p's to o/p's.

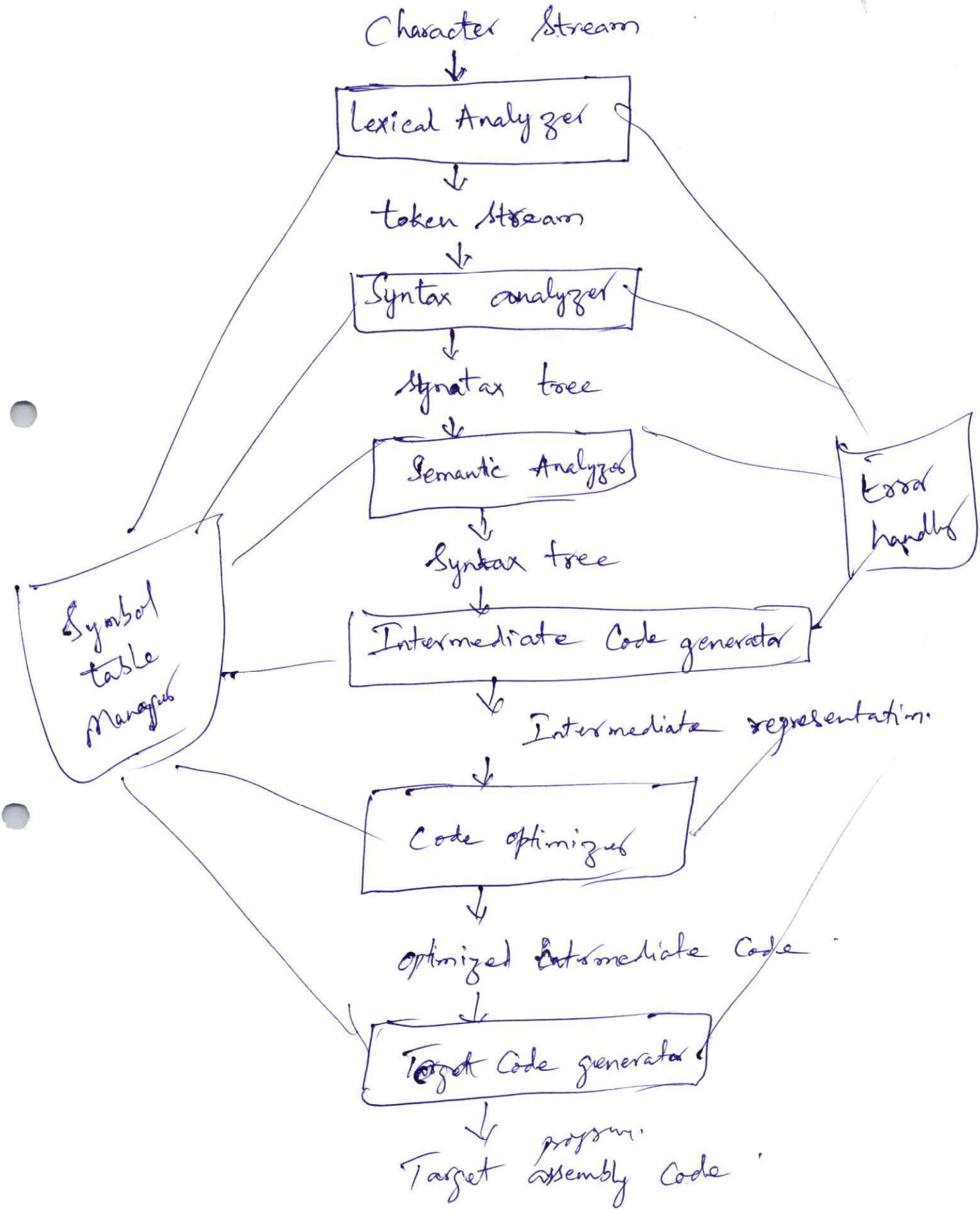
→ However interpreter gives better



→ Preprocessor directives { # include, # define, Macro expansion

→ Linker links relocatable object files and library files.

Phases of Compiler



61, 68, 75, 76, 79, 80, 83, 86, 88, 93, 94, 95, 97, 98

A4, A7, A9, B0, B3, B6, B7,

UNIT -1

1.1 OVERVIEW OF LANGUAGE PROCESSING SYSTEM

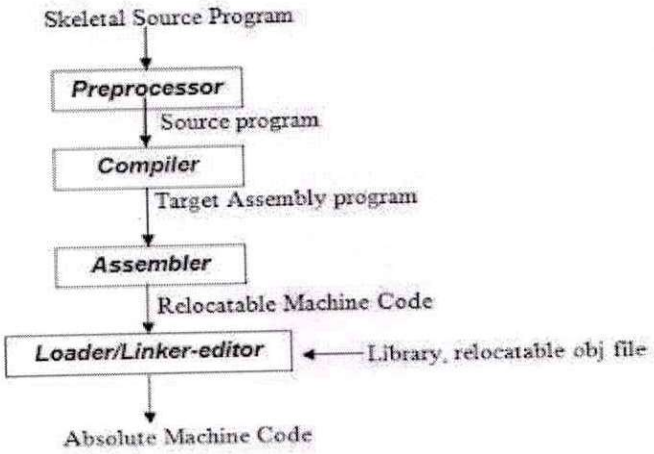


Fig 1.1 Language –processing System

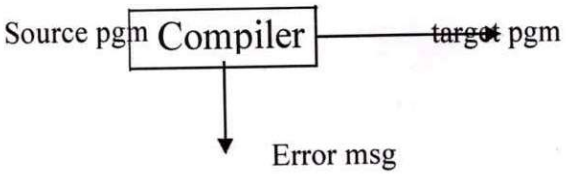
1.2 Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

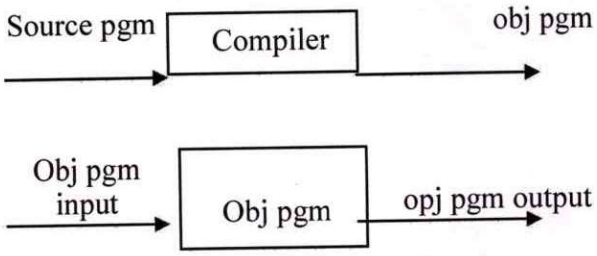
1. *Macro processing:* A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion:* A preprocessor may include header files into the program text.
3. *Rational preprocessor:* these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
4. *Language Extensions:* These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

1.3 COMPILER

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.

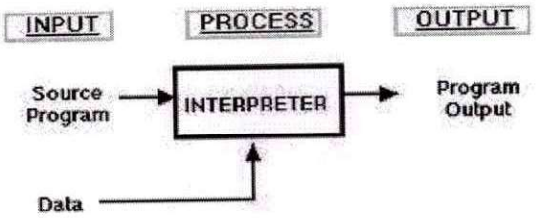


Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.



1.4 ASSEMBLER: programmers found it difficult to write or ead p og ams in machine language. They begin to use a mnemonic (symbols) for each machine inst uction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Pr grams kn wn as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source pr gram, the utput is a machine language translation (object program).

1.5 INTERPRETER: An interpreter is a program that appears to execute a source program as if it were machine language.



Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. he process of interpretation can be carried out in following phases.

1. Lexical analysis
2. Synatx analysis
3. Semantic analysis
4. Direct Execution

Advantages:

- Modification of user program can be easily made and implemented as execution proceeds.
- Type of object that denotes a various may change dynamically.
- Debugging a program and finding errors is simplified task for a program used for interpretation.
- The interpreter for the language makes it machine independent.

Disadvantages:

- The execution of the program is *slower*.
- *Memory* consumption is more.

performs another very important role,Worldtheerror-detection.AnyviolationofHLL specification would be detected and reported to the programmers. Important role of translator are:

2 ***Loader and Link-editor:***

Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program irectly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory whi e the user's program was being executed. Also the programmer would have to retrans ate his program with each execution, thus wasting translation time. To over come this p ob ems of wasted translation time and memory. System programmers developed another component ca ed loader

"A loader is a program that places programs into mem y and p epa es them for execution." It would be more efficient if subroutines c uld be t anslated into bject form the loader could"relocate" directly behind the user's pr gram. The task f adjusting programs o they may be placed in arbitrary core locations is called rel cati n. Rel cation loaders perform four functions.

1.6 TRANSLATOR

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator

1 ranslating the hll program input into an equivalent ml program.

2 Providing diagnostic messages wherever the programmer violates specification of the hll.

1.7 TYPE OF RANSLA ORS:-

- I TERPRETOR
- COMPILER
- PREPROSSESSOR

1.8 LIST OF COMPILERS

1. Ada compilers
2 .ALGOL compilers
3 .BASIC compilers
4 .C# compilers
5 .C compilers
6 .C++ compilers
7 .COBOL compilers
8 .D compilers
9 .Common Lisp compilers
10. ECMAScript interpreters
11. Eiffel compilers
12. Felix compilers
13. Fortran compilers
14. Haskell compilers
15 .Java compilers
16. Pascal compilers
17. PL/I compilers
18. Python compilers
19. Scheme compilers
20. Smalltalk compilers
21. CIL compilers

1.9 STRUCTURE OF THE COMPILER DESIGN

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called **'phases'**.

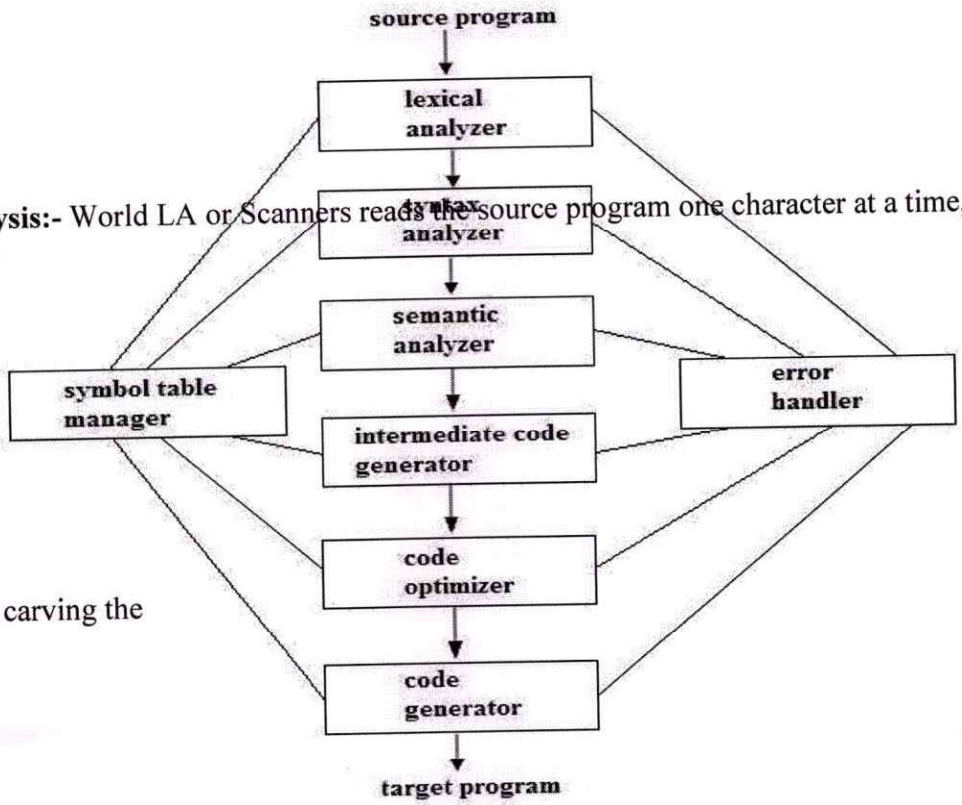


Fig 1.5 Phases of a compiler

Lexical Analysis:- World LA or Scanners reads the source program one character at a time, carving the source program into a sequence of automic units called **tokens.**

Syntax Analysis:-

The second stage of translation is called Syntax analysis parsing. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

Intermediate Code Generations:-

An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

Code Optimization :-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-

The last phase of translation is code generation. A number of optimizations to **reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

Table Management (or) Book-keeping:-

This is the portion to **keep the names** used by the program and records essential information about each. The data structure used to record this information called a 'Symbol Table'.

Error Handlers:-

It is invoked when a flaw error in the source program is detected.

The output of LA is a streamWorldoftokens,whichispassedtothenextphase,the

syntax analyzer or parser. The SA groups the tokens together into syntactic structure ca ed

as **expression.** Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

The parser has two functions. It checks if the tokens f om exical ana yzer, occur in pattern that are permitted by the specification for the sou ce anguage. It a so imposes on tokens a tree-like structure that is used by the sub-sequent phases of the compiler.

Example, if a program contains the expressi n **A+/B** after lexical analysis this expression might appear to the syntax analyzer as the t ken sequence **id+/id.** On seeing the /, the syntax analyzer should detect an error situati n, because the presence of these two adjacent binary operators violates the formulations rule f an expressi n.

Syntax analysis is to make explicit the hierarchica structure of the incoming token stream by **identifying which parts of the token stream should be grouped.**

Example, (A/B*C has two possible interpretations.)

1, divide A by B and then multiply by C or

2, multiply B by C and then use the result to divide A.

each of these two interpretations can be represented in terms of a parse tree.

Intermediate Code Generation:-

he intermediate code generation uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instruction with one operator and a small number of operands.

he output of the syntax analyzer is some representation of a parse tree. the intermediate code generation phase transforms this parse tree into an intermediate language representation of the source program.

Code Optimization

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space. Its output is another intermediate code program that does the some job as the original, but in a way that saves time and / or spaces.

1, Local Optimization:-

There are local transformations that can be applied to a program to make an improvement. For example,

If **A > B** goto **L2**

)

Goto **L3**

L2 :

This can be replaced by a single statement

If **A < B** goto **L3**

Another important local optimization is the elimination of common sub-expressions World

A := B + C + D
E := B + C + F

Might be evaluated as

T1 := B + C

A := T1 + D
E := T1 + F

Take this advantage of the common sub-exp essions **B + C.**

2, Loop Optimization:-

Another important source of ptimizati n c ncerns ab ut **increasing the speed of loops**. A typical p impr vement is to move a computation that produces the same result each time around the loop to a point, in the program just bef re the l p is entered.

Code generator :-

Cg produces the object code by deciding on the memory locations for data, selecting code to access each datum and selecting the registers in which each computation is to be done. Many computers have only a few high speed registers in which computations can be performed quickly. A good code generator would attempt to utilize registers as efficiently as possible.

Table Management OR Book -keeping :-

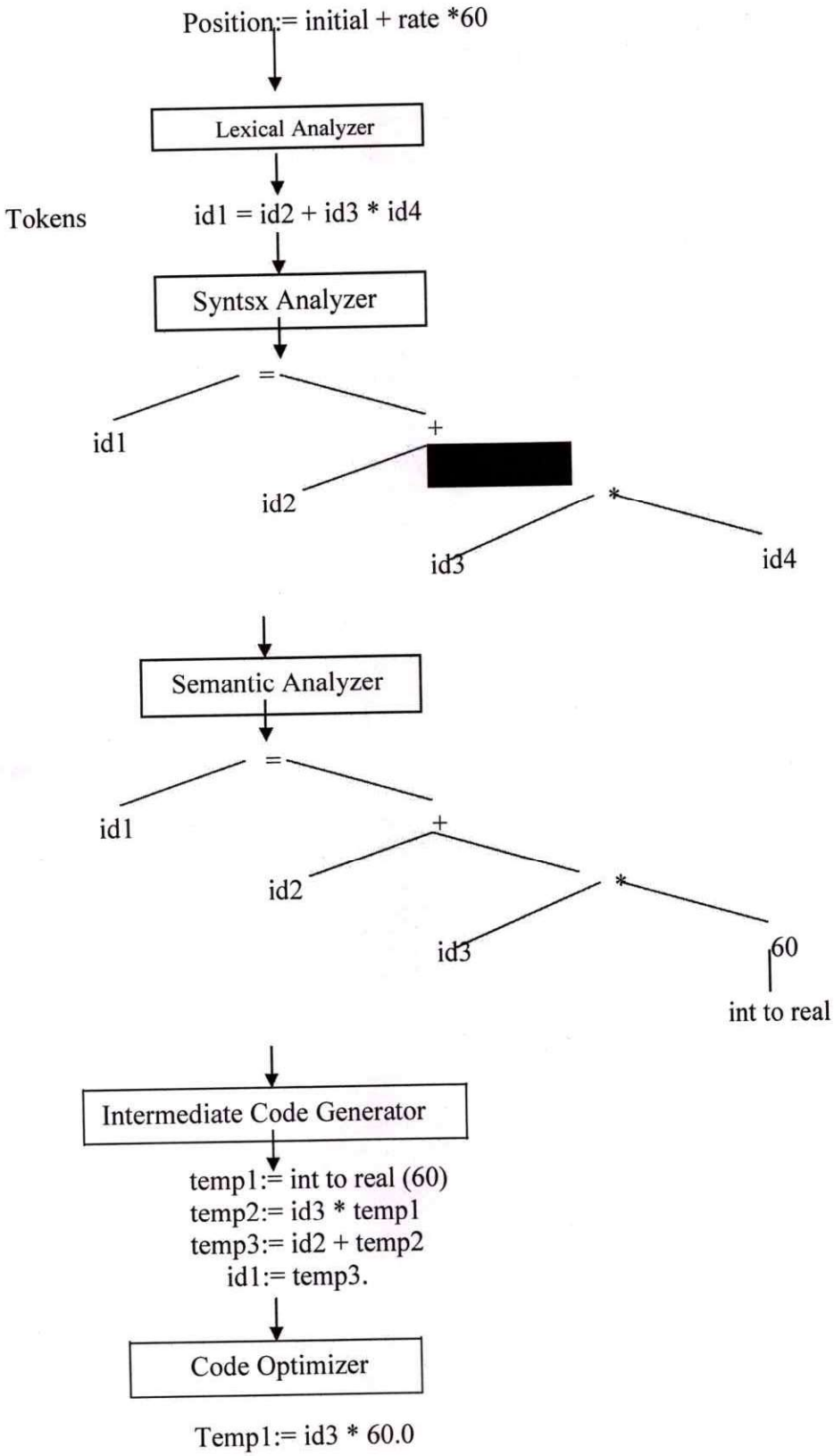
A compiler needs to collect information about all the data objects that appear in the source program. The information about data objects is collected by the early phases of the compiler-lexical and syntactic analyzers. The data structure used to record this information is called as Symbol able.

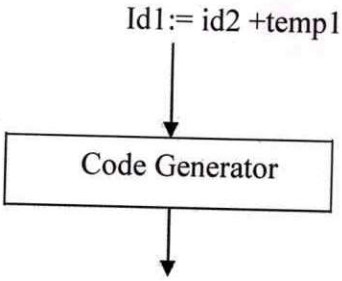
Error Handing :-

One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic msg. Both of the table-management and error-Handling routines interact with all phases of the compiler.

Example:


Position:= initial + rate *60
Lexical Analyzer
Tokens
id1 = id2 + id3 * id4
Syntsx Analyzer
=
id1
+
id2
*
id3
id4
Semantic Analyzer
=
id1
+
id2
*
id3
60
int to real
Intermediate Code Generator
temp1:= int to real (60)
temp2:= id3 * temp1
temp3:= id2 + temp2
id1:= temp3.
Code Optimizer
Temp1:= id3 * 60.0




```
MOVF  id3, r2
MULF  *60.0, r2
MOVF  id2, r2
ADDF  r2, r1
MOVF  r1, id1
```

1.10 TOKEN

LA reads the source program one character at a time, ca ving the s u ce p og am into a sequence of automatic units called 'Tokens'.

1, Type of the token.

2, Value of the token.

Type : variable, operator, keyword, constant

Value : N1ame of variable, current variable (or) pointer to symbol table.

If the symbols given in the standard format the LA accepts and produces token as output. Each token is a sub -string of the program that is to be treated as a single unit. Token are two types.

1, Specific strings such as IF (or) semicolon.

2, Classes of string such as identifiers, label, constants.

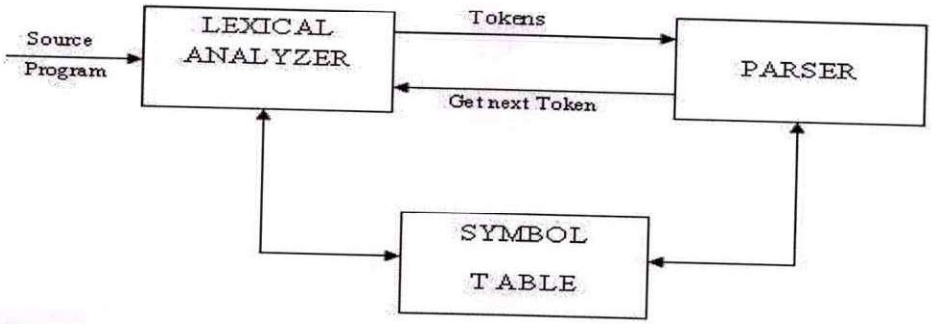
LEXICAL ANALYSIS

OVER VIEW OF LEXICAL ANALYSISWorld

- To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token ecognize s, which are designed using transition diagrams and finite automata.

ROLE OF LEXICAL ANALYZER

the LA is the first phase of a compiler. It main task is to ead the input character and produce as output a sequence of tokens that the parser uses f syntax analysis.



Upon receiving a 'get next token' command form the parser, the lexical analyzer reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.

LA may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

LEXICAL ANALYSIS VS PARSING:

Lexical analysis	Parsing
A Scanner simply turns an input String (say a file) into a list of tokens. These tokens represent things like identifiers, parentheses, operators etc. The lexical analyzer (the "lexer") parses individual symbols from the source code file into tokens. From there, the "parser" proper turns those whole tokens into sentences of your grammar	A parser converts this list of tokens into a Tree-like object to represent how the tokens fit together to form a cohesive who e (sometimes referred to as a sentence). A parser does not give the no es any meaning beyond st uctu al cohesion. The next thing to do is ext act meaning f om this structure (sometimes called contextual analysis).

TOKEN, LEXEME, PATTERN:

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example:

Description of token

oken	lexeme	pattern
const	const	const
if	if	If
relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
i	pi	any numeric constant
nun	3.14	any character b/w "and "except"
literal	"core"	pattern

A patter is a rule describing the set of lexemes that can represent a particular token in source program.

LEXICAL ERRORS:

Lexical errors are the errors thrown byWorldyourlexerwhenunabletocontinue.Whichmeans that there's no way to recognise a *lexeme* as a valid *token* for you lexer. Syntax errors, on the other side, will be thrown by your scanner when a given set of **already** recognised valid tokens don't match any of the right sides of your grammar rules. simp e panic-mo e error handling system requires that we return to a high-level parsing function when a parsing lexical error is detected.

Error-recovery actions are:

i. Delete one character from the remaining input.
ii. Insert a missing character in to the remaining input.
iii. Replace a character by another character.
iv. Transpose two adjacent characters.

DIFFERENCE BETWEEN COMPILER AND INTERPRETER

- A compiler converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.
- Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.
- List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.
- An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.
- he compiler produce object code whereas interpreter does not produce object code.
- In the process of compilation the program is analyzed only once and then the code is generated whereas source program is interpreted every time it is to be executed and every time the source program is analyzed. hence interpreter is less efficient than compiler.
- Examples of interpreter: A *UPS Debugger* is basically a graphical source level debugger but it contains built in C interpreter which can handle multiple source files. example of compiler: *Borland c compiler* or Turbo C compiler compiles the programs written in C or C++.

REGULAR EXPRESSIONS

Regular expression is a formula that describes a possible set of string.

Component of regular expression..

X	**the character x**
.	**any character, usually accept a new line**
[x y z]	**any of the characters x, y, z, ...**
R?	**a R or nothing (=optionally as R)**
R*	**zero or more occurrences.....**
R+	**one or more occurrences .....**
R1R2	**an R1 followed by an R2**
R2R1	**either an R1 or an R2.**

A token is either a single string or one of a collection of strings of a ce tain type. If we view the set of strings in each token class as an language, we can use the egu ar-expression notation to describe tokens.

Consider an identifier, which is defined to be a letter f ll wed by ze mo e letters or digits. In regular expression notation we would write.

Identifier = letter (letter | digit)*

Here are the rules that define the regular expression ver alphabet .

- is a regular expression denoting { € }, that is, the language containing only the empty string. World
- For each 'a' in ∑, is a regular expression denoting { a }, the language with only one string consisting of the single symbol 'a' .
- If R and S are regular expressions, then

(R) | (S) means LrULs
R.S means Lr.Ls
R* denotes Lr*

REGULAR DEFINI IONS

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example-1,

Ab*|cd? Is equivalent to (a(b*)) | (c(d?))

Pascal identifier

Letter - A | B | .....| Z | a | b |.....| z|
Digits - 0 | 1 | 2 | .... | 9
Id - letter (letter / digit)*

Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examins the input string and finds a prefix that is a lexeme matching one of the patterns.

Stmt →if expr then stmt
| If expr then else stmt
| ϵ
Expr →term relop term
| term
Te m →id
|number

For relop ,we use the comparison operations of languages like Pascal or SQL where = is "equals" and < > is "not equals" because it presents an interesting st uctu e of lexemes. The terminal of grammar, which are if, then , else, rel p ,id and numbe s a e the names of tokens as far as the lexical analyzer is concerned, the patterns f the t kens a e described using regular definitions.

digit -->[0,9]
digits -->digit+
number -->digit(.digit)?(e.[+-]?digits)?
letter -->[A-Z,a-z]
id -->letter(letter/digit)*
if --> if
then -->then
else -->else
relop --></>/<=/>=/==/< >

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

ws → (blank/tab/newline)$^{+}$

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. oken ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

Lexeme	Token Name	Attribute Value
Any ws	_	_
if	if	_
then	then	_
else	else	_
Any id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT

<=	relop	LE
=	relop	ET
< >	relop	NE

TRANSITION DIAGRAM:

Transition Diagram has a collection of nodes or circles, ca ed states. Each state represents a condition that could occur during the process of scanning the input ooking for a lexeme that matches one of several patterns .

Edges are directed from one state of the transition diagram to another. each e ge is abe ed by a symbol or set of symbols.

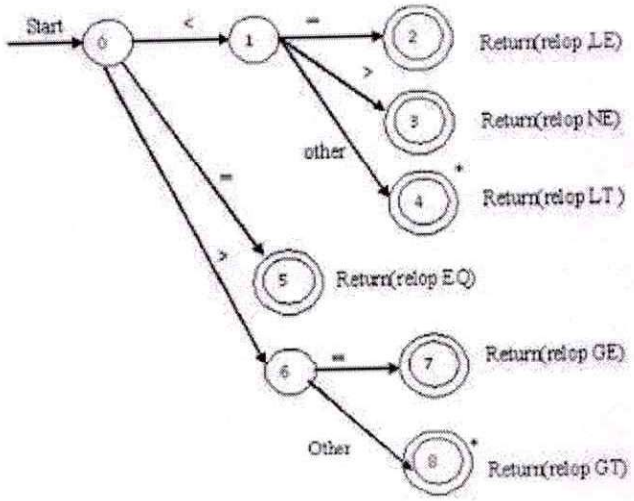
If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forwa d pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

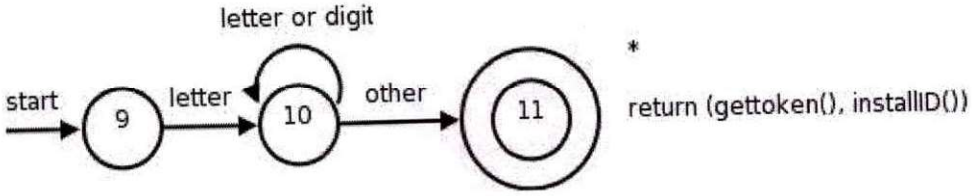
1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not c nsist f all p siti ns b/w the lexeme Begin and forward pointers we always indicate an accepting state by a d uble circle.

2. In addition, if it is necessary to return the forward p inter ne p siti n, then we shall additionally place a * near that accepting state.

3. One state is designed the state ,or initial state ., it is indicated by an edge labeled "start" entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.



As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.



The above TD for an identifier, defined to be a letter followed by any no of letters or digits.A sequence of transition diagram can be converted into program to ook for the tokens specified by the diagrams. Each state gets a segment of code.

If	=	if
Then	=	then
Else	=	else
Relop	=	< | <= | = | > | >=
Id	=	letter (letter | digit) *|
Num	=	digit |

AUTOMATA

An automation is defined as a system where inf rmati n is transmitted and used for performing some functions without direct participati n f man.

1, an automation in which the output depends only on the input is **called an automation without memory.**

2, an automation in which the output depends on the input and state also is **called as automation with memory** .

3, an automation in which the output depends only on the state of the machine is **called a Moore machine** .

3, an automation in which the output depends on the state and input at any instant of time is **called a mealy machine** .

DESCRIP ION OF A OMATA

1, an automata has a mechanism to read input from input tape,

2, any language is recognized by some automation, Hence these automation are basically language 'acceptors' or 'language recognizers'.

Types of Finite Automata

- Deterministic Automata
- on-Deterministic Automata.

DETERMI ISTIC AUTOMATA

A deterministic finite automata has at most one transition from each state on any input. A DFA is a special case of a NFA in which:-

1, it has no transitions on input €,

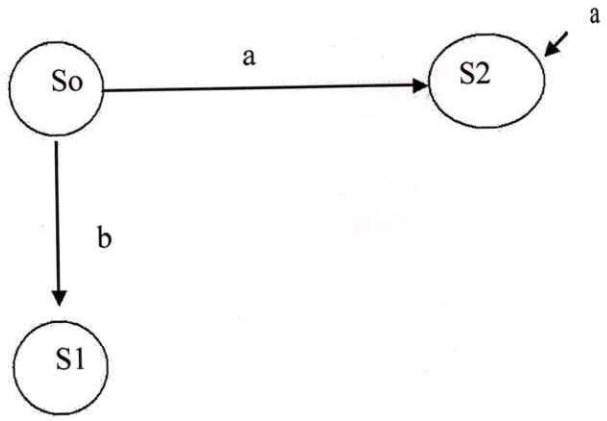
2, each input symbol has at most one transition from any state.

DFA formally defined by 5 tuple notation M = (Q, ∑, δ, qo, F), where
Q is a finite 'set of states', which is non empty.
∑ is 'input alphabets', indicates input set.
qo is an 'initial state' and qo is in Q ie, qo, ∑, Q
F is a set of 'Final states', ~~World~~
δ is a 'transmission function' mapping function, using this function the next state can be determined.

The regular expression is converted into minimized DFA by the fo owing proce ure:

Regular expression → NFA → DFA → Minimized DFA

The Finite Automata is called DFA if there is only one path for a specific input from current state to next state.



From state S0 for input 'a' there is only one path going to S2. similarly from S0 there is only one path for input going to S1.

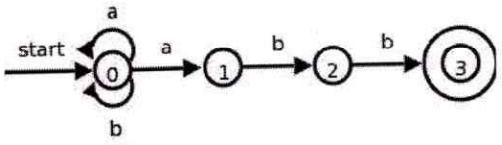
O DE ERMINIS IC AUTOMATA

- A FA is a mathematical model that consists of
  - A set of states S.
  - A set of input symbols ∑.
  - A transition for move from one state to an other.
  - A state so that is distinguished as the start (or initial) state.
  - A set of states F distinguished as accepting (or final) state.
  - A number of transition to a single symbol.

- ..... transition graph, In which the nodes are the states and the labeled edges represent the transition function.

- This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labeled by the special symbol € as well as by input symbols.

- The transition graph for an NFA that recognizes the language (a | b) * abb is shown



DEFINITION OF CFG

It involves four quantities.

CFG contain terminals, N -T, start symbol and production.

- Terminal are basic symbols form which string are formed.
- N-terminals are synthetic variables that denote sets of strings
- In a Grammar, one N -T are distinguished as the start symbol, and the set of string it denotes is the language defined by the grammar.

JNTUTheproductionofthegrammar specify the manor in which the terminal and -T can be combined to form strings.

- Each production consists of a N -T, followed by an arrow, followed by a string of one terminal and terminals.

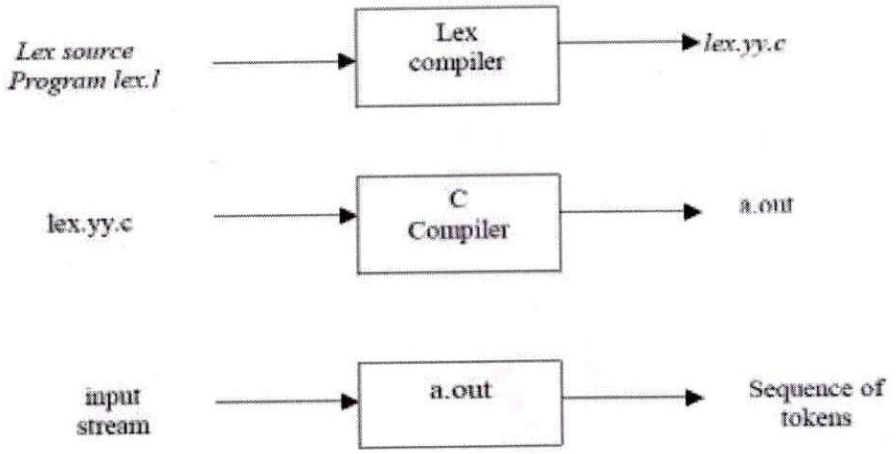
DEFINI ION OF SYMBOL TABLE

- An extensible array of records.
- he identifier and the associated records contains collected information about the identifier.

  FUNCTION identify (Identifier name)
  RETURNING a pointer to identifier information contains

- The actual string
- A macro definition
- A keyword definition
- A list of type, variable & function definition
- A list of structure and union name definition
- A list of structure and union field selected definitions.

Creating a lexical analyzer with Lex



Lex specifications:

A Lex program (the .l file) consists of three parts:

declarations
%%
translation rules
%%
auxiliary procedures

1. The *declarations* section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. *# define PIE 3.14*), and regular definitions.
2. The *translation rules* of a Lex program are statements of the form :

p1	*{action 1}*
p2	*{action 2}*
p3	*{action 3}*
...	...
...	...

where each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.

3. The third section holds whatever *auxiliary procedures* are needed by the *actions.*Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

Note: You can refer to a sample lex program given in page no. 109 of chapter 3 of the book: *Compilers: Principles, Techniques, and Tools* by Aho, Sethi & Ullman for more clarity.

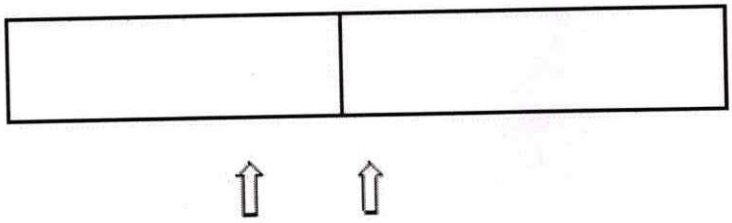
INPUT BUFFERING

The LA scans the characters of the source pgm one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Buffering techniques:

1. Buffer pairs
2. Sentinels

The lexical analyzer scans the characters of the source program one a t a time to discover tokens. Often, however, many characters beyond the next t ken many have to be examined before the next token itself can be determined. For this and ther eas ns, it is desi able for thelexical analyzer to read its input from an input buffer. Figure sh ws a buffer divided into two haves of, say 100 characters each. One pointer marks the beginning f the token being discovered. A look ahead pointer scans ahead of the beginning p int, until the token is discovered .we view the position of each pointer as being between the character last read and thecharacter next to be read. In practice each buffering scheme adopts one convention either apointer is at the symbol last read or the symbol it is ready to read.



Token beginnings look ahead pointer

oken beginnings look ahead pointerThe distance which the lookahead pointer may have to travel past the actual token may belarge. For example, in a PL/I program we may see: DECALRE (ARG1, ARG2... ARG *n*) Without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file. Since the buffer shown in above figure is of limited size there is an implied constraint on how much look ahead can be used before the next token is discovered. In the above example, ifthe look ahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been groupedinto tokens. While we can make the buffer larger if we chose or use another buffering scheme,we cannot ignore the fact that overhead is limited.

Unit-2

Parsers

- Top down Parsers (TDP)
  - TDP with Full Backtracking (Brute force approach)
  - TDP without Back tracking
    - Recursive descent
    - Non Recursive decent (LL(1))
- Bottom up Parsers (BUP)
  - Operator precedence parser (Allow ambiguous)
  - LR Parsers
    - LR(0)
    - SLR(1)
    - CLR(1)
    - LALR(1)

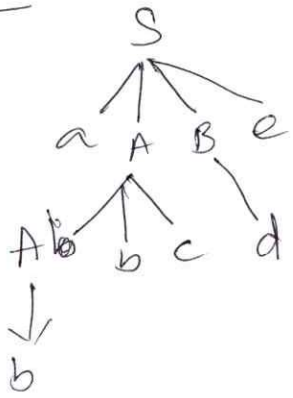
TDP

$S \rightarrow a\,ABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

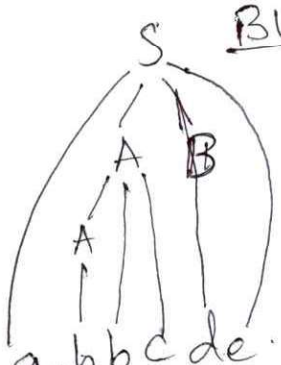
String : abbcde



Note :- In TDP we have to decide what production use.

Note :- In TDP we have to decide what production we have to use.

BUP



a b b c d e

Note :- In BUP, the task is when to reduce

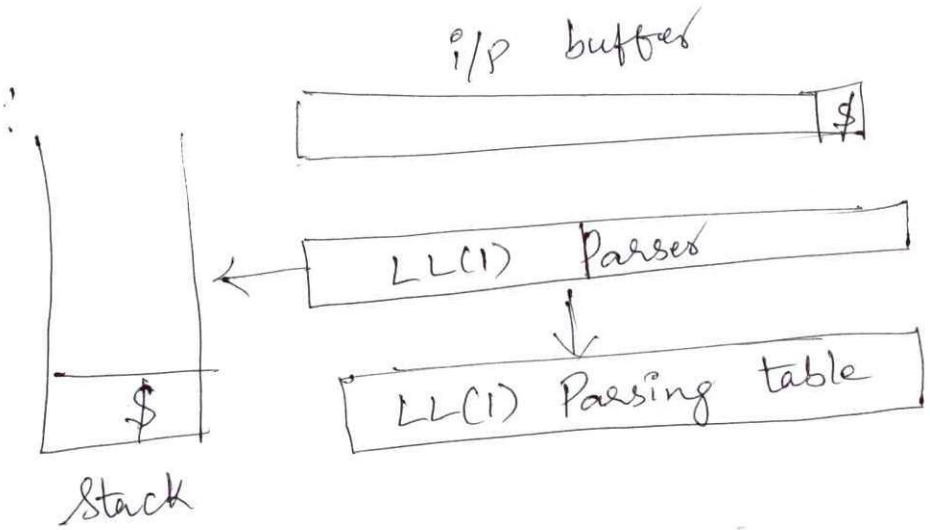
Note :-

→ TDP follows LMD.

→ BUP follows reverse of RMD.

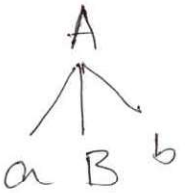
LL(1) — No. of lookaheads (how many symbols we see to take decision).

Scan i/p from left to right — Leftmost derivation



First() :- first character of the string that can be obtained by applying a production rule.

Follow() :-



$A \rightarrow aBb$

$B \rightarrow c \mid \epsilon$

Input string is "ab" to parse

→ As the first character in i/p is 'a', the passer applies $A \rightarrow aBb$

→ Now the passer checks the second character of i/p string which is 'b', but the passer can't get any string derivable from 'B' that contains 'b' as first character.

→ But grammar contain a production rule $B \rightarrow \epsilon$, if it is applied then B will vanish and the parser gets "ab". But the parser can apply it only when it knows that the character that follows B is same as the current character in the input.

(2)

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

Follow(X) to be set of terminals that can appear immediately to the right of Nonterminal X in some sential form.

Rules to Compute "Follow" set.

1) FOLLOW(S) = {$} // where S is start symbol).
2) If $A \to pBq$ is a production, where p, B and q are any grammar symbols, then everything in FIRST(q) except ϵ is in FOLLOW(B).
3) If $A \to qB$ is a production, then everything in FOLLOW(A) is in FOLLOW(B).
4) If $A \to pBq$ is a production and FIRST(q) contains ϵ, then FOLLOW(B) contains {FIRST(q) − ϵ} ∪ FOLLOW(A).

Eg :-

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

Follow () : what is the terminal followed by a non terminal.

Eg:- $A \rightarrow aBb$ Follow (B) = {b}.

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

Rules to compute FIRST

1) If X is a terminal, then FIRST(X) = {'X'}.
2) If $X \rightarrow \epsilon$, is a production rule, then add ϵ to first(X).
3) If $X \rightarrow Y_1 Y_2 Y_3 \cdots Y_n$ is a production.
   (a) FIRST(X) = FIRST(Y1).
   (b) If FIRST(Y1) contains ϵ, then FIRST(X) = {FIRST(Y_1) − ϵ} ∪ {FIRST(Y_2)}
   (c) If FIRST(Y_i) contains ϵ for all i = 1 to n then add ϵ to FIRST(X).

Eg:-

	FIRST
① $E \rightarrow TE'$	FIRST(T) = {(, id}.
$E' \rightarrow +TE' \mid \epsilon$	{+, ϵ}
$T \rightarrow FT'$	{(, id}
$T' \rightarrow *FT' \mid \epsilon$	{*, ϵ}
$F \rightarrow (E) \mid id$	{(, id}

	Action						Goto
	id	+	*	(	)	$	E
0	s_3	e_1	e_1	s_2	e_2	e_1	1
1	e_3	s_4	s_5	e_3	e_2	Acc	
2	s_3	e_1	e_1	s_2	e_2	e_1	6
3	r_4	r_4	r_4	r_4	r_4	r_4	
4	s_3	e_1	e_1	s_2	e_2	e_1	7
5	s_3	e_1	e_1	s_2	e_2	e_1	8
6	e_3	s_4	s_5	e_3	s_9	e_4	
7	r_1	r_1	s_5	r_1	r_1	r_1	
8	r_2	r_2	r_2	r_2	r_2	r_2	
9	r_3	r_3	r_3	r_3	r_3	r_3	

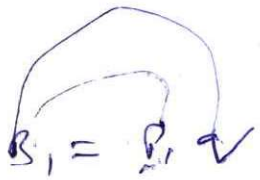
e1 :- Missing operand (+, *, $)

e2 :- Unbalanced right parenthesis. ( ))

e3 :- missing operator. (id, ( )

e4 : Missing right parenthesis ($).

id +)

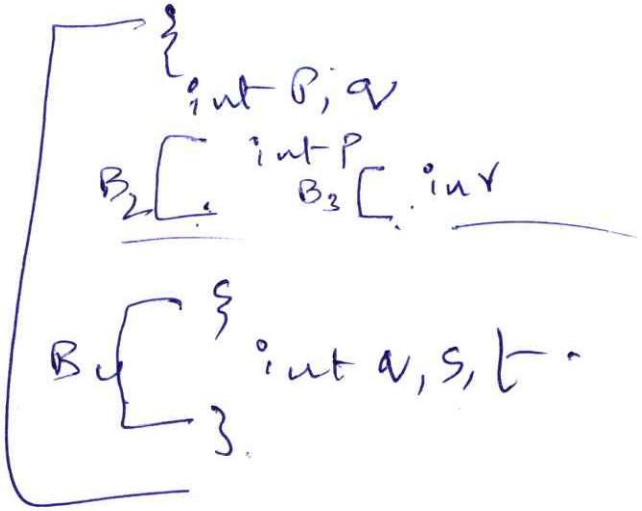


$B_1 = P, q$

$B_2 = P$

$B_3 = r$

$B_4 = q, s, t$



Symbol Tables

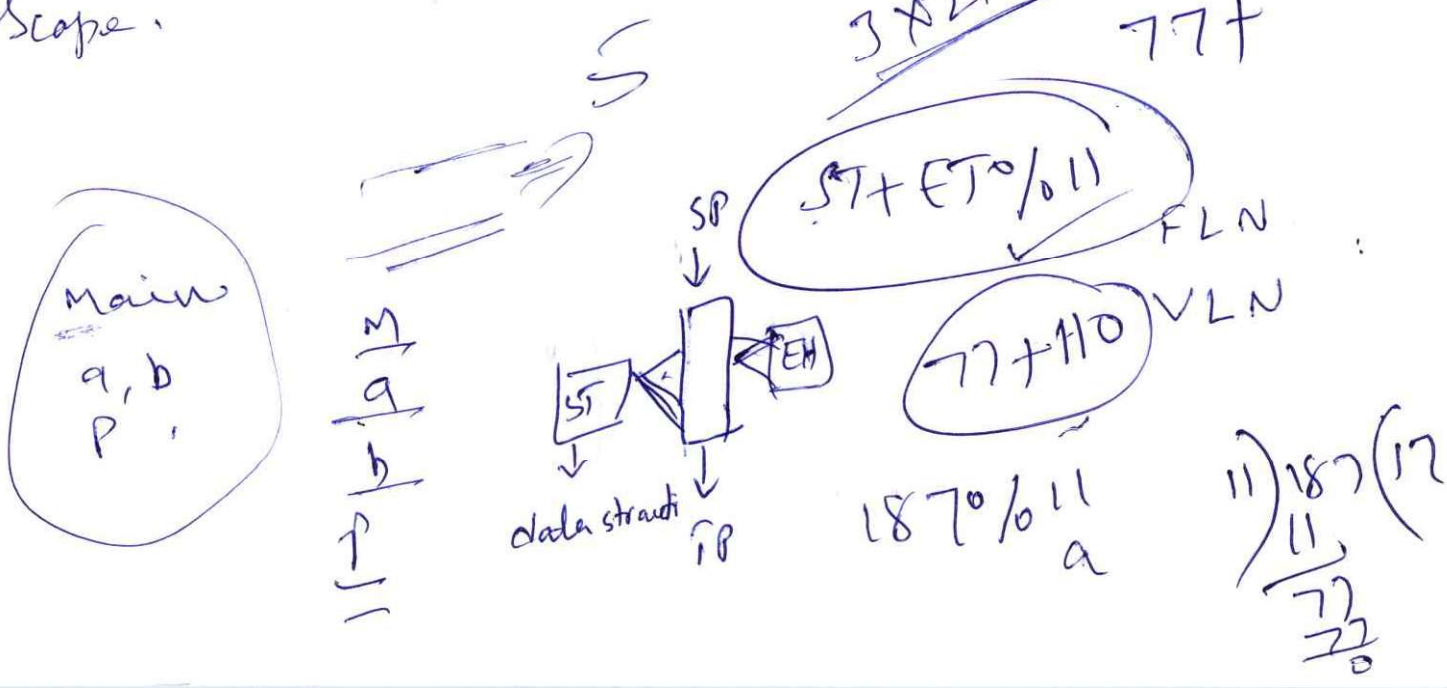
→ Data structures used to hold information about Source program Constructs.

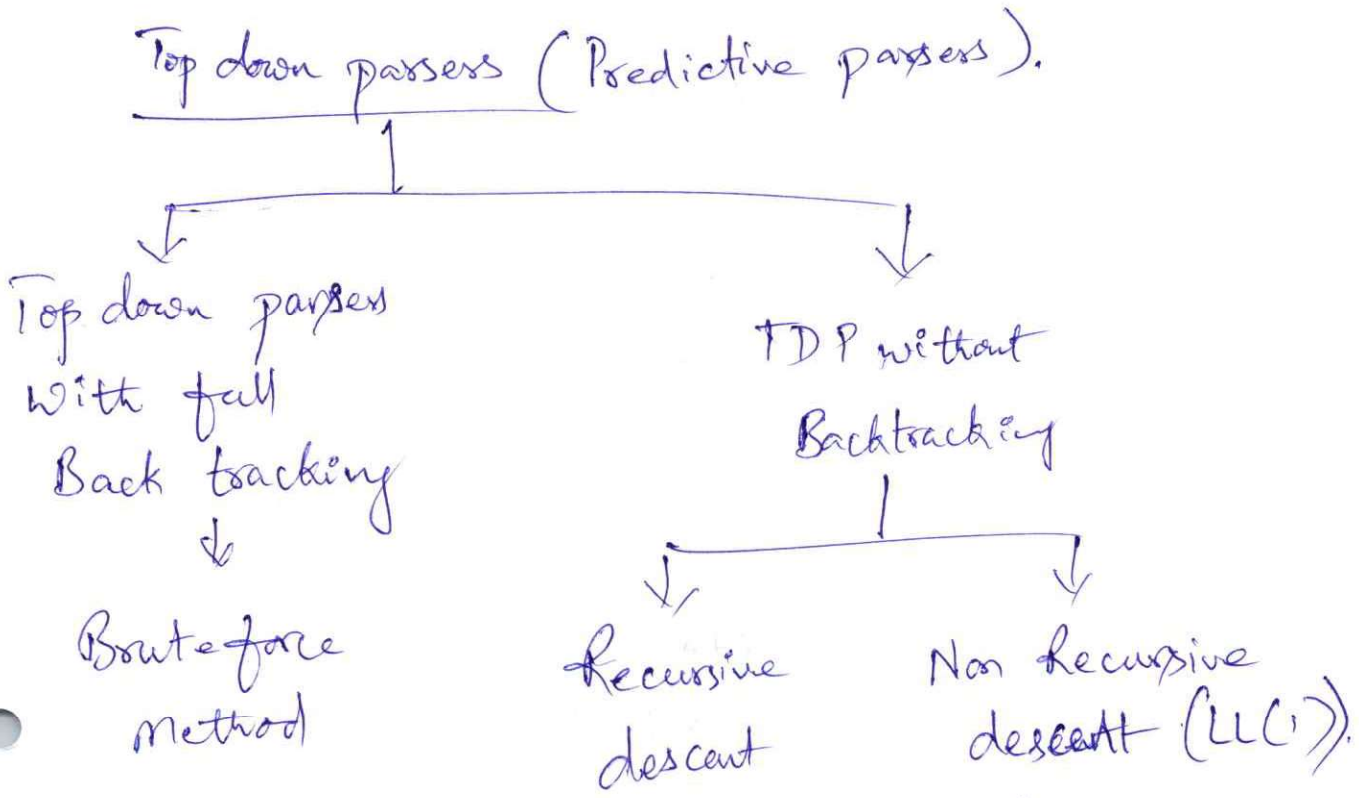
→ Information Collected incrementally by analysis phases of Compiler and used by Synthesis phases to generate target code.

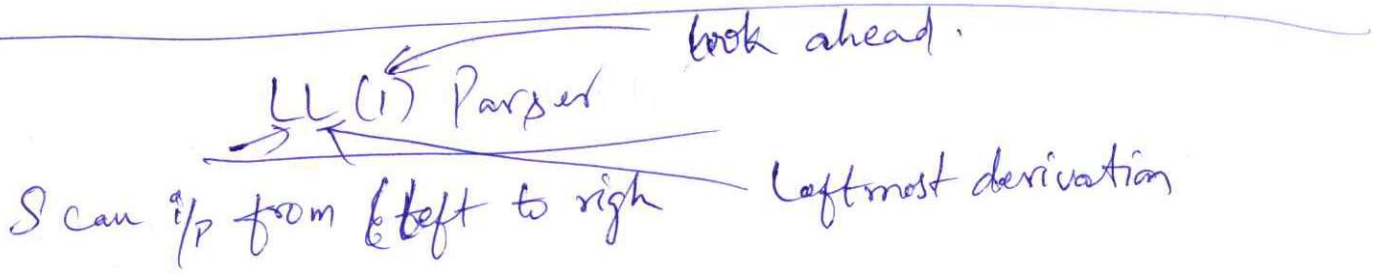
→ To record the variable names used in the Source program and Collect information about various attributes of each name.

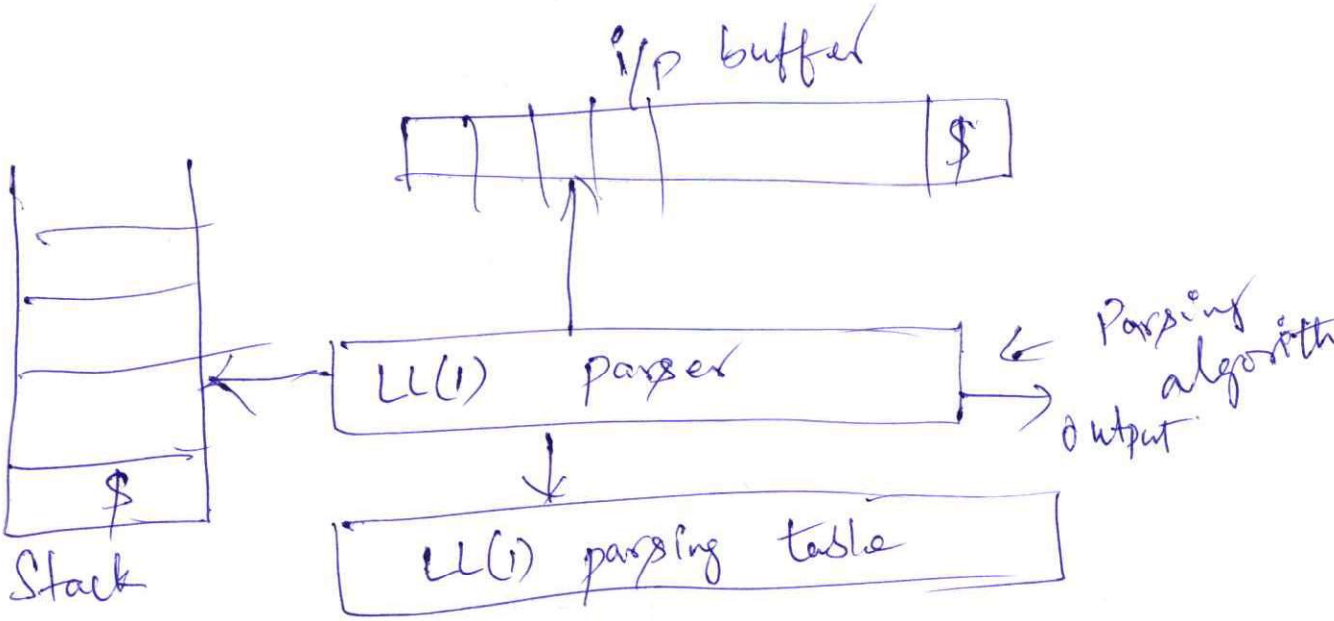
→ Attributes may provide information about Storage allocated for a name, its type, its scope, and in case of procedure names, such things as the number and types of its arguments, the method of passing each argument and return type etc.

→ Scope of a declaration is the portion of a program to which the declaration applies. The scope shall be implement by setting up a Separate Symbol table for each Scope.




Top down parsers (Predictive parsers).
Top down parsers With full Back tracking
Brute force method
TDP without Backtracking
Recursive descent
Non Recursive descent (LL(1))


look ahead.
LL(1) Parser
S can i/p from left to right
Leftmost derivation


i/p buffer
$
LL(1) parser
Parsing algorithm
output
$
Stack
LL(1) parsing table

Matched	Stack	Input	Action
	$E\$$	$id+id*id\$$	
	$TE'\$$	$id+id*id\$$	output $E \to TE'$
	$FT'E'\$$	$id+id*id\$$	output $T \to FT'$
	$idT'E\$$	$id+id*id\$$	output $F \to id$
id	$T'E\$$	$+id*id\$$	match id.
id	$E\$$	$+id*id\$$	output $T' \to \epsilon$
id	$+TE'\$$	$+id*id\$$	output $E \to +TE'$
$id+$	$TE'\$$	$id*id\$$	match +
$id+$	$FT'E'\$$	$id*id\$$	Output $T \to FT'$
$id+$	$idT'E'\$$	$id*id\$$	output $F \to id$
$id+id$	$T'E'\$$	$*id\$$	match id
$id+id$	$*FT'E'\$$	$*id\$$	output $T' \to *FT'$
$id+id$	$FT'E'\$$	$id\$$	match *
$id+id*$	$idT'E'\$$	$id\$$	Output $F \to id$
$id+id*$	$T'E'\$$	$\$$	match id.
$id+id*id$	$E'\$$	$\$$	Output $T' \to \epsilon$
$id+id*id$	$\$$	$\$$	output $E' \to \epsilon$
$id+id*id$			

→ Parser Controlled by a program that Considers 'X', the Symbol on top of Stack and 'a' the Current i/p Symbol.

→ If 'X' is non terminal, the parser chooses an X- production by Consulting entry $M[X, a]$ of the parsing table M. Otherwise, it checks for a match b/w the terminal X and Current i/p Symbol 'a'.

$E \rightarrow iE'$

$E' \rightarrow +iE' / \epsilon$

```
E()
{
  if (l=='i')
  { match('i')
    E'()
  }
  l=getchar();
E'()
{
  if (l=='+')
  {.match(+);
    match(i);
    E'()
  }
  else
  return
}
match(char t)
{ if (l==t)
    l= gtchar();
  else
    pritf("error");
}
```

```
main()
{
  E();
  if (l=='$'){
    printf("parsy Success");
}
  else
    printf("Parsing unsuccessful");
}
```

RDP: It is a type of TDP built from set of mutually recursive procedures where each nonterminal of a given grammar is implemented as a procedure.

$E \to E+T \mid E-T \mid T$

$T \to T*F \mid T/F \mid F$

$F \to (E) \mid id.$

$E \to TE'$

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

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id.$

$A \to A\alpha \mid \beta$

$A \to \beta A'$

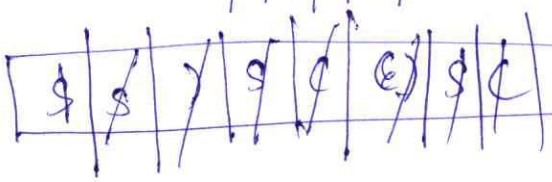
$A' \to \alpha A' \mid \epsilon$

First(α) = α is any string of grammar symbols, to be the set of terminals that begin strings derived from α.

$S \to (S) \mid \epsilon$

	(	)	$
S	$S \to (S)$	$S \to \epsilon$	$S \to \epsilon$

(( )) $

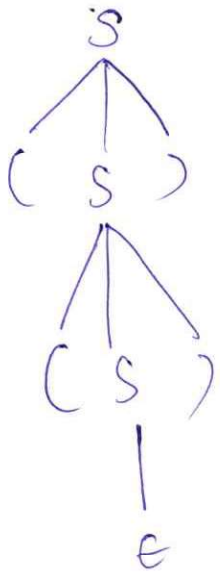


$S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

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



Recursive Descent parsing

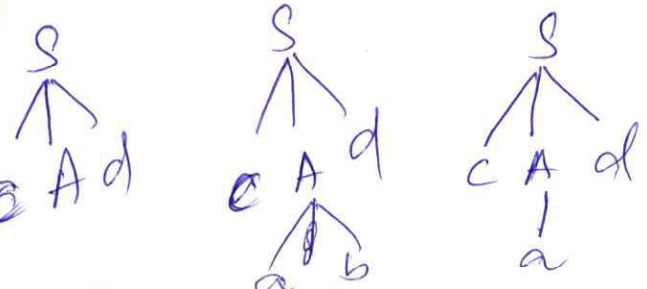
→ RDP parsing program consists of a set of procedures, one for each nonterminal.

→ Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire i/p string.

$S \to cAd$

$A \to ab \mid a$

i/p cad



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

Parsing Table

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

(2)

	First	Follow
$S \to ACB \mid Cbb \mid Ba$	$\{d, g, h, \epsilon, b, a\}$	$\{\$\}$
$A \to da \mid B\epsilon$	$\{d, g, \epsilon, h\}$	$\{h, g, \$\}$
$B \to g \mid \epsilon$	$\{g, \epsilon\}$	$\{a, h, g, \$\}$
$C \to h \mid \epsilon$	$\{h, \epsilon\}$	$\{b, h, g, \$\}$

(3)

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

(4)

	First	Follow
$S \to A$	$\{a\}$	$\{\$\}$
$A \to aB \mid Ad$	$\{a\}$	$\{\$\}$
$A \to aBA' \mid dA'$	$\{a\}$ → $\{d, \epsilon\}$	$\{\$\}$
~~$B \to b$~~ $A' \to ad \mid \epsilon$	$\{b\}$	$\{d, \$\}$
$B \to b$	$\{g\}$	NA
$C \to g$		

(5)

	First	Follow
$S \to (L) \mid a$	$\{(, a\}$	$\{\$, ,,)\}$
$L \to SL'$	$\{(, a\}$	$\{)\}$
$L' \to ,SL' \mid \epsilon$	$\{,, \epsilon\}$	$\{)\}$

(6) $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

	First	Follow
S	{a, b}	{$}
A	{ε}	{a, b}
B	{ε}	{b, a}

(1)

$E \to E+E$ — (1)

$E \to E*E$ — (2)

$E \to (E)$ — (3)

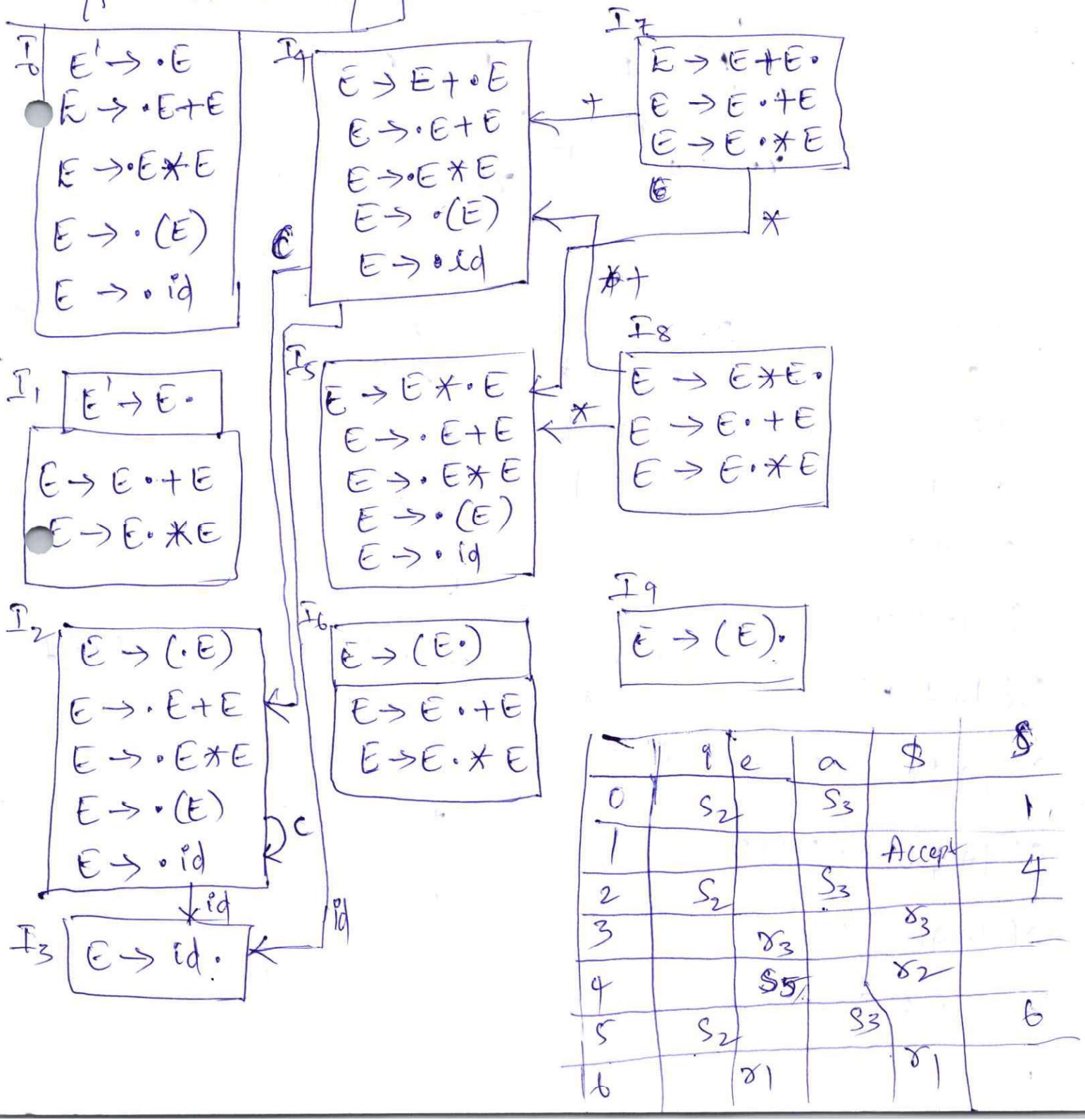
$E \to id$ — (4)

i/p = id+ id*id $

(2) $S \to iSeS \mid is \mid a$

i/p = iiaeaa$

Follow(E) = {$, +, *,)}



	id	(	a	$	S
0	S_2		S_3		1
1				Accept	
2	S_2		S_3		4
3		γ_3		γ_3	
4		S_5		γ_2	
5	S_2		S_3		6
6		γ_1		γ_1	

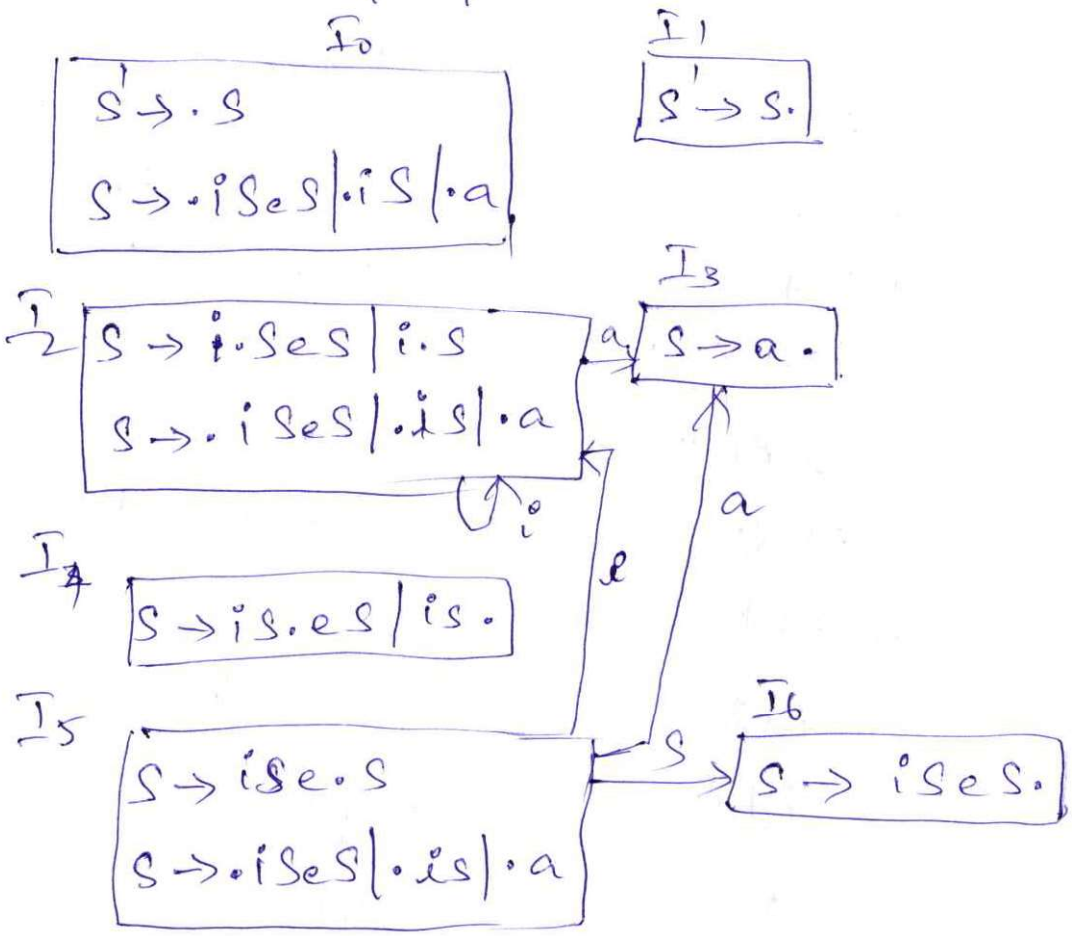
	id	+	*	(	)	$	E
0	S3			S2			1
1		S4	S5				
2	S3			S2			6
3	~~S3~~	γ_4	γ_4	~~S2~~	γ_4	γ_4	~~6~~
4	S3			S2			7
5	S3			S2			8
6		S4	S5		S9		
7		S4/γ_1	S5/γ_1		γ_1	γ_1	
8		S4/γ_2	S5/γ_2		γ_2	γ_2	
9		γ_3	γ_3		γ_3	γ_3	

Stack	Input	Action
0	id+id*id$	Shift
0id3	+id*id$	E → id
0E1	+id*id$	Shift
0E1+4	id*id$	Shift
0E1+4id3	*id$	E→id
0E1+4E7	*id$	Shift/reduce Conflict but use shift
0E1+4E7*5	id$	Shift
0E1+4E7*5id3	$	E→id
0E1+4E7*5E8	$	E → E*E
0E+4E7	$	E → E+E
0E1	$	Accept

i/p = id * id + id

Stack	Input	Action
0	id * id + id $	Shift
0 id 5	* id + id $	reduce F → id
0 F 3	* id + id $	reduce T → F
0 T 2	* id + id $	Shift
0 T 2 * 7	id + id $	Shift
0 T 2 * 7 id 5	+ id $	reduce F → id
0 T 2 * 7 F 10	+ id $	reduce T → T * F
0 T 2	+ id $	reduce E → T
0 E 1	+ id $	Shift
0 E 1 + 6	id $	Shift
0 E 1 + 6 id 5	$	reduce F → id.
0 E 1 + 6 F 3	$	reduce T → F
0 E 1 + 6 T 9	$	reduce E → E + T
0 E 1	$	Accept.

$S \to iSeS \mid iS \mid a$



Stack	Input	Action
0	iiaea$	shift
0i2	iaea$	shift
0i2i2	aea$	shift
0i2i2a3	eea$	S→a
0i2i2S4	eea$	shift
0i2i2S4e5	a$	shift
0i2i2S4e5a3	$	S→a
0i2i2S4e5S6	$	S→iSeS
0i2S4	$	S→iS
0S1	$	Accept

0i2S4 | ea$ | ea$

S→iS
~~shift~~ S→iS
~~error~~ error

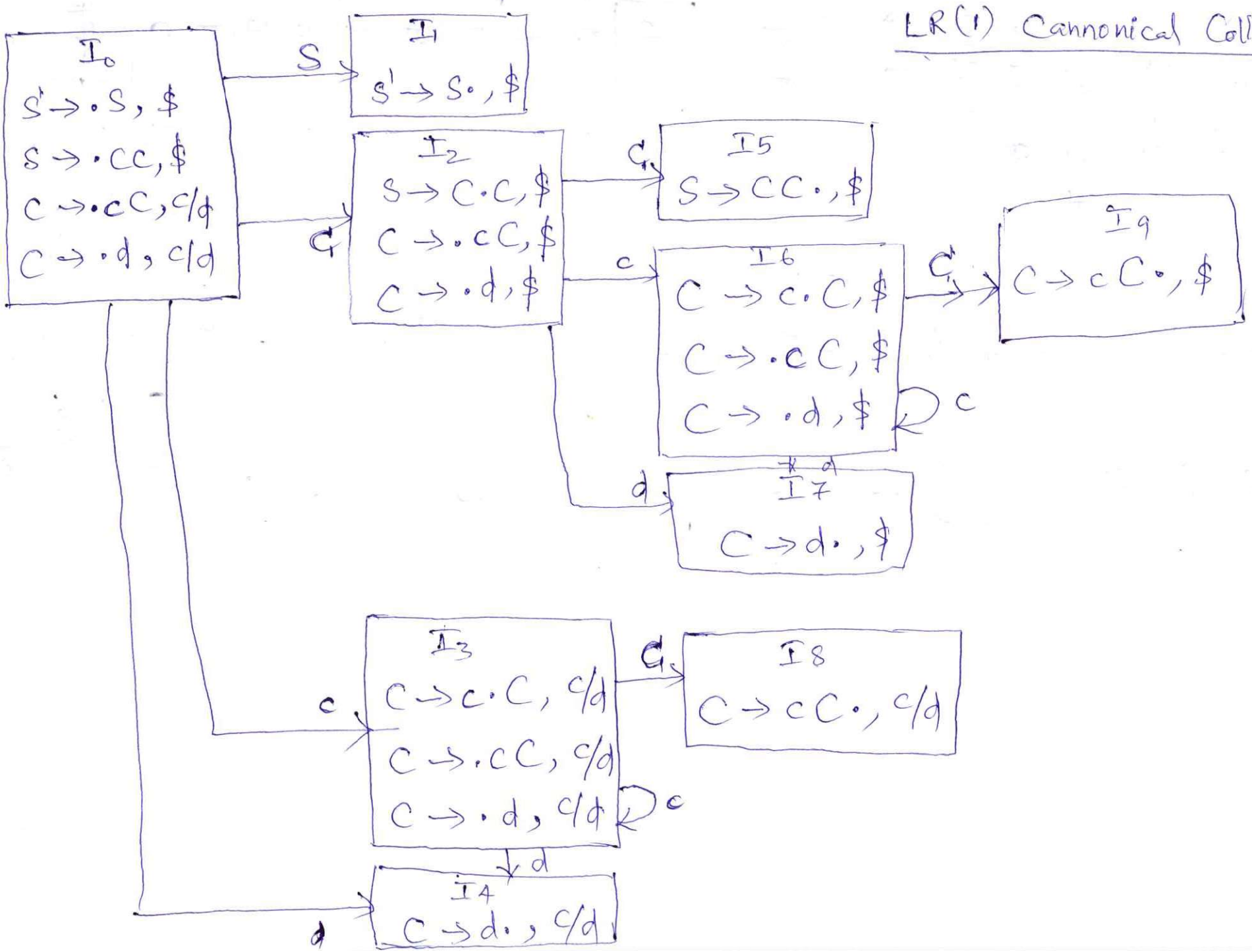
CLR(1) Parsing Table

State	Action			GOTO	
	c	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	r_3	r_3			
5			r_1		
6	S6	S7			9
7			r_3		
8	r_2	r_2			
9			r_2		

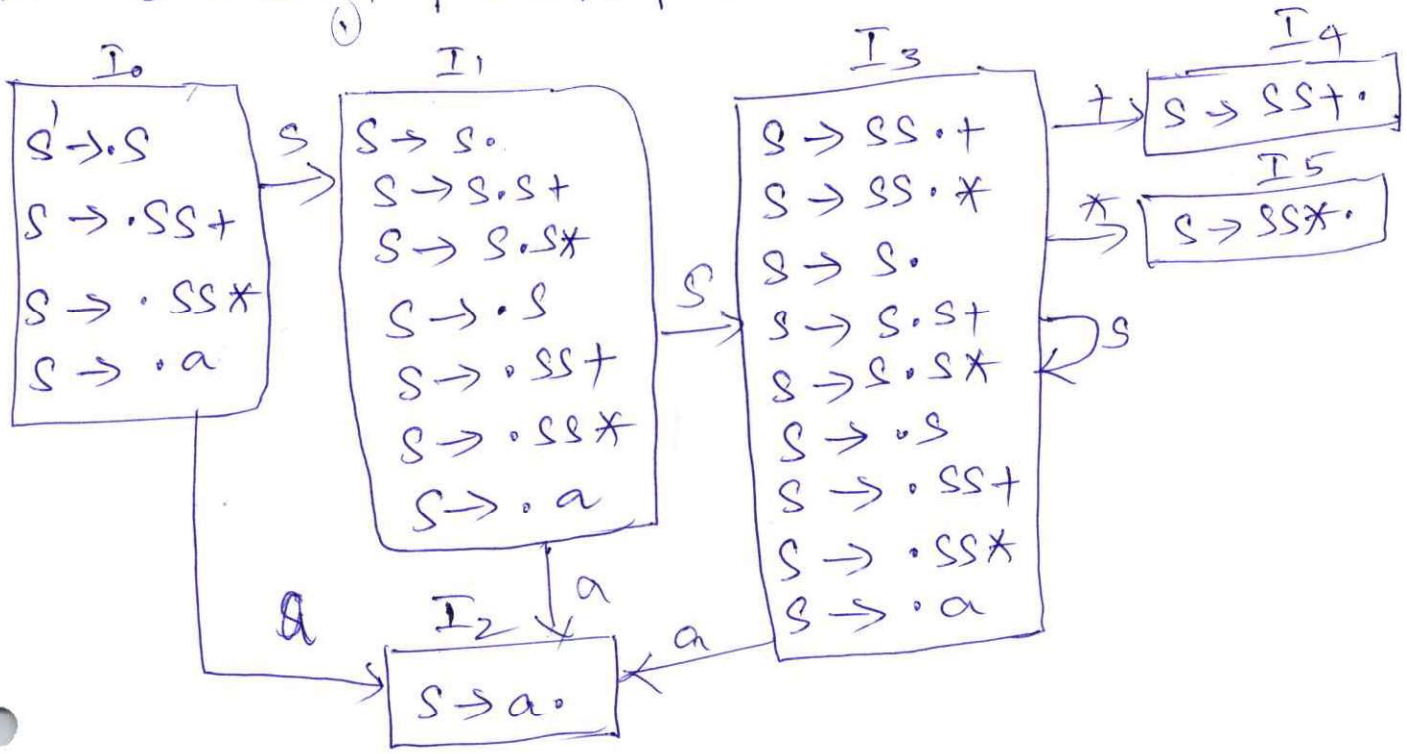
Stack	Input	Action
0	dcd$	Shift
0d4	cd$	C → d
0C2	cd$	Shift
0C2c6	d$	Shift
0C2c6d7	$	C → d
0C2c6C9	$	C → cC
0C2C5	$	S → CC
0S1	$	Accept

$S' \to S$, $S \to CC$, $C \to cC \mid d$

LR(1) Cannonical Collection



Ex:- $S \to SS+ \mid SS* \mid a$



	Action				Goto
	+	*	a	$	S
0			S2		1
1				Accept	
2	r_3	r_3	r_3	r_3	
3	S4	S5	S2		
4	r_1	r_1	r_1	r_1	
5	r_2	r_2	r_2	r_2	

Input:- a a* a+

Input = aabb

Stack	Input	Action
0	a a b b $	Shift
0 a 3	a b b $	Shift
0 a 3 a 3	b b $	Shift
0 a 3 a 3 b 4	b $	~~Shift~~ reduce
0 a 3 a 3 A 6	b $	reduce.
0 a 3 A 6	b $	reduce
0 A 2	b $.	Shift
0 A 2 b 4	$	reduce.
0 A 2 A 5	$	reduce.
0 S 1	$.	Accept

→ four possible actions a shift-reduce parser can make:

(1) Shift :- Shifts the next i/p symbol on-to 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 nonterminal to replace the string.

(3) Accept :- Announce successful completion of parsing.

(4) Error :- Discover syntax error and call an error recovery routine.

→ The handle will always appear on top of the stack, never inside.

Shift - Reduce Parsing (S-R parsing)

→ S-R parsing is a form of bottom up parsing in which a stack holds grammar symbols and an i/p buffer holds the rest of the string to be parsed.

Stack	Input
$	w $

→ The handle always appears at the top of the stack, just before it is identified as the handle.

→ During left to right scan of the i/p string, the parser shifts zero, or more i/p 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 i/p is empty.

Stack	Input
$S	$

→ Upon entering this configuration, the parser halts and announces successful completion of parsing.

Eg:-

Stack	Input	Action
$	id1 * id2 $	shift
$id1	* id2 $	reduce by F → id
$F	* id2 $	reduce by T → F
$T	* id2 $	shift
$T*	id2 $	shift
$T*id2	$	reduce by F → id
$T*F	$	reduce by T → T * F
$T	$	reduce by E → T
$E	$	accept.

Bottom-Up Parsing (BUP)

→ BUP is known as shift-reduce parsing.

→ A general method of shift reduce parsing is called LR parsing.

→ Shift reduce attempts to construct a parse tree for an i/p string beginning at leaves and working up towards the root.

→ We can think of this process as one of "reducing" a string 'w' to the start symbol of grammar.

→ At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left of that production.

Eg:- $S \rightarrow aABe$ ①
$A \rightarrow Abc \mid b$ ②③
$B \rightarrow d$ ④

String "abbcde" can be reduced to "S"

$aAbcde \longrightarrow$ ③ $A \rightarrow b$

$aAde \longrightarrow A \rightarrow Abc$

$aABe \longrightarrow B \rightarrow d$

$S \longrightarrow S \rightarrow aABe$

Handle :- A substring that matches the right side of a production.

→ A "handle" of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A.

→ The string 'w' to the right of the handle contains only terminal symbols.

→ If the grammar is unambiguous, then every right sentential form of a grammar has exactly one handle.

→ Reducing β to A in αβw can be thought of as "pruning the handle", that is removing the children of 'A' from the parse tree.

→ A reduction is the reverse of a step in derivation.

Handle Pruning

→ Bottom up parsing during left to right scan of the i/p constructs a right most derivation in reverse.

Eg:

Right Sentential form	Handle	Reducing production
$id_1 * id_2$	id_1	$F \to id$
$F * id_2$	F	$T \to F$
$T * id_2$	id_2	$F \to id$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$

$E \to E * T \mid T$; $T \to T * F \mid F$; $F \to (E) \mid id$.

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

Closure :

If I is set of items for a grammar G then

Closure (I) is the set of items constructed from I by two rules

(1) Initially, add every item in I to closure (I)

(2) If $A \to \alpha . B \beta$ is in Closure (I) and $B \to \gamma$ is a production, then add the item $B \to . \gamma$ to Closure (I), if it is not already there. Apply this rule until no more new items can be added to closure (I).

Goto

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 $[A \to \alpha X . \beta]$ such that $[A \to \alpha . X \beta]$ is in I:

	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6	S7			Accept			
2		γ_2	S7		γ_2	γ_2			
3		γ_4	γ_4		γ_4	γ_4			
4	S5			S4			8	2	3
5		γ_6	γ_6		γ_6	γ_6			
6	S5			S4				9	3
7	S5			S4					
8		S6			S11				10
9		γ_1	S7		γ_1	γ_1			
10		γ_3	γ_3		γ_3	γ_3			
11		γ_5	γ_5		γ_5	γ_5			

LR(0)

Given Grammar

$S \rightarrow A\ A$

$A \rightarrow a\ A \mid b$

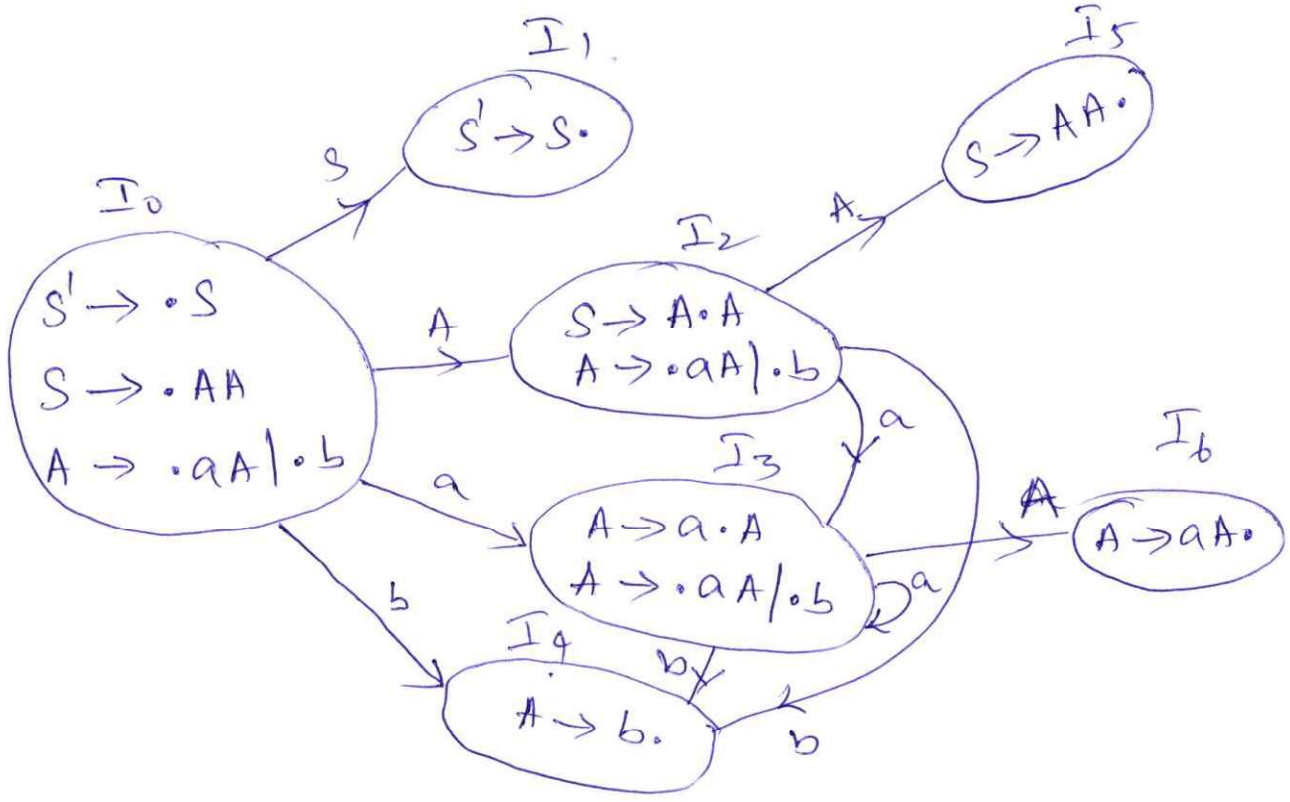
Augmented Grammar

$S' \rightarrow S$ — (1)

$S \rightarrow A\ A$ — (2)

$A \rightarrow a\ A$ (3) $\mid\ b$ (4)

Canonical Collection of LR(0)



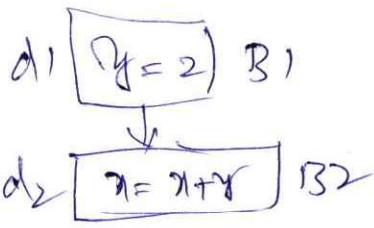
Closure :- If there is an '.' before a variable (or) non terminal then add all productions that variable.

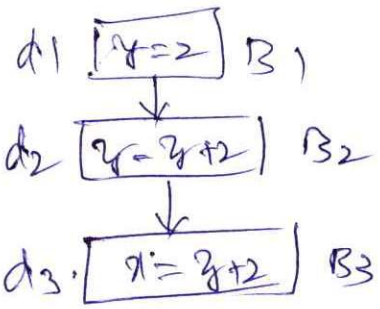
goto :- If state does not contain final item then apply goto for each production in the state.

LR(0) Parsing Table

	Action			Goto	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S3	S4			5
3	S3	S4			6
4	γ_4	γ_4	γ_4		
5	γ_2	γ_2	γ_2		
6	γ_3	γ_3	γ_3		

(2) Reaching Definition :- Def D reaches at Point P if there is a path from D to P along which D is not killed.

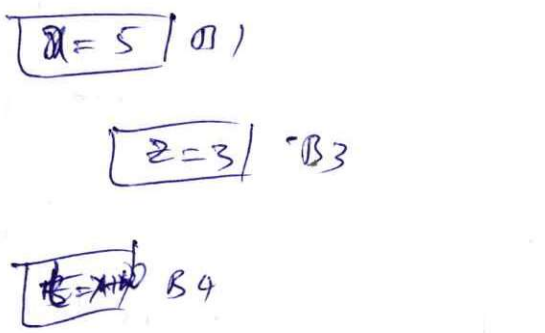




d_1: reachable to B2 but not B3.

Adv :- Used in Constant & variable propagation

(3) Live Variable :- A variable 'x' is live at some point 'p' if there is a path from p to the exit, along which the value of x is used before it is redefined. otherwise the variable is said to be dead.



Adv :- It is useful in reg alloc
" " " dead code elimination.

(4) Busy expression :- expression 'e' is said to be a busy exp along some path $P_i \dots P_j$ if it an evaluation of 'e' exists along some path $P_i \dots P_j$ and no def of any operand exists before its evaluation along the path.

Ref - Adv :- Useful in performing code movement optimization

Live Variable Analysis (or) Live Range Identification

→ This analysis is useful for code generation.

→ for register allocation.

→ Two important definitions.

(1) def(B) – set of variables to which values are assigned in block B, before the actual use.

(2) Use(B) – set of variables whose values are may be used in block B, prior to any definition of the variable.

The formulae for Computing in[B] and out[B] are

$$in[B] = Use[B] \cup (out(B) - def(B))$$

$$Out[B] = \bigcup_{S} in[S]$$ where 'S' is the successor of block B.

→ in[B] → represents variable that is live coming into a block.

→ out[B] → represents that a variable is live coming out of a block if and only if it is live coming into one of its successor block.

Algorithm:

(1) Compute def and use for each block.

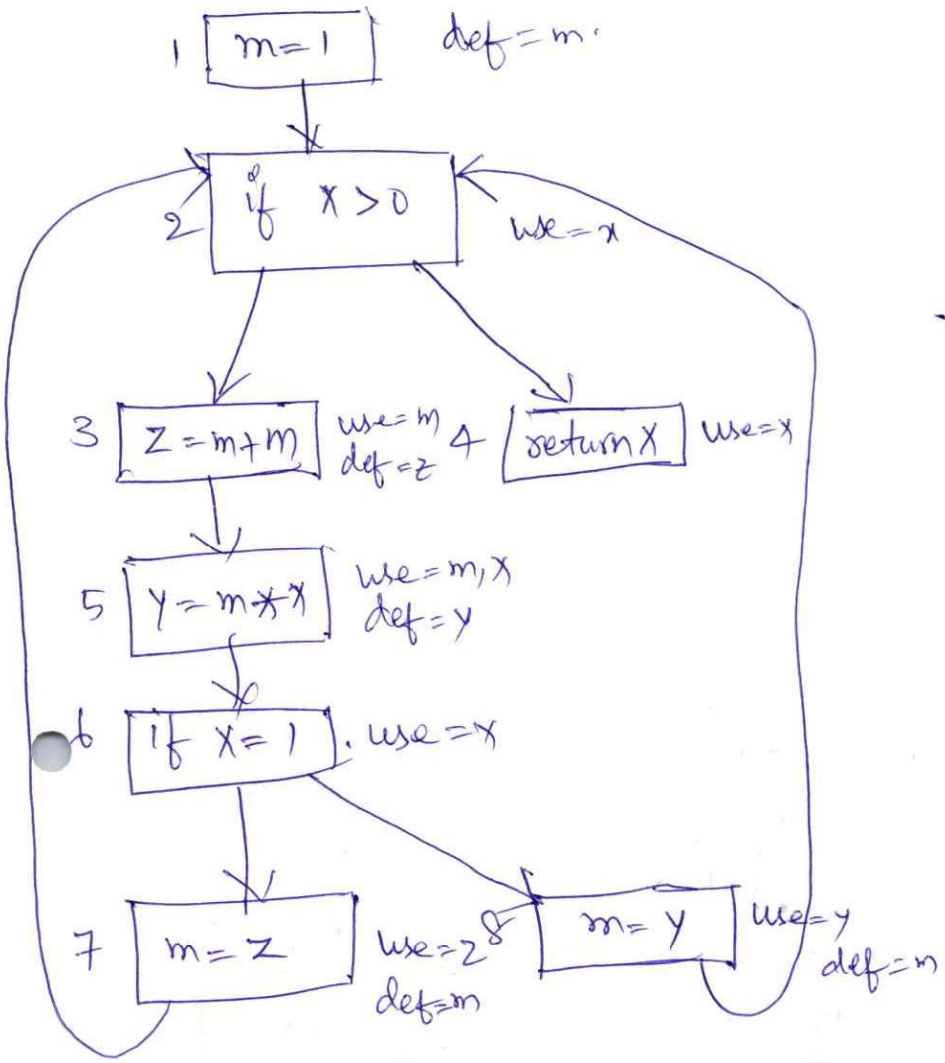
(2) For each block B initialize in[B] = ∅ and out[B] = ∅.

(3) For any change in the in[B] do step 4 to 6.

(4) For each block B do steps 5 and 6.

(5) $Out[B] = \bigcup_{S} in[S]$

(6) $in[B] = use[B] \cup (out(B) - def(B))$



Initially

	in	out
1	ϕ	ϕ
2		
3		
4		
5		
6		
7		
8	ϕ	ϕ

After 1st pass

	in	out
1	ϕ	ϕ
2	x	ϕ
3	m	ϕ
4	~~m~~,x	ϕ
5	m,x	ϕ
6	x	ϕ
7	~~x~~,z	x
8	x,y	x

Pass 2

	in	out
1	~~x~~	x
2	m,x	m,x
3	m,x	m,x
4	~~m,x~~	
5	m,x	x
6	x,y,z	x,y,z
7	m,x	m,z
8	x,y	m,x

Redundant Common Sub expression elimination

→ Algorithm

(1) Discover the evaluations of b+c that reach in the block containing statement S.

(2) Create new variable 'm'.

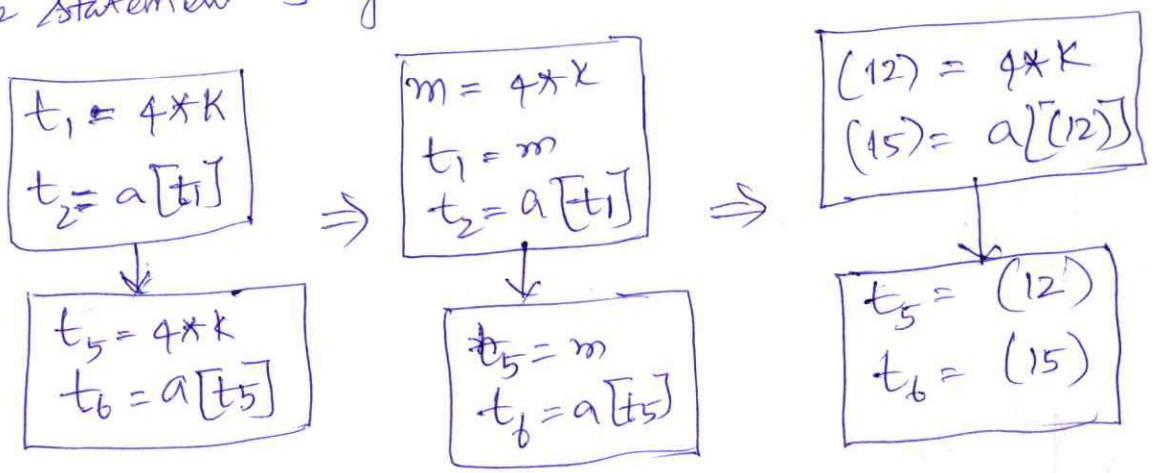
(3) Replace each statement d = b+c, which is obtained in Step 1.

$m = b + c$

$d = m$

(4) Replace statement S by $a = m$.

Eg:-



Copy propagation

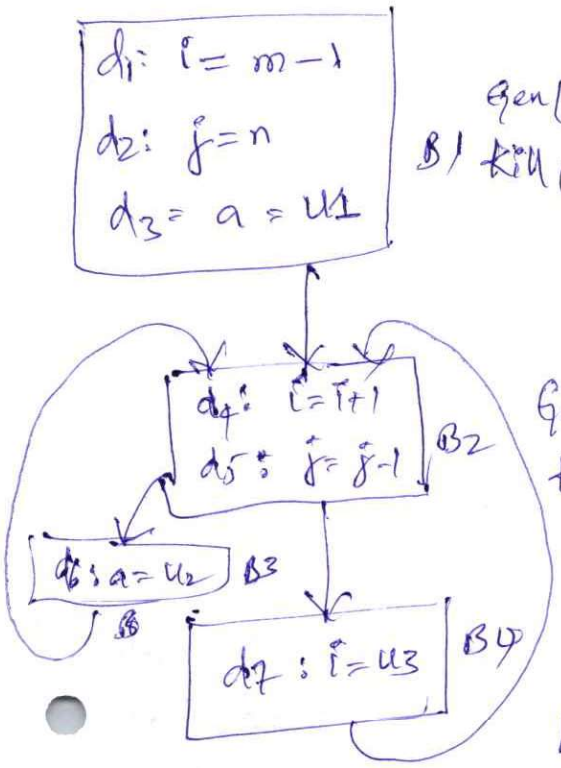
→ The assignment statement is called copy statement.

Algorithm:-

(1) Determine the statements in which x is used. These statements should be reachable from definition x.

(2) There should not be any definition of x or y occur prior to use of x in that block containing S.

(3) If S satisfies the condition mentioned in Step (2) then remove S and replace all uses of x found in (1) by y.



Gen(B1) = {1110000}
B1 kill(B1) = {0001111}

Gen (B2) = {0001100}
kill (B2) = {1100001}

Gen (B3) = {0000010}
kill (B3) = {0010000}

Gen (B4) = {0000001}
kill (B4) = {1001000}

	Initial		Pass 1	
	IN	Out	IN	Out
B1	ϕ	{1110000}	{ϕ}	{1110000}
B2	ϕ	{0001100}	{1110011}	{0001100}
B3	ϕ	{0000010}	{0001100}	{0001110}
B4	ϕ	{0000001}	{0001101}	{0000101}

3-Address Code

```
x=1
while (x<=10)
{
    y= y*2;
    x= x+1;
}
```

```
for(x=1; x<=10; x++)
{
    y= y*2;
}
```

1. x = 1
2. if x > 10 goto 8
3. t_1 = y * 2
4. y = t_1
5. t_2 = x + 1
6. x = t_2
7. goto 2
8.

---

a[10]

1. t_1 = addr(a)
2. t_2 = 10 * 4
3. t_3 = t_1[t_2]

```
int a[10], b[10], i, dp=0;
for (i=0; i<10; i++)
{
    dp += a[i] * b[i];
}
```

1. dp = 0
2. i = 0
3. if i >= 10 goto 16
4. t_1 = addr(a)
5. t_2 = i * 4
6. t_3 = t_1[t_2]
7. t_4 = addr(b)
8. t_5 = i * 4
9. t_6 = t_4[t_5]
10. t_7 = t_3 * t_6
11. t_8 = dp + t_7
12. dp = t_8
13. t_9 = i + 1
14. i = t_9
15. goto 3
16.

```
Switch(i+j)
{
  Case 1:
      x= y*8
  Case 2:
      y= 8/2;
  default:
      l= l*5
}
```

1. t = i + j
2. goto 9
3. t_1 = y * 8
4. x = t_1
5. t_2 = 8/2
6. y = t_2
7. t_3 = l * 5
8. l = t_3
9. goto 13
10. if t == 1 goto 3
11. if t == 2 goto 5
12. goto 7

---

a[i][j] m x n

00	01	02
10	11	12
20	21	22
30	31	32

4 x 3

(3x3+1)*4 ✓

(i*n+j)*4 ✓

1. t_1 = addr(a)
2. t_2 = i * n
3. t_3 = t_2 + j
4. t_4 = 4 * t_3
5. t_5 = t_1[t_4]

Data flow Analysis (DFA).

→ Data flow property property represents the certain information regarding usefulness of the data items for optimization.

(1) Available expressions (2) Reaching definitions.

(3) Live variables (4) Busy variables.

→ DFA is a process of computing values of data flow properties.

(a) Definition point contains def

(b) Reference point refers to data items.

(c) evaluation point some evaluation exp is given.

$w_1 = x = 3$ → def

$w_2 = y = x$ → ref

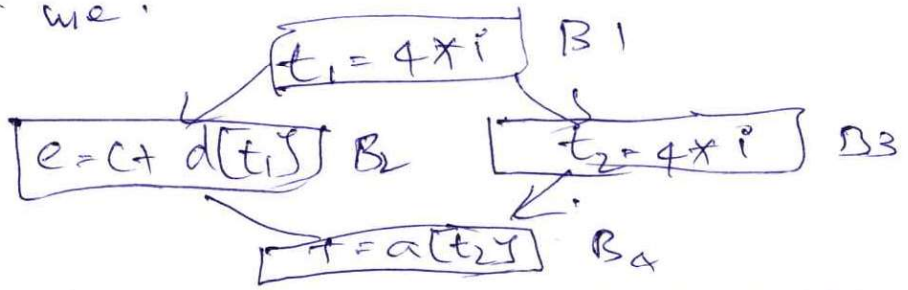
$w_3 = z = a * b$ → eval

(1) Available expression:- An exp $x+y$ is available at prog point 'w' if & only if all paths are reaching to w.

(a) The expression $x+y$ is said to be available at eval point.

(b) The expression $x+y$ is said to be available if no definition of any operand of exp (either x or y) follows its last eval only the path. In other words, if neither of the two operand get modified before their use.

Adv:- Used in elimination of CSE.



1. $t = i + j$
2. goto 10
3. $t_1 = y * 2$
4. $x = t_1$
5. $t_2 = 8/2$
6. $r = t_2$
7. $t_3 = k * 5$
8. $k = t_3$
9. goto 13
10. if $t == 1$ goto 2
11. if $t == 2$ goto 5
12. goto 7

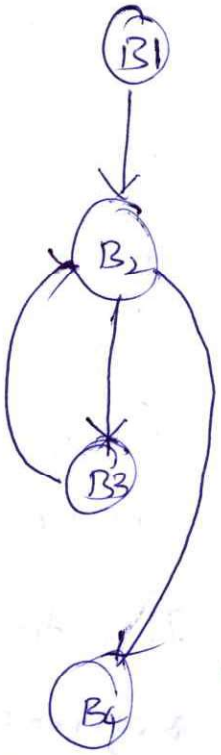
Basic block example

*1. $f = 1$
2. $x = 2$ — B1

*3. if $i <= x$ goto 8 — B2

*4. $f = f * x$
5. $t_1 = i + 1$
6. $i = t_1$
7. goto 3 — B3

*8. end — B4



Control flow graph

*1. $i = 1$ B1
*2. $j = 1$ B2
*3. $t_1 = 10 \times i$ B3
4. $t_2 = t_1 + j$
5. $t_3 = 8 * 2$
6. $t_4 = t_3 - 88$
7. $a[t_4] = 0.0$
8. $j = j + 1$
9. if $j < 10$ goto 3
*10. $i = i + 1$ B4
11. if $i <= 10$ goto 2
*12. $i = 1$ B5
*13. $t_5 = i - 1$ B6
14. $t_6 = 88 * t_5$
15. $a[t_6] = 1.0$
16. $i = i + 1$
17. if $i <= 10$ goto 13

SYNTAX ANALYSIS

ROLE OF THE PARSER

Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion. The two types of parsers employed are:

1.Top down parser: which build parse trees from top(root) to bottom(eaves)

2.Bottom up parser: which build parse trees from leaves and wo k up the oot.

Therefore there are two types of parsing methods– top-down pa sing and bottom-up parsing

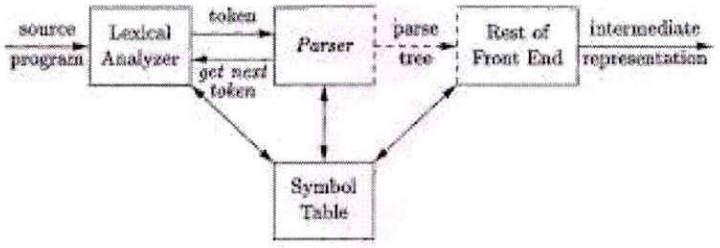


Figure 4.1: Position of parser in compiler model

TOP-DOWN PARSING

A program that performs syntax analysis is called a parser. A syntax analyzer takes tokens as input and output error message if the program syntax is wrong. The parser uses symbol-look-ahead and an approach called top-down parsing without backtracking. Top-downparsers check to see if a string can be generated by a grammar by creating a parse tree starting from the initial symbol and working down. Bottom-up parsers, however, check to see a string can be generated from a grammar by creating a parse tree from the leaves, and working up. Early parser generators such as YACC creates bottom-up parsers whereas many of Java parser generators such as JavaCC create top-down parsers.

RECURSIVE DESCENT PARSING

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for

leftmost-derivation, and k indicates k-symbol lookahead. Therefore, a parser using the single symbol look-ahead method and top-down parsing without backtracking is called LL(1) parser. In the following sections, we will also use an extended BNF notation in which some regulation expression operators are to be incorporated.

A syntax expression defines sentences World of the form , or . A syntax of the form defines sentences that consist of a sentence of the form followed by a sentence of the form followed by a sentence of the form . A syntax of the form defines zero or one occurrence of the form . A syntax of the form defines zero or more occurrences of the fo m .

A usual implementation of an LL(1) parser is:

- initialize its data structures,
- get the lookahead token by calling scanner utines, and
- call the routine that implements the start symb l.

proc syntaxAnalysis()

begin

initialize(); // initialize global data and structures

nextToken(); // get the lookahead token

program(); // parser routine that implements the start

symbol end;

FIRST AND FOLLOW

To compute FIRS (X) for all grammar symbols X, apply the following rules until no more terminals or e can be added to any FIRST set.

1. If X is terminal, then FIRST(X) is {X}.
2. If X->e is a production, then add e to FIRST(X).
3. If X is nonterminal and X->Y1Y2...Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi) and e is in all of FIRST(Y1),...,FIRST(Yi-1) that is, Y1.....Yi-1=*>e. If e is in FIRST(Yj) for all j=1,2,...,k, then add e to FIRST(X). For example, everything in FIRST(Yj) is surely in FIRST(X). If y1 does not derive e, then we add nothing more to FIRST(X), but if Y1=*>e, then we add FIRST(Y2) and so on.

To compute the FIRST(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place $ in FOLLOW(S), where S is the start symbol and $ in the input right endmarker.
2. If there is a production A=>aBs where FIRST(s) except e is placed in FOLLOW(B).
3. If there is aproduction A->aB or a production A->aBs where FIRST(s) contains e, then everything in FOLLOW(A) is in FOLLOW(B).

Consider the following example to understand the concept of First and Fo ow.Find the first and follow of all nonterminals in the Grammar-

E -> TE' E'-> +TE'|e T -> FT' T'-> *FT'|e

F -> (E)|id Then:

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

FIRST(E')={+,e} FIRST(T')={*,e}

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

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

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

For example, id and left parenthesis are added to FIRST(F) by rule 3 in definition of FIRST with i=1 in each case, since FIRST(id)=(id) and FIRST('(')= {(} by rule 1. Then by rule 3 with i=1, the production -> FT' implies that id and left parenthesis belong to FIRST(T) also.

To compute FOLLOW,we put $ in FOLLOW(E) by rule 1 for FOLLOW. By rule 2 applied toproduction F-> (E), right parenthesis is also in FOLLOW(E). By rule 3 applied to production E-> TE', $ and right parenthesis are in FOLLOW(E').

CONSTRUCTION OF PREDICTIVE PARSING TABLES

For any grammar G, the following algorithm can be used to construct the predictive parsing table. The algorithm is

Input : Grammar G
Output : Parsing table M World Method

1. 1.For each production A-> a of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(a), add A->a, to M[A,a].
3. If e is in First(a), add A->a to M[A,b] for each terminal b in FOLLOW(A). If e is in FIRST(a) and $ is in FOLLOW(A), add A->a to M[A,$].
4. Make each undefined entry of M be error.

3.6.LL(1) GRAMMAR

The above algorithm can be applied to any grammar G to pr duce a parsing table M. For some Grammars, for example if G is left recursive or ambiguous, then M will have at least one multiply-defined entry. A grammar whose parsing table has no multiply defined entries is said to be LL(1). It can be shown that the above algorithm can be used to produce for every LL(1) grammar G a parsing table M that parses all and only the sentences of G. LL(1) grammars have several distinctive properties. No ambiguous left recursive grammar can be LL(1). There remains a question of what should be done in case of multiply defined entries. One easy solution is to eliminate all left recursion and left factoring, hoping to produce a grammar which will produce no multiply defined entries in the parse tables. Unfortunately there are some grammars which will give an LL(1) grammar after any kind of alteration. In general, there are no universal rules to convert multiply defined entries into single valued entries without affecting the language recognized by the parser.

The main difficulty in using predictive parsing is in writing a grammar for the source language such that a predictive parser can be constructed from the grammar. Although left recursion elimination and left factoring are easy to do, they make the resulting grammar hard to read and difficult to use the translation purposes. To alleviate some of this difficulty, a common organization for a parser in a compiler is to use a predictive parser for control

constructs and to use operator precedence for expressions.however, if an lr parser generator is available, one can get all the benefits of predictive parsing and operator precedence automatically.

3.7.ERROR RECOVERY IN PREDICTIVE PARSING

The stack of a nonrecursive predictive parser makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. We shall therefore refer to symbols on the parser stack in the following discussion. An error is etected uring predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and the parsing table entry M[A,a] is empty.

Panic-mode error recovery is based on the idea of skipping symb ls n the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends n the choice of synchronizing set. The sets should be chosen so that the parse rec vers quickly from errors that are likely to occur in practice. Some heuristics are as f ll ws

As a starting point, we can place all symbols in FOLLO (A) into the synchronizing set for nonterminal A. If we skip tokensWorlduntilanelementofFOLLOW(A)isseenand

pop A from the stack, it is likely that parsing can continue.

- It is not enough to use FOLLOW(A) as the synchronizingset for A. Fo example , if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchica structure on constructs in a language; e.g., expressions appear within statement, which appear within bblocks,and so on. We can add to the synchronizing set of a lower construct the symbols that begin higher constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generaitn expressions.
- If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

- If a nonterminal can generate the empty string, then the production deriving e can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
- If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of a other tokens.

Semantic Analysis Unit-3.

→ Produces annotated syntax tree, after validating semantics of the program.

→ Parsers can't handle Context Sensitive features of programming languages.

→

Static Semantics
(Do not depend on Runtime System)
depends only program language definition

Eg:- (1) Variables are declared before use
(2) types of match on both sides of assignments
(3) Parameter types and number match in declaration and use. etc.

Dynamic Semantics.
(depends on Runtime and need to be checked only at execution time).

Eg:- (1) whether an overflow will occur during an arithmetic operation
(2) Array limits will be crossed during execution
(3) recursion will cross stack limits etc
(4) about variable (int, array etc)

→ During Semantic analysis "type" information is stored in the "symbol table" or the syntax tree (at nodes).

→ Usally symbol table is preferable than syntax tree because it occupy less space.

→ Type of information that is stored in symbol table is
(a) Types of variables, fuction parameters, array dimentions etc.
(b) Used not only for Semantic validation but also for subsequent phases of Compilation.

→ Static semantics of PL can be specified using attribute grammars.

→ Attribute grammars are extensions of CFG.

Terminology to understand attribute grammars

→ Every symbol(X) of a grammar (Variables and terminals) is associated with set of attributes (denoted X.a, X.b etc).

→ Two types of attributes:
Inherited attribute (AI(X)), Synthesized (AS(X))

→ Each attribute takes a value from its specific domain; which is its type.

→ Typical domains of attributes (integers, reals, chars, strings etc).

Attribute Computation Rules

→ A production has a set of attribute computation rules.

→ Rules are provided for the computation of
(a) Synthesized attributes of the LHS non-terminal of P.
(b) Inherited attributes of the RHS nonterminal of P.

→ These rules can use attributes of symbols from the production 'p' only.
* Rules are strictly local to the production P (no side effects)

→ Restrictions on the rules define different types of attribute grammars.
* L- attribute grammars (L - stands for left to right)
* S- attribute grammars (S- " Synthesized)

→ Synthesized attributes are computed in a bottom-up manner from leaves upwards.

* Always synthesized from the attribute values of the children of the node.
* leaf nodes (terminals) have synthesized attributes initialized by the lexical analyzer.
* An AG with only synthesized attributes is an S-attributed grammar.

→ Inherited attributes flow down from the parent or siblings to the node.

eg:- $S \to ABC$ ① ; $A \to aA \mid a$ ② ③ ; $B \to bB \mid b$ ④ ⑤ ; $C \to cC \mid c$ ⑥ ⑦.

$L = \{a^n b^n c^n \mid n \geq 1\}$

---

① $S \to ABC$ { S.equal↑ := if A.Count↑ = B.Count↑ & B.Count↑ := C.Count↑ then T else F }

② $A_1 \to aA_2$ { A.Count↑ := A2.Count + 1 }

③ $A \to a$ { A.Count↑ := 1 }

④ $B_1 \to bB_2$ { B.Count↑ := B.Count + 1 }

⑤ $B \to b$ { B.Count := 1 }

⑥ $C_1 \to cC_2$ { C.Count := C.count + 1 }

⑦ $C \to c$ { C.Count := 1 }

→ We defined Attributed grammar (AG) based on CFG to generate $L(G) = \{a^n b^n c^n \mid n \geq 1\}$.

$AS(S) = \{equal↑ : \{T, F\}\}$

$AS(A) = AS(B) = AS(C) = \{Count↑ : integer\}$.

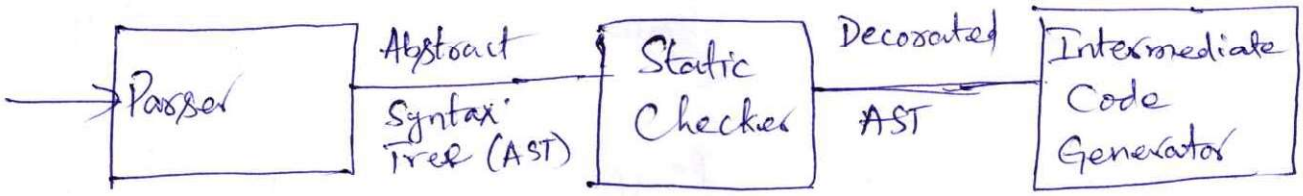
$E \rightarrow E + T$ $\{E.val = E_1.val + T.val\}$ $\{printf("+");\}$

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

$T \rightarrow T * F$ $\{T.val = T.val * F.val\}$ $\{printf("*");\}$

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

$F \rightarrow num$ $\{F.val = num.lval\}$ $\{printf("\%d", num.lval);\}$.



Static Check

(1) Type Check :- Operator applied to incompatible operand.

Eg:- 9 % 4.5 → error.

(2) Flow Control :- Statement that causes flow of Control to leave a Construct must have some place to which to transfer flow of Control.

Eg:-
```
while (1)
{
    if (Cond)
        break;
}
```
→ if we write break statement outside of while loop then it generates an error.

(3) Uniqueness Check :- There are situation where object must be defined exactly once.

Eg:- goto label;

→ Here label must be unique name.

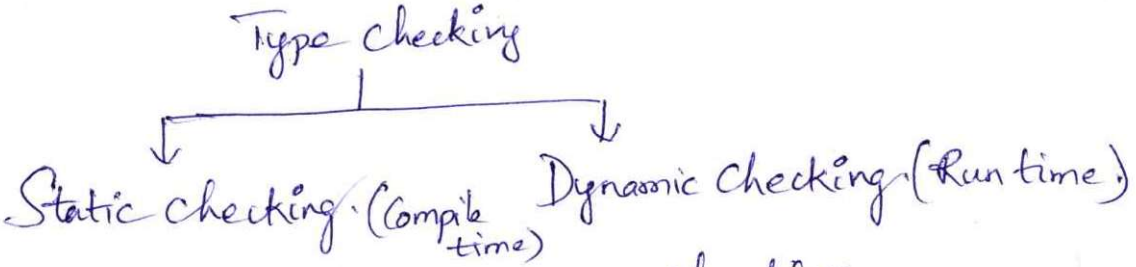
(4) Name related Check :- Some times, the same name must appear two (or) more time.

Eg:-
```
main()
{
    add(2,3);
}
```
→ There must be 'add' function definition with two integs as parameters.

Type Checking

It is a process
→ To ensure that operations are used with Correct data type.

→ Allows the programmers to limit what types may be used in certain circumstances.
→ Computes values to all types.
→ Determines whether these values are used in an appropriate manner.
→ Simplest situation: Check types of objects and report a type error in case of a violation.
→ More Complex: incorrect types may be Corrected (type Coercing)



Static checking

(1) Type checking done at Compile time.
(2) Properties Can be verified before program run.
(3) Can Catch many Common errors.
(4) Desirable when faster execution is important.

Dynamic Checking

(1) Performed during program execution.
(2) Permits programmers to be less Concerned with types.
(3) Mandatory in some situations such as, array bounds check.
(4) More robust and cleaner code.

Construction of Basic block

Algorithm: Partition into basic block

Input:- A sequence of three address statements.

Output:- A list of basic blocks with each three-address statements in exactly one block.

Method:-

1. We first determine the set of leaders. Rules are:
   (a) The first statement is a leader.
   (b) Any statement which is a target statement of Conditional or unconditional statements is a leader.
   (c) Any statement which is immediately follows a Conditional 'goto' is a leader.
2. From one leader statement to the just prior statement of the next leader (if no such leader statement is present then upto the last statement) will form a basic block.

If there is any statement which has not been included to any basic block can be removed and discarded.

Types of 3 address Codes

1. $a = b \text{ op } c$
2. $a = \text{op } b$
3. $a = b$
4. if a rel op b goto L
5. goto L.
6. (i) $A[i] = a$
   (ii) $b = A[j]$
   (iii) $a = *p$
   (iv) $b = \&c$
   (v) $*c = a$

(1)
```
x = 1;
if (x > 1)
    y = y + 3
```

1. x = 1
2. if x <= 1 goto 5
3. t = y + 3
4. y = t.

(2)
```
x = 5
if (x < 5)
    z = z + 2
else
    y = y * 2
```

1. x = 5
2. if x >= 5 goto 6
3. $t_1 = z + 2$
4. $z = t_1$
5. goto 8
6. $t_2 = y * 2$
7. $y = t_2$
8.

(3)
1. x = 1
2. $t_1 = x + 1$
3. x = t
4. if x <= 5 goto 2.
5.

(4)
```
x = 1
while (x <= 10)
{   y = y * 2
    x = x + 1
```

1. x = 1
2. if x > 10 go 8
3. $t_1 = y * 2$
4. $y = t_1$
5. $t_2 = x + 1$
6. $x = t_2$
7. goto 2
8.

(5)
```
Switch (i+j)
{
  Case 1:
      x = y * z;
      break;
  Case 2:
      z = 8/2;
      break;
  default:
      l = l * 5;
}
```

1. t = i + j
2. if t != 1 goto 6
3. $t_1 = y * 8$
4. x = t_1
5. goto 12
6. if t != 2 goto 10
7. $t_2 = 8/2$
8. z = t_2
9. goto 12
10. $t_3 = l * 5$
11. l = t_3
12.

(6)
```
x = 1
do {
    x = x + 1
} while (x <= 5);
```

Syntax Directed Translation

→ Attach rules (or) program fragments to productions in a grammar.

→ **Syntax Directed Definition (SDD):**

→ SDD specifies the values of attributes by associating semantic rules with the grammar productions.

Eg: $E_1 \rightarrow E_2 + T \quad \{ E_1.Code = E_2.Code \,||\, T.Code \,||\, '+' \}$

→ **Syntax Directed Translation (SDT):**

→ SDT embeds program fragments (also called as semantic actions) within production bodies.

→ The position of the action defines the order in which the action is executed.

Eg: $E \rightarrow E+T \quad \{ print '+' \}$

→ SDD is easier to read & specify.

→ SDT is more efficient & easy for implementation.

Eg: **SDD Vs SDT scheme** [Infix to postfix Conversion]

SDT Scheme

$E \rightarrow E+T \quad \{ print '+' \}$

$E \rightarrow E-T \quad \{ print '-' \}$

$E \rightarrow T$

$T \rightarrow 0 \quad \{ print '0' \}$

$T \rightarrow 1 \quad \{ print '1' \}$

⋮

$T \rightarrow 9 \quad \{ print '9' \}$

SDD

$E \rightarrow E+T \quad \{ E.Code = E.Code \,||\, T.Code \,||\, '+' \}$

$E \rightarrow E-T \quad \{ E.Code = E.Code \,||\, T.Code \,||\, '-' \}$

$E \rightarrow T \quad \{ E.Code = T.Code \}$

$T \rightarrow 0 \quad \{ T.Code = '0' \}$

$T \rightarrow 1 \quad \{ T.Code = '1' \}$

⋮

$T \rightarrow 9 \quad \{ T.Code = '9' \}$

Syntax Directed Definitions (SDD)

→ SDD is a Context free grammar together with attributes and semantic rules.

→ Attributes are associated with grammar symbols and rules are associated with productions.

→ Attributes may be of kind: numbers, types, table reference, string etc.

→ Attributes are of two types Inherited and Synthesized attributes.

Synthesized attribute:-

A Synthesized attribute at node 'N', is defined only in terms of attributes values at the children of 'N' and at 'N' itself.

→ For a nonterminal 'A' at a parse tree node 'N' is defined by a semantic rule associated with the production at 'N'.

→ Note that the production must have 'A' as its head.

Inherited attribute:-

An inherited attribute at node 'N' is defined only interms of attribute values at N's parent, N's siblings and 'N' itself.

→ For nonterminal 'B' at a parse tree node 'N' is defined by semantic rule associated with the production at the parent of 'N'.

→ Note that the production must have B as a symbol in its body.

Types of SDD's (or) SDT's

→ SDD's are classified into two types based on type of attributes the SDD includes.

(a) S - attributed SDD (b) L - attributed SDD.

S - attributed SDD

→ An SDD is S - attributed, if every attribute is Synthesized.

L - attributed SDD

→ An SDD is L - attributed, if it uses both Synthesized and inherited attributes with restriction of not taking values from righ siblings.

→

S - attributed SDT
[Postfix SDT]

(1) Uses only Synthesized attributes.

(2) Semantic actions are placed at right end of production.

(3) Attributes are evaluated during bottomup parsing.

(4) Computing Synthesized attributes is simple

L - attributed SDT

(1) Uses both inherited and Synthesized attributes. Each inherited attribute is restricted to inherit either from parent (or) from left sibling only.

(2) Semantic rules can be placed any where at the righ side of a production.

(3) Attributes are evaluated by traversing parse tree depth first, left to right.

(4) Computing inherited attributes is complex.

Dependency Graphs

→ A dependency graph depicts the flow of information among the attribute instances in a particular parse tree.

→ An edge from one attribute instance to another means that the value of first is needed to compute the second.

→ "Dependency graphs" are a useful tool for determining ~~the~~ an evaluation order for the attribute instances in a given parse tree.

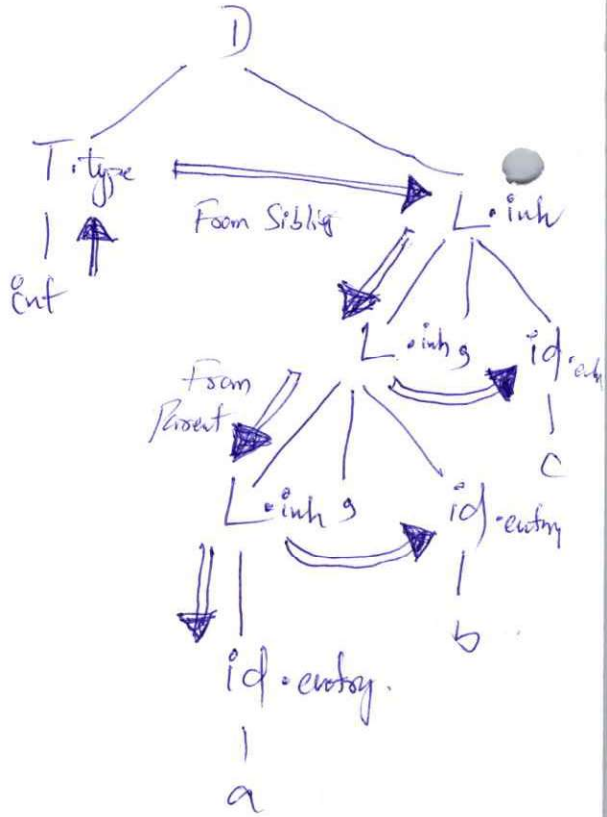
→ The topological sort of the dependency graph decides the evaluation order in a parse tree.

Eg:-

SDT

D → TL	{ L.inh = T.type }
T → int	{ T.type = integer }
T → float	{ T.type = float }
L → L1 , id	{ L1.inh = L.inh } { add(id.entry, L.inh) }
L → id	{ add(id.entry, L.inh) }

Dependency graph for a string int a, b, c is

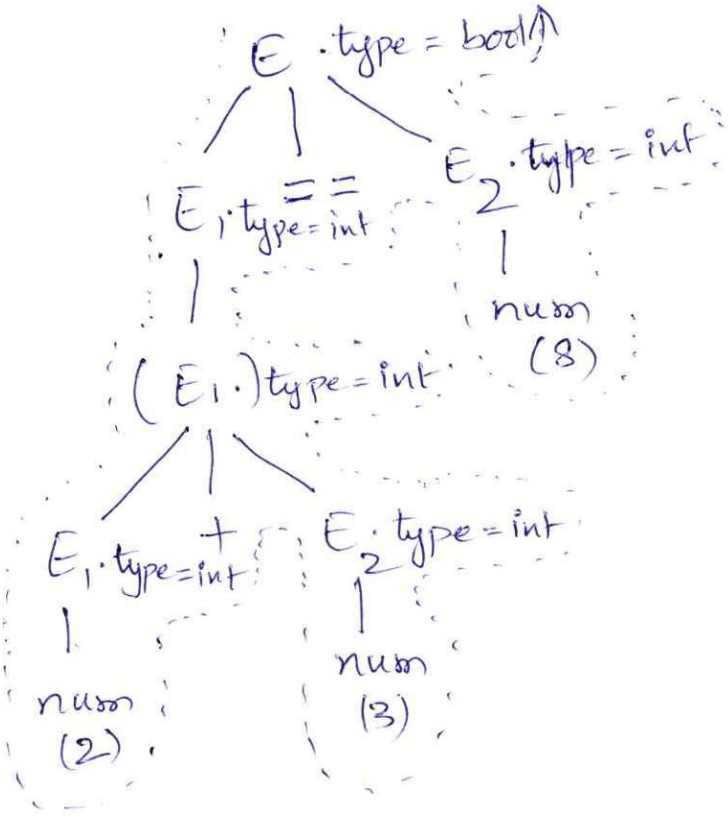


650000056

Eg:- SDD for type checking

Grammer Productions	Semantic Rules
$E \rightarrow E_1 + E_2$	{if (E_1.type == E_2.type) && (E_1.type == int) then E.type = int else error;}
$\mid E_1 == E_2$	{if (E_1.type == E_2.type) && (E_1.type == int | boolean)) then E.type = boolean else error;}
$\mid (E_1)$	{E.type = E_1.type;}
| num	{E.type = int;}
| true	{E.type = bool;}
| false	{E.type = bool;}

→ for a string (2+3) == 8, type checking will be done as follows.



Applications of SDT

Construction of Syntax Tree

SDD

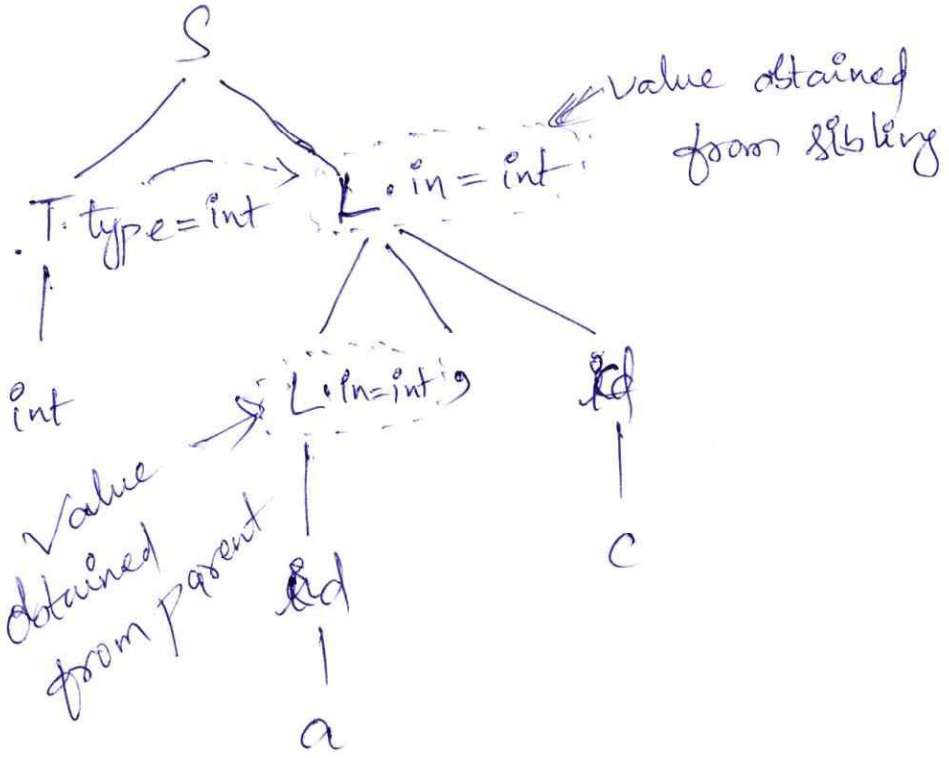
$E \rightarrow E_1 + T$	$\{E.node = new\ Node('+', E_1.node, T.node)\}$
$E \rightarrow E_1 - T$	$\{E.node = new\ Node('-', E_1.node, T.node)\}$
$E \rightarrow T$	$\{E.node = T.node\}$
$T \rightarrow (E)$	$\{T.node = E.node\}$
$T \rightarrow id$	$\{T.node = new\ Leaf(id, id.entry)\}$
$T \rightarrow num$	$\{T.node = new\ Leaf(num, num.val)\}$

→ Syntax tree for a string $a-4+c$.

Eg:- SDD for Inherited attributes

Grammar	Semantic rules
$S \rightarrow T\ L$	$\{ L.in = T.type \}$
$T \rightarrow int$	$\{ T.type = int \}$
$T \rightarrow float$	$\{ T.type = float \}$
$T \rightarrow double$	$\{ T.type = double \}$
$L \rightarrow L_1, id$	$\{ L_1.in = L.in$ $Enter\_type(id.entry, L.in) \}$
$L \rightarrow id$	$\{ Entry\_type(id.entry, L.in) \}$

→ Parse tree for the string int a, c is

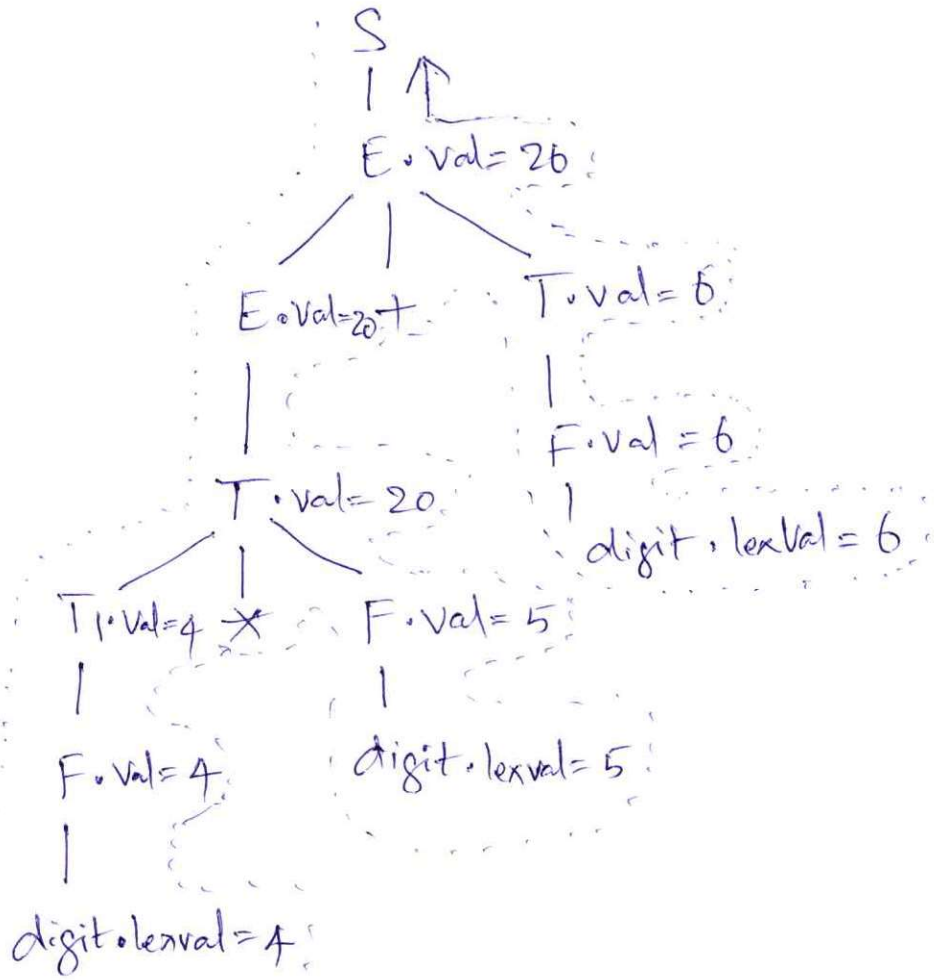


Note:-
→ Inherited attribute values are computed when the node visited during traversing of a parse tree.

Eg:- SDD for Synthesized Attributes

Grammar	Semantic rules
$S \rightarrow E$	{ Print (E.val) }
$E \rightarrow E_1 + T$	{ E.val = E_1.val + T.val }
$E \rightarrow T$	{ E.val = T.val }
$T \rightarrow T_1 * F$	{ T.val = T_1.val * F.val }
$T \rightarrow F$	{ T.val = F.val }
$F \rightarrow$ digit	{ F.val = digit.lexval }

→ Parse tree for String 4 * 5 + 6



Note:-
→ Synthesized attributes are computed during reduction by using Corresponding Semantic rules of that production.

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 have well-defined domain values, such as integer, float, character, string etc.

→ Attribute grammar is a medium to provide semantics to the Context free grammar.

→ Attribute grammar (when viewed as parse tree) can pass values or information among the nodes of a tree.

Eg: $E \rightarrow E_1 + T \quad \{ E.value = E_1.value + T.value \}$

→ The right part of the CFG contains the semantic rule that specify how grammar should interpreted.

→ Here the values of non terminals E_1 and T are added together and result is copied into the non-terminal E.

→ Based on the way the attributes get their values, they can be broadly divided into two categories

(a) Synthesized attributes.

(b) Inherited attributes.

Annotated Parse Tree

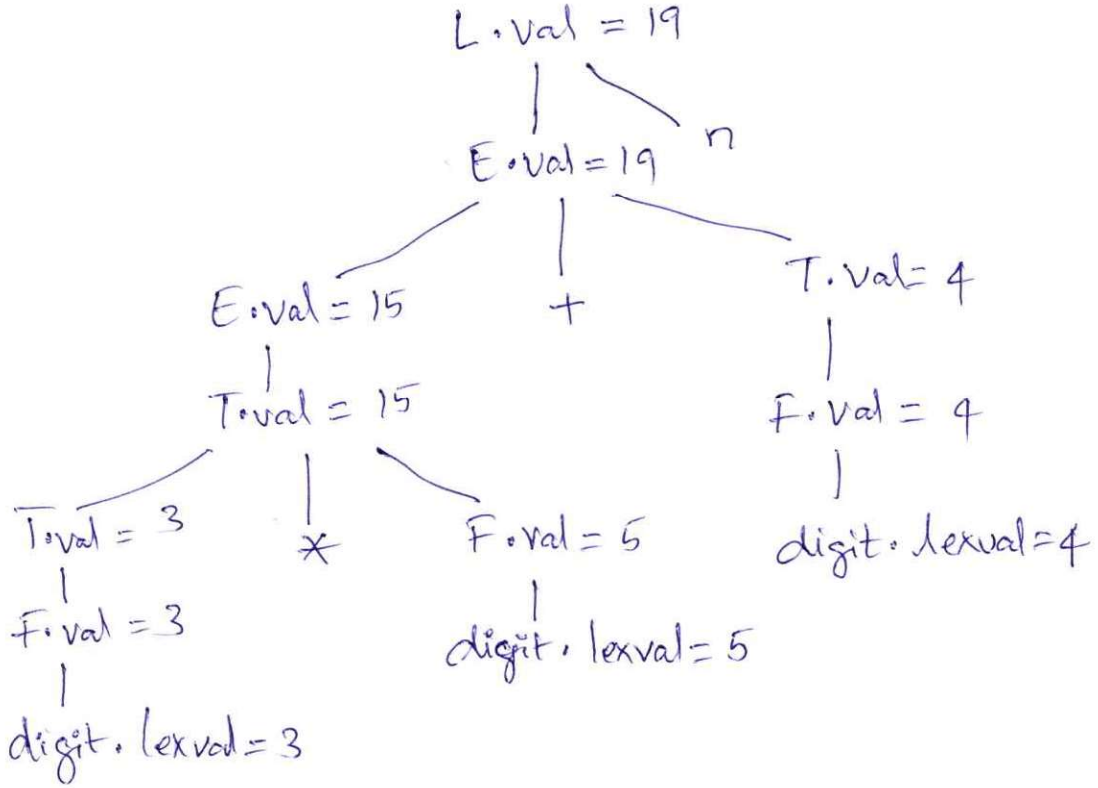
→ A parse tree, showing the values of its attributes is called an annotated parse tree.

→ The process of computing ~~values~~ the attribute values at the nodes is called annotating or decorating the parse tree.

Example :-

Productions	Semantic Rules
$L \to E\,n$	$\{L.val = E.val\}$
$E \to E_1 + T$	$\{E.val = E_1.val + T.val\}$
$E \to T$	$\{E.val = T.val\}$
$T \to T_1 * F$	$\{T.val = T_1.val * F.val\}$
$T \to F$	$\{T.val = F.val\}$
$F \to (E)$	$\{F.val = E.val\}$
$F \to digit$	$\{F.val = digit.lexval\}$

→ Annotated parse for the string $3*5+4n$ is as follows.



Abstract Syntax Tree (AST)

→ AST is a tree that represents the abstract syntactic structure of a language construct, where each interior and root node represents an operator and leaf nodes represent operands.

→ Syntax trees are abstract (or) Compact representation of parse trees. Syntax trees are also called as ASTs.

Parse Trees Vs Syntax Trees

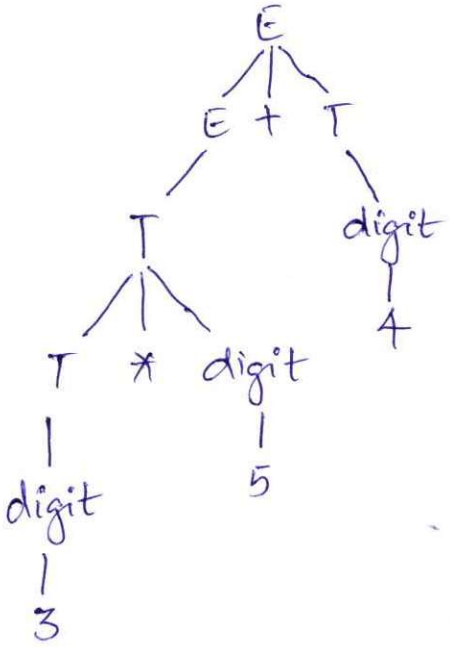
Parse Tree	Syntax Tree
(1) Parse tree is a graphical representation of the replacement process in a derivation.	(1) Syntax tree is Compact form of parse tree.
(2) Each interior node represents a grammar rule. Each leaf node represents a terminal.	(2) Each interior node represents an operator. Each leaf node represents an operand.
(3) Parse trees provide every Characteristic information from the real syntax.	(3) Syntax trees do not provide every characteristic information from the real syntax.
(4) Parse trees are Comparatively less dense than Syntax trees.	(4) Syntax trees are Comparatively more dense than parse trees.

Note:-

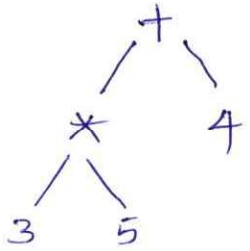
* Syntax trees are Called as AST because, they are abstract representation of the parse trees.
* They do not provide every characteristic information from the real Syntax.
  For eg:- No rules nodes, no parenthesis etc.

Eg:- ① For expression 3 * 5 + 4

Parse Tree

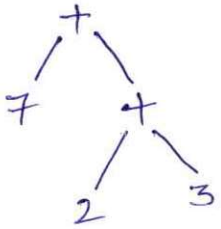


Syntax Tree

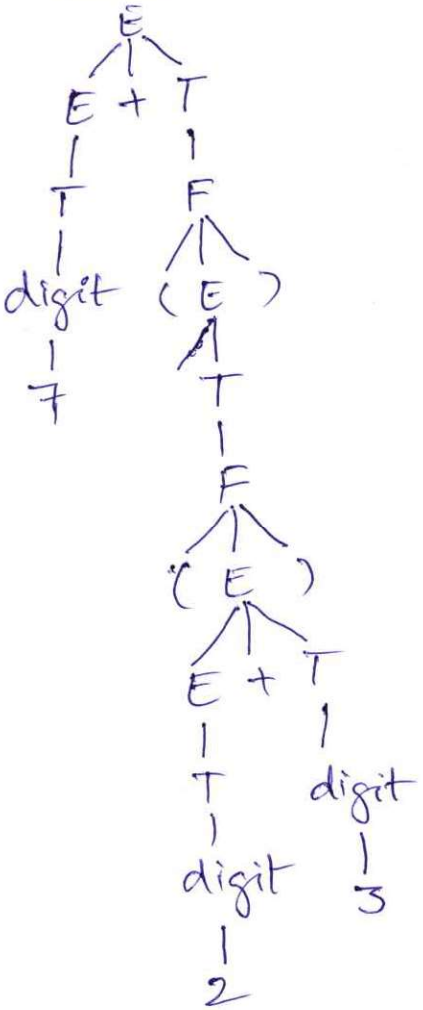


② for expression 7 + ((2+3))

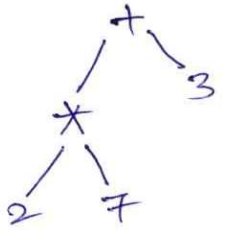
AST

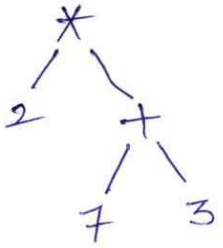


Pass Tree

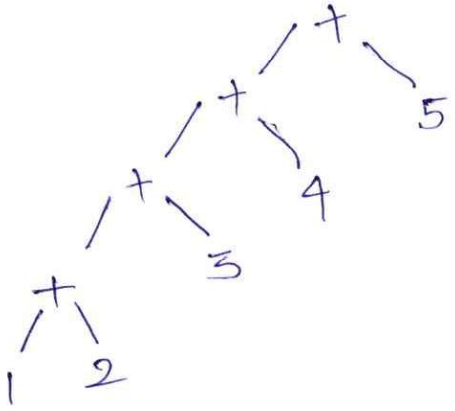


③ Expressions $2*7+3$, $2*(7+3)$



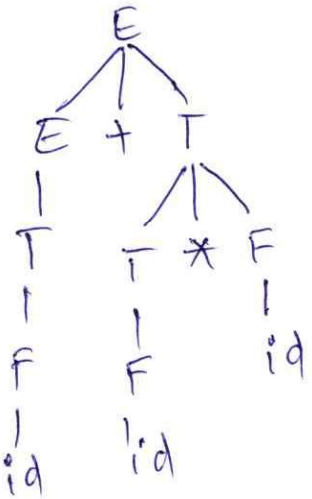


④ $1+2+3+4+5$

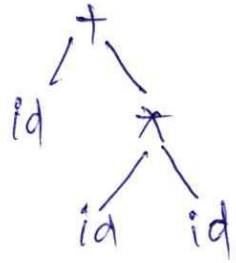


⑤ $id+id*id$

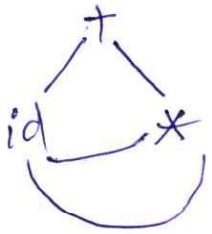
Parse Tree



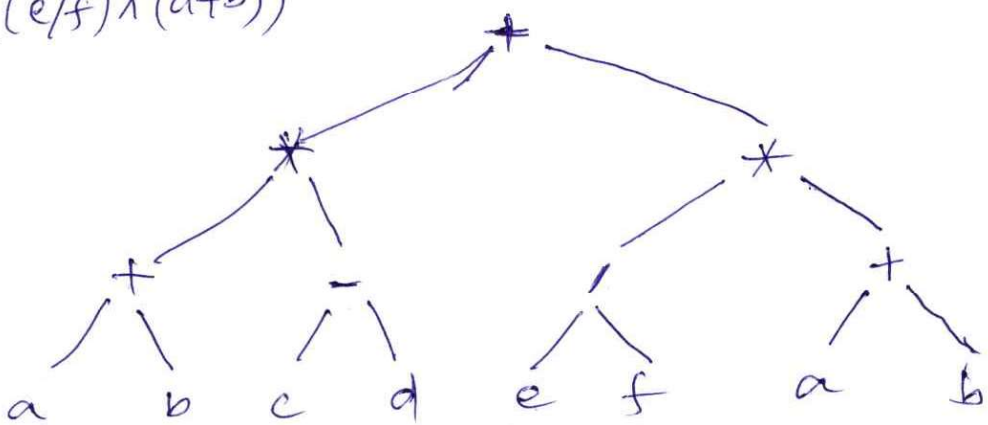
Syntax Tree



DAG



⑥ $(a+b)*(c-d)+((e/f)*(a+b))$



Polish Notation (Prefix Notation)

→ Polish notation refers to a notation in which the operator is placed before its two operands.

→ Polish notation is used to express arithmetic, logic and algebraic expressions in parenthesis free notation that makes each equation shorter and easier to parse instead of defining the evaluation priority of operators.

- Polish notations can be easily (or) readily parsed into an abstract syntax tree and stored in a stack.

Examples

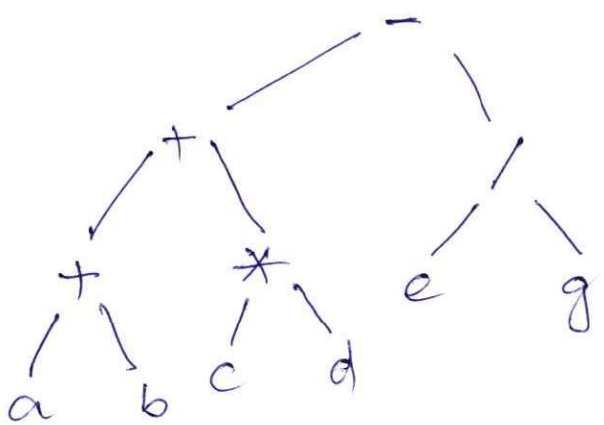
(1) Polish notation for $(3+2)*(5-1)$ is

Sol:- $*+32-51$

(2) Prefix notation for $a+b+c*d-e/g$ is

Sol:-

Syntax Tree



Prefix Notation

$-++ab*cd/eg$

Postfix Notation

→ Postfix notation is one of the useful form of intermediate code, if the given language is expressions.

→ Postfix notation is also called "suffix notation" and "reverse polish" notation.

→ Postfix notation is a linear representation of Syntax tree.

→ The postfix notation for an expression E can be defined inductively as follows.

(1) If E is a variable or constant, then the postfix notation for E is E itself.

(2) If E is an expression of the form E_1 op E_2, where op is any binary operator, then the postfix notation for E is $E_1' E_2'$ op, where E_1' and E_2' are the postfix notations for E_1 and E_2 respectively.

(3) If E is a parenthesized expression of the form (E_1), then the postfix notation for E is the same as the postfix notation for E_1.

Eg:- (1) (A+B*C) / (D*E - (F+G)/(H+I))

ABC* + DE* FG+ HI+ /-/

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

ab- cd+ ab- *+ .

Three Address Code

→ Three address code an intermediate code, which is easy to generate and can be easily converted to machine code.

→ Each three address code instruction has at most three operands and one operator on the right hand side of an instruction.

Advantages of Intermediate Code

(1) A Compiler for different machines can be created by attaching different backend to the existing front ends of each machine.

(2) A Compiler for different source languages (on the same machine) can be created by providing different front ends for corresponding source languages to existing back end.

(3) A machine independent code optimizer can be applied to intermediate code inorder to optimize the code generation.

→ Three address code will help in Code optimization.

General form of three address code

$$a = b \text{ op } c$$

where

→ a, b, c are the operands that can be names, constants or Compiler generated temporary variables.

→ 'op' represents operator.

Eg:-

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

Common Three address Instruction Forms

(1) Assignment Statement :

$$x = y \text{ op } z \quad \text{and} \quad x = \text{op } y$$

where x, y, z are operands.

'op' represents operator.

→ It assigns the result obtained after solving the right side expression of the assignment operator to the left side operand.

(2) Copy Statement :

$$x = y$$

where x and y are operands.

= is an assignment operator

→ It copy and assigns the value of operand 'y' to operand 'x'.

(3) Conditional Jump :

$$\text{if } x \text{ relop } y \text{ goto } z$$

where x, y are operands.

z is the tag (or) Label of the target statement.

'relop' is a relational operator.

'if' is a keyword.

→ If Condition " x relop y " gets satisfied then the control will be transfered to the location specified by Label 'z'. Otherwise the next statement appearing in usual sequence is executed.

(4) Unconditional Jump :

$$\text{goto } x$$

where 'x' is the tag or label of the target statement.

→ On execution of the statement the control is sent to the location specified by label 'x'. All statements in b/w are skipped.

(5) Procedure Call :-

param x Call P return y

where 'P' is a function which takes 'x' as parameter and returns 'y'.

(6) Indexed assignments :-

x = y [i] or x [i] = y

where 'x' and 'y' are variables.
'i' is an indexed variable.

→ The instruction x = y[i] sets 'x' to the value in the location 'i' memory units beyond location 'y'.
The instruction x [i] = y sets the contents of location 'i' units beyond 'x' to the value 'y'.

(7) Address and pointer assignments :-

x = &y , x = *y and *x = y

→ where 'x' and 'y' are variables.

→ The instruction x = &y sets r-value of 'x' to the location (l-value) of y. The instruction x = *y, sets r-value of 'x' to r-value of an object pointed by 'y'.

→ The instruction *x = *y sets r-value of object pointed by 'x' to the r-value of 'y'.

Problems on Three address Code (TAC)

① Write three address code for $a+b*c$.

Sol:-

$t_1 = b*c$

$t_2 = a+t_1$

② Write TAC for $-(a*b)+(c+d)-(a+b+c+d)$

Sol:-

$t_1 = a*b$

$t_2 = -t_1$

$t_3 = c+d$

$t_4 = a+b$

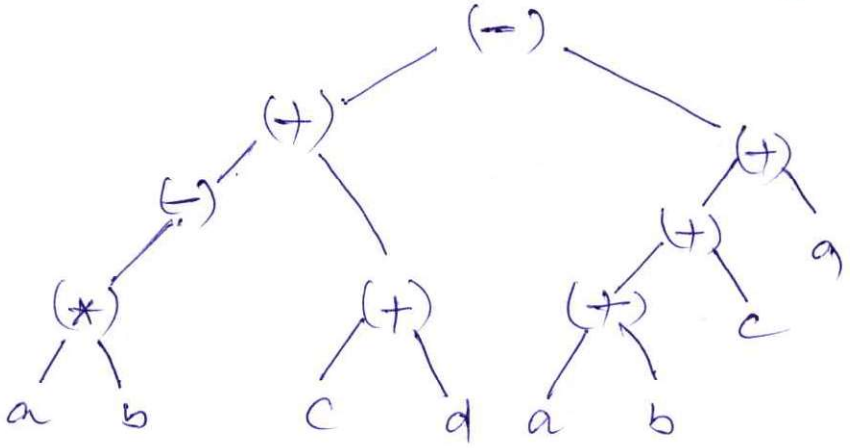
$t_5 = t_4+c$

$t_6 = t_5+d$

$t_7 = t_2+t_3$

$t_8 = t_7-t_6$

Syntax tree



③ Writ TAC for $a=(-c*b)+(-c*d)$.

Sol:-

$t_1 = -c$

$t_2 = t_1*b$

$t_3 = -c$

$t_4 = t_3*d$

$t_5 = t_2+t_4$

$a = t_5$

④ Write TAC for if A < B then 1 else 0

Sol:-

1. if A < B goto (4)
2. T1 = 0
3. goto (5)
4. T1 = 1
5.

⑤ Write TAC for "if A < B and C < D then t = 1 else t = 0"

Sol:-

1. if A < B goto (3)
2. goto (4)
3. if C < D goto (6)
4. t = 0
5. goto 7.
6. t = 1
7.

⑥ Write TAC for C = 0 do { if (a < b) then x++ else x--; C++ } while (C < 5)

Sol:-

1. C = 0
2. if (a < b) goto (4)
3. goto (7)
4. T1 = x + 1
5. x = T1
6. goto (9)
7. T2 = x - 1
8. x = T2
9. T3 = C + 1
10. C = T3
11. if (C < 5) goto (2)
12.

Implementation of Three address Code

→ Three address code is implemented in three ways, namely.

1. Quadruple
2. Triples
3. Indirect Triples.

1. Quadruple :-

In Quadruple representation, each TAC instruction is splitted into the following 4 fields.

OP, arg1, arg2, result.

Here :

→ 'OP' field is used for storing the internal code of the operator.

→ 'arg1', 'arg2' fields are used for storing two operands used.

→ 'result' field is used for storing the result of the expression.

Advantages :-

→ Easy to rearrange code for global optimization.

→ One can quickly access value of temporary values using Symbol table.

Disadvantages :-

→ Contain lot of temporaries.

→ Temporary Variable Creation increases time and space Complexity.

Eg:- Quadruple representation for the expression $a = b * -c + b * -c$

Sol:-

Three address Code

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

Quadruple representation

#	op	arg1	arg2	result
0	Uminus	c		t_1
1	*	b	t_1	t_2
2	Uminus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

(2) Triples:-

→ Triple representation does not make use of extra temporary variable to represent single operation, instead a pointer is used to refer another triple.

→ Triple uses only three fields namely

op, arg 1, arg 2.

Advantage:-

→ It saves memory by eliminating temporary variables.

Disadvantages:-

→ Difficult to rearrange the code.

→ Difficult to optimize the code.

Eg: Triple representation for $a = b * -c + b * -c$ is

Sol:

#	op	Arg1	Arg2
0	Uminus	c	
1	*	b	(0)
2	Uminus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

(3) Indirect Triple:

→ Indirect Triple representation makes use of pointer to the listing of all references to computations which is made separately and stored.

→ Temporaries are implicit and easier to rearrange code.

Eg: Indirect triple representation for $a = b * -c + b * -c$ is

#	op	arg1	arg 2
10	Uminus	c	
11	*	b	(10)
12	Uminus	c	
13	*	b	(12)
14	+	(11)	(13)
15	=	a	(14)

List of pointers to table

#	Pointers
0	(10)
1	(11)
2	(12)
3	(13)
4	(14)
5	(15)

Directed Acyclic Graph (DAG)

→ DAG is an abstract Syntax tree, with a unique node for each value.

→ DAG is a directed graph that Contains no cycles, but some nodes may have more than one parent.

Applications of DAG

(1) DAG is used in determining the Common Subexpressions.

(2) DAG is used in determining which names are used in the block and Computed outside the block.

(3) It is used for finding which statements of the block could have their Computed value outside the block.

(4) In simplifying the list of quadruples by eliminating the Common Subexpressions and not performing the assignment of the form $x = y$ unless it is must.

Rules for Construction of DAG

Rule 1: In a DAG leaf nodes represent identifiers, names or Constants. Interior nodes represent operators.

Rule 2: Before creating a node in DAG, there is a check made to find, Is is there any existing node with same childen. A new node is created only when such node does not exists.

Rule 3: The assignment of the form $x = y$ must not be performed until unless it is Compulsory.

Problems on DAG Construction

① Construct DAG for the given expression $(a+b) \times (a+b+c)$

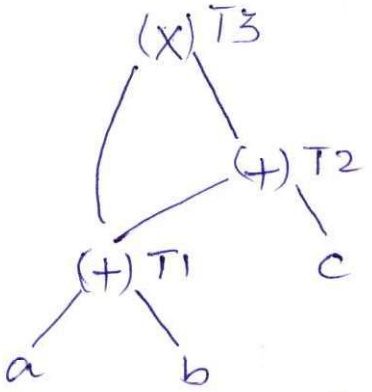
Sol:- Three address code

$T1 = a+b$

$T2 = T1+c$

$T3 = T1 \times T2$

Now DAG is



② Construct DAG for $(((a+a)+(a+a))+((a+a)+(a+a)))$

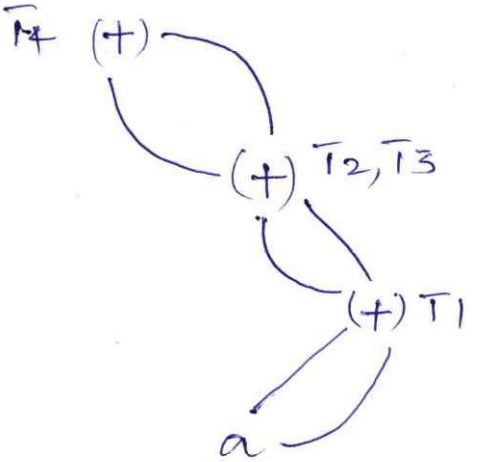
Sol:- Three address code

$T1 = a+a$

$T2 = T1+T1$

$T3 = T1+T1$

$T4 = T2+T3$



DAG for the given expression.

③ Consides the following block and construct a DAG for it.

(1) $a = b \times c$

(2) $d = b$

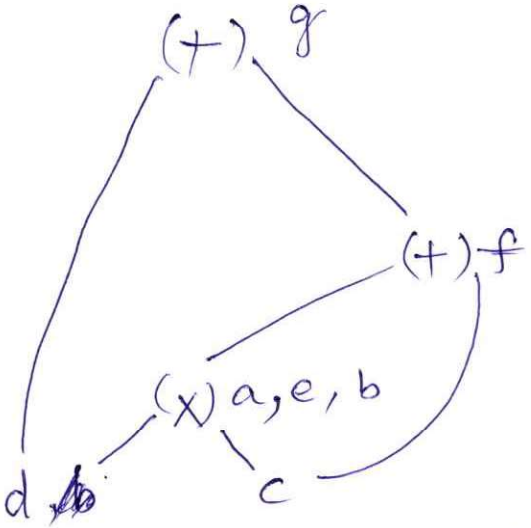
(3) $e = d \times c$

(4) $b = e$

(5) $f = b + c$

(6) $g = f + d$

Sol:- DAG for the given block



④ Construct DAG for the given block.

(1) $a = b + c$

(2) $t_1 = a \times a$

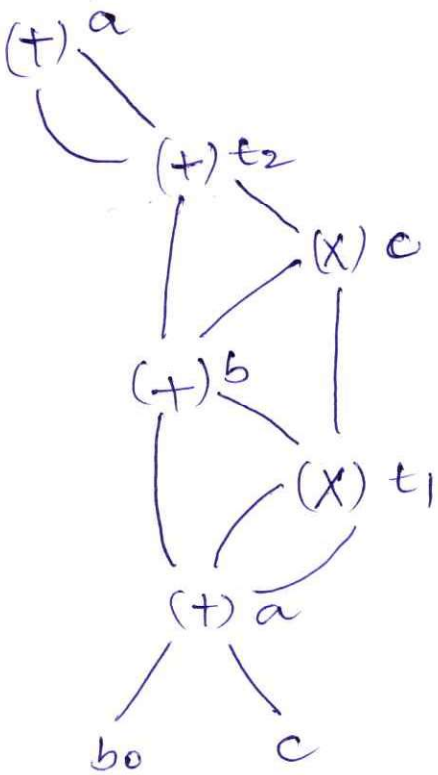
(3) $b = t_1 + a$

(4) $c = t_1 \times b$

(5) $t_2 = c + b$

(6) $a = t_2 + t_2$

Sol:-



(5) Construct DAG for the given block.

(1) $S1 = 4 * i$

(2) $S2 = a[S1]$

(3) $S3 = 4 * i$

(4) $S4 = b[S3]$

(5) $S5 = S2 * S4$

(6) $S6 = prod + S5$

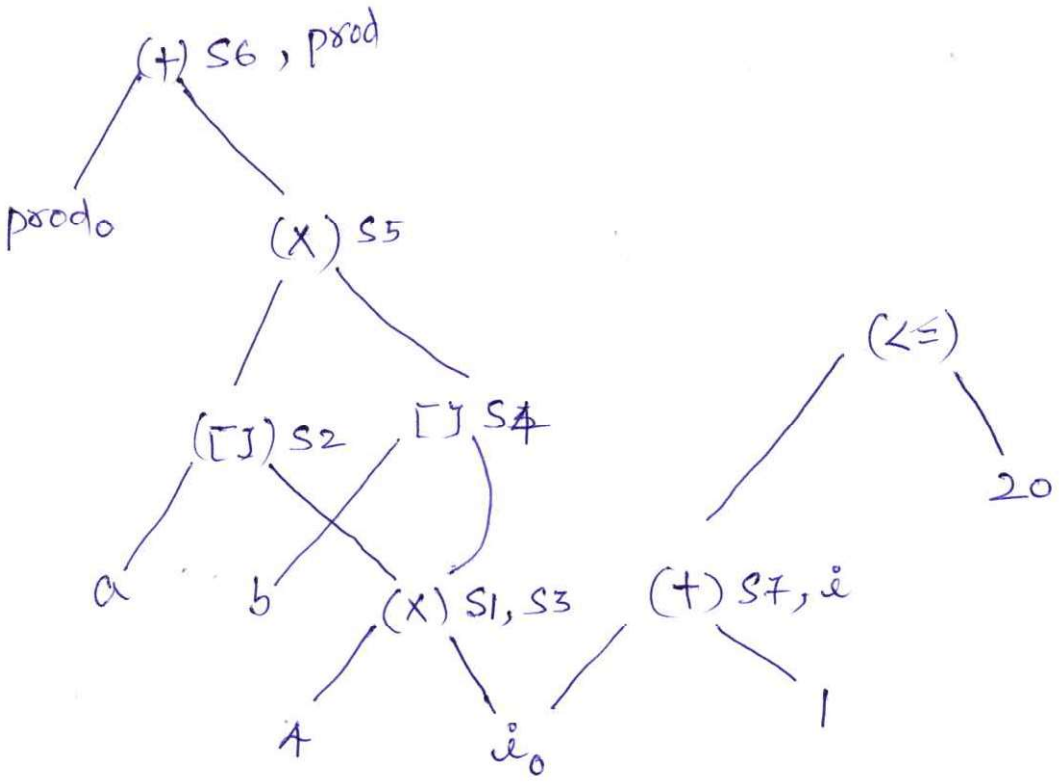
(7) $prod = S6$

(8) $S7 = i + 1$

(9) $i = S7$

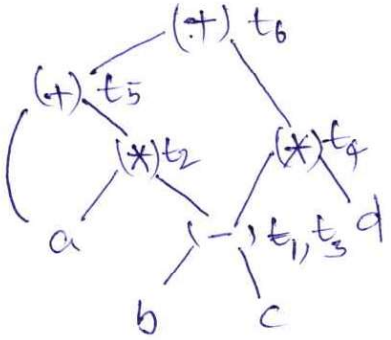
(10) if $i <= 20$ goto (1)

Sol:-



⑥ Construct DAG for $a + a * (b-c) + (b-c) * d$

Sol:-



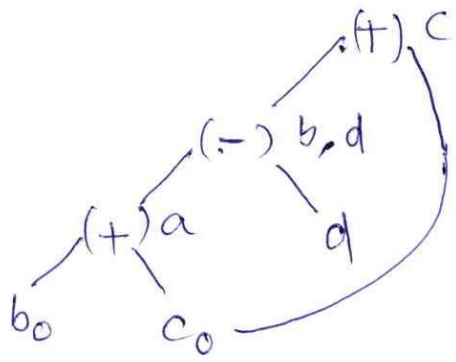
Three address code

$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = b - c$
$t_4 = t_3 * d$
$t_5 = a + t_2$
$t_6 = t_5 + t_4$

⑦ Construct DAG for

$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$

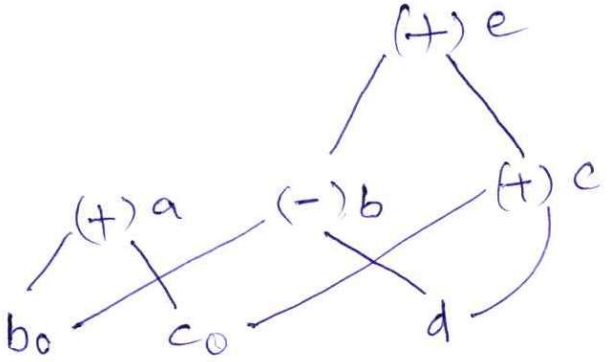
Sol:



⑧ Construct DAG for

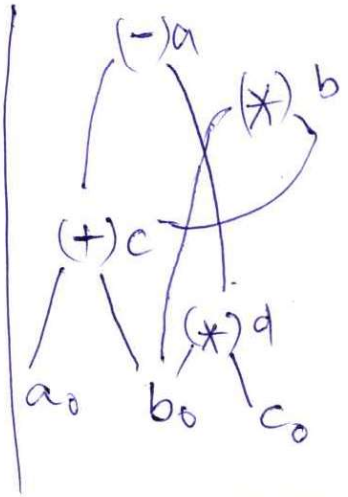
$a = b + c$
$b = b - d$
$c = c + d$
$e = b + c$

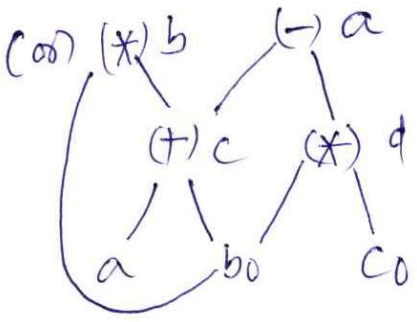
Sol:



⑨ Construct DAG for

$d = b * c$
$c = a + b$
$b = b * c$
$a = c - d$





Code generation — Unit - 5

→ It is a process of creating assembly language / machine language statements which will perform the operations specified by the source program.

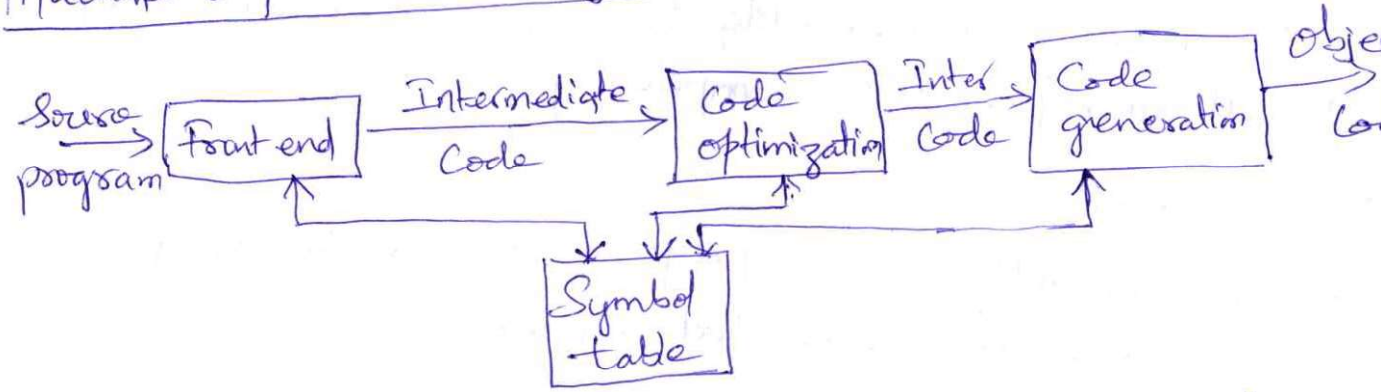
Properties of object code generation phase

① **Correctness**:- It should produce a high correct code, and do not alter the purpose of source code.

② **Efficient use of resources of the target machine**:- It is necessary to know the target machine on which it is going to run. It also should consider into account the resources available at target machine. Eg:- Memory, registers, special ckts designed to perform desired operations efficiently.

③ **Quick Code generation**:- It is necessary that code generation phase should produce the code quickly, with out taking much time.

Machine dependant Code generation:-



→ Code generator takes intermediate code as an i/p and generates target code as o/p.

Object Code forms

→ The object code that gets generated will be in one of the following forms.

① Absolute Code ② Relocatable machine code ③ Assembler code.

① Absolute Code:

→ It is a machine code that contains reference to actual memory addresses within program's address space.

→ The generated code can be placed directly in the memory and execute immediately.

→ Small programs can be compiled and executed quickly because of absolute code.

→ Disadvantage is if that memory space is not free, it has to wait.

(2) Relocatable Code:

→ It allows subprograms to be compiled separately.

→ A set of relocatable object modules can be linked together and loaded for execution with the help of 'linking loader'.

→ The advantage of generating relocatable m/c code is that we have great flexibility to compile subroutines separately and call other previously compiled programs.

(3) Assembler Code:

→ Producing assembly language program as o/p makes the process of code generation somewhat easier.

→ It uses symbolic instructions and use of macro facilities of the assembler to help in generation of code.

→ But it is slower because of assembling, linking and loading is required.

Issues in Code Generation

1. Input to the Code generator.
2. Target programs.
3. Memory Management.
4. Instruction Selection.
5. Register allocation.
6. Choice of ~~eqn~~ evaluation order
7. Approches to ~~Code~~ generation.

(1) → Takes 3 address Code as i/p.
→ Intermediate Code may be 3 address code, quadruple, postfix notation, (or) graphical rep such as Sytax tree (or) DAG etc.
→ front end should take Care of Syntax & Semantic ~~check~~ errors before submitting the i/p to the Code generator.
→ Code generation requires Complete error free intermediate Code as i/p.

(2) absolute, relocatable & assembly lang.

(3) → Mapping the names in the Source program to addresses to the data objects in run time memory.
→ type determines the amount of Storage needed.
→ By using Symbol table information, the Code generater determines the addresses in the target Code.

(4) → Uniformity and Completeness of instruction set is an important factor for the code generator

→ Selection of instruction depends upon the instruction set of target m/c.

eg: $x = a + b$.

```
Mov a, R0
ADD b, R0
Mov R0, x.
```

→ Line by line code generation leads to poor code ~~generation~~. ~~because~~ of redundancies.

$x = y + z$
$a = x + t$

```
Move y, R0 ✓
Add z, R0 ✓
Mov. R0, x
Mov x, R0
Add t, R0 ✓
Mov R0, a ✓.
```

(5) Registr allocation: Select appropriate set of variables that will reside in registers.

(2) Registr assignment: pick up the specific registr in which corresponding variable will reside

eg: $t_1 = a + b$
$t_1 = t_1 \times c$
$t_1 = t_1 / d$

```
Mov a, R0
Add b, R0
Mul c, R0
div d, R0
Mov R0, t1
```

---

absolute	M	M
registr	R	R
indexed	C(R)	C + Contents(R)
indirect registr	*R	Contents(R)
Indirect indexed	*C(R)	Contents(C + Contents(R))
literal	#C	C

Pass3

	in	out
1	x	m, x
5	m, x, z	x, y, z

Pass4

	in	out
3	m, x	m, x, z

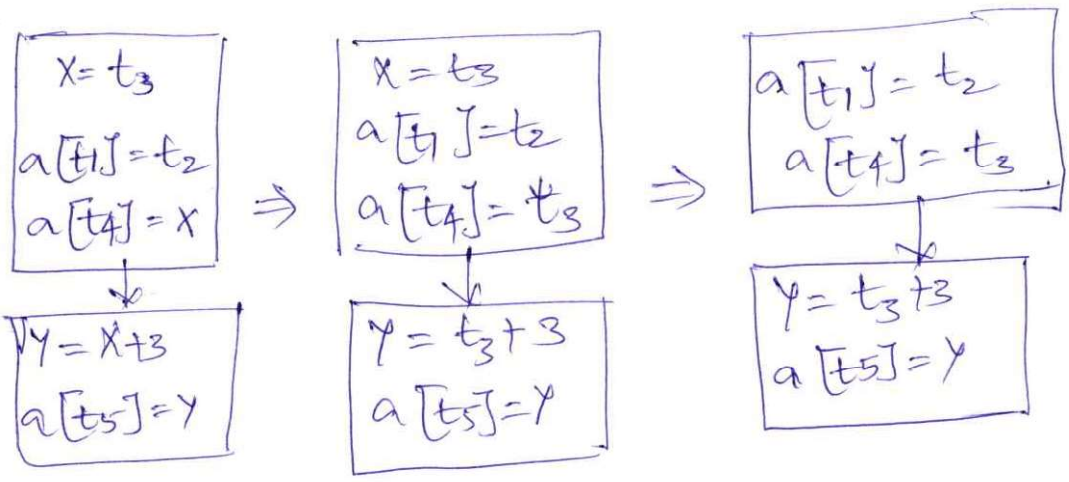
	in	out
1	{x}	{m, x}
2	{m, x}	{m, x}
3	{m, x}	{m, x, z}
4	{x}	∅
5	{m, x, z}	{x, y, z}
6	{x, y, z}	{x, y, z}
7	{m, x}	{x, z}
8	{x, y}	{m, x}

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

Mov R0, a
Add R0, b
Mov R1, c
Sub R1, d
Add

Mov t1, R1

Eg:-



Induction variable

→ A variable 'i' is called induction variable of loop, iff value of variable 'i' changes everytime either gets incremented (or) decremented.

Eg:- Generally induction variables are in the form.

$a = i * b$ $\quad a = i \pm b$ where 'b' is a ~~basic~~ Constant

$a = b * i$ $\quad a = b \pm i$ and 'i' is an induction variable

→ If b is a basic then a is in family of 'j' the 'a' depends on the definition of 'i'.

Algorithm:

1) Find the induction variable 'i' with triple (i, c, d). Consider a test form.

if i relop x goto B where 'i' is an induction variable and 'x' is not an induction variable.

$t = c * x$

$t = t + d$.

if j relop t goto B where t is a new temporary.

Finally delete all assignments to the eliminated induction variables from loop L, because these induction variables will be useless.

④ Graph Coloring for register assignment.

Register interface graph is Constructed.

→ nodes are Symbolic registers and edges Connects two nodes if one is live at a point where the other is defined.

Register descriptor : keeps track of what value is stored in each register.

address descriptor : keeps track of the location that holds the Current ~~variable~~ value of a variable.

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

$t_1 = a+b$, $t_2 = c-d$, $t_3 = e/f$, $t_4 = t_1 * t_2$, $t_5 = t_3 * t_1$,

$t_6 = t_4 + t_5$.

Mov a, R0, ~~Mov~~ Add b, R0, Mov c, R1, Sub d, R1, Move e, R2

div f, R2, Mul R0, R1, MUL R2, R0, Add R1, R0.

Generic code Algorithm (using pattern matching of Code generator Concept.

① Draw the structure for the Complete expression.

② traverse the tree in bottom up fasion and match the sub trees with the appropriate template from the tree translation scheme.

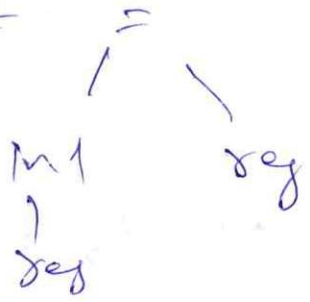
③ The Corresponding Code for the Matched template is stored in a buffer.

④ When we reach to the root node, buffer Contains the Complete m/c Code for the given expression.

Rule	Code
reg ← Const c	Move #c, Ri
regi ← Mem a	Move a, Ri
	Move Ri, a.

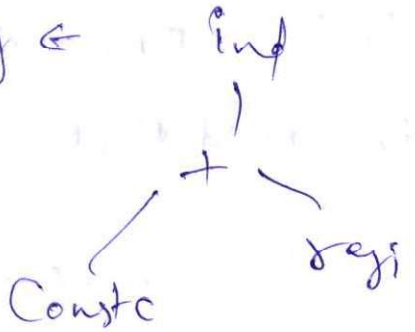
Mem ←

Mem ←



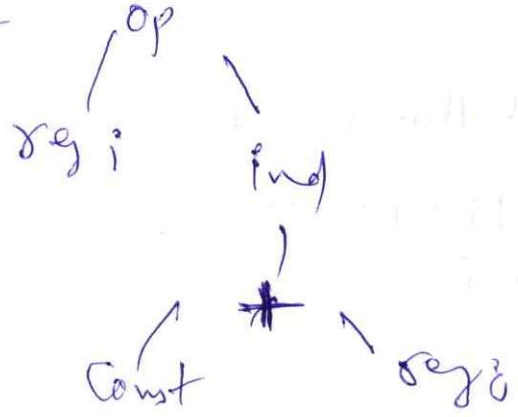
Mov Rj, Ri

reg ←



Mov c(Rj), Ri

reg ←



op c(Rj), Ri

Code generation Using DAG

→ It is much simpler than code generation from 3 add code.

→ 3-Types of algorithms are used to generate code from DAG.

1. Rearranging order
2. Heuristic Ordering
3. Labeling algorithm.

(1) Rearranging Order :-

→ The order of 3-add code affects the cost of the object code being generated.

→ In changing the order in which Computations are done we can obtain object code with min cost.

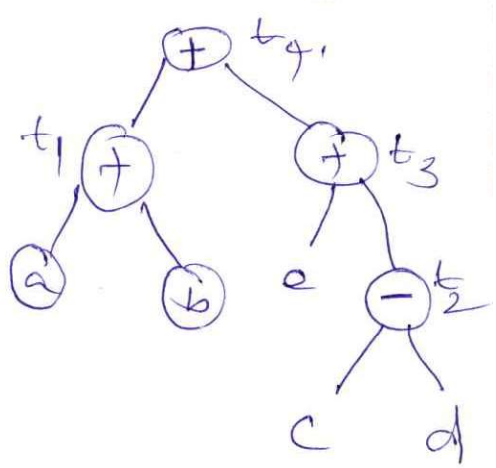
Eg:- $(a+b)+(e+(c-d))$

$t_1 = a+b$
$t_2 = c-d$
$t_3 = e+t_2$
$t_4 = t_1+t_3$

Mov a, R_0	Mov e, R_0
Add b, R_0	Add R_0, R_1
Mov c, R_1	Mov t_1, R_0
Sub d, R_1	Add R_1, R_0
Mov R_0, t_1	Mov R_0, t_4

$t_2 = c-d$
$t_3 = e+t_2$
$t_1 = a+b$
$t_4 = t_1+t_3$

Mov	c, R_0
Sub	d, R_0
~~Add~~ Mov	e, R_1
~~Add~~	R_0, R_1
Mov	a, R_0
~~Add~~	b, R_0
Add	R_1, R_0
Mov	R_0, t_4



(2) Heuristic Ordering

(1) Obtain all the interior nodes. Consider them as unlisted interior nodes.

(2) while (unlisted interior nodes remain).

(3) Pick up an unlisted node n, whose parents have been listed.

(4) list node 'n'.

(5) while the leftmost child m of n has no unlisted parent AND is not leaf.

{

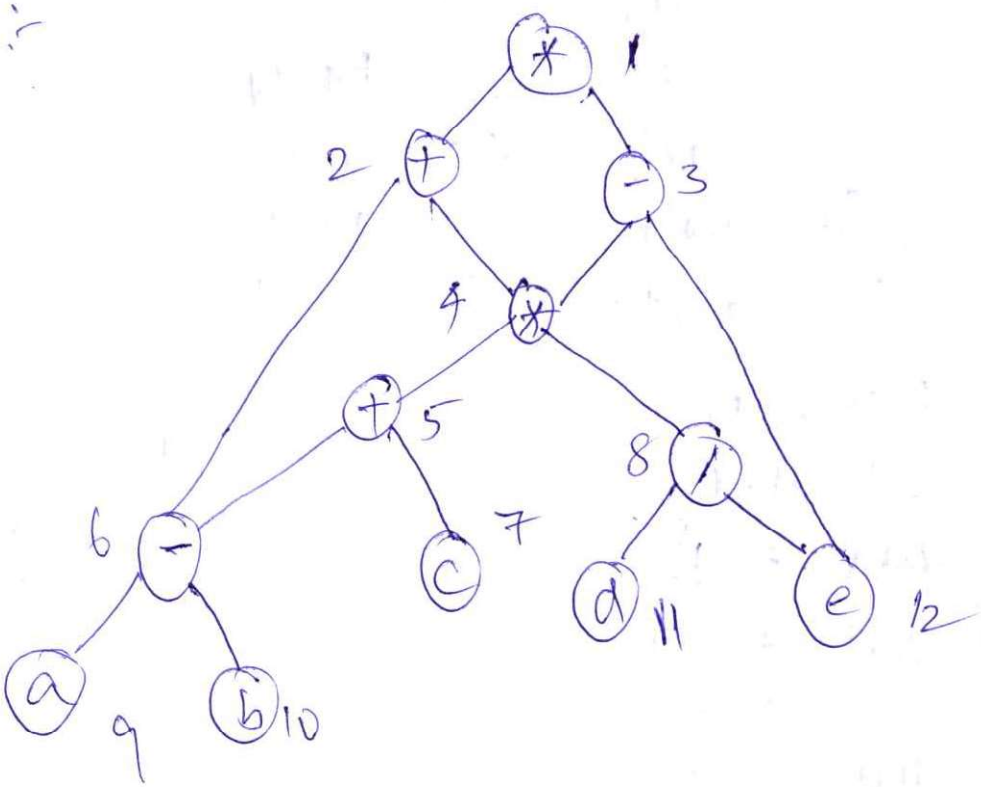
(6) list m;

(7) n = m;

}

}

Eg:-



→ Unlisted interior nodes 1, 2, 3, 4, 5, 6, 8

→ Initially only node with unlisted parent is 1

∴ Set n = 1 by line 14) which is 1

→ Now left of 1 is 2 which is unlisted and parent of 2 is listed

∴ list 2.

∴ Set n = 2

→ left of 2 is 6 which has unlististed parent 5, ∴ we cant list it, ∴ Swich to node 3 whose parent is 1 is listed

→ ∴ list 3 and set n = 3.

→ left of 3 is 4

∴ resultig list is

→ Hence 1, 2, 3, 4, 5, 6, 8.

→ Then the order of Computation is decided by reversig this list.

We get 8, 6, 5, 4, 3, 2, 1.

→ This also means that we have to perform the Computations at these nodes in the given order.

$t_8 = d/e$, $t_6 = a - b$, $t_5 = t_6 + c$, $t_4 = t_5 * t_8$, $t_3 = t_4 - e$,

$t_2 = t_6 + t_4$, $t_1 = t_2 * t_3$.

(30) Label → This gives optimized code for DAG

(3) Labeling algorithm :-

→ It gives the optimal Code for given expression in which min registers are required.

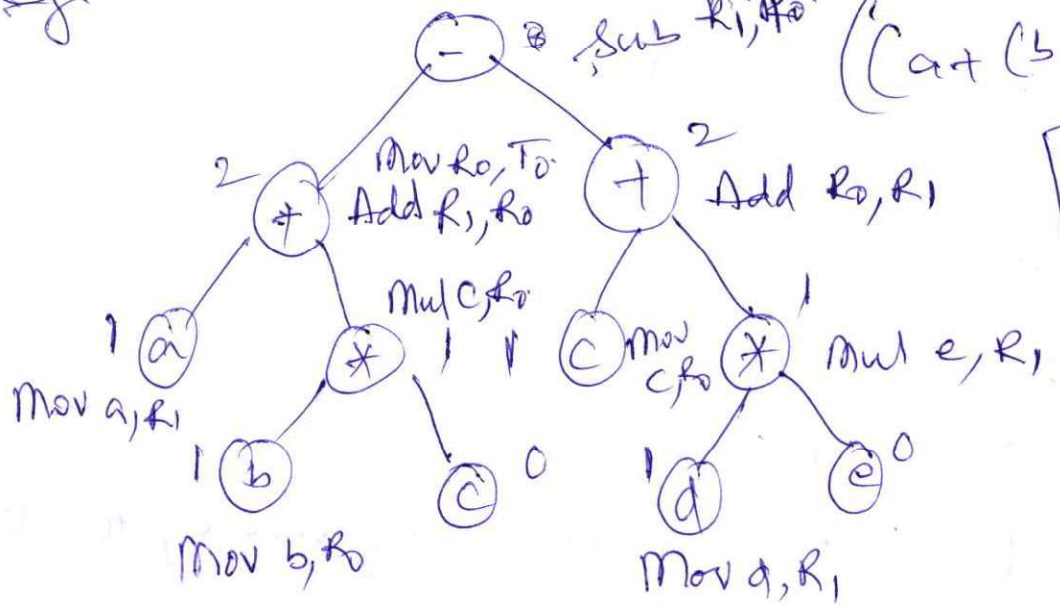
→ Using this algorithm labeling can be done to the tree by visiting nodes in bottomup order. By this all child nodes will be labelled before its parent nodes.

→ we start in bottom-up fashion and label left leaf as 1 and right leaf as 0.

→ If labels of the children of a node 'n' are L_1 & L_2 respectively then

$$\text{Label}(n) = \begin{cases} \max(L_1, L_2) & \text{if } L_1 \neq L_2 \\ L_1 + 1 & \text{if } L_1 = L_2. \end{cases}$$

Eg:



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

```
Mov b, R0
Mul c, R0
Add R1, R0
Mov R0, T0
Mov d, R1
Mul e, R1
Mov c, R0
Add R0, R1
Sub R1, R0
```

Features of DAG

→ Useful data structure for analysing Basic block.

→ It is directed with no cycle.

→ Depicts how values are calculated and used in next subsequent statements.

→ Determine Common subexpressions in basic block.

→ Determines which names are listed inside but evaluated outside of the block.

→ Determines which statements of the block could have their values outside the block.

Algorithm for Constructing a DAG

Input :- A basic block

Output :- A DAG with following information.

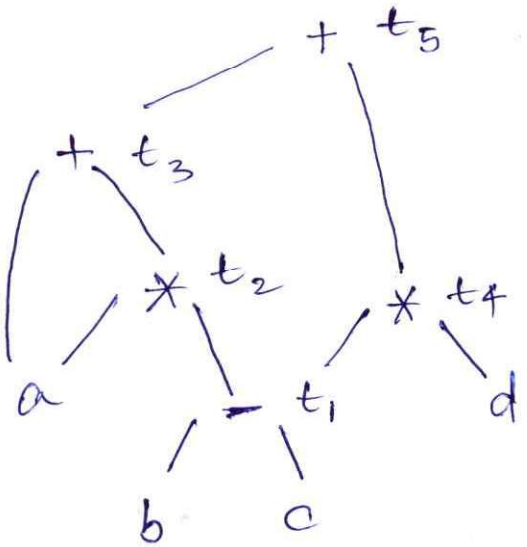
→ A label for each node for leaves the label is an identifier (Constants permitted) and for interior nodes an operator symbol.

→ For each node a (possibly empty) list of attached identifiers (Constants are not permitted here).

Method :- We assume availability of data structure to create nodes with one or two children with left and right nodes labelling should also available facility to create linked list of attached identifiers for each node.

Directed Acyclic Graph (DAG)

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



$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

(2) $((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$

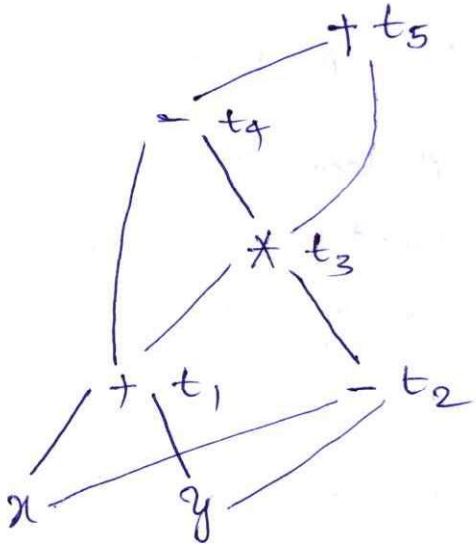
$t_1 = x + y$

$t_2 = x - y$

$t_3 = t_1 * t_2$

$t_4 = t_1 - t_3$

$t_5 = t_4 + t_3$



$S \to id = E$ ~~S.code = E.code~~ { gen (id.name = E.place); }

$E \to E_1 * T$ { E.place = new temp(); gen (E.place = E_1.place + T.place); }

$E \to T$ { E.place = T.place; }

$T \to T * F$ { T.place = new temp(); gen (T.place = T.place * F.place); }

$T \to F$

$F \to id$ { F.place = id.name }.

$S \to id = E$

$E \to E + F \mid F$

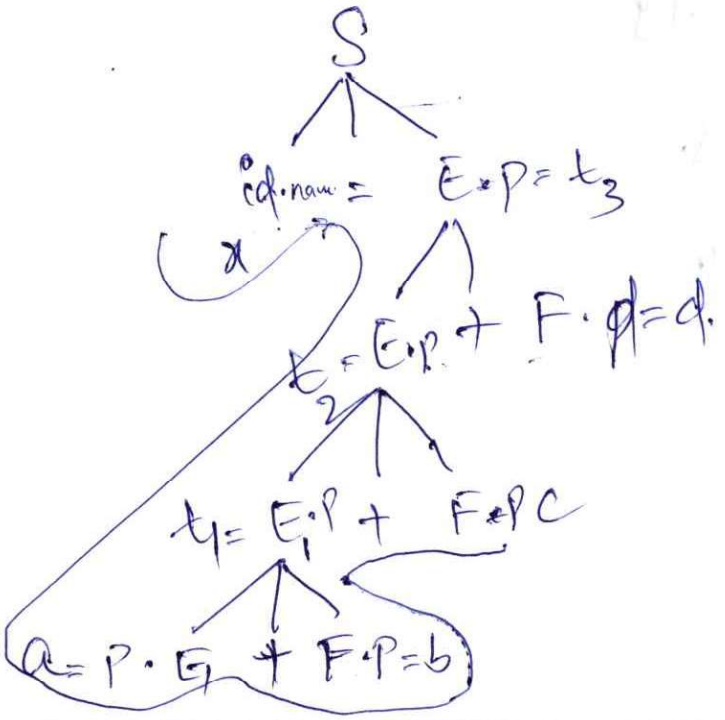
$F \to id$

$x = a + b + c + d$

$t_1 = a + b$

$t_2 = t_1 + c$

$x = t_2 + d$.



$t_1 = a + b$.

$t_2 = t_1 + c$.

$t_3 = t_2 + d$.

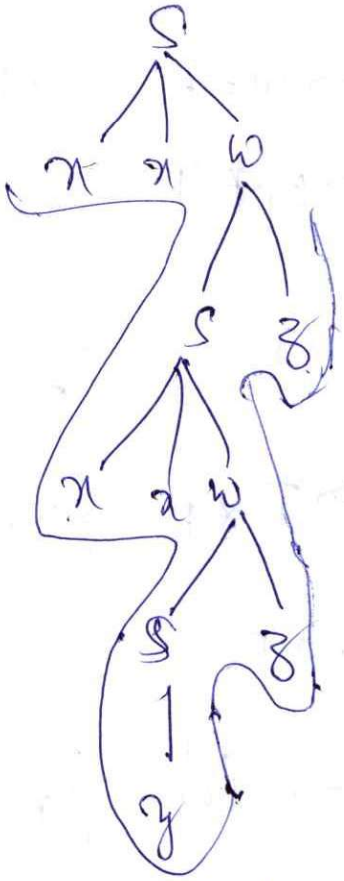
$x = t_3$.

(1) $S \to x\ x\ w$ { printf ("1"); }

$S \to y$ { printf ("2"); }

$W \to S\ z$ { printf ("3"); }.

i/p = x x x x y z z



o/p = 23131

(2) find o/p for i/p = 4 - 2 - 4 * 2 for a given SDT

$E \to E * T$ { E.val = E.val * T.val }

$E \to T$ { E.val = T.val }

$T \to F - T$ { F.val = F.val - T.val }

$T \to F$ { T.val = F.val }

$F \to 2$ { F.val = 2 }.

$F \to 4$ { F.val = 4 }.

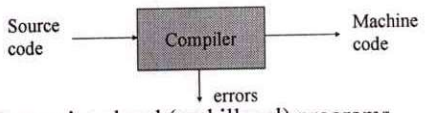
Terminology

- Compiler:
  - a program that translates an *executable* program in one language into an *executable* program in another language
  - we expect the program produced by the compiler to be better, in some way, than the original
- Interpreter:
  - a program that reads an *executable* program and produces the results of running that program
  - usually, this involves executing the source program in some fashion
- Our course is mainly about compilers but many of the same issues arise in interpreters

Disciplines involved

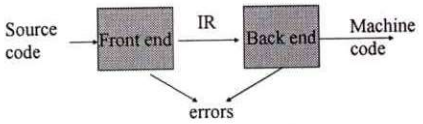
- Algorithms
- Languages and machines
- Operating systems
- Computer architectures

Abstract view



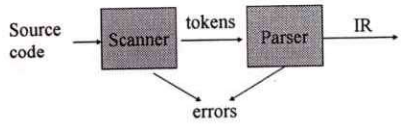
- Recognizes legal (and illegal) programs
- Generate correct code
- Manage storage of all variables and code
- Agreement on format for object (or assembly) code

Front-end, Back-end division



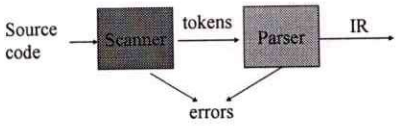
- Front end maps legal code into IR
- Back end maps IR onto target machine
- Simplify retargeting
- Allows multiple front ends
- Multiple passes -> better code

Front end



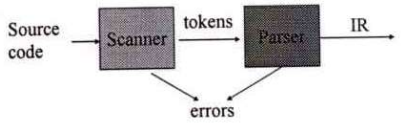
- Recognize legal code
- Report errors
- Produce IR
- Preliminary storage maps

Front end



- Scanner:
  - Maps characters into tokens – the basic unit of syntax
    - x = x + y becomes <id, x> = <id, x> + <id, y>
  - Typical tokens: number, id, +, -, *, /, do, end
  - Eliminate white space (tabs, blanks, comments)
- A key issue is speed so instead of using a tool like LEX it sometimes needed to write your own scanner

Front end



- Parser:
  - Recognize context-free syntax
  - Guide context-sensitive analysis
  - Construct IR
  - Produce meaningful error messages
  - Attempt error correction
- There are parser generators like YACC which automates much of the work

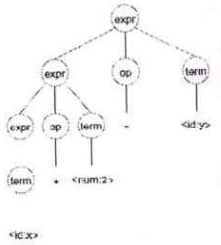
Front end

- Context free grammars are used to represent programming language syntaxes:

<expr> ::= <expr> <op> <term> | <term>
<term> ::= <number> | <id>
<op> ::= + | -

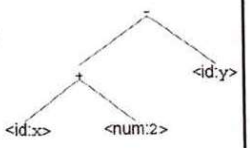
Front end

- A parser tries to map a program to the syntactic elements defined in the grammar
- A parse can be represented by a tree called a parse or syntax tree

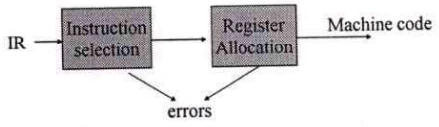


Front end

- A parse tree can be represented more compactly referred to as Abstract Syntax Tree (AST)
- AST is often used as IR between front end and back end

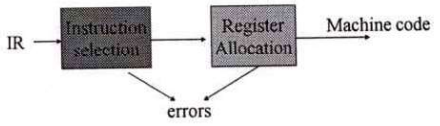


Back end



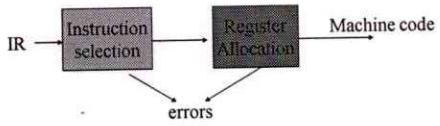
- Translate IR into target machine code
- Choose instructions for each IR operation
- Decide what to keep in registers at each point
- Ensure conformance with system interfaces

Back end



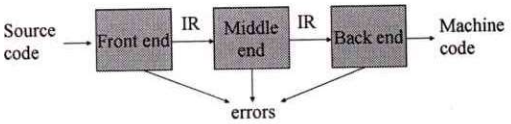
- Produce compact fast code
- Use available addressing modes

Back end



- Have a value in a register when used
- Limited resources
- Optimal allocation is difficult

Traditional three pass compiler



- Code improvement analyzes and change IR
- Goal is to reduce runtime

Middle end (optimizer)

- Modern optimizers are usually built as a set of passes
- Typical passes
  - Constant propagation
  - Common sub-expression elimination
  - Redundant store elimination
  - Dead code elimination

Compiler course

Chapter 3
Lexical Analysis

Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability

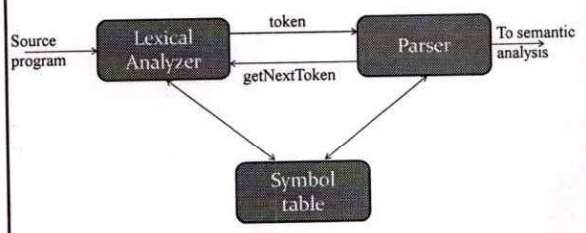
Outline

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Lexical analyzer generator
- Finite automata
- Design of lexical analyzer generator

Tokens, Patterns and Lexemes

- A token is a pair a token name and an optional token value
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token

The role of lexical analyzer



Example

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but " sorrounded by "	"core dumped"

printf("total = %d\n", score);

Attributes for tokens

- E = M * C ** 2
  - <id, pointer to symbol table entry for E>
  - <assign-op>
  - <id, pointer to symbol table entry for M>
  - <mult-op>
  - <id, pointer to symbol table entry for C>
  - <exp-op>
  - <number, integer value 2>

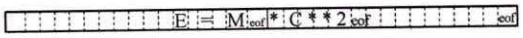
Input buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
  - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large look-aheads safely

Lexical errors

- Some errors are out of power of lexical analyzer to recognize:
  - fi (a == f(x)) ...
- However it may be able to recognize errors like:
  - d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence

Sentinels




```
Switch (*forward++) {
   case eof:
         if (forward is at end of first buffer) {
                  reload second buffer;
                  forward = beginning of second buffer;
         }
         else if (forward is at end of second buffer) {
                  reload first buffer;\
                  forward = beginning of first buffer;
         }
         else /* eof within a buffer marks the end of input */
                  terminate lexical analysis;
         break;
   cases for the other characters;
}
```

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
  - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

Regular expressions

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in Σ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) ∪ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L9r))*
- (r) is a regular expression denting L(r)

Recognition of tokens

- Starting point is the language grammar to understand the tokens:

stmt -> **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| ε
expr -> term **relop** term
| term
term -> **id**
| **number**

Regular definitions

d1 -> r1
d2 -> r2
...
dn -> rn

- Example:

letter_ -> A | B | ... | Z | a | b | ... | Z | _
digit -> 0 | 1 | ... | 9
id -> letter_ (letter_ | digit)*

Recognition of tokens (cont.)

- The next step is to formalize the patterns:

digit -> [0-9]
Digits -> digit+
number -> digit(.digits)? (E[+-]? Digit)?
letter -> [A-Za-z_]
id -> letter (letter|digit)*
If -> if
Then -> then
Else -> else
Relop -> < | > | <= | >= | = | <>

- We also need to handle whitespaces:

ws -> (blank | tab | newline)+

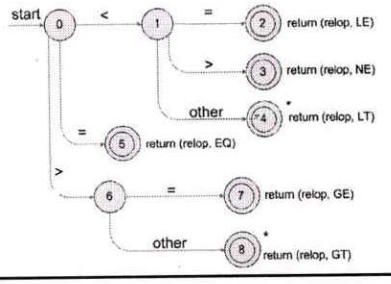
Extensions

- One or more instances: (r)+
- Zero of one instances: r?
- Character classes: [abc]

- Example:
  - letter_ -> [A-Za-z_]
  - digit -> [0-9]
  - id -> letter_(letter|digit)*

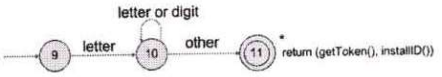
Transition diagrams

- Transition diagram for relop



Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers

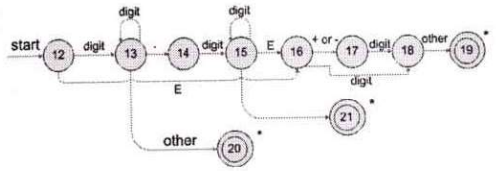


Architecture of a transition-diagram-based lexical analyzer

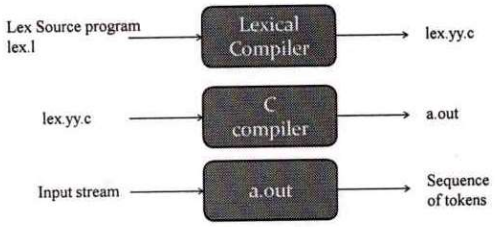
```
TOKEN getRelop()
{
    TOKEN retToken = new (RELOP)
    while (1) {          /* repeat character processing until a
                                return or failure occurs */
    switch(state) {
            case 0: c= nextchar();
                    if (c == '<') state = 1;
                    else if (c == '=') state = 5;
                    else if (c == '>') state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
    }
```

Transition diagrams (cont.)

- Transition diagram for unsigned numbers

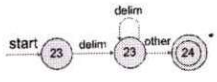


Lexical Analyzer Generator - Lex



Transition diagrams (cont.)

- Transition diagram for whitespace



Structure of Lex programs

declarations
%%
translation rules ⟶ Pattern {Action}
%%
auxiliary functions

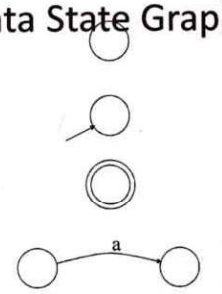
Example

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID); }
{number}  {yylval = (int) installNum(); return(NUMBER);}
...
```

```
Int installID() {/* funtion to install the
    lexeme, whose first character is
    pointed to by yytext, and whose
    length is yyleng, into the symbol
    table and return a pointer thereto
    */
}

Int installNum() { /* similar to
    installID, but puts numerical
    constants into a separate table */
}
```

Finite Automata State Graphs

- A state
- The start state
- An accepting state
- A transition



28

Finite Automata

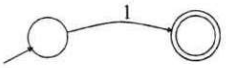
- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions state $\rightarrow^{\text{input}}$ state

26

A Simple Example

- A finite automaton that accepts only "1"



- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

29

Finite Automata

- Transition

$$s_1 \rightarrow^a s_2$$

- Is read

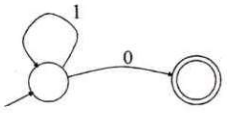
In state s_1 on input "a" go to state s_2

- If end of input
  - If in accepting state => accept, othewise => reject
- If no transition possible => reject

27

Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}

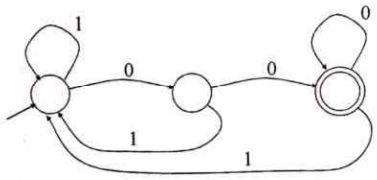


- Check that "1110" is accepted but "110..." is not

30

And Another Example

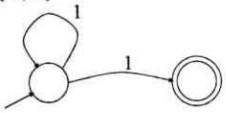
- Alphabet {0,1}
- What language does this recognize?



31

And Another Example

- Alphabet still { 0, 1 }

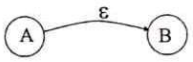


- The operation of the automaton is not completely defined by the input
  - On input "11" the automaton could be in either state

32

Epsilon Moves

- Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input

33

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- *Finite* automata have *finite* memory
  - Need only to encode the current state

34

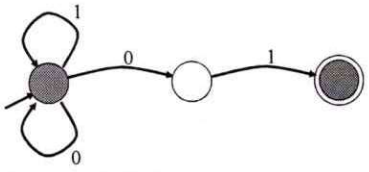
Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

35

Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 1
- Rule: NFA accepts if it <u>can</u> get in a final state

36

NFA vs. DFA (1)

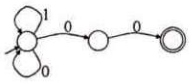
- NFAs and DFAs recognize the same set of languages (regular languages)
- DFAs are easier to implement
  - There are no choices to consider

37

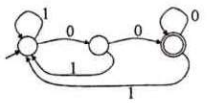
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA



DFA

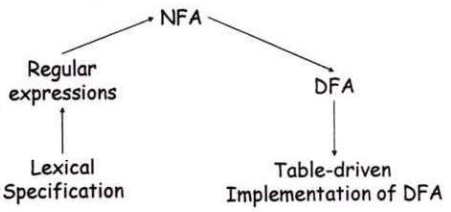


- DFA can be exponentially larger than NFA

38

Regular Expressions to Finite Automata

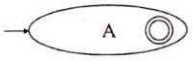
- High-level sketch



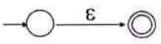
39

Regular Expressions to NFA (1)

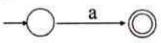
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A



- For ε



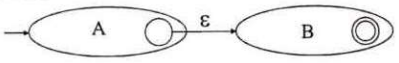
- For input a



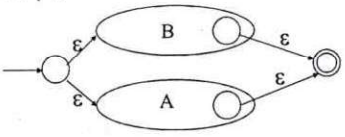
40

Regular Expressions to NFA (2)

- For AB



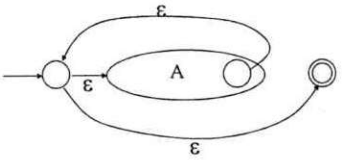
- For A | B



41

Regular Expressions to NFA (3)

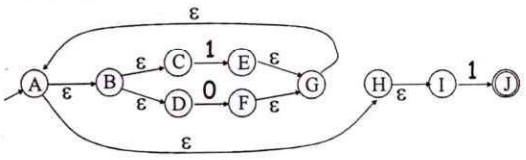
- For A*



42

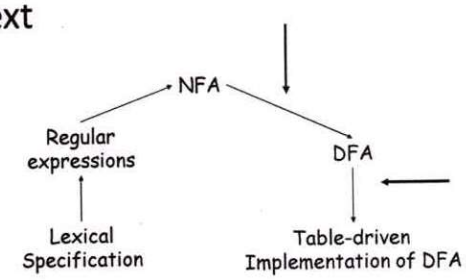
Example of RegExp -> NFA conversion

- Consider the regular expression
  (1 | 0)*1
- The NFA is



43

Next



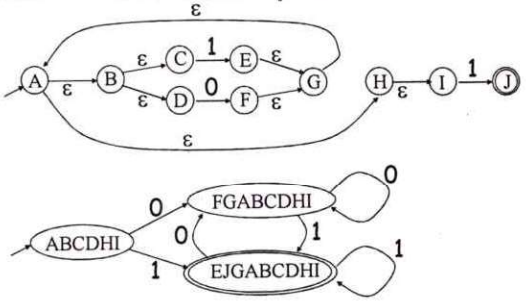
44

NFA to DFA. The Trick

- Simulate the NFA
- Each state of resulting DFA
  = a non-empty subset of states of the NFA
- Start state
  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^a S'$ to DFA iff
  - S' is the set of NFA states reachable from the states in S after seeing the input a
    - considering ε-moves as well

45

NFA -> DFA Example



46

NFA to DFA. Remark

- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many non-empty subsets are there?
  - $2^N - 1$ = finitely many, but exponentially many

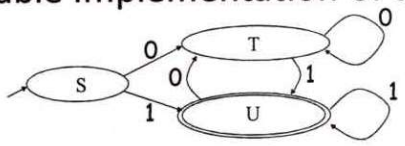
47

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $S_i \rightarrow^a S_k$ define T[i,a] = k
- DFA "execution"
  - If in state S_i and input a, read T[i,a] = k and skip to state S_k
  - Very efficient

48

Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

49

Implementation (Cont.)

- NFA -> DFA conversion is at the heart of tools such as flex or jflex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations

50

Readings

- Chapter 3 of the book

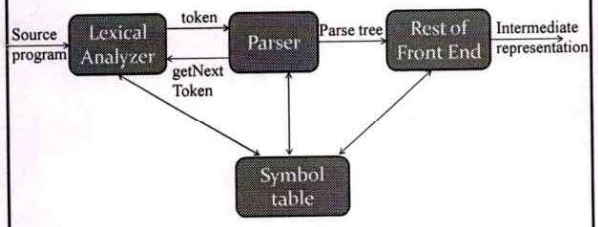
Compiler course

Chapter 4
Syntax Analysis

Outline

- Role of parser
- Context free grammars
- Top down parsing
- Bottom up parsing
- Parser generators

The role of parser



Uses of grammars

E -> E + T | T
T -> T * F | F
F -> (E) | **id**

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

Error handling

- Common programming errors
  - Lexical errors
  - Syntactic errors
  - Semantic errors
  - Lexical errors
- Error handler goals
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect subsequent errors
  - Add minimal overhead to the processing of correct progrms

Error-recover strategies

- Panic mode recovery
  - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- Phrase level recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue
- Error productions
  - Augment the grammar with productions that generate the erroneous constructs
- Global correction
  - Choosing minimal sequence of changes to obtain a globally least-cost correction

Context free grammars

- Terminals
- Nonterminals
- Start symbol
- productions

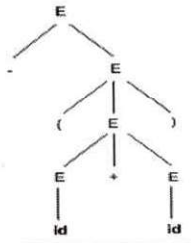
expression -> expression + term
expression -> expression – term
expression -> term
term -> term * factor
term -> term / factor
term -> factor
factor -> (expression)
factor -> **id**

Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
  - E -> E + E | E * E | -E | (E) | **id**
  - Derivations for **–(id+id)**
    - E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id+id**)

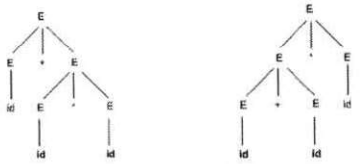
Parse trees

- **-(id+id)**
- E => -E => -(E) => -(E+E) => -(**id**+E)=>-(id+id)



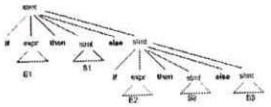
Ambiguity

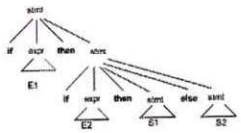
- For some strings there exist more than one parse tree
- Or more than one leftmost derivation
- Or more than one rightmost derivation
- Example: id+id*id

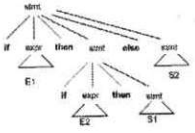


Elimination of ambiguity

stmt ⟶ **If** expr **then** stmt
| **If** expr **then** stmt **else** stmt
| **other**

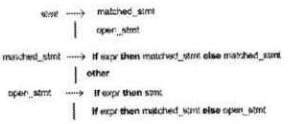






Elimination of ambiguity (cont.)

- Idea:
  - A statement appearing between a **then** and an **else** must be matched



Elimination of left recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \stackrel{+}{\Rightarrow} A\,\alpha$
- Top down parsing methods cant handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:
    - $A \rightarrow A\,\alpha \mid \beta$
  - We may replace it with
    - $A \rightarrow \beta\ A'$
    - $A' \rightarrow \alpha\ A' \mid \varepsilon$

Left recursion elimination (cont.)

- There are cases like following
  - S -> Aa | b
  - A -> Ac | Sd | ε
- Left recursion elimination algorithm:
  - Arrange the nonterminals in some order A1,A2,...,An.
  - For (each i from 1 to n) {
    - For (each j from 1 to i-1) {
      - Replace each production of the form Ai-> Aj γ by the production Ai-> $\delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where Aj-> $\delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all current Aj productions
      - }
      - Eliminate left recursion among the Ai-productions
    - }

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Consider following grammar:
  - Stmt -> **if** expr **then** stmt **else** stmt
  - | **if** expr **then** stmt
- On seeing input **if** it is not clear for the parser which production to use
- We can easily perform left factoring:
  - If we have A-> $\alpha\,\beta_1 \mid \alpha\,\beta_2$ then we replace it with
    - A -> αA'
    - A'-> $\beta_1 \mid \beta_2$

Left factoring (cont.)

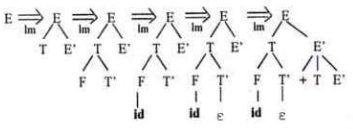
- Algorithm
  - For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha <> \varepsilon$, then replace all of A-productions A-> $\alpha\,\beta_1 \mid \alpha\,\beta_2 \mid \ldots \mid \alpha\,\beta_n \mid \gamma$ by
    - $A \rightarrow \alpha A' \mid \gamma$
    - $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
- Example:
  - S -> I E t S | i E t S e S | a
  - E -> b

Top Down Parsing

Introduction

- A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right
- It can be also viewed as finding a leftmost derivation for an input string
- Example: id+id*id

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



Recursive descent parsing

- Consists of a set of procedures, one for each nonterminal
- Execution begins with the procedure for start symbol
- A typical procedure for a non-terminal

```
void A() {
        choose an A-production, A->X1X2..Xk
        for (i=1 to k) {
                if (Xi is a nonterminal
                        call procedure Xi();
                else if (Xi equals the current input symbol a)
                        advance the input to the next symbol;
                else /* an error has occurred */
        }
}
```

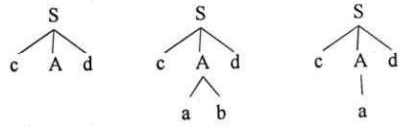
Recursive descent parsing (cont)

- General recursive descent may require backtracking
- The previous code needs to be modified to allow backtracking
- In general form it cant choose an A-production easily.
- So we need to try all alternatives
- If one failed the input pointer needs to be reset and another alternative should be tried
- Recursive descent parsers cant be used for left-recursive grammars

Example

S->cAd
A->ab | a

Input: cad



First and Follow

- First() is set of terminals that begins strings derived from
- If $\alpha \stackrel{*}{\Rightarrow} \varepsilon$ then is also in First(ε)
- In predictive parsing when we have A-> $\alpha \mid \beta$, if First(α) and First(β) are disjoint sets then we can select appropriate A-production by looking at the next input
- Follow(A), for any nonterminal A, is set of terminals a that can appear immediately after A in some sentential form
  - If we have $S \stackrel{*}{\Rightarrow} \alpha A a \beta$ for some α and β then a is in Follow(A)
- If A can be the rightmost symbol in some sentential form, then $ is in Follow(A)

Computing First

- To compute First(X) for all grammar symbols X, apply following rules until no more terminals or ε can be added to any First set:
  1. If X is a terminal then First(X) = {X}.
  2. If X is a nonterminal and X->$Y_1Y_2...Y_k$ is a production for some k>=1, then place a in First(X) if for some i a is in First(Yi) and ε is in all of First(Y_1),...,First(Yi-1) that is $Y_1...Y_{i-1} \stackrel{*}{\Rightarrow} \varepsilon$. if ε is in First(Yj) for j=1,...,k then add ε to First(X).
  3. If X-> ε is a production then add ε to First(X)
- Example!

Computing follow

- To compute First(A) for all nonterminals A, apply following rules until nothing can be added to any follow set:
  1. Place $ in Follow(S) where S is the start symbol
  2. If there is a production A-> $\alpha B \beta$ then everything in First(β) except ε is in Follow(B).
  3. If there is a production A->B or a production A-> $\alpha B \beta$ where First(β) contains ε, then everything in Follow(A) is in Follow(B)
- Example!

LL(1) Grammars

- Predictive parsers are those recursive descent parsers needing no backtracking
- Grammars for which we can create predictive parsers are called LL(1)
  - The first L means scanning input from left to right
  - The second L means leftmost derivation
  - And 1 stands for using one input symbol for lookahead
- A grammar G is LL(1) if and only if whenever A-> α | β are two distinct productions of G, the following conditions hold:
  - For no terminal a do α and β both derive strings beginning with a
  - At most one of α or β can derive empty string
  - If $\alpha \Rightarrow^* \varepsilon$ then β does not derive any string beginning with a terminal in Follow(A).

Construction of predictive parsing table

- For each production A-> α in grammar do the following:
  1. For each terminal a in First(α) add A-> in M[A,a]
  2. If ε is in First(α), then for each terminal b in Follow(A) add A-> ε to M[A,b]. If ε is in First(α) and $ is in Follow(A), add A-> ε to M[A,$] as well
- If after performing the above, there is no production in M[A,a] then set M[A,a] to error

Example

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

	First	Follow
F	{(,id}	{+, *,), $}
T	{(,id}	{+,), $}
E	{(,id}	{), $}
E'	{+,ε}	{), $}
T'	{*,ε}	{+,), $}

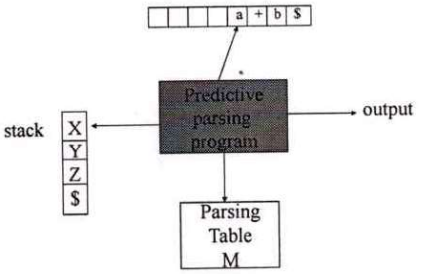
Non-terminal	Input Symbol: id	+	*	(	)	$
E	E -> TE'			E -> TE'		
E'		E' -> +TE'			E' -> ε	E' -> ε
T	T -> FT'			T -> FT'		
T'		T' -> ε	T' -> *FT'		T' -> ε	T' -> ε
F	F -> **id**			F -> (E)		

Another example

S -> iEtSS' | a
S' -> eS | ε
E -> b

Non-terminal	Input Symbol: a	b	e	i	t	$
S	S -> a			S -> iEtSS'		
S'			S' -> ε S' -> eS			S' -> ε
E		E -> b				

Non-recursive predicting parsing



Predictive parsing algorithm

```
Set ip point to the first symbol of w;
Set X to the top stack symbol;
While (X<>$) { /* stack is not empty */
  if (X is a) pop the stack and advance ip;
  else if (X is a terminal) error();
  else if (M[X,a] is an error entry) error();
  else if (M[X,a] = X->Y1Y2..Yk) {
        output the production X->Y1Y2..Yk;
        pop the stack;
        push Yk,...,Y2,Y1 on to the stack with Y1 on top;
  }
  set X to the top stack symbol;
}
```

Example

- id+id*id$

Matched	Stack	Input	Action
	E$	id+id*id$	

Error recovery in predictive parsing

- Panic mode
  - Place all symbols in Follow(A) into synchronization set for nonterminal A: skip tokens until an element of Follow(A) is seen and pop A from stack.
  - Add to the synchronization set of lower level construct the symbols that begin higher level constructs
  - Add symbols in First(A) to the synchronization set of nonterminal A
  - If a nonterminal can generate the empty string then the production deriving can be used as a default
  - If a terminal on top of the stack cannot be matched, pop the terminal, issue a message saying that the terminal was insterted

Example

Non-terminal	Input Symbol: id	+	*	(	)	$
E	E -> TE'			E -> TE	synch	synch
E'		E' -> +TE'			E' -> ε	E' -> ε
T	T -> FT'	synch		T -> FT'	synch	synch
T'		T' -> ε	T' -> *FT'		T' -> ε	T' -> ε
F	F -> id	synch	synch	F -> (E)	synch	synch

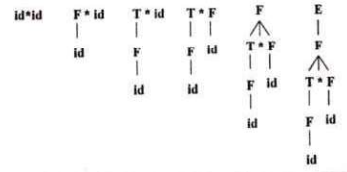
Stack	Input	Action
E$	)id*+id$	Error, Skip)
E$	id*+id$	id is in First(E)
TE'$	id*+id$	
FT'E'$	id*+id$	
idT'E'$	id*+id$	
T'E'$	*+id$	
*FT'E'$	*+id$	
FT'E'$	+id$	Error, M[F,+]=synch
T'E'$	+id$	F has been poped

Bottom-up Parsing

Introduction

- Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)
- Example: id*id

E -> E + T | T
T -> T * F | F
F -> (E) | **id**



Shift-reduce parser

- The general idea is to shift some symbols of input to the stack until a reduction can be applied
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply
- A reduction is a reverse of a step in a derivation
- The goal of a bottom-up parser is to construct a derivation in reverse:
  - E=>T=>T*F=>T*id=>F*id=>id*id

Handle pruning

- 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

Right sentential form	Handle	Reducing production
id*id	id	F->id
F*id	F	T->F
T*id	id	F->id
T*F	T*F	E->T*F

Shift reduce parsing

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

Stack	Input
$	w$

- Acceptance configuration

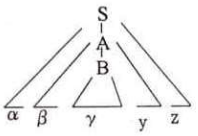
Stack	Input
$S	$

Shift reduce parsing (cont.)

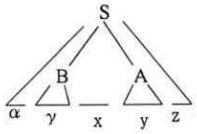
- Basic operations:
  - Shift
  - Reduce
  - Accept
  - Error
- Example: id*id

Stack	Input	Action
$	id*id$	shift
$id	*id$	reduce by F->id
$F	*id$	reduce by T->F
$T	*id$	shift
$T*	id$	shift
$T*id	$	reduce by F->id
$T*F	$	reduce by T->T*F
$T	$	reduce by E->T
$E	$	accept

Handle will appear on top of the stack



Stack	Input
$ α β γ	yz$
$ α β B	yz$
$ α β By	z$



Stack	Input
$ α γ	xyz$
$ α Bxy	z$

Conflicts during shit reduce parsing

- Two kind of conflicts
  - Shift/reduce conflict
  - Reduce/reduce conflict
- Example:

```
stmt ⟶ if expr then stmt
     | if expr then stmt else stmt
     | other
```

Stack	Input
… if expr then stmt	else …$

Reduce/reduce conflict

stmt -> id(parameter_list)
stmt -> expr:=expr
parameter_list->parameter_list, parameter
parameter_list->parameter
parameter->id
expr->id(expr_list)
expr->id
expr_list->expr_list, expr
expr_list->expr

Stack	Input
… id(id	,id) …$

LR Parsing

- The most prevalent type of bottom-up parsers
- LR(k), mostly interested on parsers with k<=1
- Why LR parsers?
  - Table driven
  - Can be constructed to recognize all programming language constructs
  - Most general non-backtracking shift-reduce parsing method
  - Can detect a syntactic error as soon as it is possible to do so
  - Class of grammars for which we can construct LR parsers are superset of those which we can construct LL parsers

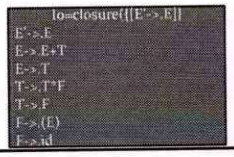
States of an LR parser

- States represent set of items
- An LR(o) item of G is a production of G with the dot at some position of the body:
  - For A->XYZ we have following items
    - A->.XYZ
    - A->X.YZ
    - A->XY.Z
    - A->XYZ.
  - In a state having A->.XYZ we hope to see a string derivable from XYZ next on the input.
  - What about A->X.YZ?

Constructing canonical LR(0) item sets

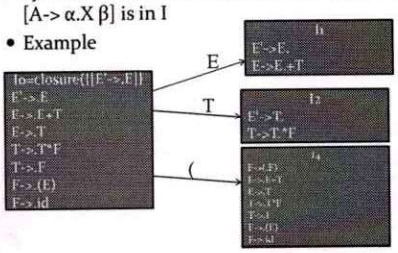
- Augmented grammar:
  - G with addition of a production: S'->S
- Closure of item sets:
  - If I is a set of items, closure(I) is a set of items constructed from I by the following rules:
    - Add every item in I to closure(I)
    - If A->α.Bβ is in closure(I) and B->γ is a production then add the item B->.γ to clsoure(I).
- Example:

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



Constructing canonical LR(0) item sets (cont.)

- Goto (I,X) where I is an item set and X is a grammar symbol is closure of set of all items [A-> αX. β] where [A-> α.X β] is in I
- Example



Closure algorithm

```
SetOfItems CLOSURE(I) {
  J=I;
  repeat
      for (each item A-> α.Bβ in J)
            for (each prodcution B->γ of G)
                  if (B->.γ is not in J)
                        add B->.γ to J;
  until no more items are added to J on one round;
  return J;
```

GOTO algorithm

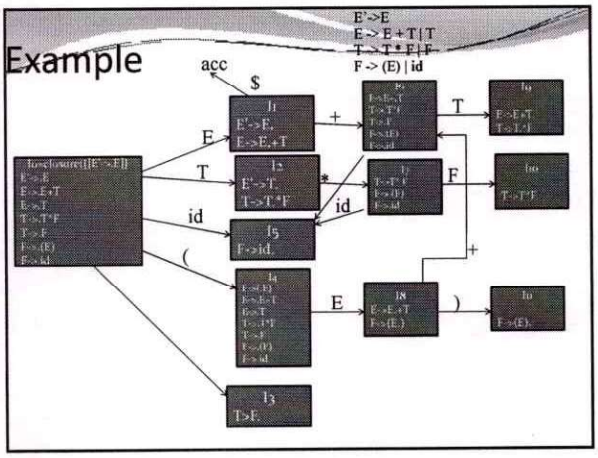
```
SetOfItems  GOTO(I,X) {
  J=empty;
  if (A-> α.X β is in I)
       add CLOSURE(A-> αX. β ) to J;
  return J;
}
```

Canonical LR(0) items

```
Void items(G') {
  C= CLOSURE({[S'->.S]});
  repeat
      for (each set of items I in C)
        for (each grammar symbol X)
          if (GOTO(I,X) is not empty and not in C)
              add GOTO(I,X) to C;
  until no new set of items are added to C on a round;
}
```

Example

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

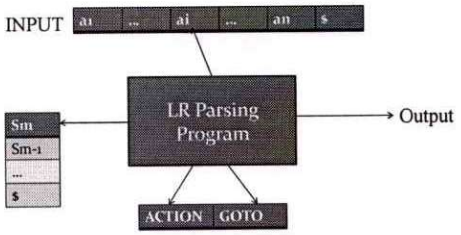


Use of LR(0) automaton

- Example: id*id

Line	Stack	Symbols	Input	Action
(1)	0	$	id*id$	Shift to 5
(2)	05	$id	*id$	Reduce by F->id
(3)	03	$F	*id$	Reduce by T->F
(4)	02	$T	*id$	Shift to 7
(5)	027	$T*	id$	Shift to 5
(6)	0275	$T*id	$	Reduce by F->id
(7)	02710	$T*F	$	Reduce by T->T*F
(8)	02	$T	$	Reduce by E->T
(9)	01	$E	$	accept

LR-Parsing model



LR parsing algorithm

```
let a be the first symbol of w$;
while(1) { /*repeat forever */
   let s be the state on top of the stack;
   if (ACTION[s,a] = shift t) {
         push t onto the stack;
         let a be the next input symbol;
   } else if (ACTION[s,a] = reduce A->β) {
         pop |β| symbols of the stack;
         let state t now be on top of the stack;
         push GOTO[t,A] onto the stack;
         output the production A->β;
   } else if (ACTION[s,a]=accept) break; /* parsing is done */
   else call error-recovery routine;
}
```

Example

(0) E'->E
(1) E -> E + T
(2) E-> T
(3) T -> T * F
(4) T-> F
(5) F -> (E)
(6) F->id

id*id+id?

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		R3	S7		R2	R2			
3		R4	R7		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		R2	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

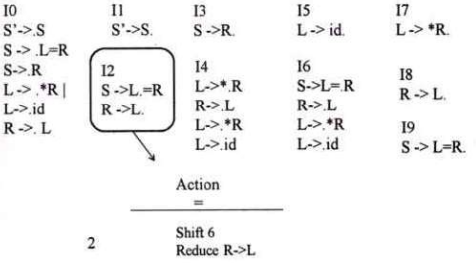
Line	Stack	Symbols	Input	Action
(1)	0		id*id+id$	Shift to 5
(2)	05	id	*id+id$	Reduce by F->id
(3)	03	F	*id+id$	Reduce by T->F
(4)	02	T	*id+id$	Shift to 7
(5)	027	T*	id+id$	Shift to 5
(6)	0275	T*id	+id$	Reduce by F->id
(7)	02710	T*F	+id$	Reduce by T->T*F
(8)	02	T	+id$	Reduce by E->T
(9)	01	E	+id$	Shift
(10)	016	E+	id$	Shift
(11)	0165	E+id	$	Reduce by F->id
(12)	0163	E+F	$	Reduce by T->F
(13)	0169	E+T	$	Reduce by E->E+T
(14)	01	E	$	accept

Constructing SLR parsing table

- Method
  - Construct C={Io,I1, ... , In}, the collection of LR(o) items for G'
  - State i is constructed from state Ii:
    - If [A->α.aβ] is in Ii and Goto(Ii,a)=Ij, then set ACTION[i,a] to "shift j"
    - If [A->α.] is in Ii, then set ACTION[i,a] to "reduce A->α" for all a in follow(A)
    - If [S'->.S] is in Ii, then set ACTION[I,$] to "Accept"
  - If any conflicts appears then we say that the grammar is not SLR(1).
  - If GOTO(Ii,A) = Ij then GOTO[i,A]=j
  - All entries not defined by above rules are made "error"
  - The initial state of the parser is the one constructed from the set of items containing [S'->.S]

Example grammar which is not SLR(1)

S -> L=R | R
L -> *R | id
R -> L



	Action
2	Shift 6 Reduce R->L

More powerful LR parsers

- Canonical-LR or just LR method
  - Use lookahead symbols for items: LR(1) items
  - Results in a large collection of items
- LALR: lookaheads are introduced in LR(o) items

Canonical LR(1) items

- In LR(1) items each item is in the form: [A->α.β,a]
- An LR(1) item [A->α.β,a] is valid for a viable prefix γ if there is a derivation $S \overset{*}{\Rightarrow}_{rm} \delta A w \Rightarrow_{rm} \delta\alpha\beta w$, where
  - Γ= δα
  - Either a is the first symbol of w, or w is ε and a is $
- Example:
  - S->BB
  - B->aB|b

$S \overset{*}{\Rightarrow}_{rm} aaBab \Rightarrow_{rm} aaaBab$

Item [B->a.B,a] is valid for γ=aaa and w=ab

Constructing LR(1) sets of items

```
SetOfItems Closure(I) {
    repeat
        for (each item [A->α.Bβ,a] in I)
            for (each production B->γ in G')
                for (each terminal b in First(βa))
                    add [B->.γ, b] to set I;
    until no more items are added to I;
    return I;
}

SetOfItems Goto(I,X) {
    initialize J to be the empty set;
    for (each item [A->α.Xβ,a] in I)
        add item [A->αX.β,a] to set J;
    return closure(J);
}

void items(G'){
    initialize C to Closure({[S'->.S,$]});
    repeat
        for (each set of items I in C)
            for (each grammar symbol X)
                if (Goto(I,X) is not empty and not in C)
                    add Goto(I,X) to C;
    until no new sets of items are added to C;
}
```

Example

S'->S
S->CC
C->cC
C->d

Canonical LR(1) parsing table

- Method
  - Construct C={Io,I1, ... , In}, the collection of LR(1) items for G'
  - State i is constructed from state Ii:
    - If [A->α.aβ, b] is in Ii and Goto(Ii,a)=Ij, then set ACTION[i,a] to "shift j"
    - If [A->α., a] is in Ii, then set ACTION[i,a] to "reduce A->α"
    - If [S'->.S,$] is in Ii, then set ACTION[I,$] to "Accept"
  - If any conflicts appears then we say that the grammar is not LR(1).
  - If GOTO(Ii,A) = Ij then GOTO[i,A]=j
  - All entries not defined by above rules are made "error"
  - The initial state of the parser is the one constructed from the set of items containing [S'->.S,$]

Example

S'->S
S->CC
C->cC
C->d

LALR Parsing Table

- For the previous example we had:

- State merges cant produce Shift-Reduce conflicts. Why?
- But it may produce reduce-reduce conflict

Example of RR conflict in state merging

S'->S
S -> aAd | bBd | aBe | bAe
A -> c
B -> c

An easy but space-consuming LALR table construction

- Method:
  1. Construct C={Io,I1,...,In} the collection of LR(1) items.
  2. For each core among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
  3. Let C'={Jo,J1,...,Jm} be the resulting sets. The parsing actions for state i, is constructed from Ji as before. If there is a conflict grammar is not LALR(1).
  4. If J is the union of one or more sets of LR(1) items, that is J = I1 UI2...IIk then the cores of Goto(I1,X), ..., Goto(Ik,X) are the same and is a state like K, then we set Goto(J,X) =k.
- This method is not efficient, a more efficient one is discussed in the book

Compaction of LR parsing table

- Many rows of action tables are identical
  - Store those rows separately and have pointers to them from different states
  - Make lists of (terminal-symbol, action) for each state
  - Implement Goto table by having a link list for each nonterinal in the form (current state, next state)

Using ambiguous grammars

E->E+E

E->E*E

E->(E)

E->id

STATE	ACTON						GO TO
	id	+	*	(	)	$	E
0	S3			S2			1
1		S4	S5			Acc	
2	S3			S2			6
3		R4	R4		R4	R4	
4	S3			S2			7
5	S3			S2			8
6		S4	S5				
7		R1	S5		R1	R1	
8		R2	R2		R2	R2	
9		R3	R3		R3	R3	

I0: E'->.E
E->.E+E
E->.E*E
E->.(E)
E->.id

I1: E'->E.
E->E.+E
E->E.*E

I2: E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id

I3: E->.id

I4: E->E+.E
E->.E+E
E->.E*E
E->.(E)
E->.id

I5: E->E*.E
E->(E)
E->.E+E
E->.E*E
E->.(E)
E->.id

I6: E->(E.)
E->E.+E
E->E.*E

I7: E->E+E.
E->E.+E
E->E.*E

I8: E->E*E.
E->E.+E
E->E.*E

I9: E->(E).

Readings

- Chapter 4 of the book

Compiler course

Chapter 5

Syntax Directed Translation

Outline

- Syntax Directed Definitions
- Evaluation Orders of SDD's
- Applications of Syntax Directed Translation
- Syntax Directed Translation Schemes

Introduction

- We can associate information with a language construct by attaching attributes to the grammar symbols.
- A syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.

Production	Semantic Rule
E->E1+T	E.code=E1.code‖T.code‖'+'

- We may alternatively insert the semantic actions inside the grammar

E -> E1+T {print '+'}

Syntax Directed Definitions

- A SDD is a context free grammar with attributes and rules
- Attributes are associated with grammar symbols and rules with productions
- Attributes may be of many kinds: numbers, types, table references, strings, etc.
- Synthesized attributes
  - A synthesized attribute at node N is defined only in terms of attribute values of children of N and at N it
- Inherited attributes
  - An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings

Example of S-attributed SDD

Production	Semantic Rules
1) L -> E n	L.val = E.val
2) E -> E1 + T	E.val = E1.val + T.val
3) E -> T	E.val = T.val
4) T -> T1 * F	T.val = T1.val * F.val
5) T -> F	T.val = F.val
6) F -> (E)	F.val = E.val
7) F -> digit	F.val = digit.lexval

Example of mixed attributes

Production	Semantic Rules
1) T -> FT'	T'.inh = F.val
	T.val = T'.syn
2) T' -> *FT'1	T'1.inh = T'.inh*F.val
	T'.syn = T'1.syn
3) T' -> ε	T'.syn = T'.inh
1) F -> digit	F.val = F.val = digit.lexval

Evaluation orders for SDD's

- A dependency graph is used to determine the order of computation of attributes
- Dependency graph
  - For each parse tree node, the parse tree has a node for each attribute associated with that node
  - If a semantic rule defines the value of synthesized attribute A.b in terms of the value of X.c then the dependency graph has an edge from X.c to A.b
  - If a semantic rule defines the value of inherited attribute B.c in terms of the value of X.a then the dependency graph has an edge from X.c to B.c
- Example!

Ordering the evaluation of attributes

- If dependency graph has an edge from M to N then M must be evaluated before the attribute of N
- Thus the only allowable orders of evaluation are those sequence of nodes $N_1,N_2,...,N_k$ such that if there is an edge from Ni to Nj then i<j
- Such an ordering is called a topological sortof a graph
- Example!

S-Attributed definitions

- An SDD is S-attributed if every attribute is synthesized
- We can have a post-order traversal of parse-tree to evaluate attributes in S-attributed definitions

```
postorder(N) {
        for (each child C of N, from the left) postorder(C);
        evaluate the attributes associated with node N;
}
```

- S-Attributed definitions can be implemented during bottom-up parsing without the need to explicitly create parse trees

L-Attributed definitions

- A SDD is L-Attributed if the edges in dependency graph goes from Left to Right but not from Right to Left.
- More precisely, each attribute must be either
  - Synthesized
  - Inherited, but if there us a production $A->X_1X_2...X_n$ and there is an inherited attribute Xi.a computed by a rule associated with this production, then the rule may only use:
    - Inherited attributes associated with the head A
    - Either inherited or synthesized attributes associated with the occurrences of symbols $X_1,X_2,...,X_{i-1}$ located to the left of Xi
    - Inherited or synthesized attributes associated with this occurrence of Xi itself, but in such a way that there is no cycle in the graph

Application of Syntax Directed Translation

- Type checking and intermediate code generation (chapter 6)
- Construction of syntax trees
  - Leaf nodes: Leaf(op,val)
  - Interior node: Node(op,c1,c2,...,ck)
- Example:

Production	Semantic Rules
1) E -> E1 + T	E.node=new node('+', E1.node,T.node)
2) E -> E1 - T	E.node=new node('-', E1.node,T.node)
3) E -> T	E.node = T.node
4) T -> (E)	T.node = E.node
5) T -> id	T.node = new Leaf(id,id.entry)
6) T -> num	T.node = new Leaf(num,num.val)

Syntax tree for L-attributed definition

Production	Semantic Rules
1) E -> TE'	E.node=E'.syn E'.inh=T.node
2) E' -> + TE1'	E1'.inh=new node('+', E'.inh,T.node) E'.syn=E1'.syn
3) E' -> -TE1'	E1'.inh=new node('+', E'.inh,T.node) E'.syn=E1'.syn
4) E' -> ϵ	E'.syn = E'.inh
5) T -> (E)	T.node = E.node
6) T -> id	T.node=new Leaf(id,id.entry)
7) T -> num	T.node = new Leaf(num,num.val)

Syntax directed translation schemes

- An SDT is a Context Free grammar with program fragments embedded within production bodies
- Those program fragments are called semantic actions
- They can appear at any position within production body
- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth first order
- Typically SDT's are implemented during parsing without building a parse tree

Postfix translation schemes

- Simplest SDDs are those that we can parse the grammar bottom-up and the SDD is s-attributed
- For such cases we can construct SDT where each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production
- SDT's with all actions at the right ends of the production bodies are called postfix SDT's

Example of postfix SDT

```
1) L -> E n        {print(E.val);}
2) E -> E1 + T     {E.val=E1.val+T.val;}
3) E -> T          {E.val = T.val;}
4) T -> T1 * F     {T.val=T1.val*F.val;}
5) T -> F          {T.val=F.val;}
6) F -> (E)        {F.val=E.val;}
7) F -> digit      {F.val=digit.lexval;}
```

Parse-Stack implementation of postfix SDT's

- In a shift-reduce parser we can easily implement semantic action using the parser stack
- For each nonterminal (or state) on the stack we can associate a record holding its attributes
- Then in a reduction step we can execute the semantic action at the end of a production to evaluate the attribute(s) of the non-terminal at the leftside of the production
- And put the value on the stack in replace of the rightside of production

Example

```
L -> E n        {print(stack[top-1].val);
                  top=top-1;}
E -> E1 + T     {stack[top-2].val=stack[top-2].val+stack.val;
                  top=top-2;}
E -> T
T -> T1 * F     {stack[top-2].val=stack[top-2].val+stack.val;
                  top=top-2;}
T -> F
F -> (E)        {stack[top-2].val=stack[top-1].val
          top=top-2;}
F -> digit
```

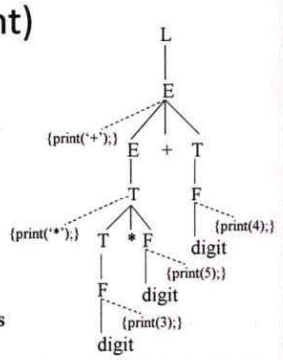
SDT's with actions inside productions

- For a production B->X {a} Y
  - If the parse is bottom-up then we perform action "a" as soon as this occurrence of X appears on the top of the parser stack
  - If the parser is top down we perform "a" just before we expand Y
- Sometimes we cant do things as easily as explained above
- One example is when we are parsing this SDT with a bottom-

```
1) L -> E n
2) E -> {print('+');} E1 + T
3) E -> T
4) T -> {print('*');} T1 * F
5) T -> F
6) F -> (E)
7) F -> digit {print(digit.lexval);}
```

SDT's with actions inside productions (cont)

- Any SDT can be implemented as follows
  1. Ignore the actions and produce a parse tree
  2. Examine each interior node N and add actions as new children at the correct position
  3. Perform a postorder traversal and execute actions when their nodes are visited



SDT's for L-Attributed definitions

- We can convert an L-attributed SDD into an SDT using following two rules:
  - Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A. if several inherited attributes of A are dpendent on one another in an acyclic fashion, order them so that those needed first are computed first
  - Place the action of a synthesized attribute for the head of a production at the end of the body of the production

Example

S -> while (C) S1

L1=new();
L2=new();
S1.next=L1;
C.false=S.next;
C.true=L2;

S.code=label||L1||C.code||label||L2||S1.code

S -> while ({L1=new();L2=new();C.false=S.next;C.true=L2;}
C) {S1.next=L1;}
S1{S.code=label||L1||C.code||label||L2||S1.code;}

Readings

- Chapter 5 of the book

Compiler course

Chapter 8

Code Generation

Outline

- Code Generation Issues
- Target language Issues
- Addresses in Target Code
- Basic Blocks and Flow Graphs
- Optimizations of Basic Blocks
- A Simple Code Generator
- Peephole optimization
- Register allocation and assignment
- Instruction selection by tree rewriting

Introduction

- The final phase of a compiler is code generator
- It receives an intermediate representation (IR) with supplementary information in symbol table
- Produces a semantically equivalent target program
- Code generator main tasks:
  - Instruction selection
  - Register allocation and assignment
  - Insrtuction ordering

Issues in the Design of Code Generator

- The most important criterion is that it produces correct code
- Input to the code generator
  - IR + Symbol table
  - We assume front end produces low-level IR, i.e. values of names in it can be directly manipulated by the machine instructions.
  - Syntactic and semantic errors have been already detected
- The target program
  - Common target architectures are: RISC, CISC and Stack based machines
  - In this chapter we use a very simple RISC-like computer with addition of some CISC-like addressing modes

complexity of mapping

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code.

x=y+z

```
LD    R0, y
ADD   R0, R0, z
ST    x, R0
```

a=b+c
d=a+e

```
LD    R0, b
ADD   R0, R0, c
ST    a, R0
LD    R0, a
ADD   R0, R0, e
ST    d, R0
```

Register allocation

- Two subproblems
  - Register allocation: selecting the set of variables that will reside in registers at each point in the program
  - Resister assignment: selecting specific register that a variable reside in
- Complications imposed by the hardware architecture
  - Example: register pairs for multiplication and division

```
t=a+b
t=t*c
T=t/d

L     R1, a
A     R1, b
M     R0, c
D     R0, d
ST    R1, t
```

```
t=a+b
t=t+c
T=t/d

L     R0, a
A     R0, b
M     R0, c
SRDA  R0, 32
D     R0, d
ST    R1, t
```

A simple target machine model

- Load operations: LD r,x and LD r1, r2
- Store operations: ST x,r
- Computation operations: OP dst, src1, src2
- Unconditional jumps: BR L
- Conditional jumps: Bcond r, L like BLTZ r, L

Addressing Modes

- variable name: x
- indexed address: a(r) like LD R1, a(R2) means R1=contents(a+contents(R2))
- integer indexed by a register : like LD R1, 100(R2)
- Indirect addressing mode: *r and *100(r)
- immediate constant addressing mode: like LD R1, #100

b = a [i]

```
LD R1, i                //R1 = i
MUL R1, R1, 8           //R1 = Rl * 8
LD R2, a(R1)
      //R2=contents(a+contents(R1))
ST b, R2                //b = R2
```

a[j] = c

```
LD R1, c                //R1 = c
LD R2, j                // R2 = j
MUL R2, R2, 8           //R2 = R2 * 8
ST a(R2), R1
      //contents(a+contents(R2))=R1
```

x=*p

```
LD R1, p                //R1 = p
LD R2, o(R1)            // R2 =
  contents(o+contents(R1))
ST x, R2                // x=R2
```

conditional-jump three-address instruction

If x<y goto L

```
LD R1, x        // R1 = x
LD R2, y        // R2 = y
SUB R1, R1, R2  // R1 = R1 - R2
BLTZ R1, M      // i f R1 < o jump t o M
```

costs associated with the addressing modes

- LD R0, R1 cost = 1
- LD R0, M cost = 2
- LD R1, *100(R2) cost = 3

Addresses in the Target Code

- A statically determined area Code
- A statically determined data area Static
- A dynamically managed area Heap
- A dynamically managed area Stack

three-address statements for procedure calls and returns

- call callee
- Return
- Halt
- action

Target program for a sample call and return

```
                                    // code for c
                100:  ACTION1       // code for action1
        // code for c
                120:  ST 364, #140  // save return address 140 in location 364
action1             132:  BR 200    // call p
call p              140:  ACTION2
action2             160:  HALT      // return to operating system
halt                ...
                                    // code for p
        // code for p
                200:  ACTION3
action3             220:  BR *364   // return to address saved in location 364
return              ...
                                    // 300-363 hold activation record for c
                300:                // return address
                304:                // local data for c
                ...
                                    // 364-451 hold activation record for p
                364:                // return address
                368:                // local data for p
```

Stack Allocation

```
LD   SP, #stackStart          // initialize the stack
code for the first procedure
HALT                          // terminate execution

ADD  SP, SP, #caller.recordSize  // increment stack pointer
ST   *SP, #here + 16             // save return address
BR   callee.codeArea             // return to caller
                                    Branch to called procedure
```

Return to caller

in Callee: BR *0(SP)

in caller: SUB SP, SP, #caller.recordsize

Target code for stack allocation

```
                                    // code for m
                100:  LD SP, #600   // initialize the stack
                108:  ACTION1       // code for action1
                128:  ADD SP, SP, #sizeM  // call sequence begins
                136:  ST +SP, #152  // push return address
                144:  BR 300        // call q
        // code for m               // restore SP
                152:  SUB SP, SP, #sizeM
action1         160:  ACTION2
call q          ...
action2                             // code for p
halt
                200:  ACTION3
                220:  BR *0(SP)     // return
        // code for p
                ...
action3                             // code for q
return          300:  ACTION4       // contains a conditional jump to 456
                320:  ADD SP, SP, #sizeQ
        // code for q
                328:  ST *SP, #344  // push return address
action4         336:  BR 200        // call p
call p          341:  SUB SP, SP, #sizeQ
action5         352:  ACTION5
call q          372:  ADD SP, SP, #sizeQ
action6         380:  BR *SP, #396  // push return address
call q          388:  BR 300        // call q
return          396:  SUB SP, SP, #sizeQ
                404:  ACTION6
                424:  ADD SP, SP, #sizeQ
                432:  ST *SP, #440  // push return address
                440:  BR 300        // call q
                448:  SUB SP, SP, #sizeQ
                456:  BR *0(SP)     // return
                ...
                600                 // stack starts here
```

Basic blocks and flow graphs

- Partition the intermediate code into basic blocks
  - The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
  - Control will leave the block without halting or branching, except possibly at the last instruction in the block.
- The basic blocks become the nodes of a flow graph

rules for finding leaders

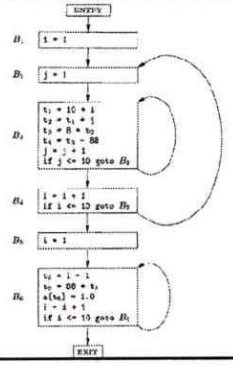
- The first three-address instruction in the intermediate code is a leader.
- Any instruction that is the target of a conditional or unconditional jump is a leader.
- Any instruction that immediately follows a conditional or unconditional jump is a leader.

Intermediate code to set a 10*10 matrix to an identity matrix

for i from 1 to 10 **do**
    for j from 1 to 10 **do**
        $a[i, j] = 0.0;$
for i from 1 to 10 **do**
    $a[i, i] = 1.0;$

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Flow graph based on Basic Blocks



liveness and next-use information

- We wish to determine for each three address statement x=y+z what the next uses of x, y and z are.
- Algorithm:
  - Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
  - In the symbol table, set x to "not live" and "no next use."
  - In the symbol table, set y and z to "live" and the next uses of y and z to i.

DAG representation of basic blocks

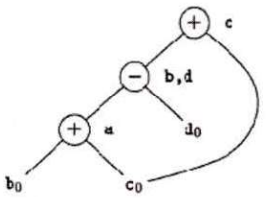
- There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
- There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
- Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
- Certain nodes are designated *output nodes*. These are the nodes whose variables are *live on exit* from the block.

Code improving transformations

- We can eliminate *local common subexpressions,* that is, instructions that compute a value that has already been computed.
- We can eliminate *dead code,* that is, instructions that compute a value that is never used.
- We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
- We can apply algebraic laws to reorder operands of three-address instructions, and sometimes t hereby simplify t he computation.

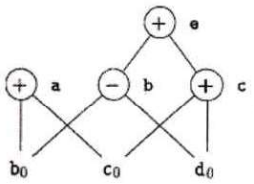
DAG for basic block

```
a = b + c
b = a - d
c = b + c
d = a - d
```



DAG for basic block

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

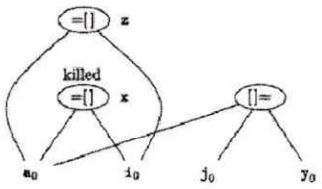


array accesses in a DAG

- An assignment from an array, like x = a [i], is represented by creating a node with operator =[] and two children representing the initial value of the array, ao in this case, and the index i. Variable x becomes a label of this new node.
- An assignment to an array, like a [j] = y, is represented by a new node with operator []= and three children representing ao, j and y. There is no variable labeling this node. What is different is that the creation of this node *kills* all currently constructed nodes whose value depends on ao. A node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.

DAG for a sequence of array assignments

```
x = a[i]
a[j] = y
z = a[i]
```



Rules for reconstructing the basic block from a DAG

- The order of instructions must respect the order of nodes in the DAG. That is, we cannot compute a node's value until we have computed a value for each of its children.
- Assignments to an array must follow all previous assignments to, or evaluations from, the same array, according to the order of these instructions in the original basic block.
- Evaluations of array elements must follow any previous (according to the original block) assignments to the same array. The only permutation allowed is that two evaluations from the same array may be done in either order, as long as neither crosses over an assignment to that array.
- Any use of a variable must follow all previous (according to the original block) procedure calls or indirect assignments through a pointer.
- Any procedure call or indirect assignment through a pointer must follow all previous (according to the original block) evaluations of any variable.

principal uses of registers

- In most machine architectures, some or all of the operands of an operation must be in registers in order to perform the operation.
- Registers make good temporaries - places to hold the result of a subexpression while a larger expression is being evaluated, or more generally, a place to hold a variable that is used only within a single basic block.
- Registers are often used to help with run-time storage management, for example, to manage the run-time stack, including the maintenance of stack pointers and possibly the top elements of the stack itself.

Descriptors for data structure

- For each available register, a register descriptor keeps track of the variable names whose current value is in that register. Since we shall use only those registers that are available for local use within a basic block, we assume that initially, all register descriptors are empty. As the code generation progresses, each register will hold the value of zero or more names.
- For each program variable, an address descriptor keeps track of the location or locations where the current value of that variable can be found. The location might be a register, a memory address, a stack location, or some set of more than one of these. The information can be stored in the symbol-table entry for that variable name.

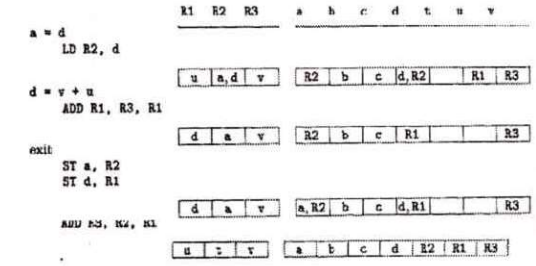
Machine Instructions for Operations

- Use getReg(x = y + z) to select registers for x, y, and z. Call these R_x, R_y and R_z.
- If y is not in R_y (according to the register descriptor for R_y), then issue an instruction LD R_y, y', where y' is one of the memory locations for y (according to the address descriptor for y).
- Similarly, if z is not in R_z, issue and instruction LD R_z, z', where z' is a location for *x* .
- Issue the instruction ADD R_x , R_y, R_z.

Rules for updating the register and address descriptors

- For the instruction LD R, x
  - Change the register descriptor for register R so it holds only x.
  - Change the address descriptor for x by adding register R as an additional location.
- For the instruction ST x, R, change the address descriptor for x to include its own memory location.
- For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction x = y + *x*
  - Change the register descriptor for R_x so that it holds only *x*.
  - Change the address descriptor for x so that its only location is R_x. Note that the memory location for x is *not* now in the address descriptor for x.
  - Remove R_x from the address descriptor of any variable other than x.
- When we process a copy statement x = y, after generating the load for y into register R_y, if needed, and after managing descriptors as for all load statements (per rule 1):
  - Add x to the register descriptor for R_y
  - Change the address descriptor for x so that its only location is R_y .

Instructions generated and the changes in the register and address descriptors



Rules for picking register R_y for y

- If y is currently in a register, pick a register already containing y as R_y. Do not issue a machine instruction to load this register, as none is needed.
- If y is not in a register, but there is a register that is currently empty, pick one such register as R_y.
- The difficult case occurs when y is not in a register, and there is no register that is currently empty. We need to pick one of the allowable registers anyway, and we need to make it safe to reuse.

Possibilities for value of R

- If the address descriptor for v says that v is somewhere besides R, then we are OK.
- If v is x, the value being computed by instruction I, and x is not also one of the other operands of instruction I (z in this example), then we are OK. The reason is that in this case, we know this value of x is never again going to be used, so we are free to ignore it.
- Otherwise, if v is not used later (that is, after the instruction I, there are no further uses of v, and if v is live on exit from the block, then v is recomputed within the block), then we are OK.
- If we are not OK by one of the first two cases, then we need to generate the store instruction **ST** v, *R* to place a copy of v in its own memory location. This operation is called a spill.

Selection of the register Rx

1. Since a new value of x is being computed, a register that holds only x is always an acceptable choice for Rx.
2. If y is not used after instruction I, and Ry holds only y after being loaded, Ry can also be used as Rx. A similar option holds regarding z and Rx.

Possibilities for value of R

- If the address descriptor for v says that v is somewhere besides R, then we are OK.
- If v is x, the value being computed by instruction I, and x is not also one of the other operands of instruction I (z in this example), then we are OK. The reason is that in this case, we know this value of x is never again going to be used, so we are free to ignore it.
- Otherwise, if v is not used later (that is, after the instruction I, there are no further uses of v, and if v is live on exit from the block, then v is recomputed within the block), then we are OK.
- If we are not OK by one of the first two cases, then we need to generate the store instruction **ST** v, *R* to place a copy of v in its own memory location. This operation is called a spill.

Characteristic of peephole optimizations

- Redundant-instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Redundant-instruction elimination

- LD a, Ro
  ST Ro, a
- if debug == 1 goto L1
  goto L2
  L I : print debugging information
  L2:

Flow-of-control optimizations

goto L1
...
Ll: goto L2

Can be replaced by:
goto L2
...
Ll: goto L2

if a<b goto L1
...
Ll: goto L2

Can be replaced by:
if a<b goto L2
...
Ll: goto L2

Algebraic simplifications

- x=x+0
- x=x*1

Register Allocation and Assignment

- Global Register Allocation
- Usage Counts
- Register Assignment for Outer Loops
- Register Allocation by Graph Coloring

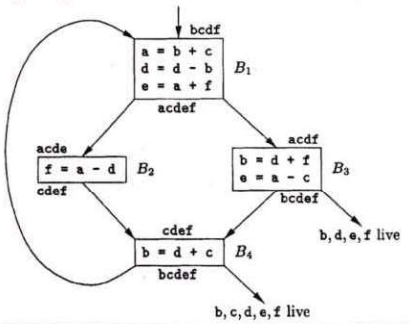
Global register allocation

- Previously explained algorithm does local (block based) register allocation
- This resulted that all live variables be stored at the end of block
- To save some of these stores and their corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (globally)
- Some options are:
  - Keep values of variables used in loops inside registers
  - Use graph coloring approach for more globally allocation

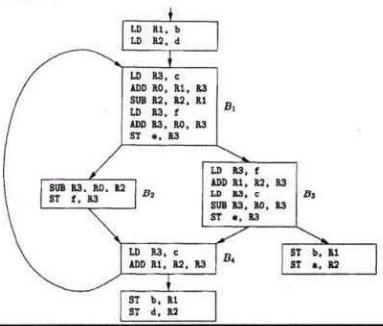
Usage counts

- For the loops we can approximate the saving by register allocation as:
  - Sum over all blocks (B) in a loop (L)
  - For each uses of x before any definition in the block we add one unit of saving
  - If x is live on exit from B and is assigned a value in B, then we ass 2 units of saving

Flow graph of an inner loop



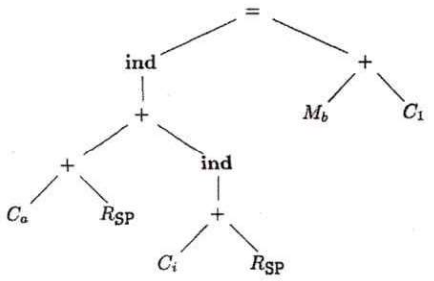
Code sequence using global register assignment



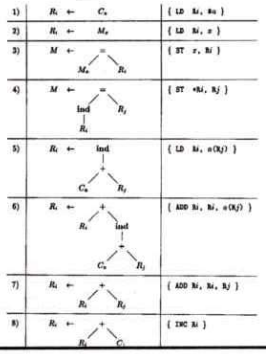
Register allocation by Graph coloring

- Two passes are used
  - Target-machine instructions are selected as though there are an infinite number of symbolic registers
  - Assign physical registers to symbolic ones
    - Create a register-interference graph
    - Nodes are symbolic registers and edges connects two nodes if one is live at a point where the other is defined.
    - For example in the previous example an edge connects a and d in the graph
    - Use a graph coloring algorithm to assign registers.

Intermediate-code tree for a[i]=b+1



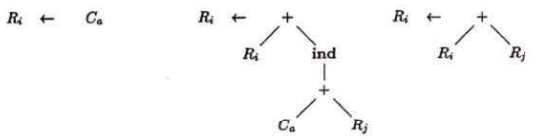
Tree-rewriting rules



Syntax-directed translation scheme

1)	$R_i \rightarrow \mathbf{c}_a$	{ LD Ri, #a }
2)	$R_i \rightarrow M_x$	{ LD Ri, x }
3)	$M \rightarrow = M_x\ R_i$	{ ST x, Ri }
4)	$M \rightarrow = \mathbf{ind}\ R_i\ R_j$	{ ST *Ri, Rj }
5)	$R_i \rightarrow \mathbf{ind} + \mathbf{c}_a\ R_j$	{ LD Ri, a(Rj) }
6)	$R_i \rightarrow + R_i\ \mathbf{ind} + \mathbf{c}_a\ R_j$	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \rightarrow + R_i\ R_j$	{ ADD Ri, Ri, Rj }
8)	$R_i \rightarrow + R_i\ \mathbf{c}_1$	{ INC Ri }
9)	$R \rightarrow \mathbf{sp}$	
10)	$M \rightarrow \mathbf{m}$	

An instruction set for tree matching

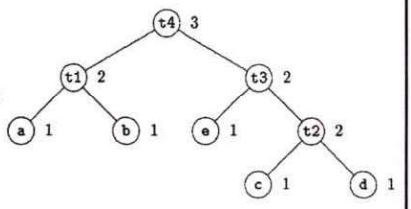


Ershov Numbers

- Label any leaf 1.
- The label of an interior node with one child is the label of its child.
- The label of an interior node with two children is
  - The larger of the labels of its children, if those labels are different.
  - One plus the label of its children if the labels are the same.

A tree labeled with Ershov numbers

```
t1 = a - b
t2 = c + d
t3 = e * t2
t4 = t1 + t3
```



Generating code from a labeled expression tree

- To generate machine code for an interior node with label k and two children with equal labels (which must be k - 1) do the following:
  - Recursively generate code for the right child, using base $b+1$. The result of the right child appears in register R_{b+k}.
  - Recursively generate code for the left child, using base b; the result appears in R_{b+k-1}.
  - Generate the instruction OP R_{b+k}, R_{b+k-1}, R_{b+k}, where OP is the appropriate operation for the interior node in question.
- Suppose we have an interior node with label k and children with unequal labels. Then one of the children, which we'll call the "big" child, has label k, and the other child, the "little" child, has some label $m < k$. Do the following to generate code for this interior node, using base b:
  - Recursively generate code for the big child, using base b; the result appears in register R_{b+k-1}.
  - Recursively generate code for the small child, using base b; the result appears in register R_{b+m-1}. Note that since $m < k$, neither R_{b+k-1} nor any higher-numbered register is used.
  - Generate the instruction OP R_{b+k-1}, R_{b+m-1}, R_{b+k-1} or the instruction OP R_{b+k-1}, R_{b+k-1}, R_{b+m-1}, depending on whether the big child is the right or left child, respectively.
- For a leaf representing operand x, if the base is b generate the instruction LD R_b, x.

Optimal three-register code

```
LD  R3, d
LD  R2, c
ADD R3, R2, R3
LD  R2, e
MUL R3, R2, R3
LD  R2, b
LD  R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

Evaluating Expressions with an Insufficient Supply of Registers

- Node N has at least one child with label r or greater. Pick the larger child (or either if their labels are the same) to be the "big" child and let the other child be the "little" child.
- Recursively generate code for the big child, using base b = 1. The result of this evaluation will appear in register R_r.
- Generate the machine instruction ST t_k, R_r, where t_k is a temporary variable used for temporary results used to help evaluate nodes with label k.
- Generate code for the little child as follows. If the little child has label r or greater, pick base b=1. If the label of the little child is j<r, then pick b=r-j. Then recursively apply this algorithm to the little child; the result appears in R_r.
- Generate the instruction LD R_r-l, t_k.
- If the big child is the right child of N, then generate the instruction OP R_r, R_r, R_{r-1}. If the big child is the left child, generate OP R_r, R_{r-1}, R_r.

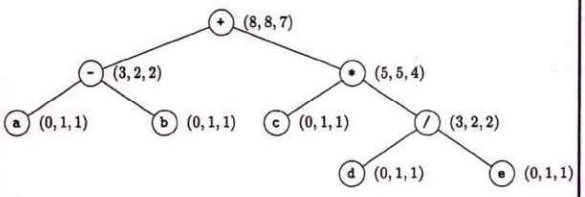
Optimal three-register code using only two registers

```
LD  R2, d
LD  R1, c
ADD R2, R1, R2
LD  R1, e
MUL R2, R1, R2
ST  t3, R2
LD  R2, b
LD  R1, a
SUB R2, R1, R2
LD  R1, t3
ADD R2, R2, R1
```

Dynamic Programming Algorithm

- Compute bottom-up for each node n of the expression tree T an array C of costs, in which the ith component C[i] is the optimal cost of computing the subtree S rooted at n into a register, assuming i registers are available for the computation, for 1 ≤ i ≤ r.
- Traverse T, using the cost vectors to determine which subtrees of T must be computed into memory.
- Traverse each tree using the cost vectors and associated instructions to generate the final target code. The code for the subtrees computed into memory locations is generated first.

Syntax tree for (a-b)+c*(d/e) with cost vector at each node



minimum cost of evaluating the root with two registers available

- Compute the left subtree with two registers available into register Ro, compute the right subtree with one register available into register R1, and use the instruction ADD Ro, Ro, R1 to compute the root. This sequence has cost 2+5+1=8.
- Compute the right subtree with two registers available into R l , compute the left subtree with one register available into Ro, and use the instruction ADD Ro, Ro, R1. This sequence has cost 4+2+1=7.
- Compute the right subtree into memory location M, compute the left subtree with two registers available into register RO, and use the instruction ADD Ro, Ro, M. This sequence has cost 5+2+1=8

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Semester End Question Papers

COMPILER DESIGN

Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)

(Aziz Nagar, C.B.Post, Hyderabad -500075) **Subject code: A16523**

III B. Tech II SEM REGULAR EXAMINATION - MAY 2018

COMPILER DESIGN

(FOR CSE)

Time: 3hrs **Max.Marks:75**

Note: This question paper contains two PARTS A and B.
PART A is compulsory which carries 25 marks. Answer all questions.
PART B consists of 5 Units. Answer all full question.

PART - A

ANSWER ALL THE QUESTIONS **25 M**

1. What do you mean by LL(1) grammar? Give example. 3M
2. What are the problems in top down parsing? 2M
3. what are the various conflicts that occur during shift reduce parsing? 2M
4. Draw the model of an LR parser. 3M
5. Generate three address code for the given pseudo code 3M
   while(i<=100) { A=A/B*20; ++i; print(A value) }
6. How to generate polish notation using translation schemes? 2M
7. At what levels code can be optimized by user and compiler? 2M
8. Describe Loop Optimization. 3M
9. Mention the properties that a code generator should possess. 2M
10. Construct a DAG for the expression a=b*-c + b*-c. 3M

ANSWER ALL THE QUESTIONS **5X10M=50M**

11.i) Describe the various phases of a compiler and trace it with the program segment (position:=initial + rate*60). 10M

OR

ii) a. What are the preprocessing steps required for predictive parse table construction? 5M
b. Construct the predictive parser for the following grammar. 5M
S→ (L)/a
L→L,S/S

12. i) a. With neat sketch explain the structure of LR parser and the rules to compute LR item.5M
b. Consider the following grammar: 5M
E → E + E
E → E * E
E → (E)
E → id
Show the shift-reduce parser action for the string: id*(id+id).

OR

ii). a. Differentiate between Top down and Bottom up Parsing methods. 5M
b. Write the steps for the efficient construction of LALR parsing table. 5M

13. i) a. Explain heap storage allocation. With an Example 5M
b. What is an Abstract syntax tree? How to construct it? Explain by writing syntax directed definition. 5M

OR

ii). a. Can we reuse the symbol table space? Explain through an example. 5M
b. Write short notes on functions of semantic analysis. 5M

14. i) Explain in detail about inter procedural optimization with an example. 10M

OR

ii) Explain reducible and non-reducible flow graphs with an example. 10M

15. i) Efficient Register allocation and assignment improves the performance of object code-Justify this statement with suitable examples. 10M

OR

ii). Explain in detail about machine dependent code optimization techniques with their drawbacks. 10M

*** VJTT(A) ***

Vidya Jyothi Institute of Technology(Autonomous)

(Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)

(Aziz Nagar, C.B.Post, Hyderabad -500075) **Subject code: A16523**

III B. Tech II SEM SUPPLEMENTARYEXAMINATION – DECEMBER 2018 **R15**

COMPILER DESIGN

(CSE)

Time: 3hrs **Max.Marks:75**

Note: This question paper contains two PARTs A and B.
PART A is compulsory which carries 25 marks. Answer all questions.
PART B consists of 5 questions. Answer all the questions.

PART - A

ANSWER ALL THE QUESTIONS **25 M**

1. Write the role of preprocessor in language processing. 2M
2. What is the role of lexical analyzer? 3M
3. Discuss the role of Action and Goto functions in LR parser? 3M
4. Explain the various actions performed by shift-reduce parsers 2M
5. Write the fields and uses of symbol table. 3M
6. What is the need for symbol table? 2M
7. Describe scope of variable? 2M
8. Give theorganization of optimizing compiler. 3M
9. What are the issues in the design of code generator? 3M
10. How to allocate registers to instruction? 2M

PART-B

ANSWER ALL THE QUESTIONS **5QX10M=50M**

11. i) a) What is the role of regular expression in lexical analysis? Explain with examples. 5M
    b) Explain bootstrapping a compiler with suitable diagrams. 5M

OR

ii) a) Verify whether the following grammar is LL(1) or not? 7M

$E \rightarrow E + T \mid T$
$T \rightarrow T^* F / F$
$F \rightarrow (F) \mid a \mid b.$

b) Why lexical and syntax analyzer are separated out? 3M

12. i) a) Parse the input string intid,id;using shift reduce parser for the grammar. 5M

$S \rightarrow TL;$
$T \rightarrow int \mid float$
$L \rightarrow L,id \mid id$

b) Show that the grammar $S \rightarrow 0S1 \mid SS \mid \varepsilon$ is ambiguous. 5M

OR

ii) a) Explain the compaction of LR parsing tables with an example. 5M
b) Explain the process of handling "Dangling-ELSE" ambiguity. 5M

13. i) a) Translate the assignment x := a+b*c+dinto three address statement. 5M
    b) Discuss storage allocation for block structured languages. 5M

OR

ii) a) Write down the specification of a simple Type Checker. 5M
b) Explain in detail the symbol table organization for Block-Structured languages. 5M

14. i) Construct the syntax tree and draw the DAG for the expression. 10M
    (a*b) + (c-d) * (a*b) + b.

OR

ii) Explain how loop optimization can be done? How they are different from local optimizations. 10M

15. i) a) Discuss various object code forms. 5M
    b) write a short note on code generating algorithms. 5M

OR

ii). Explain in detail register allocation and assignment. 10M

VJIT(A)

Code No: 136AQ **R16**

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B. Tech III Year II Semester Examinations, May - 2019
COMPILER DESIGN
(Common to CSE, IT)

Time: 3 hours **Max. Marks: 75**

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 25 marks. Answer all questions in Part A. Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

PART - A

(25 Marks)

1.a) Define regular expression. [2]
b) Define linker and loader and explain briefly. [3]
c) Define ambiguous grammar. [2]
d) Compare SLR, CLR and LACR. [3]
e) What is coercion? [2]
f) How to find evaluation order for SDD's? [3]
g) What are the limitations of static allocation? [2]
h) Write the fields and uses of symbol table. [3]
i) What is common sub-expression elimination? Explain. [2]
j) What are induction variables? What is induction variable elimination? [3]

PART - B

(50 Marks)

2.a) Explain the procedure to convert regular expression to Finite automata.
b) Explain various phases in the construction of compiler with a neat sketch. [5+5]

OR

3.a) What is the functionality of preprocessing and input buffering?
b) Explain compiler construction tools. [5+5]

4.a) What is left recursion? Describe the algorithm used for eliminating left recursion?
b) Eliminate left recursion in the following: [5+5]
$E \rightarrow E+T|T,\ T \rightarrow T*F|F,\ F \rightarrow (E)|id$

OR

5.a) What is ambiguous grammar? Show that following grammar is ambiguous or not.
$A \rightarrow A + A \mid A - A \mid A * A \mid a$
b) Verify whether the following grammar is LL(1) or not? [5+5]
$E \rightarrow E+T|T$
$T \rightarrow T*F/F$
$F \rightarrow (F)\ |a|b.$

6.a) What are three address codes? Explain different types of representations of three address code.

b) Write three codes for x:=A[y, z] [5+5]

OR

7.a) What is type checker? Explain the specification of a simple type checker.

b) Explain translation schema for array elements. [5+5]

8.a) Explain about Heap management.

b) Define reference counting. What is the role of reference counting in garbage collection? [5+5]

OR

9.a) Give the detailed description on DAG.

b) Explain different methods for register allocation and assignment. [5+5]

10.a) Explain redundancy elimination techniques.

b) Write the principal sources of optimization. [5+5]

OR

11.a) Explain loop optimization technique with example.

b) Explain constant propagation with example. [5+5]

---ooOoo---

Code No: 115AP **R13**

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B. Tech III Year I Semester Examinations, May/June - 2019
COMPILER DESIGN
(Computer Science and Engineering)

Time: 3 hours **Max. Marks: 75**

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 25 marks. Answer all questions in Part A. Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

PART - A

(25 Marks)

1.a) Why lexical and syntax analyzers are separated? [2]
b) List the various error recovery strategies for a lexical analysis. [3]
c) Mention the types of LR parser. [2]
d) Define LR(0) items with examples. [3]
e) What are the benefits of intermediate code generation? [2]
f) Explain about hashing. [3]
g) What is a basic block? [2]
h) Discuss about common sub expression elimination. [3]
i) How do you calculate the cost of an instruction? [2]
j) List out the common issues in the design of code generator. [3]

PART - B

(50 Marks)

2. Explain the various phases of a compiler in detail. Also write down the output for the following expression after each phase a: =b*cd. [10]

OR

3. What is FIRST and FOLLOW? Explain the steps to compute FIRST and FOLLOW with an example. [10]

4. Check whether the following grammar is SLR (1) or not. Explain your answer with Reasons. [10]
S→ L=R S→R L→*R L→id R→L.

OR

5. Consider the grammar.
$E \rightarrow E + T$ $E \rightarrow T$ $T \rightarrow T * F$ $T \rightarrow F$ $F \rightarrow (E) / id$
Construct CLR parsing table for the above grammar. Give the moves of the CLR parser on id * id + id. [10]

6. What is a three address code? Mention its types. How would you implement the three address statements? Explain with examples. [10]

OR

7. Describe in detail the syntax directed translation of case statements. [10]

8. What are steps needed to compute the next use information? [10]

OR

9. Discuss about the following:
   a) Copy Propagation
   b) Dead code Elimination and
   c) Code motion. [10]

10. Write the code generation for the d:=(a-b)+(a-c)+(a-c). [10]

OR

11. Write a code generation algorithm. Explain about the descriptor and function getreg(). Give an example. [10]

---ooOoo---

Code No: 115AP

R13

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B. Tech III Year I Semester Examinations, November/December - 2018
COMPILER DESIGN
(Computer Science and Engineering)

Time: 3 hours **Max. Marks: 75**

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 25 marks. Answer all questions in Part A. Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

PART - A

(25 Marks)

1.a) Define Boot strapping? [2]
b) What are the draw backs of predictive parsing? [3]
c) What are the actions performed by Shift reduce parser? [2]
d) Differentiate between SLR, LALR and CLR parsers. [3]
e) What are the applications of DAG? [2]
f) What are the advantages of stack storage allocation strategy? [3]
g) Define Basic block. What are the rules for defining a basic block? [2]
h) What is common sub expression elimination? [3]
i) Define Dead code elimination? [2]
j) What is register allocation? Give a brief description. [3]

PART - B

(50 Marks)

2.a) Define Regular Expression. Explain about the Properties of Regular Expressions.
b) Differentiate between Top down and bottom up parsing techniques. [5+5]

OR

3.a) Define Compiler? Explain in brief about the types of lexical errors with example.
b) Construct a Predictive parsing table for the Grammar
E→E+T/T, T→T*F/F, F→ (E)/id? [4+6]

4.a) Construct LALR Parsing table for the grammar
S→L=R/R
L→*R/id
R→L
b) Define Ambiguous Grammar? Check whether the grammar
S→aAB,
A→bC/cd,
C→cd,
B→c/d Is Ambiguous or not? [6+4]

OR

5.a) Construct SLR Parsing table for the grammar
S→ (L)|a
L→L,s|s
b) Discuss in brief about model of LR parser. [6+4]

6.a) Construct a Quadruple, Triple and Indirect Triple for the statement a+a*(b-c) +(b-c)*d?

b) Differentiate between Static and Dynamic Storage allocation Strategies. [5+5]

OR

7.a) Construct an annotated parse tree for real id1, id2, id3.

b) Explain in brief about equivalence of type expressions. [5+5]

8.a) Explain in brief about different Principal sources of optimization techniques with suitable examples.

b) Define Flow Graph? Explain how a given program can be converted into flow graph? [5+5]

OR

9.a) Explain in brief about function preserving transformations on basic blocks.

b) Explain in brief about optimization of basic blocks. [5+5]

10.a) Explain in detail about global common sub expression elimination technique.

b) Explain in brief about Induction variable elimination. [5+5]

OR

11.a) Explain in brief about the issues in the design of code generator.

b) Explain in detail about peep hole optimization. [5+5]

---oo0oo---

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Extra Topics Delivered

COMPILER DESIGN

S. No	Name of the Topic
1	LEX program
2	DAG(Directed acyclic graph)

1. LEX program

```
/*lex program to count number of words*/
%{
#include<stdio.h>
#include<string.h>
int i = 0;
%}

/* Rules Section*/
%%
([a-zA-Z0-9])*    {i++;} /* Rule for counting
                    number of words*/

"\n" {printf("%d\n", i); i = 0;}
%%

int yywrap(void){}

int main()
{
  // The function that starts the analysis
  yylex();

  return 0;
}
```

Output:

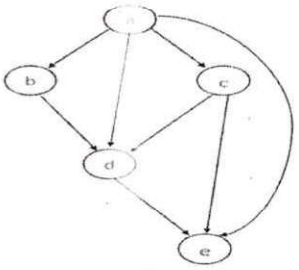
```
shubham@gfg-desktop:~$ lex words.l
shubham@gfg-desktop:~$ cc lex.yy.c -lfl
shubham@gfg-desktop:~$ ./a.out
Hello Everyone
 2
This is GeeksforGeeks
 3
```

2. Directed Acyclic Graph :

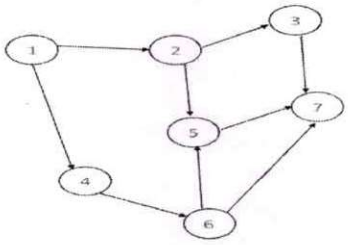
The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.

- Directed acyclic graphs are a type of data structure and they are used to apply transformations to basic blocks.
- The Directed Acyclic Graph (DAG) facilitates the transformation of basic blocks.
- DAG is an efficient method for identifying common sub-expressions.
- It demonstrates how the statement's computed value is used in subsequent statements.

Examples of directed acyclic graph :



(A)



(B)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Innovation in Teaching and Learning

COMPILER DESIGN

Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi) permanently affliated JNTUH

(Aziz Nagar, C.B.Post, Hyderabad -500075)

(AUTONOMOUS)

Innovation in Teaching Learning: Role Play

Subject: Compiler Design

Name of the Faculty: Dr.Aruna Kmari

Topic: Phases of compiler

Class/ Section: III B.Tech II-Sem CSE-B

Teaching is an art and science. Teaching is a process of imparting knowledge and skills. It is a systematic process based on some educational objectives to communicate.

Interactive learning is a hands-on, real-world approach to education. 'Interactive learning actively engages the students in wrestling with the material. It reinvigorates the classroom for both students and faculty. Lectures are changed into discussions, and students and teachers become partners in the journey of knowledge acquisition.'

Role-playing is the changing of one's behaviour to assume a role, either unconsciously to fill a social role, or consciously to act out an adopted role.

- To refer to the playing of roles generally such as in a theatre, or educational setting;
- To refer to taking a role of an existing character or person and acting it out with a partner taking someone else's role, often involving different genres of practice

Topic: Phases of compiler

Compiler: A **compiler** is a computer program that transforms computer code written in one programming language (the source language) into another programming language (the target language).

Lexical Analysis. The first phase of scanner works as a text scanner

Syntax Analysis. The next phase is called the syntax analysis or parsing

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, and adding string to an integer. ...

Intermediate Code Generation After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine.

Code Optimization The next phase does code optimization of the intermediate code.

Code Generation In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.

Instructor

Dr.D.Aruna Kumari

CSE-HOD

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

Vidya Jyothi Institute of Technology

(An Autonomous Institution)

(Accredited by NAAC, Approved by AICTE New Delhi & Permanently Affiliated to JNTUH)

Aziz Nagar Gate, C.B. Post, Hyderabad-500 075

Department of Computer Science & Engineering

(Accredited by NBA)

Academic Year: 2019-20

III B.Tech- II Sem

Course: CD

Branch:CSE

BATCH: 2017-21

S.No	Reg.No	MID I Threshold 60%							MID II Threshold 60%						Threshold 60%
		ASM - I (5)	PART-A			PART-B			ASM - II (5)	PART-A	PART-B				End Exam (75M)
			Q1(2M)(CO1)	Q2(2M)(CO2)	Q3(2M)(CO3)	Q4(5M)(CO1)	Q5(5M)(CO2)	Q6(4M)(CO3)		Q1(6M)(CO3)	Q2(7M)(CO4)	Q3(7M)(CO4)	Q4(7M)(CO5)	Q5(7M)(CO5)	
1	17601A0511	5	1	1	1	2	2	1	5	4	7			7	1
2	17601A0575	5	2	2	2	5	5	3	5	5		7		7	41
3	17911A0501	5	0	0	0	0	0	0	5	0	0	0			29
4	17911A0502	5	1	1	1	3	3	2	5	4	7			7	48
5	17911A0503	5	1	1	1	2	2	1	5	4	7			7	54
6	17911A0504	5	2	2	2	5	5	3	5	5		7		7	46
7	17911A0505	5	0	0	0	0	0	0	5	0	0	0			52
8	17911A0506	5	1	1	1	3	3	2	5	4	7			7	45
9	17911A0507	5	1	1	1	2	2	1	5	4	7			7	53
10	17911A0508	5	2	2	2	5	5	3	5	5		7		7	44
11	17911A0509	5	0	0	0	0	0	0	5	0	0	0			52
12	17911A0510	5	1	1	1	3	3	2	5	4	7			7	53
13	17911A0511	5	1	1	1	2	2	1	5	4	7			7	48
14	17911A0512	5	2	2	2	5	5	3	5	5		7		7	49
15	17911A0513	5	0	0	0	0	0	0	5	0	0	0			56
16	17911A0514	5	1	1	1	3	3	2	5	4	7			7	42
17	17911A0515	5	1	1	1	2	2	1	5	4	7			7	40
18	17911A0516	5	2	2	2	5	5	3	5	5		7		7	45
19	17911A0517	5	0	0	0	0	0	0	5	0	0	0			53
20	17911A0518	5	1	1	1	3	3	2	5	4	7			7	37
21	17911A0520	5	1	1	1	2	2	1	5	4	7			7	40
22	17911A0521	5	2	2	2	5	5	3	5	5		7		7	53
23	17911A0523	5	0	0	0	0	0	0	5	0	0	0			39
24	17911A0524	5	1	1	1	3	3	2	5	4	7			7	48
25	17911A0526	5	1	1	1	5	5	1	5	2		6	6		37
26	17911A0527	5	1	1	1	5	5	1	5	1	7		7		44
27	17911A0528	5	2	2	2	5	5	3	5	2		7	7		52
28	17911A0529	5	2	2	2	5	5	2	5	5		7		7	53
29	17911A0530	5	2	2	2	5	5	3	5	5		7		7	53
30	17911A0531	5	2	2	2	5	5	3	5	5		7		7	55
31	17911A0532	5	0	0	0	0	0	0	5	0	0	0			51
32	17911A0533	5	1	1	1	3	3	2	5	4	7			7	33
33	17911A0534	5	0	0	0	0	0	0	5	1		5	5		47
34	17911A0536	5	1	1	2	5	5	2	5	2	5	5			49
35	17911A0537	5	1	1	1	5	5	1	5	2		6	6		47
36	17911A0538	5	1	1	1	5	5	1	5	1	7		7		36
37	17911A0539	5	2	2	2	5	5	3	5	2		7	7		45
38	17911A0540	5	2	2	2	5	5	2	5	5		7		7	43
39	17911A0541	5	2	2	2	5	5	3	5	5		7		7	45
40	17911A0542	5	2	2	2	5	5	3	5	5		7		7	49
41	17911A0543	5	0	0	0	0	0	0	5	0	0	0			46
42	17911A0544	5	1	1	1	3	3	2	5	4	7			7	44
43	17911A0545	5	0	0	0	0	0	0	5	1		5	5		30
44	17911A0546	5	1	1	2	5	5	2	5	2	5	5			49
45	17911A0547	5	1	1	1	5	5	1	5	2		6	6		42
46	17911A0548	5	1	1	1	5	5	1	5	1	7		7		56
47	17911A0549	5	2	2	2	5	5	3	5	2		7	7		47
48	17911A0550	5	2	2	2	5	5	2	5	5		7		7	41
49	17911A0552	5	2	2	2	5	5	3	5	5		7		7	51
50	17911A0553	5	2	2	2	5	5	3	5	5		7		7	50
51	17911A0554	5	0	0	0	0	0	0	5	0	0	0			46
52	17911A0555	5	1	1	1	3	3	2	5	4	7			7	52
53	17911A0556	5	0	0	0	0	0	0	5	1		5	5		52
54	17911A0557	5	1	1	2	5	5	2	5	2	5	5			52
55	17911A0558	5	1	1	1	5	5	1	5	2		6	6		48
56	17911A0559	5	1	1	1	5	5	1	5	1	7		7		53
57	17911A0560	5	2	2	2	5	5	3	5	2		7	7		54
58	17911A0561	5	2	2	2	5	5	2	5	5		7		7	54
59	17911A0562	5	2	2	2	5	5	3	5	5		7		7	51
60	17911A0563	5	2	2	2	5	5	3	5	5		7		7	50
61	17911A0564	5	0	0	0	0	0	0	5	0	0	0			59
62	17911A0565	5	1	1	1	3	3	2	5	4	7			7	31
63	17911A0566	5	0	0	0	0	0	0	5	1		5	5		51
64	17911A0567	5	1	1	2	5	5	2	5	2	5	5			53
65	17911A0569	5	1	1	1	5	5	1	5	2		6	6		33
66	17911A0570	5	1	1	1	5	5	1	5	1	7		7		40
67	17911A0571	5	2	2	2	5	5	3	5	2		7	7		46
68	17911A0572	5	2	2	2	5	5	2	5	5		7		7	44

69	17911A0573	5	2	2	2	5	5	3	5	5		7		7	36
70	17911A0574	5	2	2	2	5	5	4	5	6		7	7		43
71	17911A0575	5	2	2	2	5	5	3	5	5		7		7	53
72	17911A0576	5	1	1	1	3	3	1	5	2		7	7		34
73	17911A0577	5	2	2	2	5	5	3	5	5		7		7	55
74	17911A0578	5	1	1	2	5	5	2	5	1	7		7		52
75	17911A0579	5	2	2	2	5	5	2	5	6		7	7		61
76	17911A0580	5	1	1	1	5	5	1	5	3		7		7	58
77	17911A0581	5	2	2	2	5	5	4	5	3		7		7	58
78	17911A0582	5	1	1	1	5	5	1	5	4	7			7	62
79	17911A0583	5	1	1	1	3	3	2	5	3		7		7	64
80	17911A0584	5	1	1	1	4	4	2	5	2		7	7		61
81	17911A0585	5	1	1	1	2	2	2	5	3		7		7	26
82	17911A0586	5	1	1	2	5	5	2	5	2	4			4	55
83	17911A0587	5	1	1	2	5	5	2	5	2	5	5			59
84	17911A0589	5	1	1	1	3	3	1	5	5		7		7	67
85	17911A0590	5	2	2	2	5	5	1	5	4	7			7	56
86	17911A0591	5	2	2	2	5	5	1	5	2	5	5			59
87	17911A0594	5	1	1	1	3	3	1	5	2		6	5		31
88	17911A0595	5	2	2	2	5	5	1	5	1		5	5		56
89	17911A0596	5	2	2	2	5	5	1	5	4	7			7	58
90	17911A0597	5	1	1	1	4	4	1	5	2	4			4	11
91	17911A0598	5	1	1	2	5	5	2	5	2	4			4	35
92	17911A0599	5	1	1	1	3	3	2	5	4	7			7	63
93	17911A05A0	5	1	1	1	4	4	2	5	2		7	7		29
94	17911A05A2	5	2	2	2	5	5	2	5	6		7	7		57
95	17911A05A3	5	1	1	2	5	5	2	5	6		7	7		59
96	17911A05A5	5	1	1	1	5	5	1	5	3		7		7	42
97	17911A05A6	5	1	1	1	3	3	2	5	5		7		7	31
98	17911A05A7	5	1	1	1	4	4	2	5	6		7	7		64
99	17911A05A8	5	2	2	2	5	5	1	5	6		7	7		54
100	17911A05A9	5	1	1	1	4	4	1	5	5		7		7	63
101	17911A05B0	5	2	2	2	5	5	2	5	0	0	0			31
102	17911A05B1	5	1	1	1	4	4	2	5	2		6	6		55
103	17911A05B2	5	1	1	1	5	5	1	5	6		7	7		40
104	17911A05B3	5	1	1	1	5	5	2	5	2		6	6		65
105	17911A05B4	5	1	1	1	5	5	1	5	5		7		7	40
106	17911A05B6	5	2	2	2	5	5	1	5	6		7	7		59
107	17911A05B7	5	1	1	1	5	5	2	5	4	7			7	34
108	17911A05B8	5	1	1	2	5	5	2	5	2		6	5		38
109	17911A05B9	5	1	1	1	2	2	2	5	2		6	5		A
110	17911A05C0	5	1	1	1	5	5	1	5	2		6	5		56
111	17911A05C1	5	2	2	2	5	5	2	5	6		7	7		47
112	17911A05C2	5	1	1	1	2	2	2	5	5		7		7	26
113	17911A05C3	5	2	2	2	5	5	1	5	1		5	5		48
114	17911A05C4	5	1	1	2	1	5	2	5	6		7	7		61
115	17911A05C5	5	1	1	1	1	5	2	5	3		7		7	46
116	17911A05C7	5	1	1	1	1	5	1	5	2		6	6		47
117	17911A05C8	5	1	1	1	2	5	1	5	4	7			7	54
118	17911A05C9	5	2	2	2	2	5	1	5	6		7	7		37
119	17911A05D0	5	1	1	2	2	5	2	5	4	7			7	47
120	17911A05D1	5	1	1	2	2	5	2	5	2		2	2		53
121	17911A05D2	5	1	1	2	2	5	2	5	5		7		7	56
122	17911A05D3	5	2	2	2	1	5	1	5	4	7			7	67
123	17911A05D4	5	1	1	1	2	3	2	5	1	7		7		71
124	17911A05D5	5	2	2	2	2	5	4	5	6		7	7		64
125	17911A05D6	5	2	2	2	2	5	2	5	6		7	7		53
126	17911A05D7	5	2	2	2	1	5	4	5	4	7			7	63
127	17911A05D8	5	1	1	1	1	5	2	5	0		0		0	64
128	17911A05D9	5	1	1	1	1	4	2	5	2	5	5			71
129	17911A05E0	5	1	1	1	2	2	2	5	2	5	5			59
130	17911A05E1	5	2	2	2	5	5	1	5	6		7	7		68
131	17911A05E2	5	1	1	1	4	4	2	5	2		7	7		68
132	17911A05E3	5	1	1	1	1	5	2	5	5		7		7	67
133	17911A05E4	5	2	2	2	1	5	1	5	5		7		7	73
134	17911A05E5	5	1	1	2	1	5	2	5	5		7		7	70
135	17911A05E6	5	2	2	2	2	5	1	5	6		7	7		70
136	17911A05E7	5	1	1	1	2	4	2	5	4	7			7	72
137	17911A05E8	5	2	2	2	2	5	1	5	0	0		0		33
138	17911A05E9	5	2	2	2	2	5	4	5	5		7		7	38
139	17911A05F0	5	2	2	2	2	5	1	5	4	7			7	69
140	17911A05F1	5	2	2	2	1	5	3	5	6		7	7		52
141	17911A05F2	5	1	1	2	2	5	2	5	4	7			7	68
142	17911A05F3	5	2	2	2	2	5	1	5	5		7		7	46
143	17911A05F4	5	1	1	1	2	4	1	5	5		7		7	67
144	17911A05F5	5	2	2	2	1	5	2	5	6		7	7		72
145	17911A05F6	5	2	2	2	1	5	3	5	2		6	6		43
146	17911A05F7	5	2	2	2	1	5	3	5	4	7			7	62
147	17911A05F8	5	1	1	1	1	1	0	5	1		4	4		63
148	17911A05F9	5	1	1	1	4	4	1	5	2	5	5			67
149	17911A05G0	5	2	2	2	5	5	3	5	5		7		7	60
150	17911A05G1	5	2	2	2	5	5	4	5	6		7	7		46
151	17911A05G2	5	2	2	2	5	5	4	5	5		7		7	67
152	17911A05G3	5	2	2	2	5	5	1	5	6		7	7		75
153	17911A05G4	5	2	2	2	5	5	3	5	5		7		7	36
154	17911A05G5	5	1	1	1	2	2	1	5	6		7	7		46

155	17911A05G6	5	2	2	2	5	5	4	5	6		7	7		68
156	17911A05G7	5	2	2	2	5	5	3	5	4	7			7	49
157	17911A05G8	5	1	1	2	5	5	2	5	2		6	5		45
158	17911A05G9	5	1	1	1	1	5	1	5	4	7			7	39
159	17911A05H0	5	1	1	1	1	3	2	5	1		5	5		68
160	17911A05H1	5	1	1	1	1	5	2	5	4	7			7	69
161	17911A05H2	5	1	1	1	2	3	2	5	3		7		7	60
162	17911A05H3	5	1	1	1	2	5	1	5	5		7		7	57
163	17911A05H4	5	2	2	2	2	5	3	5	4	7			7	38
164	17911A05H5	5	2	2	2	2	5	4	5	2		6	5		45
165	17911A05H6	5	2	2	2	2	5	2	5	5		7		7	50
166	17911A05H7	5	2	2	2	1	5	2	5	6		7	7		63
167	17911A05H8	5	1	1	1	2	5	1	5	1		5	5		45
168	17911A05H9	5	2	2	2	2	5	2	5	4	7			7	47
169	17911A05J1	5	1	1	1	2	4	2	5	2	4			4	74
170	17911A05J2	5	2	2	2	1	5	3	5	5		7		7	55
171	17911A05J3	5	2	2	2	1	5	4	5	3		7		7	65
172	17911A05J4	5	1	1	2	1	5	2	5	2		7	7		75
173	17911A05J5	5	2	2	2	5	5	4	5	4	7			7	74
174	17911A05J6	5	2	2	2	5	5	4	5	5		7		7	65
175	17911A05J7	5	1	1	1	1	1	0	5	2		6	6		69
176	17911A05J8	5	1	1	1	4	4	1	5	2	3		3		61
177	17911A05J9	5	2	2	2	5	5	4	5	4	7			7	67
178	17911A05K0	5	2	2	2	5	5	4	5	5		7		7	63
179	17911A05K1	5	1	1	1	5	5	1	5	4	7			7	75
180	17911A05K2	5	2	2	2	5	5	1	5	6		7	7		64
181	17911A05K3	5	1	1	1	5	5	2	5	6		7		7	39
182	17911A05K4	5	1	1	1	3	3	2	5	6		7	7		61
183	17911A05K5	5	2	2	2	5	5	1	5	6		6	6		70
184	17911A05K6	5	1	1	1	5	5	2	5	6		5	5		40
185	17911A05K7	5	1	1	1	5	5	1	5	6	5	5			50
186	17911A05K8	5	2	2	2	5	5	3	5	6		7	7		73
187	17911A05K9	5	1	1	1	5	5	2	5	6	7			7	47
188	17911A05L0	5	1	1	1	5	5	1	5	6		6	5		57
189	17911A05L1	5	2	2	2	5	5	3	5	6		7		7	67
190	17911A05L2	5	2	2	2	5	5	1	5	6		6	6		57
191	17911A05L3	5	1	1	1	2	2	2	5	6	3		3		54
192	17911A05L4	5	2	2	2	5	5	1	5	6	7			7	68
193	17911A05L5	5	2	2	2	5	5	1	5	6	7			7	73
194	17911A05L6	5	1	1	2	5	5	2	5	6		6	6		53
195	17911A05L7	5	2	2	2	5	5	2	5	6	7		7		54
196	17911A05L8	5	2	2	2	5	5	3	5	6	7			7	53
197	17911A05L9	5	1	1	1	5	5	2	5	6	7			7	46
198	17911A05M0	5	2	2	2	5	5	4	5	6		7	7		48
199	17911A05M1	5	2	2	2	5	5	4	5	6	7			7	29
200	17911A05M2	5	1	1	2	5	5	4	5	6	5	5			44
201	17911A05M3	5	1	1	1	3	3	4	5	6	4			4	60
202	17911A05M4	5	1	1	1	3	3	4	5	6		6	5		49
203	17911A05M5	5	1	1	1	5	5	4	5	6		7	7		38
204	17911A05M6	5	1	1	2	5	5	4	5	6		7	7		49
205	17911A05M7	5	1	1	2	5	5	4	5	6		7	7		53
206	17911A05M8	5	1	1	1	4	4	4	5	6	7			7	41
207	17911A05M9	5	2	2	2	5	5	4	5	6	7			7	46
208	17911A05N0	5	1	1	1	4	4	4	5	6		6	5		44
209	17911A05N1	5	1	1	1	3	3	4	5	6	7			7	55
210	17911A05N2	5	1	1	1	2	2	4	5	6		6	5		44
211	17911A05N3	5	1	1	1	4	4	4	5	6		7	7		36
212	17911A05N5	5	2	2	2	5	5	4	5	6	7			7	44
213	17911A05N6	5	1	1	1	5	5	4	5	6	7			7	53
214	17911A05N7	5	1	1	1	5	5	4	5	6	7			7	43
215	17911A05N8	5	2	2	2	5	5	4	5	6	7			7	49
216	17911A05N9	5	2	2	2	5	5	4	5	6	7			7	34
217	17911A05P0	5	1	1	1	5	5	4	5	6	7		7		41
218	17911A05P2	5	1	1	1	4	4	4	5	6	7			7	44
219	17911A05P3	5	2	2	2	5	5	4	5	6	7			7	42
220	17911A05P4	5	2	2	2	5	5	4	5	6		6	6		32
221	17911A05P5	5	1	1	2	5	5	4	5	6	7			7	31
222	17911A05P6	5	1	1	1	0	0	4	5	6		3	3		41
223	17911A05P7	5	1	1	1	3	3	4	5	6		6	5		48
224	17911A05P8	5	1	1	1	4	4	4	5	6		6	6		47
225	17911A05P9	5	2	2	2	5	5	4	5	6		7		7	55
226	17911A05Q0	5	1	1	1	5	5	4	5	6		2	2		42
227	18915A0501	5	1	1	1	3	3	4	5	6	4			4	40
228	18915A0502	5	2	2	2	5	5	4	5	6	7			7	41
229	18915A0503	5	2	2	2	5	5	4	5	6		7		7	46
230	18915A0504	5	1	1	1	5	5	4	5	6		4	4		A
231	18915A0501	5	2	2	2	5	5	4	5	6		7	7		48
232	18915A0502	5	1	1	1	5	5	4	5	6		7	7		38
233	18915A0503	5	1	1	1	4	4	4	5	6	3		3		32
234	18915A0504	5	1	1	2	5	5	4	5	6		7		7	45
Average marks		5	1.346154	1.346154	1.448718	3.559829	4.183761	2.183761	5	4.025641	5.583333	5.9375	6.010526	6.786325	50.76724
No of students attemped		234	234	234	234	234	234	234	234	234	96	160	95	117	232
%of students scored 60%		100	94.017094	94.017094	94.017094	67.948718	87.179487	67.521368	100	70.08547	87.5	91.25	96.842105	99.145299	81.03
CO ATTAINMENT		3	3.0	3.0	3.0	2.0	3.0	2.0	3	3.0	3.0	3.0	3.0	3.0	3.0

ASSESSMENT OF COs FOR THE COURSE

CO	Method			value		CO Attainment (Internal)		CO Attainment (End Exam)	Overall CO Attainment
CO 1	ASM I			3	2.67	2.89		3.00	2.97
	MID I - PART A - Q1			3.0					
	MID I - PART B Q4			2.0					
CO 2	ASM I			3	3.00				
	MID I - PART A - Q2			3.0					
	MID I - PART B Q5			3.0					
CO 3	ASM I			3	2.8				
	ASM II			3.0					
	MID I - PART A - Q3			3.0					
	MID I - PART B Q6			2.0					
	MID II - PART A - Q1			3.0					
CO 4	ASM II			3	3				
	MID II - PART B - Q2			3.0					
	MID II - PART B- Q3			3.0					
CO 5	ASM II			3	3				
	MID II- PART B- Q4			3.0					
	MID II - PART B- Q5			3.0					

COURSE-COORDINATOR

HOD-CSE

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Course End Survey Form

COMPILER DESIGN

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

(Accredited by NBA, Approved by AICTE New Delhi & Permanently Affiliated to JNTUH)
Aziz Nagar Gate, C.B. Post, Hyderabad-500 075.

Department of Computer Science & Engineering

Course End Survey Form Academic year: 2019-20

Name of the student		Year &sem	III – II
Roll number		Regulations	R 15

Dear Student,

We need your help in evaluating the courses offered, by responding the short survey below.

Your feedback is very valuable for us in order to continually improve our program. The aim of this survey is to evaluate how well each of the courses has prepared you to have necessary skills.

Your responses will be kept confidential and will not be revealed to anyone outside the department without your permission.

Please indicate (√) the level to which you agree with the following criterion:
(3: Strongly agree 2: Agree 1: Strongly disagree)

Name of The Course: COMPILER DESIGN		RATING		
After completing this course the student must demonstrate the knowledge and ability to		3	2	1
CO 1	Differentiate the phases in compilation & parsing.			
CO 2	Identify the process in parsing and semantic analysis.			
CO 3	Explain about symbol tables and code optimization methods.			
CO 4	Explain about code optimization methods.			
CO 5	Analyze data flow and generate object code.			

Any other comments / suggestions: _____

_____

_____

_____

Signature

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Assessment Sheet – Co Wise (Direct Attainment)

COMPILER DESIGN

Vidya Jyothi Institute of Technology(Autonomous)

Department of Computer Science Engineering

Academic Year: 2018-19 BATCH: 2016-20

III B.Tech - II Sem

Course: CD

Faculty:

S.No	Reg.No	MID I Threshold 60%							MID II Threshold 60%							Threshold 60%
			PART-A			PART-B				PART-A			PART-B			
		ASM - I (5)	Q1(2M)	Q2(2M)	Q3 B (2M)	Q4(5M)	Q5(5M)	Q6(4M)	ASM - II (5)	Q1(2M)	Q2(2M)	Q3 A (2M)	Q4(4M)	Q5(5M)	Q6(5M)	End Exam (75M)
1	15911A0521	5	2	2	2	4	4	4	5	2	2	2	4	4	4	51
2	15911A0548	5	2	2	2	2	2	3	5	2	2	2	3	2	2	48
3	15911A0599	5	2	2	2	2	2	2	5	2	2	2	2	2	2	53
4	15911A05B9	5	2	2	2	2	2	2	5	2	2	2	2	2	2	8
5	15911A05D9	5	2	2	2	2	4	4	5	2	2	2	4	2	4	15
6	15911A05L4	5	2	2	2	3	3	3	5	2	2	2	3	3	3	60
7	15911A05L6	5	2	2	2	3	3	2	5	2	2	2	2	3	3	52
8	15911A05P9	5	2	2	2	4	4	4	5	2	2	2	4	4	4	48
9	16911A0501	5	2	2	2	5	5	4	5	2	2	2	4	5	5	59
10	16911A0502	5	2	2	2	3	3	3	5	2	2	2	1	3	3	57
11	16911A0503	5	2	2	2	1	3	3	5	2	2	2	3	1	3	45
12	16911A0505	5	2	2	2	3	3	4	5	2	2	2	4	3	3	48
13	16911A0506	5	2	2	2	4	2	4	5	2	2	2	4	4	5	54
14	16911A0507	5	2	2	2	5	3	4	5	2	2	2	4	5	5	61
15	16911A0508	5	2	2	2	3	3	3	5	2	2	2	1	3	3	58
16	16911A0509	5	2	2	2	1	3	3	5	2	2	2	3	1	3	61
17	16911A0510	5	2	2	2	3	3	4	5	2	2	2	4	3	3	60
18	16911A0511	5	2	2	2	4	5	4	5	2	2	2	4	4	5	48
19	16911A0512	5	2	2	2	3	3	3	5	2	2	2	3	3	3	54
20	16911A0513	5	2	2	2	3	3	2	5	2	2	2	2	3	3	47
21	16911A0514	5	2	2	2	4	4	4	5	2	2	2	4	4	4	51
22	16911A0515	5	2	2	2	3	3	3	5	2	2	2	3	3	3	49
23	16911A0516	5	2	2	2	4	4	4	5	2	2	2	4	4	4	51
24	16911A0517	5	2	2	2	2	2	3	5	2	2	2	3	2	2	47
25	16911A0518	5	2	2	2	2	2	2	5	2	2	2	2	2	2	58
26	16911A0519	5	2	2	2	2	2	2	5	2	2	2	2	2	2	55
27	16911A0520	5	2	2	2	2	4	4	5	2	2	2	4	2	4	45
28	16911A0521	5	2	2	2	3	3	3	5	2	2	2	3	3	3	60
29	16911A0522	5	2	2	2	3	3	2	5	2	2	2	2	3	3	49
30	16911A0523	5	2	2	2	4	4	4	5	2	2	2	4	4	4	53
31	16911A0524	5	2	2	2	5	5	4	5	2	2	2	4	5	5	55
32	16911A0525	5	2	2	2	3	3	3	5	2	2	2	1	3	3	48
33	16911A0526	5	2	2	2	1	3	3	5	2	2	2	3	1	3	54
34	16911A0527	5	2	2	2	3	3	4	5	2	2	2	4	3	3	17
35	16911A0528	5	2	2	2	4	2	4	5	2	2	2	4	4	5	49

36	16911A0529	5	2	2	2	5	3	4	5	2	2	2	4	5	5	60
37	16911A0530	5	2	2	2	3	3	3	5	2	2	2	1	3	3	58
38	16911A0532	5	2	2	2	1	3	3	5	2	2	2	3	1	3	56
39	16911A0533	5	2	2	2	3	3	4	5	2	2	2	4	3	3	48
40	16911A0534	5	2	2	2	4	5	4	5	2	2	2	4	4	5	61
41	16911A0535	5	2	2	2	3	3	3	5	2	2	2	3	3	3	49
42	16911A0536	5	2	2	2	3	3	2	5	2	2	2	2	3	3	48
43	16911A0537	5	2	2	2	4	4	4	5	2	2	2	4	4	4	45
44	16911A0538	5	2	2	2	3	3	3	5	2	2	2	3	3	3	54
45	16911A0540	5	2	2	2	4	4	4	5	2	2	2	4	4	4	49
46	16911A0541	5	2	2	2	2	2	3	5	2	2	2	3	2	2	56
47	16911A0542	5	2	2	2	2	2	2	5	2	2	2	2	2	2	52
48	16911A0543	5	2	2	2	2	2	2	5	2	2	2	2	2	2	48
49	16911A0544	5	2	2	2	2	4	4	5	2	2	2	4	2	4	52
50	16911A0545	5	1	1	1	3	3	3	5	1	1	1	3	3	3	54
51	16911A0546	5	2	2	2	3	3	2	5	2	2	2	2	3	3	59
52	16911A0548	5	0	0	0	4	4	4	5	0	0	0	4	4	4	45
53	16911A0549	5	0	0	0	5	5	4	5	0	0	0	4	5	5	60
54	16911A0550	5	0	0	0	3	3	3	5	0	0	0	1	3	3	34
55	16911A0551	5	0	0	0	1	3	3	5	0	0	0	3	1	3	40
56	16911A0552	5	0	0	0	3	3	4	5	0	0	0	4	3	3	42
57	16911A0553	5	0	0	0	4	2	4	5	0	0	0	4	4	5	34
58	16911A0554	5	2	2	2	5	3	4	5	2	2	2	4	5	5	28
59	16911A0555	5	1	1	1	3	3	3	5	1	1	1	1	3	3	37
60	16911A0556	5	1	1	1	1	3	3	5	1	1	1	3	1	3	50
61	16911A0558	5	1	1	1	3	3	4	5	1	1	1	4	3	3	47
62	16911A0559	5	0	0	0	4	5	4	5	0	0	0	4	4	5	45
63	16911A0560	5	0	0	0	3	3	3	5	0	0	0	3	3	3	29
64	16911A0561	5	0	0	0	3	3	2	5	0	0	0	2	3	3	32
65	16911A0562	5	2	2	2	4	4	4	5	2	2	2	4	4	4	45
66	16911A0563	5	2	2	2	3	3	3	5	2	2	2	3	3	3	30
67	16911A0564	5	2	2	2	4	4	4	5	2	2	2	4	4	4	45
68	16911A0565	5	0	0	0	2	2	3	5	0	0	0	3	2	2	28
69	16911A0566	5	0	0	0	2	2	2	5	0	0	0	2	2	2	47
70	16911A0567	5	2	2	2	2	2	2	5	2	2	2	2	2	2	33
71	16911A0568	5	2	2	2	2	4	4	5	2	2	2	4	2	4	50
72	16911A0569	5	2	2	2	3	3	3	5	2	2	2	3	3	3	29
73	16911A0570	5	2	2	2	3	3	2	5	2	2	2	2	3	3	45
74	16911A0572	5	2	2	2	4	4	4	5	2	2	2	4	4	4	58
75	16911A0573	5	2	2	2	5	5	4	5	2	2	2	4	5	5	32
76	16911A0575	5	2	2	2	3	3	3	5	2	2	2	1	3	3	15
77	16911A0576	5	2	2	2	1	3	3	5	2	2	2	3	1	3	47
78	16911A0577	5	2	2	2	3	3	4	5	2	2	2	4	3	3	55
79	16911A0579	5	2	2	2	4	2	4	5	2	2	2	4	4	5	42
80	16911A0580	5	2	2	2	5	3	4	5	2	2	2	4	5	5	28
81	16911A0581	5	2	2	2	3	3	3	5	2	2	2	1	3	3	57
82	16911A0582	5	2	2	2	1	3	3	5	2	2	2	3	1	3	12

83	16911A0583	5	2	2	2	3	3	4	5	2	2	2	4	3	3	41
84	16911A0584	5	2	2	2	4	5	4	5	2	2	2	4	4	5	43
85	16911A0585	5	2	2	2	3	3	3	5	2	2	2	3	3	3	49
86	16911A0586	5	2	2	2	3	3	2	5	2	2	2	2	3	3	43
87	16911A0587	5	1	1	1	4	4	4	5	1	1	1	4	4	4	48
88	16911A0588	5	1	1	1	3	3	3	5	1	1	1	3	3	3	42
89	16911A0590	5	1	1	1	4	4	4	5	1	1	1	4	4	4	37
90	16911A0591	5	1	1	1	2	2	3	5	1	1	1	3	2	2	49
91	16911A0592	5	1	1	1	2	2	2	5	1	1	1	2	2	2	56
92	16911A0593	5	1	1	1	2	2	2	5	1	1	1	2	2	2	36
93	16911A0594	5	0	0	0	2	4	4	5	0	0	0	4	2	4	45
94	16911A0595	5	0	0	0	3	3	3	5	0	0	0	3	3	3	37
95	16911A0596	5	0	0	0	3	3	2	5	0	0	0	2	3	3	26
96	16911A0597	5	0	0	0	4	4	4	5	0	0	0	4	4	4	7
97	16911A0598	5	0	0	0	5	5	4	5	0	0	0	4	5	5	32
98	16911A0599	5	2	2	2	3	3	3	5	2	2	2	1	3	3	31
99	16911A05A0	5	2	2	2	1	3	3	5	2	2	2	3	1	3	12
100	16911A05A1	5	2	2	2	3	3	4	5	2	2	2	4	3	3	35
101	16911A05A2	5	2	2	2	4	2	4	5	2	2	2	4	4	5	33
102	16911A05A3	5	2	2	2	5	3	4	5	2	2	2	4	5	5	26
103	16911A05A4	5	2	2	2	3	3	3	5	2	2	2	1	3	3	33
104	16911A05A6	5	2	2	2	1	3	3	5	2	2	2	3	1	3	33
105	16911A05A7	5	2	2	2	3	3	4	5	2	2	2	4	3	3	29
106	16911A05A8	5	2	2	2	4	5	4	5	2	2	2	4	4	5	29
107	16911A05A9	5	2	2	2	3	3	3	5	2	2	2	3	3	3	28
108	16911A05B0	5	2	2	2	3	3	2	5	2	2	2	2	3	3	34
109	16911A05B1	5	2	2	2	4	4	4	5	2	2	2	4	4	4	28
110	16911A05B2	5	2	2	2	3	3	3	5	2	2	2	3	3	3	29
111	16911A05B3	5	2	2	2	4	4	4	5	2	2	2	4	4	4	30
112	16911A05B4	5	2	2	2	2	2	3	5	2	2	2	3	2	2	33
113	16911A05B5	5	2	2	2	2	2	2	5	2	2	2	2	2	2	28
114	16911A05B6	5	2	2	2	2	2	2	5	2	2	2	2	2	2	44
115	16911A05B7	5	2	2	2	2	4	4	5	2	2	2	4	2	4	40
116	16911A05B8	5	2	2	2	3	3	3	5	2	2	2	3	3	3	40
117	16911A05B9	5	2	2	2	3	3	2	5	2	2	2	2	3	3	39
118	16911A05C0	5	2	2	2	4	4	4	5	2	2	2	4	4	4	28
119	16911A05C1	5	2	2	2	5	5	4	5	2	2	2	4	5	5	30
120	16911A05C2	5	1	1	1	3	3	3	5	1	1	1	1	3	3	18
121	16911A05C3	5	2	2	2	1	3	3	5	2	2	2	3	1	3	38
122	16911A05C4	5	0	0	0	3	3	4	5	0	0	0	4	3	3	42
123	16911A05C5	5	0	0	0	4	2	4	5	0	0	0	4	4	5	36
124	16911A05C6	5	0	0	0	5	3	4	5	0	0	0	4	5	5	38
125	16911A05C7	5	0	0	0	3	3	3	5	0	0	0	1	3	3	46
126	16911A05C9	5	0	0	0	1	3	3	5	0	0	0	3	1	3	41
127	16911A05D0	5	1	1	1	3	3	4	5	1	1	1	4	3	3	31
128	16911A05D1	5	2	2	2	4	5	4	5	2	2	2	4	4	5	33
129	16911A05D2	5	0	0	0	3	3	3	5	0	0	0	3	3	3	56

130	16911A05D3	5	0	0	0	3	3	2	5	0	0	0	2	3	3	39
131	16911A05D4	5	0	0	0	4	4	4	5	0	0	0	4	4	4	48
132	16911A05D5	5	0	0	0	3	3	3	5	0	0	0	3	3	3	49
133	16911A05D6	5	2	2	2	4	4	4	5	2	2	2	4	4	4	27
134	16911A05D7	5	2	2	2	2	2	3	5	2	2	2	3	2	2	33
135	16911A05D8	5	2	2	2	2	2	2	5	2	2	2	2	2	2	51
136	16911A05D9	5	1	1	1	2	2	2	5	1	1	1	2	2	2	44
137	16911A05E0	5	1	1	1	2	4	4	5	1	1	1	4	2	4	42
138	16911A05E1	5	1	1	1	3	3	3	5	1	1	1	3	3	3	31
139	16911A05E2	5	2	2	2	3	3	2	5	2	2	2	2	3	3	3
140	16911A05E3	5	2	2	2	4	4	4	5	2	2	2	4	4	4	41
141	16911A05E4	5	2	2	2	5	5	4	5	2	2	2	4	5	5	14
142	16911A05E5	5	0	0	0	3	3	3	5	0	0	0	1	3	3	33
143	16911A05E6	5	0	0	0	1	3	3	5	0	0	0	3	1	3	A
144	16911A05E7	5	0	0	0	3	3	4	5	0	0	0	4	3	3	33
145	16911A05E8	5	0	0	0	4	2	4	5	0	0	0	4	4	5	51
146	16911A05E9	5	0	0	0	5	3	4	5	0	0	0	4	5	5	60
147	16911A05F0	5	0	0	0	3	3	3	5	0	0	0	1	3	3	48
148	16911A05F1	5	2	2	2	1	3	3	5	2	2	2	3	1	3	51
149	16911A05F2	5	1	1	1	3	3	4	5	1	1	1	4	3	3	59
150	16911A05F3	5	1	1	1	4	5	4	5	1	1	1	4	4	5	65
151	16911A05F4	5	1	1	1	3	3	3	5	1	1	1	3	3	3	48
152	16911A05F6	5	1	1	1	3	3	2	5	1	1	1	2	3	3	58
153	16911A05F7	5	1	1	1	4	4	4	5	1	1	1	4	4	4	52
154	16911A05F8	5	2	2	2	3	3	3	5	2	2	2	3	3	3	55
155	16911A05F9	5	1	1	1	4	4	4	5	1	1	1	4	4	4	52
156	16911A05G0	5	1	1	1	2	2	3	5	1	1	1	3	2	2	60
157	16911A05G1	5	1	1	1	2	2	2	5	1	1	1	2	2	2	45
158	16911A05G2	5	1	1	1	2	2	2	5	1	1	1	2	2	2	62
159	16911A05G3	5	0	0	0	2	4	4	5	0	0	0	4	2	4	48
160	16911A05G4	5	0	0	0	3	3	3	5	0	0	0	3	3	3	56
161	16911A05G5	5	2	2	2	3	3	2	5	2	2	2	2	3	3	59
162	16911A05G6	5	0	0	0	4	4	4	5	0	0	0	4	4	4	54
163	16911A05G7	5	0	0	0	5	5	4	5	0	0	0	4	5	5	49
164	16911A05G8	5	2	2	2	3	3	3	5	2	2	2	1	3	3	53
165	16911A05G9	5	2	2	2	1	3	3	5	2	2	2	3	1	3	45
166	16911A05H0	5	2	2	2	3	3	4	5	2	2	2	4	3	3	59
167	16911A05H1	5	0	0	0	4	2	4	5	0	0	0	4	4	5	61
168	16911A05H2	5	0	0	0	5	3	4	5	0	0	0	4	5	5	A
169	16911A05H3	5	0	0	0	3	3	3	5	0	0	0	1	3	3	48
170	16911A05H4	5	0	0	0	1	3	3	5	0	0	0	3	1	3	56
171	16911A05H6	5	0	0	0	3	3	4	5	0	0	0	4	3	3	50
172	16911A05H7	5	2	2	2	4	5	4	5	2	2	2	4	4	5	48
173	16911A05H8	5	2	2	2	3	3	3	5	2	2	2	3	3	3	51
174	16911A05J0	5	2	2	2	3	3	2	5	2	2	2	2	3	3	48
175	16911A05J1	5	2	2	2	4	4	4	5	2	2	2	4	4	4	58
176	16911A05J2	5	2	2	2	3	3	3	5	2	2	2	3	3	3	57

177	16911A05J3	5	2	2	2	4	4	4	5	2	2	2	4	4	4	60
178	16911A05J4	5	2	2	2	2	2	3	5	2	2	2	3	2	2	15
179	16911A05J5	5	2	2	2	2	2	2	5	2	2	2	2	2	2	46
180	16911A05J6	5	2	2	2	2	2	2	5	2	2	2	2	2	2	48
181	16911A05J7	5	2	2	2	2	4	4	5	2	2	2	4	2	4	51
182	16911A05J8	5	2	2	2	3	3	3	5	2	2	2	3	3	3	49
183	16911A05J9	5	2	2	2	3	3	2	5	2	2	2	2	3	3	2
184	16911A05K0	5	2	2	2	4	4	4	5	2	2	2	4	4	4	48
185	16911A05K2	5	2	2	2	5	5	4	5	2	2	2	4	5	5	51
186	16911A05K3	5	2	2	2	3	3	3	5	2	2	2	1	3	3	49
187	16911A05K4	5	2	2	2	1	3	3	5	2	2	2	3	1	3	58
188	16911A05K5	5	2	2	2	3	3	4	5	2	2	2	4	3	3	56
189	16911A05K6	5	2	2	2	4	2	4	5	2	2	2	4	4	5	49
190	16911A05K7	5	2	2	2	5	3	4	5	2	2	2	4	5	5	66
191	16911A05K8	5	2	2	2	3	3	3	5	2	2	2	1	3	3	48
192	16911A05L0	5	2	2	2	1	3	3	5	2	2	2	3	1	3	45
193	16911A05L1	5	2	2	2	3	3	4	5	2	2	2	4	3	3	54
194	16911A05L2	5	2	2	2	4	5	4	5	2	2	2	4	4	5	48
195	16911A05L3	5	2	2	2	3	3	3	5	2	2	2	3	3	3	61
196	16911A05L4	5	2	2	2	3	3	2	5	2	2	2	2	3	3	54
197	16911A05L5	5	2	2	2	4	4	4	5	2	2	2	4	4	4	58
198	16911A05L6	5	2	2	2	3	3	3	5	2	2	2	3	3	3	49
199	16911A05L7	5	2	2	2	4	4	4	5	2	2	2	4	4	4	47
200	16911A05L8	5	2	2	2	2	2	3	5	2	2	2	3	2	2	49
201	16911A05L9	5	2	2	2	2	2	2	5	2	2	2	2	2	2	58
202	16911A05M0	5	2	2	2	2	2	2	5	2	2	2	2	2	2	54
203	16911A05M1	5	2	2	2	2	4	4	5	2	2	2	4	2	4	46
204	16911A05M2	5	2	2	2	3	3	3	5	2	2	2	3	3	3	45
205	16911A05M3	5	2	2	2	3	3	2	5	2	2	2	2	3	3	51
206	16911A05M4	5	2	2	2	4	4	4	5	2	2	2	4	4	4	49
207	16911A05M6	5	2	2	2	5	5	4	5	2	2	2	4	5	5	58
208	16911A05M7	5	1	1	1	3	3	3	5	1	1	1	1	3	3	9
209	16911A05M8	5	2	2	2	1	3	3	5	2	2	2	3	1	3	54
210	16911A05M9	5	0	0	0	3	3	4	5	0	0	0	4	3	3	49
211	16911A05N0	5	0	0	0	4	2	4	5	0	0	0	4	4	5	45
212	16911A05N1	5	0	0	0	5	3	4	5	0	0	0	4	5	5	52
213	16911A05N2	5	0	0	0	3	3	3	5	0	0	0	1	3	3	49
214	16911A05N3	5	0	0	0	1	3	3	5	0	0	0	3	1	3	53
215	16911A05N4	5	0	0	0	3	3	4	5	0	0	0	4	3	3	55
216	16911A05N5	5	2	2	2	4	5	4	5	2	2	2	4	4	5	50
217	16911A05N6	5	1	1	1	3	3	3	5	1	1	1	3	3	3	60
218	16911A05N7	5	1	1	1	3	3	2	5	1	1	1	2	3	3	45
219	16911A05N8	5	1	1	1	4	4	4	5	1	1	1	4	4	4	61
220	16911A05N9	5	1	1	1	3	3	3	5	1	1	1	3	3	3	5
221	16911A05P0	5	1	1	1	5	3	4	5	1	1	1	4	5	5	54
222	16911A05P1	5	2	2	2	3	3	3	5	2	2	2	1	3	3	46
223	16911A05P2	5	2	2	2	1	3	3	5	2	2	2	3	1	3	48

224	16911A05P3	5	2	2	2	3	3	4	5	2	2	2	4	3	3	62
225	16911A05P4	5	0	0	0	4	5	4	5	0	0	0	4	4	5	16
226	16911A05P5	5	0	0	0	3	3	3	5	0	0	0	3	3	3	45
227	16911A05P6	5	0	0	0	3	3	2	5	0	0	0	2	3	3	60
228	16911A05P7	5	2	2	2	4	4	4	5	2	2	2	4	4	4	52
229	16911A05P8	5	2	2	2	3	3	3	5	2	2	2	3	3	3	15
230	16911A05P9	5	2	2	2	3	3	3	5	2	2	2	3	3	3	48
231	16911A05Q0	5	2	2	2	3	3	3	5	2	2	2	3	3	3	65
232	17915A0501	5	2	2	2	3	3	3	5	2	2	2	3	3	3	60
233	17915A0502	5	2	2	2	3	3	3	5	2	2	2	3	3	3	18
234	17915A0503	5	2	2	2	3	3	3	5	2	2	2	3	3	3	48
235	17918A0501	5	2	2	2	3	3	3	5	2	2	2	3	3	3	59
Average marks		5	1.5	1.5	1.5	3.1	3.2	3.3	5	1.5	1.5	1.5	3.1	3.1	3.4	44.7
No of students attemped		**235**	**235**	**235**	**235**	**235**	**235**	**235**	**235**	**235**	**235**	**235**	**235**	**235**	**235**	**233**
No of students scored		**235**	156.00	156.00	156.00	174.00	195.00	194.00	**235**	156.00	156.00	156.00	173.00	174.00	205.00	155.00
%of students scored		**100**	66.38	66.38	66.38	74.04	82.98	82.55	**100**	66.38	66.38	66.38	73.62	74.04	87.23	66.52
CO ATTAINMENT		**3**	**2.0**	**2.0**	**2.0**	**3.0**	**3.0**	**3.0**	**3**	**2.0**	**2.0**	**2.0**	**3.0**	**3.0**	**3.0**	**2.0**

ASSESSMENT OF COs FOR THE COURSE							
CO	Method		value	Avg	CO Attainment	CO Attainment (End	Overall CO Attainment
CO 1	ASM I		3	2.7	2.67	2.00	2.17
	MID I - PART A - Q1		2.0				
	MID I - PART B - Q4		3.0				
CO 2	ASM I		3	2.7			
	MID I - PART A - Q2		2.0				
	MID I - PART B - Q5		3.0				
CO 3	ASM I		3	2.7			
	ASM II		3				
	MID I - PART A - Q3		2.0				
	MID I - PART B - Q6		3.0				
	MID II - PART A - Q1		2.0				
	MID II - PART B - Q4		3.0				
CO 4	ASM II		3	2.7			
	MID II - PART A - Q2		2.0				
	MID II - PART B - Q5		3.0				
CO 5	ASM II		3	2.7			
	MID II- PART A - Q3		2.0				
	MID II - PART B - Q6		3.0				

Vidya Jyothi Institute of Technology

(An Autonomous Institution)

(Accredited by NAAC. Approved by AICTE New Delhi & Permanently Affiliated to JNTUH)

Aziz Nagar Gate, C.B. Post, Hyderabad-500 075

Department of Computer Science & Engineering

(Accredited by NBA)

Academic Year: 2019-20 **BATCH: 2017-21**

III B.Tech- II Sem

Course: CD

Branch:CSE

S.No	Reg.No	MID I Threshold 60%							MID II Threshold 60%						Threshold 60%
			PART-A			PART-B				PART-A	PART-B				
		ASM - I (5)	Q1(2M)(CO1)	Q2(2M)(CO2)	Q3(2M)(CO3)	Q4(5M)(CO1)	Q5(5M)(CO2)	Q6(4M)(CO3)	ASM - II (5)	Q1(6M)(CO3)	Q2(7M)(CO4)	Q3(7M)(CO4)	Q4(7M)(CO5)	Q5(7M)(CO5)	End Exam (75M)
1	14911A0516	5	1	1	1	2	2	1	5	4	7			7	1
2	16911A0531	5	2	2	2	5	5	3	5	5		7		7	41
3	16911A0547	5	0	0	0	0	0	0	5	0	0	0			29
4	16911A0557	5	1	1	1	3	3	2	5	4	7			7	48
5	16911A05F5	5	1	1	1	2	2	1	5	4	7			7	54
6	16911A05H9	5	2	2	2	5	5	3	5	5		7		7	46
7	16911A05K9	5	0	0	0	0	0	0	5	0	0	0			52
8	16911A05M5	5	1	1	1	3	3	2	5	4	7			7	45
9	17601A0511	5	1	1	1	2	2	1	5	4	7			7	53
10	17601A0575	5	2	2	2	5	5	3	5	5		7		7	44
11	17911A0501	5	0	0	0	0	0	0	5	0	0	0			52
12	17911A0502	5	1	1	1	3	3	2	5	4	7			7	53
13	17911A0503	5	1	1	1	2	2	1	5	4	7			7	48
14	17911A0504	5	2	2	2	5	5	3	5	5		7		7	49
15	17911A0505	5	0	0	0	0	0	0	5	0	0	0			56
16	17911A0506	5	1	1	1	3	3	2	5	4	7			7	42
17	17911A0507	5	1	1	1	2	2	1	5	4	7			7	40
18	17911A0508	5	2	2	2	5	5	3	5	5		7		7	45
19	17911A0509	5	0	0	0	0	0	0	5	0	0	0			53
20	17911A0510	5	1	1	1	3	3	2	5	4	7			7	37
21	17911A0511	5	1	1	1	2	2	1	5	4	7			7	40
22	17911A0512	5	2	2	2	5	5	3	5	5		7		7	53
23	17911A0513	5	0	0	0	0	0	0	5	0	0	0			39
24	17911A0514	5	1	1	1	3	3	2	5	4	7			7	48
25	17911A0515	5	1	1	1	5	5	1	5	2		6	6		37
26	17911A0516	5	1	1	1	5	5	1	5	1	7		7		44
27	17911A0517	5	2	2	2	5	5	3	5	2		7	7		52
28	17911A0518	5	2	2	2	5	5	2	5	5		7		7	53
29	17911A0520	5	2	2	2	5	5	3	5	5		7		7	53
30	17911A0521	5	2	2	2	5	5	3	5	5		7		7	55

31	17911A0523	5	0	0	0	0	0	0	5	0	0	0			51
32	17911A0524	5	1	1	1	3	3	2	5	4	7			7	33
33	17911A0526	5	0	0	0	0	0	0	5	1		5	5		47
34	17911A0527	5	1	1	2	5	5	2	5	2	5	5			49
35	17911A0528	5	1	1	1	5	5	1	5	2		6	6		47
36	17911A0529	5	1	1	1	5	5	1	5	1	7		7		36
37	17911A0530	5	2	2	2	5	5	3	5	2		7	7		45
38	17911A0531	5	2	2	2	5	5	2	5	5		7		7	43
39	17911A0532	5	2	2	2	5	5	3	5	5		7		7	45
40	17911A0533	5	2	2	2	5	5	3	5	5		7		7	49
41	17911A0534	5	0	0	0	0	0	0	5	0	0	0			46
42	17911A0536	5	1	1	1	3	3	2	5	4	7			7	44
43	17911A0537	5	0	0	0	0	0	0	5	1		5	5		30
44	17911A0538	5	1	1	2	5	5	2	5	2	5	5			49
45	17911A0539	5	1	1	1	5	5	1	5	2		6	6		42
46	17911A0540	5	1	1	1	5	5	1	5	1	7		7		56
47	17911A0541	5	2	2	2	5	5	3	5	2		7	7		47
48	17911A0542	5	2	2	2	5	5	2	5	5		7		7	41
49	17911A0543	5	2	2	2	5	5	3	5	5		7		7	51
50	17911A0544	5	2	2	2	5	5	3	5	5		7		7	50
51	17911A0545	5	0	0	0	0	0	0	5	0	0	0			46
52	17911A0546	5	1	1	1	3	3	2	5	4	7			7	52
53	17911A0547	5	0	0	0	0	0	0	5	1		5	5		52
54	17911A0548	5	1	1	2	5	5	2	5	2	5	5			52
55	17911A0549	5	1	1	1	5	5	1	5	2		6	6		48
56	17911A0550	5	1	1	1	5	5	1	5	1	7		7		53
57	17911A0552	5	2	2	2	5	5	3	5	2		7	7		54
58	17911A0553	5	2	2	2	5	5	2	5	5		7		7	54
59	17911A0554	5	2	2	2	5	5	3	5	5		7		7	51
60	17911A0555	5	2	2	2	5	5	3	5	5		7		7	50
61	17911A0556	5	0	0	0	0	0	0	5	0	0	0			59
62	17911A0557	5	1	1	1	3	3	2	5	4	7			7	31
63	17911A0558	5	0	0	0	0	0	0	5	1		5	5		51
64	17911A0559	5	1	1	2	5	5	2	5	2	5	5			53
65	17911A0560	5	1	1	1	5	5	1	5	2		6	6		33
66	17911A0561	5	1	1	1	5	5	1	5	1	7		7		40
67	17911A0562	5	2	2	2	5	5	3	5	2		7	7		46
68	17911A0563	5	2	2	2	5	5	2	5	5		7		7	44
69	17911A0564	5	2	2	2	5	5	3	5	5		7		7	36
70	17911A0565	5	2	2	2	5	5	4	5	6		7	7		43
71	17911A0566	5	2	2	2	5	5	3	5	5		7		7	53
72	17911A0567	5	1	1	1	3	3	1	5	2		7	7		34
73	17911A0569	5	2	2	2	5	5	3	5	5		7		7	55
74	17911A0570	5	1	1	2	5	5	2	5	1	7		7		52
75	17911A0571	5	2	2	2	5	5	2	5	6		7	7		61
76	17911A0572	5	1	1	1	5	5	1	5	3		7		7	58
77	17911A0573	5	2	2	2	5	5	4	5	3		7		7	58

78	17911A0574	5	1	1	1	5	5	1	5	4	7			7	62
79	17911A0575	5	1	1	1	3	3	2	5	3		7		7	64
80	17911A0576	5	1	1	1	4	4	2	5	2		7	7		61
81	17911A0577	5	1	1	1	2	2	2	5	3		7		7	26
82	17911A0578	5	1	1	2	5	5	2	5	2	4			4	55
83	17911A0579	5	1	1	2	5	5	2	5	2	5	5			59
84	17911A0580	5	1	1	1	3	3	1	5	5		7		7	67
85	17911A0581	5	2	2	2	5	5	1	5	4	7			7	56
86	17911A0582	5	2	2	2	5	5	1	5	2	5	5			59
87	17911A0583	5	1	1	1	3	3	1	5	2		6	5		31
88	17911A0584	5	2	2	2	5	5	1	5	1		5	5		56
89	17911A0585	5	2	2	2	5	5	1	5	4	7			7	58
90	17911A0586	5	1	1	1	4	4	1	5	2	4			4	11
91	17911A0587	5	1	1	2	5	5	2	5	2	4			4	35
92	17911A0589	5	1	1	1	3	3	2	5	4	7			7	63
93	17911A0590	5	1	1	1	4	4	2	5	2		7	7		29
94	17911A0591	5	2	2	2	5	5	2	5	6		7	7		57
95	17911A0594	5	1	1	2	5	5	2	5	6		7	7		59
96	17911A0595	5	1	1	1	5	5	1	5	3		7		7	42
97	17911A0596	5	1	1	1	3	3	2	5	5		7		7	31
98	17911A0597	5	1	1	1	4	4	2	5	6		7	7		64
99	17911A0598	5	2	2	2	5	5	1	5	6		7	7		54
100	17911A0599	5	1	1	1	4	4	1	5	5		7		7	63
101	17911A05A0	5	2	2	2	5	5	2	5	0	0	0			31
102	17911A05A2	5	1	1	1	4	4	2	5	2		6	6		55
103	17911A05A3	5	1	1	1	5	5	1	5	6		7	7		40
104	17911A05A5	5	1	1	1	5	5	2	5	2		6	6		65
105	17911A05A6	5	1	1	1	5	5	1	5	5		7		7	40
106	17911A05A7	5	2	2	2	5	5	1	5	6		7	7		59
107	17911A05A8	5	1	1	1	5	5	2	5	4	7			7	34
108	17911A05A9	5	1	1	2	5	5	2	5	2		6	5		38
109	17911A05B0	5	1	1	1	2	2	2	5	2		6	5		A
110	17911A05B1	5	1	1	1	5	5	1	5	2		6	5		56
111	17911A05B2	5	2	2	2	5	5	2	5	6		7	7		47
112	17911A05B3	5	1	1	1	2	2	2	5	5		7		7	26
113	17911A05B4	5	2	2	2	5	5	1	5	1		5	5		48
114	17911A05B6	5	1	1	2	1	5	2	5	6		7	7		61
115	17911A05B7	5	1	1	1	1	5	2	5	3		7		7	46
116	17911A05B8	5	1	1	1	1	5	1	5	2		6	6		47
117	17911A05B9	5	1	1	1	2	5	1	5	4	7			7	54
118	17911A05C0	5	2	2	2	2	5	1	5	6		7	7		37
119	17911A05C1	5	1	1	2	2	5	2	5	4	7			7	47
120	17911A05C2	5	1	1	2	2	5	2	5	2		2	2		53
121	17911A05C3	5	1	1	2	2	5	2	5	5		7		7	56
122	17911A05C4	5	2	2	2	1	5	1	5	4	7			7	67
123	17911A05C5	5	1	1	1	2	3	2	5	1	7		7		71
124	17911A05C7	5	2	2	2	2	5	4	5	6		7	7		64

125	17911A05C8	5	2	2	2	2	5	2	5	6		7	7		53
126	17911A05C9	5	2	2	2	1	5	4	5	4	7			7	63
127	17911A05D0	5	1	1	1	1	5	2	5	0		0		0	64
128	17911A05D1	5	1	1	1	1	4	2	5	2	5	5			71
129	17911A05D2	5	1	1	1	2	2	2	5	2	5	5			59
130	17911A05D3	5	2	2	2	5	5	1	5	6		7	7		68
131	17911A05D4	5	1	1	1	4	4	2	5	2		7	7		68
132	17911A05D5	5	1	1	1	1	5	2	5	5		7		7	67
133	17911A05D6	5	2	2	2	1	5	1	5	5		7		7	73
134	17911A05D7	5	1	1	2	1	5	2	5	5		7		7	70
135	17911A05D8	5	2	2	2	2	5	1	5	6		7	7		70
136	17911A05D9	5	1	1	1	2	4	2	5	4	7			7	72
137	17911A05E0	5	2	2	2	2	5	1	5	0	0		0		33
138	17911A05E1	5	2	2	2	2	5	4	5	5		7		7	38
139	17911A05E2	5	2	2	2	2	5	1	5	4	7			7	69
140	17911A05E3	5	2	2	2	1	5	3	5	6		7	7		52
141	17911A05E4	5	1	1	2	2	5	2	5	4	7			7	68
142	17911A05E5	5	2	2	2	2	5	1	5	5		7		7	46
143	17911A05E6	5	1	1	1	2	4	1	5	5		7		7	67
144	17911A05E7	5	2	2	2	1	5	2	5	6		7	7		72
145	17911A05E8	5	2	2	2	1	5	3	5	2		6	6		43
146	17911A05E9	5	2	2	2	1	5	3	5	4	7			7	62
147	17911A05F0	5	1	1	1	1	1	0	5	1		4	4		63
148	17911A05F1	5	1	1	1	4	4	1	5	2	5	5			67
149	17911A05F2	5	2	2	2	5	5	3	5	5		7		7	60
150	17911A05F3	5	2	2	2	5	5	4	5	6		7	7		46
151	17911A05F4	5	2	2	2	5	5	4	5	5		7		7	67
152	17911A05F5	5	2	2	2	5	5	1	5	6		7	7		75
153	17911A05F6	5	2	2	2	5	5	3	5	5		7		7	36
154	17911A05F7	5	1	1	1	2	2	1	5	6		7	7		46
155	17911A05F8	5	2	2	2	5	5	4	5	6		7	7		68
156	17911A05F9														
157	17911A05G0														
158	17911A05G1														
159	17911A05G2														
160	17911A05G3														
161	17911A05G4														
162	17911A05G5														
163	17911A05G6														
164	17911A05G7														
165	17911A05G8														
166	17911A05G9	5	2	2	2	5	5	3	5	4	7			7	49
167	17911A05H0	5	1	1	2	5	5	2	5	2		6	5		45
168	17911A05H1	5	1	1	1	1	5	1	5	4	7			7	39
169	17911A05H2	5	1	1	1	1	3	2	5	1		5	5		68
170	17911A05H3	5	1	1	1	1	5	2	5	4	7			7	69
171	17911A05H4	5	1	1	1	2	3	2	5	3		7		7	60

172	17911A05H5	5	1	1	1	2	5	1	5	5		7		7	57
173	17911A05H6	5	2	2	2	2	5	3	5	4	7			7	38
174	17911A05H7	5	2	2	2	2	5	4	5	2		6	5		45
175	17911A05H8	5	2	2	2	2	5	2	5	5		7		7	50
176	17911A05H9	5	2	2	2	1	5	2	5	6		7	7		63
177	17911A05J1	5	1	1	1	2	5	1	5	1		5	5		45
178	17911A05J2	5	2	2	2	2	5	2	5	4	7			7	47
179	17911A05J3	5	1	1	1	2	4	2	5	2	4			4	74
180	17911A05J4	5	2	2	2	1	5	3	5	5		7		7	55
181	17911A05J5	5	2	2	2	1	5	4	5	3		7		7	65
182	17911A05J6	5	1	1	2	1	5	2	5	2		7	7		75
183	17911A05J7	5	2	2	2	5	5	4	5	4	7			7	74
184	17911A05J8	5	2	2	2	5	5	4	5	5		7		7	65
185	17911A05J9	5	1	1	1	1	1	0	5	2		6	6		69
186	17911A05K0	5	1	1	1	4	4	1	5	2	3		3		61
187	17911A05K1	5	2	2	2	5	5	4	5	4	7			7	67
188	17911A05K2	5	2	2	2	5	5	4	5	5		7		7	63
189	17911A05K3	5	1	1	1	5	5	1	5	4	7			7	75
190	17911A05K4	5	2	2	2	5	5	1	5	6		7	7		64
191	17911A05K5	5	1	1	1	5	5	2	5	6		7		7	39
192	17911A05K6	5	1	1	1	3	3	2	5	6		7	7		61
193	17911A05K7	5	2	2	2	5	5	1	5	6		6	6		70
194	17911A05K8	5	1	1	1	5	5	2	5	6		5	5		40
195	17911A05K9	5	1	1	1	5	5	1	5	6	5	5			50
196	17911A05L0	5	2	2	2	5	5	3	5	6		7	7		73
197	17911A05L1	5	1	1	1	5	5	2	5	6	7			7	47
198	17911A05L2	5	1	1	1	5	5	1	5	6		6	5		57
199	17911A05L3	5	2	2	2	5	5	3	5	6		7		7	67
200	17911A05L4	5	2	2	2	5	5	1	5	6		6	6		57
201	17911A05L5	5	1	1	1	2	2	2	5	6	3		3		54
202	17911A05L6	5	2	2	2	5	5	1	5	6	7			7	68
203	17911A05L7	5	2	2	2	5	5	1	5	6	7			7	73
204	17911A05L8	5	1	1	2	5	5	2	5	6		6	6		53
205	17911A05L9	5	2	2	2	5	5	2	5	6	7		7		54
206	17911A05M0	5	2	2	2	5	5	3	5	6	7			7	53
207	17911A05M1	5	1	1	1	5	5	2	5	6	7			7	46
208	17911A05M2	5	2	2	2	5	5	4	5	6		7	7		48
209	17911A05M3	5	2	2	2	5	5	4	5	6	7			7	29
210	17911A05M4	5	1	1	2	5	5	4	5	6	5	5			44
211	17911A05M5	5	1	1	1	3	3	4	5	6	4			4	60
212	17911A05M6	5	1	1	1	3	3	4	5	6		6	5		49
213	17911A05M7	5	1	1	1	5	5	4	5	6		7	7		38
214	17911A05M8	5	1	1	2	5	5	4	5	6		7	7		49
215	17911A05M9	5	1	1	2	5	5	4	5	6		7	7		53
216	17911A05N0	5	1	1	1	4	4	4	5	6	7			7	41
217	17911A05N1	5	2	2	2	5	5	4	5	6	7			7	46
218	17911A05N2	5	1	1	1	4	4	4	5	6		6	5		44

219	17911A05N3	5	1	1	1	3	3	4	5	6	7			7	55
220	17911A05N5	5	1	1	1	2	2	4	5	6		6	5		44
221	17911A05N6	5	1	1	1	4	4	4	5	6		7	7		36
222	17911A05N7	5	2	2	2	5	5	4	5	6	7			7	44
223	17911A05N8	5	1	1	1	5	5	4	5	6	7			7	53
224	17911A05N9	5	1	1	1	5	5	4	5	6	7			7	43
225	17911A05P0	5	2	2	2	5	5	4	5	6	7			7	49
226	17911A05P2	5	2	2	2	5	5	4	5	6	7			7	34
227	17911A05P3	5	1	1	1	5	5	4	5	6	7		7		41
228	17911A05P4	5	1	1	1	4	4	4	5	6	7			7	44
229	17911A05P5	5	2	2	2	5	5	4	5	6	7			7	42
230	17911A05P6	5	2	2	2	5	5	4	5	6		6	6		32
231	17911A05P7	5	1	1	2	5	5	4	5	6	7			7	31
232	17911A05P8	5	1	1	1	0	0	4	5	6		3	3		41
233	17911A05P9	5	1	1	1	3	3	4	5	6		6	5		48
234	17911A05Q0	5	1	1	1	4	4	4	5	6		6	6		47
235	18915A0501	5	2	2	2	5	5	4	5	6		7		7	55
236	18915A0502	5	1	1	1	5	5	4	5	6		2	2		42
237	18915A0503	5	1	1	1	3	3	4	5	6	4			4	40
238	18915A0504	5	2	2	2	5	5	4	5	6	7			7	41
Average marks		5	1.346491	1.346491	1.447368	3.526316	4.166667	2.135965	5	3.973684	5.610526	5.922581	6.043956	6.782609	50.96476
No of students attemped		228	228	228	228	228	228	228	228	228	95	155	91	115	227
%of students scored 60%		100	93.859649	93.859649	93.859649	67.105263	86.842105	66.666667	100	69.298246	87.368421	90.967742	96.703297	99.130435	81.50
CO ATTAINMENT		3	3.0	3.0	3.0	2.0	3.0	2.0	3	2.0	3.0	3.0	3.0	3.0	3.0

ASSESSMENT OF COs FOR THE COURSE									
CO	Method			value		CO Attainment (Internal)		CO Attainment (End Exam)	Overall CO Attainment
CO 1	ASM I			3	2.67	2.85		3.00	2.96
	MID I - PART A Q1			3.0					
	MID I - PART B Q4			2.0					
CO 2	ASM I			3	3.00				
	MID I - PART A Q2			3.0					
	MID I - PART B Q5			3.0					
CO 3	ASM I			3	2.6				
	ASM II			3.0					
	MID I - PART A Q3			3.0					
	MID I - PART B Q6			2.0					
	MID II - PART A - Q1			2.0					
CO 4	ASM II			3	3				
	MID II - PART B - Q2			3.0					
	MID II - PART B Q3			3.0					
CO 5	ASM II			3	3				
	MID II- PART B Q4			3.0					
	MID II - PART B Q5			3.0					

COURSE-COORDINATOR

HOD-CSE

Vidya Jyothi Institute of Technology(Autonomous)

Department of Computer Science Engineering

Academic Year: 2020-21 BATCH: 2018-2022

III B.Tech - II Sem

Course: CD

Faculty:

S.No	Reg.No	MID I Threshold 60%							MID II Threshold 60%							Threshold 60%
			PART-A			PART-B				PART-A			PART-B			
		ASM - I (5)	Q1(2M)	Q2(2M)	Q3 B (2M)	Q4(5M)	Q5(5M)	Q6(4M)	ASM - II (5)	Q1(2M)	Q2(2M)	Q3 A (2M)	Q4(4M)	Q5(5M)	Q6(5M)	End Exam (75M)
1	16911A0574	5	2	2	2	4	4	4	5	2	2	1	4	4	2	10
2	17911A0551	5	2	2	2	2	2	3	5	2	0	1	3	2	3	8
3	17911A0568	5	2	2	2	2	2	2	5	2	0	1	2	2	2	14
4	17911A0593	5	2	2	0	2	2	2	5	2	0	1	2	2	2	50
5	17911A05A1	5	2	2	2	2	4	4	5	2	0	1	4	2	2	48
6	18911A0501	5	2	2	2	3	3	3	5	2	0	2	3	3	2	65
7	18911A0502	5	2	2	2	3	3	2	5	2	1	2	2	3	2	73
8	18911A0503	5	2	2	2	4	4	4	5	2	1	2	4	4	4	53
9	18911A0504	5	2	2	2	5	5	4	5	1	1	2	4	5	0	45
10	18911A0505	5	2	2	2	3	3	3	5	1	1	2	1	3	0	62
11	18911A0506	5	2	2	2	1	3	3	5	1	1	2	3	1	0	73
12	18911A0507	5	2	2	2	3	3	4	5	2	0	2	4	3	3	56
13	18911A0508	5	2	2	2	4	2	4	5	2	0	2	4	4	5	53
14	18911A0509	5	2	2	2	5	3	4	5	2	0	2	4	5	5	62
15	18911A0510	5	2	2	2	3	3	3	5	2	0	2	1	3	3	72
16	18911A0511	5	2	2	2	1	3	3	5	2	0	2	3	1	3	59
17	18911A0512	5	2	2	2	3	3	4	5	2	1	2	4	3	3	49
18	18911A0513	5	2	2	2	4	5	4	5	2	1	2	4	4	5	54
19	18911A0515	5	2	2	2	3	3	3	5	1	1	2	3	3	3	73
20	18911A0516	5	2	2	2	3	3	2	5	1	1	2	2	3	3	8
21	18911A0518	5	2	2	2	4	4	4	5	2	2	2	4	4	4	72
22	18911A0519	5	2	2	2	4	4	4	5	2	2	2	4	4	4	75
23	18911A0521	5	2	2	2	5	5	4	5	1	2	2	4	5	5	48
24	18911A0522	5	2	2	2	3	3	3	5	1	2	2	1	3	3	61
25	18911A0523	5	2	2	2	1	3	3	5	1	2	2	3	1	3	58
26	18911A0524	5	2	2	2	3	3	4	5	2	2	2	4	3	3	62
27	18911A0525	5	2	2	2	4	2	4	5	2	2	2	4	4	5	52
28	18911A0527	5	2	2	2	5	3	4	5	2	2	2	4	5	5	73
29	18911A0528	5	2	2	2	3	3	3	5	2	2	2	1	3	3	48
30	18911A0529	5	2	2	2	1	3	3	5	2	2	2	3	1	3	46
53	18911A0530	5	2	2	2	3	3	4	5	2	0	2	4	3	3	56
32	18911A0531	5	2	2	2	4	5	4	5	2	0	2	4	4	5	49
33	18911A0532	5	2	2	2	3	3	3	5	1	0	2	3	3	3	51
34	18911A0533	5	2	2	2	3	3	2	5	1	0	2	2	3	3	46
35	18911A0534	5	2	2	2	4	4	4	5	2	0	2	4	4	4	71

57	18911A0535	5	2	2	2	4	4	4	5	2	1	2	4	4	4	65	
59	18911A0536	5	2	2	2	5	5	4	5	1	1	2	4	5	5	69	
58	18911A0537	5	2	2	2	3	3	3	5	1	1	2	1	3	3	56	
51	18911A0538	5	2	2	2	1	3	3	5	1	1	2	3	1	3	60	
40	18911A0539	5	2	2	2	3	3	4	5	2	1	2	4	3	3	A	
41	18911A0540	5	2	2	2	4	2	4	5	2	0	2	4	4	5	59	
42	18911A0541	5	2	2	2	5	3	4	5	2	0	2	4	5	5	75	
43	18911A0542	5	2	2	2	3	3	3	5	2	0	2	1	3	3	52	
56	18911A0543	5	2	2	2	1	3	3	5	2	0	2	3	1	3	75	
45	18911A0544	5	2	2	2	3	3	4	5	2	0	2	4	3	3	57	
46	18911A0546	5	2	2	2	4	5	4	5	2	1	2	4	4	5	48	
47	18911A0547	5	2	2	2	3	3	3	5	1	1	2	3	3	3	62	
48	18911A0548	5	2	2	2	3	3	2	5	1	1	2	2	3	3	68	
49	18911A0549	5	2	2	2	4	4	4	5	2	1	2	4	4	4	63	
50	18911A0550	5	2	2	2	4	4	4	5	2	2	2	4	4	4	57	
51	18911A0551	5	2	2	2	5	5	4	5	1	2	2	4	5	5	68	
52	18911A0552	5	2	2	2	3	3	3	5	1	2	2	1	3	3	49	
53	18911A0553	5	2	2	2	1	3	3	5	1	0	2	3	1	3	61	
54	18911A0554	5	2	2	2	3	3	4	5	2	0	2	4	3	3	70	
55	18911A0555	5	2	2	2	4	2	4	5	2	0	2	4	4	5	74	
56	18911A0556	5	2	2	2	5	3	4	5	2	0	2	4	5	5	59	
57	18911A0557	5	2	2	2	3	3	3	5	2	0	2	1	3	3	73	
58	18911A0559	5	2	2	2	1	3	3	5	2	1	2	3	1	3	53	
59	18911A0560	5	2	2	2	3	3	4	5	2	1	2	4	3	3	55	
60	18911A0561	5	2	2	2	4	5	4	5	2	1	2	4	4	5	48	
61	18911A0562	5	2	2	2	3	3	3	5	1	1	2	3	3	3	46	
62	18911A0563	5	2	2	2	3	3	2	5	1	1	2	2	3	3	50	
63	18911A0564	5	2	2	2	4	4	4	5	2	0	2	4	4	4	49	
64	18911A0565	5	2	2	2	4	4	4	5	2	0	2	4	4	4	71	
65	18911A0566	5	2	2	2	5	5	4	5	1	0	2	4	5	5	52	
66	18911A0567	5	2	2	2	3	3	3	5	1	0	2	1	3	3	46	
67	18911A0568	5	2	2	2	1	3	3	5	1	0	2	3	1	3	53	
68	18911A0569	5	2	2	2	3	3	4	5	2	1	2	4	3	3	56	
69	18911A0570	5	2	2	2	4	2	4	5	2	1	2	4	4	5	47	
70	18911A0571	5	2	2	2	5	3	4	5	2	1	2	4	5	5	64	
71	18911A0572	5	2	2	2	3	3	3	5	2	1	2	1	3	3	56	
72	18911A0573	5	2	2	2	1	3	3	5	2	2	2	3	1	3	62	
73	18911A0574	5	2	2	2	3	3	4	5	2	2	2	4	3	3	55	
74	18911A0575	5	2	2	2	4	5	4	5	2	2	2	4	4	5	61	
75	18911A0576	5	2	2	2	3	3	3	5	1	2	2	3	3	3	45	
76	18911A0577	5	2	2	2	3	3	2	5	1	0	2	2	3	3	61	
77	18911A0578	5	2	2	2	4	4	4	5	2	0	2	4	4	4	69	
78	18911A0579	5	2	2	2	4	4	4	5	2	0	2	4	4	4	64	
79	18911A0580	5	2	2	2	5	5	4	5	1	0	2	4	5	5	58	
80	18911A0581	5	2	2	2	3	3	3	5	1	0	2	1	3	3	51	
81	18911A0582	5	2	2	2	1	3	3	5	1	1	2	3	1	3	65	
82	18911A0583	5	2	2	2	3	3	4	5	2	1	2	4	3	3	61	

83	18911A0584	5	1	2	2	4	2	4	5	2	1	2	4	4	5	49	
84	18911A0585	5	1	2	2	5	3	4	5	2	1	2	4	5	5	49	
85	18911A0586	5	1	2	2	3	3	3	5	2	1	2	1	3	3	58	
86	18911A0587	5	0	2	2	1	3	3	5	2	0	2	3	1	3	45	
87	18911A0588	5	0	2	2	3	3	4	5	2	0	2	4	3	3	57	
88	18911A0589	5	0	2	2	4	5	4	5	2	0	2	4	4	5	56	
89	18911A0590	5	0	2	2	3	3	3	5	1	0	2	3	3	3	45	
90	18911A0591	5	0	2	2	3	3	2	5	1	0	2	2	3	3	52	
91	18911A0592	5	1	2	2	4	4	4	5	2	1	2	4	4	4	46	
92	18911A0593	5	1	2	2	4	4	4	5	2	1	2	4	4	4	55	
93	18911A0594	5	1	2	2	5	5	4	5	1	1	2	4	5	5	50	
94	18911A0595	5	1	2	2	3	3	3	5	1	1	2	1	3	3	61	
95	18911A0596	5	2	2	2	1	3	3	5	1	2	2	3	1	3	46	
96	18911A0597	5	2	2	2	3	3	4	5	2	2	2	4	3	3	48	
97	18911A0598	5	2	2	2	4	2	4	5	2	2	2	4	4	5	53	
98	18911A0599	5	1	2	2	5	3	4	5	2	2	2	4	5	5	45	
99	18911A05A0	5	1	2	2	3	3	3	5	2	0	2	1	3	3	51	
100	18911A05A1	5	1	2	2	1	3	3	5	2	0	2	3	1	3	40	
101	18911A05A2	5	0	2	2	3	3	4	5	2	0	2	4	3	3	56	
102	18911A05A3	5	0	2	2	4	5	4	5	2	0	2	4	4	5	53	
103	18911A05A4	5	0	2	2	3	3	3	5	1	0	2	3	3	3	57	
104	18911A05A5	5	0	2	2	3	3	2	5	1	1	2	2	3	3	54	
105	18911A05A6	5	0	2	2	4	4	4	5	2	1	2	4	4	4	69	
106	18911A05A7	5	1	2	2	4	4	4	5	2	1	2	4	4	4	48	
107	18911A05A8	5	1	2	2	5	5	4	5	1	1	2	4	5	5	51	
108	18911A05A9	5	1	2	2	3	3	3	5	1	1	2	1	3	3	55	
109	18911A05B0	5	1	2	2	1	3	3	5	1	0	2	3	1	3	59	
110	18911A05B1	5	2	2	2	3	3	4	5	2	0	2	4	3	3	47	
111	18911A05B2	5	2	2	2	4	2	4	5	0	0	2	4	4	5	59	
112	18911A05B3	5	2	2	2	5	3	4	5	0	0	2	4	5	5	71	
113	18911A05B4	5	1	2	2	3	3	3	5	0	0	2	1	3	3	30	
114	18911A05B6	5	1	2	2	1	3	3	5	0	1	2	3	1	3	57	
115	18911A05B7	5	1	2	2	3	3	4	5	0	1	2	4	3	3	69	
116	18911A05B8	5	0	2	2	4	5	4	5	1	1	2	4	4	5	50	
117	18911A05B9	5	0	2	2	3	3	3	5	1	1	2	3	3	3	66	
118	18911A05C1	5	0	2	2	3	3	2	5	1	2	2	2	3	3	64	
119	18911A05C2	5	0	2	2	4	4	4	5	1	2	2	4	4	4	67	
120	18911A05C3	5	0	2	2	3	3	3	5	1	0	2	3	3	3	65	
121	18911A05C4	5	1	2	2	4	4	4	5	0	0	1	4	4	4	35	
122	18911A05C5	5	1	2	2	2	2	3	5	0	0	1	3	2	2	58	
123	18911A05C6	5	1	2	2	2	2	2	5	0	0	1	2	2	2	30	
124	18911A05C7	5	1	2	0	2	2	2	5	0	0	1	2	2	2	51	
125	18911A05C8	5	2	1	1	2	4	4	5	0	1	1	4	2	4	27	
126	18911A05C9	5	1	1	1	3	3	3	5	1	1	2	3	3	3	35	
127	18911A05D0	5	2	1	1	3	3	2	5	1	1	2	2	3	3	53	
128	18911A05D1	5	1	0	0	4	4	4	5	1	1	2	4	4	4	48	
129	18911A05D2	5	1	0	0	5	5	4	5	1	1	2	4	5	5	34	

130	18911A05D4	5	1	0	0	3	3	3	5	1	0	2	1	3	3	47
153	18911A05D5	5	0	0	0	1	3	3	5	1	0	2	3	1	3	32
132	18911A05D7	5	0	0	0	3	3	4	5	2	0	2	4	3	3	51
133	18911A05D8	5	0	1	1	4	2	4	5	2	0	2	4	4	5	52
134	18911A05D9	5	0	1	1	5	3	4	5	0	0	2	4	5	5	47
135	18911A05E0	5	0	1	1	3	3	3	5	0	1	2	1	3	3	17
157	18911A05E1	5	1	1	1	1	3	3	5	0	1	2	3	1	3	41
159	18911A05E2	5	1	2	2	3	3	4	5	0	1	2	4	3	3	56
158	18911A05E3	5	1	2	2	4	5	4	5	0	1	2	4	4	5	26
151	18911A05E4	5	1	2	2	3	3	3	5	1	2	2	3	3	3	56
140	18911A05E5	5	0	1	1	3	3	2	5	1	0	2	2	3	3	53
141	18911A05E6	5	2	1	1	4	4	4	5	1	0	2	4	4	4	59
142	18911A05E7	5	2	1	1	3	3	3	5	1	0	2	3	3	3	48
143	18911A05E8	5	1	0	0	4	4	4	5	1	0	1	4	4	4	15
156	18911A05E9	5	1	0	0	2	2	3	5	0	0	1	3	2	2	34
145	18911A05F0	5	1	0	0	2	2	2	5	0	1	1	2	2	2	59
146	18911A05F1	5	0	0	0	2	2	2	5	0	1	1	2	2	2	35
147	18911A05F2	5	0	0	0	2	4	4	5	0	1	1	4	2	4	26
148	18911A05F3	5	0	1	1	3	3	3	5	0	1	2	3	3	3	18
149	18911A05F4	5	0	1	1	3	3	2	5	1	1	2	2	3	3	42
150	18911A05F5	5	0	1	1	4	4	4	5	1	0	2	4	4	4	32
151	18911A05F6	5	1	1	1	5	5	4	5	1	0	2	4	5	5	32
152	18911A05F7	5	1	2	2	3	3	3	5	1	0	2	1	3	3	34
153	18911A05F8	5	1	2	2	1	3	3	5	1	0	2	3	1	3	51
154	18911A05F9	5	1	2	2	3	3	4	5	2	0	2	4	3	3	43
155	18911A05G0	5	2	2	1	4	2	4	5	2	1	2	4	4	5	32
156	18911A05G1	5	2	2	1	5	3	4	5	2	1	2	4	5	5	35
157	18911A05G2	5	2	1	1	3	3	3	5	2	1	2	1	3	3	45
158	18911A05G3	5	2	1	0	1	3	3	5	2	1	2	3	1	3	42
159	18911A05G4	5	2	1	0	3	3	4	5	2	2	2	4	3	3	30
160	18911A05G5	5	2	0	0	4	5	4	5	2	2	2	4	4	5	29
161	18911A05G6	5	2	0	0	3	3	3	5	1	2	2	3	3	3	35
162	18911A05G7	5	2	0	0	3	3	2	5	1	2	2	2	3	3	58
163	18911A05G8	5	1	0	1	4	4	4	5	2	2	2	4	4	4	56
164	18911A05G9	5	1	0	1	3	3	3	5	2	2	2	3	3	3	50
165	18911A05H0	5	1	1	1	4	4	4	5	2	2	1	4	4	4	48
166	18911A05H1	5	1	1	1	2	2	3	5	2	2	1	3	2	2	56
167	18911A05H2	5	1	1	2	2	2	2	5	2	2	1	2	2	2	48
168	18911A05H3	5	1	1	0	2	2	2	5	2	2	1	2	2	2	42
169	18911A05H5	5	0	2	2	2	4	4	5	0	2	1	4	2	4	40
170	18911A05H6	5	0	2	2	3	3	3	5	0	2	2	3	3	3	32
171	18911A05H7	5	0	2	2	3	3	2	5	0	2	2	2	3	3	58
172	18911A05H8	5	0	2	2	4	4	4	5	0	2	2	4	4	4	27
173	18911A05H9	5	1	2	2	5	5	4	5	0	2	2	4	5	5	54
174	18911A05J0	5	1	2	2	3	3	3	5	1	2	2	1	3	3	47
175	18911A05J1	5	1	2	2	1	3	3	5	1	2	2	3	1	3	51
176	18911A05J2	5	0	2	2	3	3	4	5	1	2	2	4	3	3	59

177	18911A05J3	5	0	2	2	4	2	4	5	1	2	2	4	4	5	55
178	18911A05J4	5	0	2	2	5	3	4	5	1	2	2	4	5	5	49
179	18911A05J5	5	0	2	2	3	3	3	5	0	2	2	1	3	3	56
180	18911A05J6	5	0	2	2	1	3	3	5	0	2	2	3	1	3	62
181	18911A05J7	5	1	2	2	3	3	4	5	0	2	2	4	3	3	49
182	18911A05J8	5	1	2	2	4	5	4	5	0	2	2	4	4	5	50
183	18911A05J9	5	1	2	2	3	3	3	5	0	2	2	3	3	3	52
184	18911A05K0	5	1	2	2	3	3	2	5	1	2	2	2	3	3	46
185	18911A05K1	5	2	2	2	4	4	4	5	1	2	2	4	4	4	55
186	18911A05K2	5	2	2	2	3	3	3	5	1	2	2	3	3	3	60
187	18911A05K4	5	2	2	2	4	4	4	5	1	2	1	4	4	4	56
188	18911A05K5	5	2	2	2	2	2	3	5	2	2	1	3	2	2	49
189	18911A05K6	5	2	2	2	2	2	2	5	2	2	1	2	2	2	45
190	18911A05K7	5	2	2	0	2	2	2	5	2	2	1	2	2	2	56
191	18911A05K8	5	2	2	2	0	4	4	5	2	2	1	4	2	4	48
192	18911A05K9	5	2	2	2	0	3	3	5	2	2	2	3	3	3	61
193	18911A05L0	5	2	2	2	0	3	2	5	2	2	2	2	3	3	53
194	18911A05L1	5	2	0	0	0	4	4	5	2	2	2	4	4	4	52
195	18911A05L2	5	2	0	0	0	5	4	5	1	2	2	4	5	5	49
196	18911A05L3	5	1	0	0	1	3	3	5	1	2	2	1	3	3	45
197	18911A05L4	5	2	0	0	1	3	3	5	1	2	2	3	1	3	59
198	18911A05L5	5	0	0	0	1	3	4	5	2	2	2	4	3	3	50
199	18911A05L7	5	0	1	1	1	2	4	5	2	2	2	4	4	5	32
200	18911A05L8	5	0	1	1	1	3	4	5	2	2	2	4	5	5	33
201	18911A05L9	5	0	1	1	0	3	3	5	2	2	2	1	3	3	32
202	18911A05M0	5	0	1	1	0	3	3	5	2	2	2	3	1	3	42
203	18911A05M1	5	1	1	1	0	3	4	5	2	2	2	4	3	3	43
204	18911A05M2	5	2	0	0	0	5	4	5	2	2	2	4	4	5	41
205	18911A05M3	5	1	0	0	0	3	3	5	1	2	2	3	3	3	34
206	18911A05M4	5	1	0	0	1	3	2	5	1	2	2	2	3	3	35
207	18911A05M5	5	1	0	0	1	4	4	5	2	2	2	4	4	4	50
208	18911A05M6	5	0	0	0	1	3	3	5	2	2	2	3	3	3	58
209	18911A05M7	5	0	1	1	1	4	4	5	2	2	1	4	4	4	57
210	18911A05M8	5	0	1	1	2	2	3	5	2	2	1	3	2	2	34
211	18911A05M9	5	0	1	1	2	2	2	5	2	2	1	2	2	2	57
212	18911A05N0	5	0	1	1	2	2	2	5	2	2	1	2	2	2	26
213	18911A05N1	5	1	2	2	2	4	4	5	2	2	1	4	2	4	40

214	18911A05N2	5	1	1	2	3	3	3	5	2	2	2	3	3	3	29
215	18911A05N3	5	1	1	2	3	3	2	5	2	2	2	2	3	3	42
216	18911A05N5	5	1	1	2	4	4	4	5	2	2	2	4	4	4	42
217	18911A05N6	5	2	0	2	5	5	4	5	1	2	2	4	5	5	59
218	18911A05N7	5	0	0	2	3	3	3	5	1	2	2	1	3	3	30
219	18911A05N8	5	0	0	2	1	3	3	5	1	2	2	3	1	3	26
220	18911A05N9	5	0	0	2	3	3	4	5	2	2	2	4	3	3	43
221	18911A05P0	5	0	0	2	4	2	4	5	2	2	2	4	4	5	34
222	18911A05P1	5	0	1	2	5	3	4	5	2	2	2	4	5	5	28
223	18911A05P2	5	0	1	2	3	3	3	5	2	2	2	1	3	3	51
224	18911A05P3	5	2	1	2	1	3	3	5	2	2	2	3	1	3	42
225	18911A05P4	5	1	1	2	3	3	4	5	2	2	2	4	3	3	58
226	18911A05P5	5	1	2	2	4	5	4	5	2	2	2	4	4	5	35
227	18911A05P6	5	1	2	2	3	3	3	5	1	2	2	3	3	3	50
228	18911A05P7	5	1	2	2	3	3	2	5	1	2	2	2	3	3	51
229	18911A05P8	5	1	2	2	4	4	4	5	2	2	2	4	4	4	30
230	18911A05P9	5	2	0	2	3	3	3	5	2	2	2	3	3	3	51
253	18911A05Q0	5	1	0	0	4	4	4	5	2	2	1	4	4	4	29
232	19915A0501	5	1	0	0	2	2	3	5	2	2	1	3	2	2	47
233	19915A0502	5	1	0	0	0	2	2	5	2	2	1	2	2	2	27
234	19915A0503	5	1	0	0	0	2	2	5	2	2	1	2	2	2	46
235	19915A0504	5	0	1	0	0	4	4	5	2	2	1	4	2	4	32
257	19915A0505	5	0	1	1	0	3	3	5	2	2	2	3	3	3	34
259	19915A0506	5	2	1	1	0	3	2	5	2	2	2	2	3	3	59
258	19915A0507	5	0	1	1	1	4	4	5	2	2	2	4	4	4	58
251	19915A0508	5	0	1	1	1	5	4	5	1	1	2	4	5	5	33
240	19915A0509	5	2	0	1	1	3	3	5	1	1	2	1	3	3	42
241	19915A0510	5	2	0	0	1	3	3	5	1	1	2	3	1	3	28
242	19915A0511	5	2	0	0	1	3	4	5	2	1	2	4	3	3	45
243	19915A0512	5	0	0	0	0	2	4	5	2	1	2	4	4	5	52
256	19915A0513	5	0	0	0	0	3	4	5	2	1	2	4	5	5	67
245	19915A0514	5	0	1	0	0	3	3	5	2	1	2	1	3	3	57
246	19915A0515	5	0	1	1	0	3	3	5	2	1	2	3	1	3	60
247	19915A0516	5	0	1	1	0	3	4	5	2	1	2	4	3	3	55
248	19915A0517	5	2	1	1	1	5	4	5	2	2	2	4	4	5	48
249	19915A0518	5	2	2	1	1	3	3	5	1	2	2	3	3	3	52
250	19915A0519	5	2	2	2	1	3	2	5	1	2	2	2	3	3	46
251	19915A0520	5	2	2	2	1	4	4	5	2	2	2	4	4	4	56
252	19915A0521	5	2	2	2	3	3	3	5	2	2	2	3	3	3	54
253	19915A0522	5	2	2	2	4	4	4	5	2	2	1	4	4	4	45
254	19915A0523	5	2	2	2	2	2	3	5	2	2	1	3	2	2	45
255	19915A0524	5	2	2	2	2	2	2	5	2	2	1	2	2	2	63
Average marks		5	1.3	1.5	1.5	2.7	3.2	3.4	5	1.4	1.2	1.9	3.1	3.1	3.5	49.7
No of students attemped		255	255	255	255	255	255	255	255	255	255	255	255	255	255	254
No of students scored		255	128.00	169.00	177.00	167.00	218.00	219.00	255	140.00	113.00	217.00	191.00	197.00	226.00	190.00
%of students scored		100	50.20	66.27	69.41	65.49	85.49	85.88	100	54.90	44.31	85.10	74.90	77.25	88.63	74.80
CO ATTAINMENT LEVEL		3	1.0	2.0	2.0	2.0	3.0	3.0	3	1.0	0.0	3.0	3.0	3.0	3.0	3.0

ASSESSMENT OF COs FOR THE COURSE							
CO	Method		value	Avg	CO Attainment	CO Attainment (End	Overall CO Attainment
CO 1	ASM I		3	2.0	2.43	3.00	2.86
	MID I - PART A - Q1		1.0				
	MID I - PART B - Q4		2.0				
CO	ASM I		3	2.7			
	MID I - PART A - Q2		2.0				
	MID I - PART B - Q5		3.0				
CO	ASM I		3	2.5			
	ASM II		3				
	MID I - PART A - Q3		2.0				
	MID I - PART B - Q6		3.0				
	MID II - PART A - Q1		1.0				
	MID II - PART B - Q4		3.0				
CO 4	ASM II		3	2.0			
	MID II - PART A - Q2		0.0				
	MID II - PART B - Q5		3.0				
CO 5	ASM II		3	3.0			
	MID II- PART A - Q3		3.0				
	MID II - PART B - Q6		3.0				

COURSE-COORDINATOR

HOD-CSE

Vidya Jyothi Institute of Technology

(An Autonomous Institution)

(Accredited by NAAC. Approved by AICTE New Delhi & Permanently Affiliated to JNTUH)

Aziz Nagar Gate, C.B. Post, Hyderabad-500 075

Department of Computer Science & Engineering

(Accredited by NBA)

Academic Year: 2019-20 **BATCH: 2017-21**

III B.Tech- II Sem

Course: CD

Branch:CSE

S.No	Reg.No	MID I Threshold 60%							MID II Threshold 60%							Threshold 60%
			PART-A			PART-B				PART-A	PART-B					
		ASM - I (5)	Q1(2M)(CO1)	Q2(2M)(CO2)	Q3(2M)(CO3)	Q4(5M)(CO1)	Q5(5M)(CO2)	Q6(4M)(CO3)	ASM - II (5)	Q1(6M)(CO3)	Q2(7M)(CO4)	Q3(7M)(CO4)	Q4(7M)(CO5)	Q5(7M)(CO5)		End Exam (75M)
1	14911A05L6	5	1	1	1	2	2	1	5	4	7			7		1
2	16911A0531	5	2	2	2	5	5	3	5	5		7		7		41
3	16911A0547	5	0	0	0	0	0	0	5	0	0	0				29
4	16911A0557	5	1	1	1	3	3	2	5	4	7			7		48
5	16911A05F5	5	1	1	1	2	2	1	5	4	7			7		54
6	16911A05H9	5	2	2	2	5	5	3	5	5		7		7		46
7	16911A05K9	5	0	0	0	0	0	0	5	0	0	0				52
8	16911A05M5	5	1	1	1	3	3	2	5	4	7			7		45
9	17601A0511	5	1	1	1	2	2	1	5	4	7			7		53
10	17601A0575	5	2	2	2	5	5	3	5	5		7		7		44
11	17911A0501	5	0	0	0	0	0	0	5	0	0	0				52
12	17911A0502	5	1	1	1	3	3	2	5	4	7			7		53
13	17911A0503	5	1	1	1	2	2	1	5	4	7			7		48
14	17911A0504	5	2	2	2	5	5	3	5	5		7		7		49
15	17911A0505	5	0	0	0	0	0	0	5	0	0	0				56
16	17911A0506	5	1	1	1	3	3	2	5	4	7			7		42
17	17911A0507	5	1	1	1	2	2	1	5	4	7			7		40
18	17911A0508	5	2	2	2	5	5	3	5	5		7		7		45
19	17911A0509	5	0	0	0	0	0	0	5	0	0	0				53
20	17911A0510	5	1	1	1	3	3	2	5	4	7			7		37
21	17911A0511	5	1	1	1	2	2	1	5	4	7			7		40
22	17911A0512	5	2	2	2	5	5	3	5	5		7		7		53
23	17911A0513	5	0	0	0	0	0	0	5	0	0	0				39
24	17911A0514	5	1	1	1	3	3	2	5	4	7			7		48
25	17911A0515	5	1	1	1	5	5	1	5	2		6	6			37
26	17911A0516	5	1	1	1	5	5	1	5	1	7		7			44
27	17911A0517	5	2	2	2	5	5	3	5	2		7	7			52
28	17911A0518	5	2	2	2	5	5	2	5	5		7		7		53
29	17911A0520	5	2	2	2	5	5	3	5	5		7		7		53
30	17911A0521	5	2	2	2	5	5	3	5	5		7		7		55

31	17911A0523	5	0	0	0	0	0	0	5	0	0	0			51
32	17911A0524	5	1	1	1	3	3	2	5	4	7			7	33
33	17911A0526	5	0	0	0	0	0	0	5	1		5	5		47
34	17911A0527	5	1	1	2	5	5	2	5	2	5	5			49
35	17911A0528	5	1	1	1	5	5	1	5	2		6	6		47
36	17911A0529	5	1	1	1	5	5	1	5	1	7		7		36
37	17911A0530	5	2	2	2	5	5	3	5	2		7	7		45
38	17911A0531	5	2	2	2	5	5	2	5	5		7		7	43
39	17911A0532	5	2	2	2	5	5	3	5	5		7		7	45
40	17911A0533	5	2	2	2	5	5	3	5	5		7		7	49
41	17911A0534	5	0	0	0	0	0	0	5	0	0	0			46
42	17911A0536	5	1	1	1	3	3	2	5	4	7			7	44
43	17911A0537	5	0	0	0	0	0	0	5	1		5	5		30
44	17911A0538	5	1	1	2	5	5	2	5	2	5	5			49
45	17911A0539	5	1	1	1	5	5	1	5	2		6	6		42
46	17911A0540	5	1	1	1	5	5	1	5	1	7		7		56
47	17911A0541	5	2	2	2	5	5	3	5	2		7	7		47
48	17911A0542	5	2	2	2	5	5	2	5	5		7		7	41
49	17911A0543	5	2	2	2	5	5	3	5	5		7		7	51
50	17911A0544	5	2	2	2	5	5	3	5	5		7		7	50
51	17911A0545	5	0	0	0	0	0	0	5	0	0	0			46
52	17911A0546	5	1	1	1	3	3	2	5	4	7			7	52
53	17911A0547	5	0	0	0	0	0	0	5	1		5	5		52
54	17911A0548	5	1	1	2	5	5	2	5	2	5	5			52
55	17911A0549	5	1	1	1	5	5	1	5	2		6	6		48
56	17911A0550	5	1	1	1	5	5	1	5	1	7		7		53
57	17911A0552	5	2	2	2	5	5	3	5	2		7	7		54
58	17911A0553	5	2	2	2	5	5	2	5	5		7		7	54
59	17911A0554	5	2	2	2	5	5	3	5	5		7		7	51
60	17911A0555	5	2	2	2	5	5	3	5	5		7		7	50
61	17911A0556	5	0	0	0	0	0	0	5	0	0	0			59
62	17911A0557	5	1	1	1	3	3	2	5	4	7			7	31
63	17911A0558	5	0	0	0	0	0	0	5	1		5	5		51
64	17911A0559	5	1	1	2	5	5	2	5	2	5	5			53
65	17911A0560	5	1	1	1	5	5	1	5	2		6	6		33
66	17911A0561	5	1	1	1	5	5	1	5	1	7		7		40
67	17911A0562	5	2	2	2	5	5	3	5	2		7	7		46
68	17911A0563	5	2	2	2	5	5	2	5	5		7		7	44
69	17911A0564	5	2	2	2	5	5	3	5	5		7		7	36
70	17911A0565	5	2	2	2	5	5	4	5	6		7	7		43
71	17911A0566	5	2	2	2	5	5	3	5	5		7		7	53
72	17911A0567	5	1	1	1	3	3	1	5	2		7	7		34
73	17911A0569	5	2	2	2	5	5	3	5	5		7		7	55
74	17911A0570	5	1	1	2	5	5	2	5	1	7		7		52
75	17911A0571	5	2	2	2	5	5	2	5	6		7	7		61
76	17911A0572	5	1	1	1	5	5	1	5	3		7		7	58
77	17911A0573	5	2	2	2	5	5	4	5	3		7		7	58

78	17911A0574	5	1	1	1	5	5	1	5	4	7			7	62
79	17911A0575	5	1	1	1	3	3	2	5	3		7		7	64
80	17911A0576	5	1	1	1	4	4	2	5	2		7	7		61
81	17911A0577	5	1	1	1	2	2	2	5	3		7		7	26
82	17911A0578	5	1	1	2	5	5	2	5	2	4			4	55
83	17911A0579	5	1	1	2	5	5	2	5	2	5	5			59
84	17911A0580	5	1	1	1	3	3	1	5	5		7		7	67
85	17911A0581	5	2	2	2	5	5	1	5	4	7			7	56
86	17911A0582	5	2	2	2	5	5	1	5	2	5	5			59
87	17911A0583	5	1	1	1	3	3	1	5	2		6	5		31
88	17911A0584	5	2	2	2	5	5	1	5	1		5	5		56
89	17911A0585	5	2	2	2	5	5	1	5	4	7			7	58
90	17911A0586	5	1	1	1	4	4	1	5	2	4			4	11
91	17911A0587	5	1	1	2	5	5	2	5	2	4			4	35
92	17911A0589	5	1	1	1	3	3	2	5	4	7			7	63
93	17911A0590	5	1	1	1	4	4	2	5	2		7	7		29
94	17911A0591	5	2	2	2	5	5	2	5	6		7	7		57
95	17911A0594	5	1	1	2	5	5	2	5	6		7	7		59
96	17911A0595	5	1	1	1	5	5	1	5	3		7		7	42
97	17911A0596	5	1	1	1	3	3	2	5	5		7		7	31
98	17911A0597	5	1	1	1	4	4	2	5	6		7	7		64
99	17911A0598	5	2	2	2	5	5	1	5	6		7	7		54
100	17911A0599	5	1	1	1	4	4	1	5	5		7		7	63
101	17911A05A0	5	2	2	2	5	5	2	5	0	0	0			31
102	17911A05A2	5	1	1	1	4	4	2	5	2		6	6		55
103	17911A05A3	5	1	1	1	5	5	1	5	6		7	7		40
104	17911A05A5	5	1	1	1	5	5	2	5	2		6	6		65
105	17911A05A6	5	1	1	1	5	5	1	5	5		7		7	40
106	17911A05A7	5	2	2	2	5	5	1	5	6		7	7		59
107	17911A05A8	5	1	1	1	5	5	2	5	4	7			7	34
108	17911A05A9	5	1	1	2	5	5	2	5	2		6	5		38
109	17911A05B0	5	1	1	1	2	2	2	5	2		6	5		A
110	17911A05B1	5	1	1	1	5	5	1	5	2		6	5		56
111	17911A05B2	5	2	2	2	5	5	2	5	6		7	7		47
112	17911A05B3	5	1	1	1	2	2	2	5	5		7		7	26
113	17911A05B4	5	2	2	2	5	5	1	5	1		5	5		48
114	17911A05B6	5	1	1	2	1	5	2	5	6		7	7		61
115	17911A05B7	5	1	1	1	1	5	2	5	3		7		7	46
116	17911A05B8	5	1	1	1	1	5	1	5	2		6	6		47
117	17911A05B9	5	1	1	1	2	5	1	5	4	7			7	54
118	17911A05C0	5	2	2	2	2	5	1	5	6		7	7		37
119	17911A05C1	5	1	1	2	2	5	2	5	4	7			7	47
120	17911A05C2	5	1	1	2	2	5	2	5	2		2	2		53
121	17911A05C3	5	1	1	2	2	5	2	5	5		7		7	56
122	17911A05C4	5	2	2	2	1	5	1	5	4	7			7	67
123	17911A05C5	5	1	1	1	2	3	2	5	1	7		7		71
124	17911A05C7	5	2	2	2	2	5	4	5	6		7	7		64

125	17911A05C8	5	2	2	2	2	5	2	5	6		7	7		53
126	17911A05C9	5	2	2	2	1	5	4	5	4	7			7	63
127	17911A05D0	5	1	1	1	1	5	2	5	0		0		0	64
128	17911A05D1	5	1	1	1	1	4	2	5	2	5	5			71
129	17911A05D2	5	1	1	1	2	2	2	5	2	5	5			59
130	17911A05D3	5	2	2	2	5	5	1	5	6		7	7		68
131	17911A05D4	5	1	1	1	4	4	2	5	2		7	7		68
132	17911A05D5	5	1	1	1	1	5	2	5	5		7		7	67
133	17911A05D6	5	2	2	2	1	5	1	5	5		7		7	73
134	17911A05D7	5	1	1	2	1	5	2	5	5		7		7	70
135	17911A05D8	5	2	2	2	2	5	1	5	6		7	7		70
136	17911A05D9	5	1	1	1	2	4	2	5	4	7			7	72
137	17911A05E0	5	2	2	2	2	5	1	5	0	0		0		33
138	17911A05E1	5	2	2	2	2	5	4	5	5		7		7	38
139	17911A05E2	5	2	2	2	2	5	1	5	4	7			7	69
140	17911A05E3	5	2	2	2	1	5	3	5	6		7	7		52
141	17911A05E4	5	1	1	2	2	5	2	5	4	7			7	68
142	17911A05E5	5	2	2	2	2	5	1	5	5		7		7	46
143	17911A05E6	5	1	1	1	2	4	1	5	5		7		7	67
144	17911A05E7	5	2	2	2	1	5	2	5	6		7	7		72
145	17911A05E8	5	2	2	2	1	5	3	5	2		6	6		43
146	17911A05E9	5	2	2	2	1	5	3	5	4	7			7	62
147	17911A05F0	5	1	1	1	1	1	0	5	1		4	4		63
148	17911A05F1	5	1	1	1	4	4	1	5	2	5	5			67
149	17911A05F2	5	2	2	2	5	5	3	5	5		7		7	60
150	17911A05F3	5	2	2	2	5	5	4	5	6		7	7		46
151	17911A05F4	5	2	2	2	5	5	4	5	5		7		7	67
152	17911A05F5	5	2	2	2	5	5	1	5	6		7	7		75
153	17911A05F6	5	2	2	2	5	5	3	5	5		7		7	36
154	17911A05F7	5	1	1	1	2	2	1	5	6		7	7		46
155	17911A05F8	5	2	2	2	5	5	4	5	6		7	7		68
156	17911A05F9														
157	17911A05G0														
158	17911A05G1														
159	17911A05G2														
160	17911A05G3														
161	17911A05G4														
162	17911A05G5														
163	17911A05G6														
164	17911A05G7														
165	17911A05G8														
166	17911A05G9	5	2	2	2	5	5	3	5	4	7			7	49
167	17911A05H0	5	1	1	2	5	5	2	5	2		6	5		45
168	17911A05H1	5	1	1	1	1	5	1	5	4	7			7	39
169	17911A05H2	5	1	1	1	1	3	2	5	1		5	5		68
170	17911A05H3	5	1	1	1	1	5	2	5	4	7			7	69
171	17911A05H4	5	1	1	1	2	3	2	5	3		7		7	60

172	17911A05H5	5	1	1	1	2	5	1	5	5		7		7	57
173	17911A05H6	5	2	2	2	2	5	3	5	4	7			7	38
174	17911A05H7	5	2	2	2	2	5	4	5	2		6	5		45
175	17911A05H8	5	2	2	2	2	5	2	5	5		7		7	50
176	17911A05H9	5	2	2	2	1	5	2	5	6		7	7		63
177	17911A05J1	5	1	1	1	2	5	1	5	1		5	5		45
178	17911A05J2	5	2	2	2	2	5	2	5	4	7			7	47
179	17911A05J3	5	1	1	1	2	4	2	5	2	4			4	74
180	17911A05J4	5	2	2	2	1	5	3	5	5		7		7	55
181	17911A05J5	5	2	2	2	1	5	4	5	3		7		7	65
182	17911A05J6	5	1	1	2	1	5	2	5	2		7	7		75
183	17911A05J7	5	2	2	2	5	5	4	5	4	7			7	74
184	17911A05J8	5	2	2	2	5	5	4	5	5		7		7	65
185	17911A05J9	5	1	1	1	1	1	0	5	2		6	6		69
186	17911A05K0	5	1	1	1	4	4	1	5	2	3		3		61
187	17911A05K1	5	2	2	2	5	5	4	5	4	7			7	67
188	17911A05K2	5	2	2	2	5	5	4	5	5		7		7	63
189	17911A05K3	5	1	1	1	5	5	1	5	4	7			7	75
190	17911A05K4	5	2	2	2	5	5	1	5	6		7	7		64
191	17911A05K5	5	1	1	1	5	5	2	5	6		7		7	39
192	17911A05K6	5	1	1	1	3	3	2	5	6		7	7		61
193	17911A05K7	5	2	2	2	5	5	1	5	6		6	6		70
194	17911A05K8	5	1	1	1	5	5	2	5	6		5	5		40
195	17911A05K9	5	1	1	1	5	5	1	5	6	5	5			50
196	17911A05L0	5	2	2	2	5	5	3	5	6		7	7		73
197	17911A05L1	5	1	1	1	5	5	2	5	6	7			7	47
198	17911A05L2	5	1	1	1	5	5	1	5	6		6	5		57
199	17911A05L3	5	2	2	2	5	5	3	5	6		7		7	67
200	17911A05L4	5	2	2	2	5	5	1	5	6		6	6		57
201	17911A05L5	5	1	1	1	2	2	2	5	6	3		3		54
202	17911A05L6	5	2	2	2	5	5	1	5	6	7			7	68
203	17911A05L7	5	2	2	2	5	5	1	5	6	7			7	73
204	17911A05L8	5	1	1	2	5	5	2	5	6		6	6		53
205	17911A05L9	5	2	2	2	5	5	2	5	6	7		7		54
206	17911A05M0	5	2	2	2	5	5	3	5	6	7			7	53
207	17911A05M1	5	1	1	1	5	5	2	5	6	7			7	46
208	17911A05M2	5	2	2	2	5	5	4	5	6		7	7		48
209	17911A05M3	5	2	2	2	5	5	4	5	6	7			7	29
210	17911A05M4	5	1	1	2	5	5	4	5	6	5	5			44
211	17911A05M5	5	1	1	1	3	3	4	5	6	4			4	60
212	17911A05M6	5	1	1	1	3	3	4	5	6		6	5		49
213	17911A05M7	5	1	1	1	5	5	4	5	6		7	7		38
214	17911A05M8	5	1	1	2	5	5	4	5	6		7	7		49
215	17911A05M9	5	1	1	2	5	5	4	5	6		7	7		53
216	17911A05N0	5	1	1	1	4	4	4	5	6	7			7	41
217	17911A05N1	5	2	2	2	5	5	4	5	6	7			7	46
218	17911A05N2	5	1	1	1	4	4	4	5	6		6	5		44

219	17911A05N3	5	1	1	1	3	3	4	5	6	7			7	55
220	17911A05N5	5	1	1	1	2	2	4	5	6		6	5		44
221	17911A05N6	5	1	1	1	4	4	4	5	6		7	7		36
222	17911A05N7	5	2	2	2	5	5	4	5	6	7			7	44
223	17911A05N8	5	1	1	1	5	5	4	5	6	7			7	53
224	17911A05N9	5	1	1	1	5	5	4	5	6	7			7	43
225	17911A05P0	5	2	2	2	5	5	4	5	6	7			7	49
226	17911A05P2	5	2	2	2	5	5	4	5	6	7			7	34
227	17911A05P3	5	1	1	1	5	5	4	5	6	7		7		41
228	17911A05P4	5	1	1	1	4	4	4	5	6	7			7	44
229	17911A05P5	5	2	2	2	5	5	4	5	6	7			7	42
230	17911A05P6	5	2	2	2	5	5	4	5	6		6	6		32
231	17911A05P7	5	1	1	2	5	5	4	5	6	7			7	31
232	17911A05P8	5	1	1	1	0	0	4	5	6		3	3		41
233	17911A05P9	5	1	1	1	3	3	4	5	6		6	5		48
234	17911A05Q0	5	1	1	1	4	4	4	5	6		6	6		47
235	18915A0501	5	2	2	2	5	5	4	5	6		7		7	55
236	18915A0502	5	1	1	1	5	5	4	5	6		2	2		42
237	18915A0503	5	1	1	1	3	3	4	5	6	4			4	40
238	18915A0504	5	2	2	2	5	5	4	5	6	7			7	41
Average marks		5	1.346491	1.346491	1.447368	3.526316	4.166667	2.135965	5	3.973684	5.610526	5.922581	6.043956	6.782609	50.96476
No of students attemped		228	228	228	228	228	228	228	228	228	95	155	91	115	227
%of students scored 60%		100	93.859649	93.859649	93.859649	67.105263	86.842105	66.666667	100	69.298246	87.368421	90.967742	96.703297	99.130435	81.50
CO ATTAINMENT		3	3.0	3.0	3.0	2.0	3.0	2.0	3	2.0	3.0	3.0	3.0	3.0	3.0

ASSESSMENT OF COs FOR THE COURSE									
CO	Method			value		CO Attainment (Internal)		CO Attainment (End Exam)	Overall CO Attainment
CO 1	ASM I			3	2.67	2.85		3.00	2.96
	MID I - PART A Q1			3.0					
	MID I - PART B Q4			2.0					
CO 2	ASM I			3	3.00				
	MID I - PART A Q2			3.0					
	MID I - PART B Q5			3.0					
CO 3	ASM I			3	2.6				
	ASM II			3.0					
	MID I - PART A Q3			3.0					
	MID I - PART B Q6			2.0					
	MID II - PART A - Q1			2.0					
CO 4	ASM II			3	3				
	MID II - PART B - Q2			3.0					
	MID II - PART B Q3			3.0					
CO 5	ASM II			3	3				
	MID II- PART B Q4			3.0					
	MID II - PART B Q5			3.0					

COURSE-COORDINATOR

HOD-CSE

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

R15 16-20

CD					
	OP1	OP2	OP3	TOTAL	ATTAIN
CO1	11	12	212	235	2.86
CO2	11	24	200	235	2.80
CO3	12	24	199	235	2.80
CO4	10	24	201	235	2.81
CO5	25	20	190	235	2.70
					2.79

CO OVERALL ATTAINMENT=80% of Direct + 20% of Indirect
2.20

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.