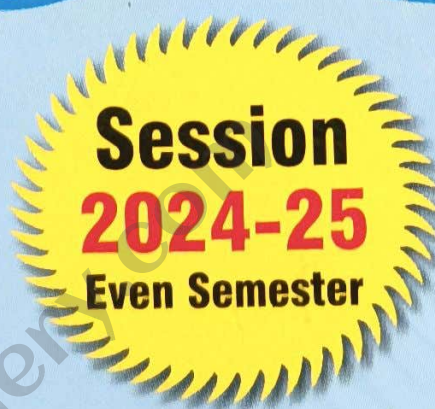
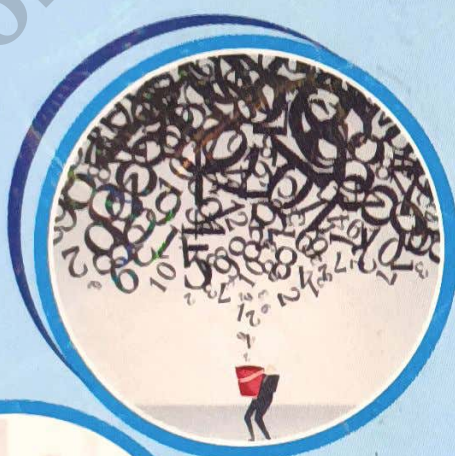


QUANTUM *Series*

Semester - 6

COMPUTER SCIENCE

Compiler Design



- Topic-wise coverage of entire syllabus in Question-Answer form.
- Short Questions (2 Marks)

Includes solution of following AKTU Question Papers:

2020-21 • 2021-22 • 2022-23 • 2023-24

सजग सूचना

छात्रों, डीलरों, दुकानदारों, कॉपियर्स और जिनसे यह संबंधित हो सकता है क्वांटम पुस्तकों के कॉपीराइट उल्लंघन के संबंध में दिल्ली के माननीय उच्च न्यायालय का आदेश

यह छात्रों, डीलरों, दुकानदारों, कॉपियर्स और आम जनता को सूचित करना है कि क्वांटम पेज प्राइवेट लिमिटेड पुस्तकों की क्वांटम श्रृंखला में कॉपीराइट का मालिक है।

क्वांटम पुस्तकों (चाहे सॉफ्ट कॉपी या हार्ड कॉपी में) की कोई भी अनधिकृत प्रतिलिपि, स्कैनिंग, पुनरुत्पादन, वितरण या संचलन, क्वांटम पेज प्राइवेट लिमिटेड के कॉपीराइट और ट्रेडमार्क अधिकारों के उल्लंघन के बराबर है, जो दिवानी अपराध के साथ-साथ दंडिक अपराध है, जिनकी सजा कारावास तक है।

क्वांटम पेज प्राइवेट लिमिटेड ने दिल्ली के माननीय उच्च न्यायालय के समक्ष क्वांटम पुस्तकों के उल्लंघनकर्ताओं के खिलाफ क्वांटम पेज प्राइवेट लिमिटेड बनाम टेलीग्राम एफजेड एलएलसी और अन्य, सीएस (कॉम) ९२१/२०२२ नामक एक मुकदमा शुरू किया है। 23 दिसंबर, 2022 के अपने आदेश में, माननीय उच्च न्यायालय ने माना है कि क्वांटम पुस्तकों का अनधिकृत पुनरुत्पादन कॉपीराइट उल्लंघन के बराबर है और टेलीग्राम चैनलों सहित विभिन्न स्रोतों से क्वांटम पुस्तकों की उल्लंघनकारी प्रतियों को हटाने का निर्देश दिया है।

छात्रों, डीलरों, दुकानदारों, कॉपीयरों और आम जनता को एतद्वारा आगाह किया जाता है कि वे क्वांटम पुस्तकों (चाहे सॉफ्ट कॉपी या हार्ड कॉपी में) की कोई भी अनधिकृत नकल, स्कैनिंग, पुनरुत्पादन, वितरण और वितरण न करें।

क्वांटम पुस्तकों के ऐसे किसी भी अनधिकृत उपयोग से उल्लंघनकर्ताओं पर क्वांटम पेज प्राइवेट लिमिटेड द्वारा दिवानी और आपराधिक कार्यवाही शुरू की जाएगी।

QUANTUM SERIES

For

B.Tech Students of Third Year
of All Engineering Colleges Affiliated to
Dr. A.P.J. Abdul Kalam Technical University,
Uttar Pradesh, Lucknow
(Formerly Uttar Pradesh Technical University)

Compiler Design

By

Aditya Kumar



QUANTUM PAGE PVT. LTD.
Ghaziabad ■ New Delhi

PUBLISHED BY :

Apram Singh
Quantum Publications®
 (A Unit of Quantum Page Pvt. Ltd.)
 Plot No. 59/2/7, Site - 4, Industrial Area,
 Sahibabad, Ghaziabad-201 010

Phone : 0120-4160479

Email : pagequantum@gmail.com Website: www.quantumpage.co.in

Delhi Office : M-28, Naveen Shahdara, Delhi-110032

© QUANTUM PAGE PVT. LTD.

No part of this publication may be reproduced or transmitted,
 in any form or by any means, without permission.

ALL RIGHTS RESERVED

Information contained in this work is derived from sources believed to be reliable. Every effort has been made to ensure accuracy, however neither the publisher nor the authors guarantee the accuracy or completeness of any information published herein, and neither the publisher nor the authors shall be responsible for any errors, omissions, or damages arising out of use of this information.

Compiler Design (CS : Sem-6)1st Edition : 2010-112nd Edition : 2011-123rd Edition : 2012-134th Edition : 2013-145th Edition : 2014-156th Edition : 2015-167th Edition : 2016-178th Edition : 2017-189th Edition : 2018-1910th Edition : 2019-2011th Edition : 2020-2112th Edition : 2021-2213th Edition : 2022-2314th Edition : 2023-2415th Edition : 2024-25

(Thoroughly Revised Edition)

Price: Rs. 120/- only

Printed at : Narula Printers, Delhi-110093.

CONTENTS**BCS602 : Compiler Design****UNIT-1 : INTRODUCTION TO COMPILER (1-1 C to 1-37 C)**

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

UNIT-2 : BASIC PARSING TECHNIQUES (2-1 C to 2-44 C)

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

UNIT-3 : SYNTAX-DIRECTED TRANSLATION**(3-1 C to 3-35 C)**

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

UNIT-4 : SYMBOL TABLES**(4-1 C to 4-25 C)**

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

UNIT-5 : CODE GENERATION**(5-1 C to 5-29 C)**

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

SHORT QUESTIONS**(SQ-1 C to SQ-20 C)****SOLVED PAPERS (2020-21 TO 2023-24) (SP-1 C to SP-14 C)**

Introduction to Compiler

CONTENTS

Part-1	: Introduction to Compiler :	1-2C to 1-8C
	Phases and Passes	
Part-2	: Bootstrapping.....	1-8C to 1-9C
Part-3	: Finite State Machines and	1-9C to 1-21C
	Regular Expressions and their Application to Lexical Analysis, Optimization of DFA Based Pattern Matchers	
Part-4	: Implementation of	1-21C to 1-26C
	Lexical Analyzers, Lexical Analyzer Generator, LEX Compiler	
Part-5	: Formal Grammars and	1-27C to 1-29C
	their Application to Syntax Analysis, BNF Notation	
Part-6	: Ambiguity, YACC.....	1-30C to 1-33C
Part-7	: The Syntactic Specification.....	1-33C to 1-35C
	of Programming Languages : Context Free Grammar (CFG), Derivation and Parse Trees, Capabilities of CFG	

PART-1*Introduction to Compiler, Phases and Passes.*

Que 1.1. Explain in detail the process of compilation. Illustrate the output of each phase of compilation of the input " $a = (b + c) * (b + c) * 2$ ".

Answer

A compiler contains 6 phases which are as follows :

i. Phase 1 (Lexical analyzer) :

- The lexical analyzer is also called scanner.
- The lexical analyzer phase takes source program as an input and separates characters of source language into groups of strings called token.
- These tokens may be keywords identifiers, operator symbols and punctuation symbols.

ii. Phase 2 (Syntax analyzer) :

- The syntax analyzer phase is also called parsing phase.
- The syntax analyzer groups tokens together into syntactic structures.
- The output of this phase is parse tree.

iii. Phase 3 (Semantic analyzer) :

- The semantic analyzer phase checks the source program for semantic errors and gathers type information for subsequent code generation phase.
- It uses parse tree and symbol table to check whether the given program is semantically consistent with language definition.
- The output of this phase is annotated syntax tree.

iv. Phase 4 (Intermediate code generation) :

- The intermediate code generation takes syntax tree as an input from semantic phase and generates intermediate code.
- It generates variety of code such as three address code, quadruple, triple.

v. Phase 5 (Code optimization) : This phase is designed to improve the intermediate code so that the ultimate object program runs faster and takes less space.

vi. **Phase 6 (Code generation) :**

- a. It is the final phase for compiler.
- b. It generates the assembly code as target language.
- c. In this phase, the address in the binary code is translated from logical address.

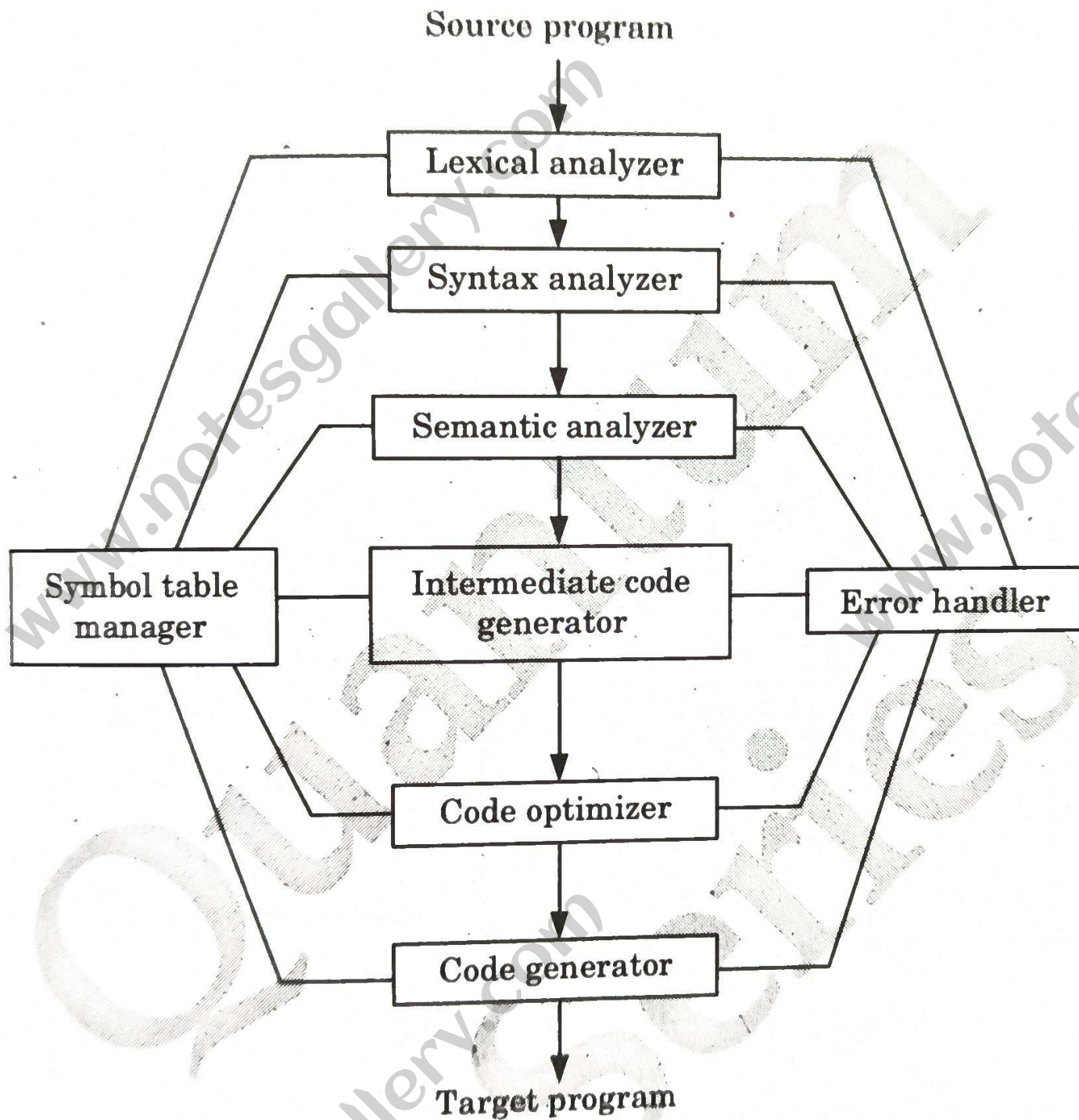


Fig. 1.1.1.

Symbol table / table management : A symbol table is a data structure containing a record that allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

Error handler : The error handler is invoked when a flaw in the source program is detected.

Input processing in compiler

Input processing in compiler	Output
$a = (b + c) * (b + c) + 2$ ↓ Lexical analyzer ↓ $id_1 = (id_2 + id_3) * (id_2 + id_3) + 2$	Token stream
↓ Syntax analyzer ↓ ↓ Semantic analyzer ↓ ↓ Intermediate code generation ↓ $t_1 = b + c$ $t_2 = t_1 * t_1$ $t_3 = \text{int_to_real}(2)$ $t_4 = t_2 * t_3$ $id_1 = t_4$	Parse tree Annotated syntax tree Intermediate code

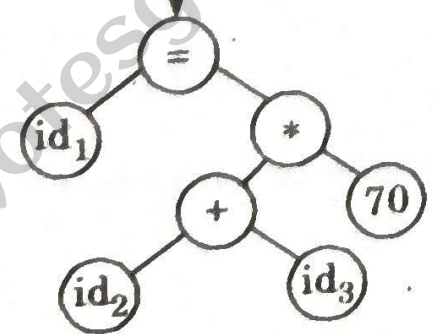
Code optimization $t_1 = b + c$ $t_2 = t_1 * t_1$ $id_1 = t_2 * 2$	Optimized code
Machine code MOV R₁, b ADD R₁, R₁, c MUL R₂, R₁, R₁ MUL R₂, R₁, # 2.0 ST id₁, R₂	Machine code

Que 1.2.

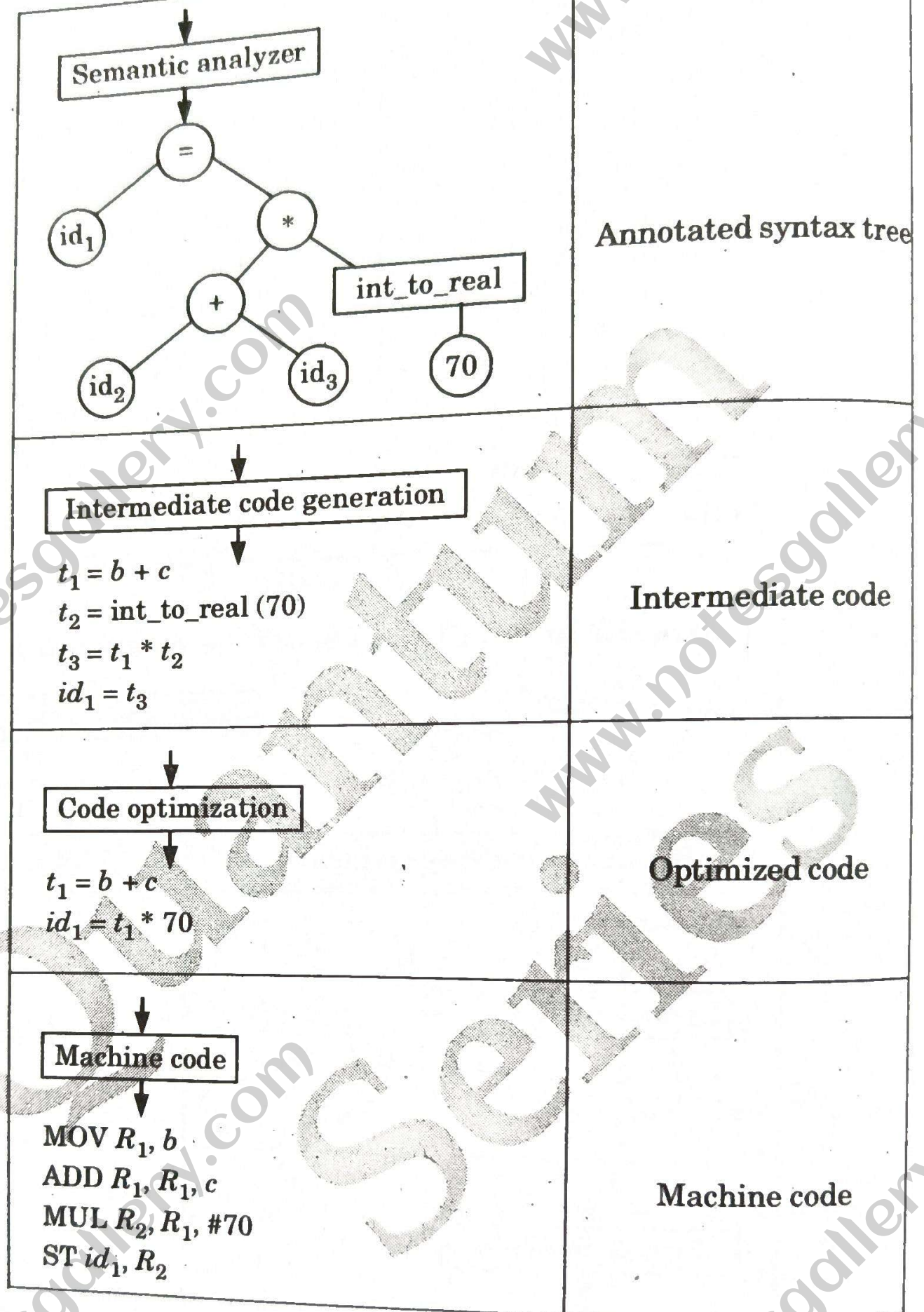
Explain in detail the process of compilation for the statement $a = b + c * 70$.

AKTU 2020-21, Marks 10

Answer

Input processing in compiler	Output
$a = b + c * 70$ Lexical analyzer $id_1 = (id_2 + id_3) * 70$	Token stream
Syntax analyzer 	Parse tree

1-6 C (CS-Sem-6)

**Que 1.3.****What are the types of passes in compiler ?****Answer****Types of passes :****1. Single-pass compiler :**

- In a single-pass compiler, when a line source is processed it is scanned and the tokens are extracted.

- b. Then the syntax of the line is analyzed and the tree structure, some tables containing information about each token are built.
2. **Multi-pass compiler :** In multi-pass compiler, it scan the input source once and produces first modified form, then scans the modified form and produce a second modified form and so on, until the object form is produced.

Que 1.4. Discuss the role of compiler writing tools. Describe various compiler writing tools.

Answer

Role of compiler writing tools :

1. Compiler writing tools are used for automatic design of compiler component.
2. Every tool uses specialized language.
3. Writing tools are used as debuggers, version manager.

Various compiler construction/writing tools are :

1. **Parser generator :** The procedure produces syntax analyzer, normally from input that is based on context free grammar.
2. **Scanner generator :** It automatically generates lexical analyzer, normally from specification based on regular expressions.
3. **Syntax directed translation engine :**
 - a. It produces collection of routines that are used in parse tree.
 - b. These translations are associated with each node of parse tree, and each translation is defined in terms of translations at its neighbour nodes in the tree.
4. **Automatic code generator :** These tools take a collection of rules that define translation of each operation of the intermediate language into the machine language for target machine.
5. **Data flow engine :** The data flow engine is use to optimize the code involved and gathers the information about how values are transmitted from one part of the program to another.

Que 1.5. Discuss about cross compiler.

AKTU 2020-21, Marks 05

Answer

1. A cross-compiler is a compiler that runs on one machine and produces object code for another machine.
2. The cross-compiler is used to implement the compiler, which is characterized by three languages :

- a. The source language,
 - b. The object language, and
 - c. The language in which it is written.
3. If a compiler has been implemented in its own language, then this arrangement is called a "bootstrap" arrangement.

PART-2

Bootstrapping.

Que 1.6. Define bootstrapping with the help of an example.

OR

What is a cross compiler ? How is bootstrapping of a compiler done to a second machine ?

Answer

Cross compiler : A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.

Bootstrapping :

1. Bootstrapping is the process of writing a compiler (or assembler) in the source programming language that it intends to compile.
2. Bootstrapping leads to a self-hosting compiler.
3. An initial minimal core version of the compiler is generated in a different language.
4. A compiler is characterized by three languages :
 - a. Source language (S)
 - b. Target language (T)
 - c. Implementation language (I)
5. ${}^S C_I^T$ represents a compiler for Source S, Target T, implemented in I. The T-diagram shown in Fig. 1.6.1 is also used to depict the same compiler :

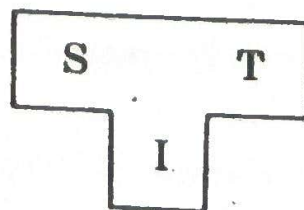


Fig. 1.6.1.

6. To create a new language, L, for machine A :
 - a. Create ${}^S C_A^A$ a compiler for a subset, S, of the desired language, L, using language A, which runs on machine A. (Language A may be assembly language.)

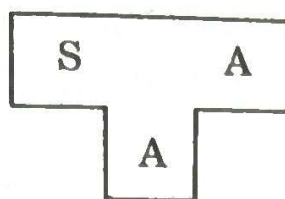


Fig. 1.6.2.

- b. Create ${}^L C_S^A$, a compiler for language L written in a subset of L .

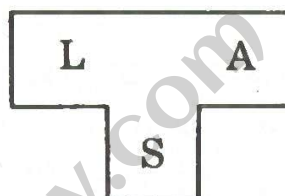


Fig. 1.6.3.

- c. Compile ${}^L C_S^A$ using ${}^S C_A^A$ to obtain ${}^L C_A^A$, a compiler for language L , which runs on machine A and produces code for machine A .

$${}^L C_S^A \rightarrow {}^S C_A^A \rightarrow {}^L C_A^A$$

The process illustrated by the T -diagrams is called bootstrapping and can be summarized by the equation:

$$L_S A + S_A A = L_A A$$

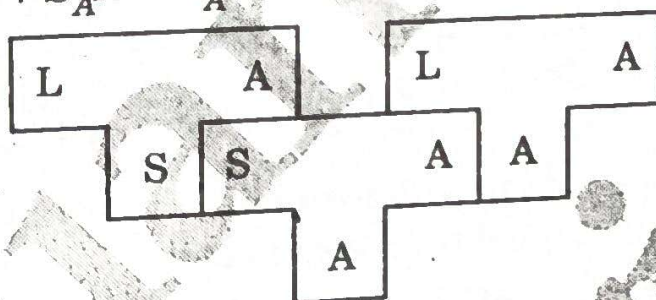


Fig. 1.6.4.

PART-3

Finite State Machines and Regular Expressions and their Application to Lexical Analysis, Optimization of DFA Based Pattern Matchers.

Que 1.7. What do you mean by regular expression ? Write the formal recursive definition of a regular expression.

Answer

1. Regular expression is a formula in a special language that is used for specifying simple classes of strings.
2. A string is a sequence of symbols; for the purpose of most text-based search techniques, a string is any sequence of alphanumeric characters (letters, numbers, spaces, tabs, and punctuation).

Formal recursive definition of regular expression :

Formally, a regular expression is an algebraic notation for characterizing a set of strings.

1. Any terminals, i.e., the symbols belong to S are regular expression. Null string ($\wedge$, ϵ) and null set (ϕ) are also regular expression.
2. If P and Q are two regular expressions then the union of the two regular expressions, denoted by $P + Q$ is also a regular expression.
3. If P and Q are two regular expressions then their concatenation denoted by PQ is also a regular expression.
4. If P is a regular expression then the iteration (repetition or closure) denoted by P^* is also a regular expression.
5. If P is a regular expression then P , is a regular expression.
6. The expressions got by repeated application of the rules from (1) to (5) over Σ are also regular expression.

Que 1.8.

Define and differentiate between DFA and NFA with an example.

Answer**DFA:**

1. A finite automata is said to be deterministic if we have only one transition on the same input symbol from some state.
2. A DFA is a set of five tuples and represented as :

$$M = (Q, \Sigma, \delta, q_0, F)$$

where,

 Q = A set of non-empty finite states Σ = A set of non-empty finite input symbols q_0 = Initial state of DFA F = A non-empty finite set of final state

$$\delta = Q \times \Sigma \rightarrow Q$$

NFA:

1. A finite automata is said to be non-deterministic, if we have more than one possible transition on the same input symbol from some state.
2. A non-deterministic finite automata is set of five tuples and represented as :

$$M = (Q, \Sigma, \delta, q_0, F)$$

where,

 Q = A set of non-empty finite states Σ = A set of non-empty finite input symbols q_0 = Initial state of NFA and member of Q

$F = A$ non-empty finite set of final states and member of Q

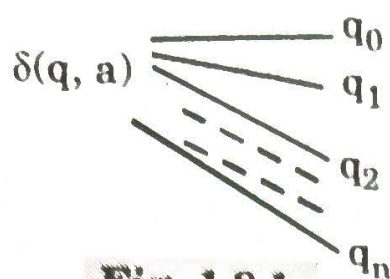


Fig. 1.8.1.

δ = It is transition function that takes a state from Q and an input symbol from Σ and returns a subset of Q . The δ is represented as :

$$\delta = Q * (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$$

Difference between DFA and NFA :

S.No.	DFA	NFA
1.	It stands for deterministic finite automata.	It stands for non-deterministic finite automata.
2.	Only one transition is possible from one state to another on same input symbol.	More than one transition is possible from one state to another on same input symbol.
3.	Transition function δ is written as : $\delta : Q \times \Sigma \rightarrow Q$	Transition function δ is written as : $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$
4.	In DFA, ϵ -transition is not possible.	In NFA, ϵ -transition is possible.
5.	DFA cannot be converted into NFA.	NFA can be converted into DFA.

Example ; DFA for the language that contains the strings ending with 0 over $\Sigma = \{0, 1\}$.

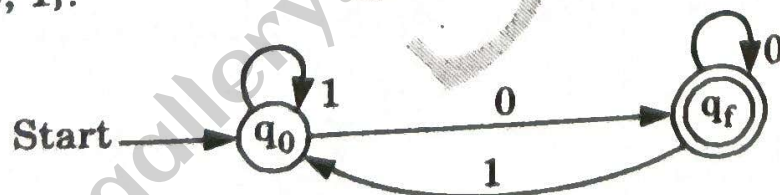


Fig. 1.8.2.

NFA for the language L which accept all the strings in which the third symbol from right end is always a over $\Sigma = \{a, b\}$.

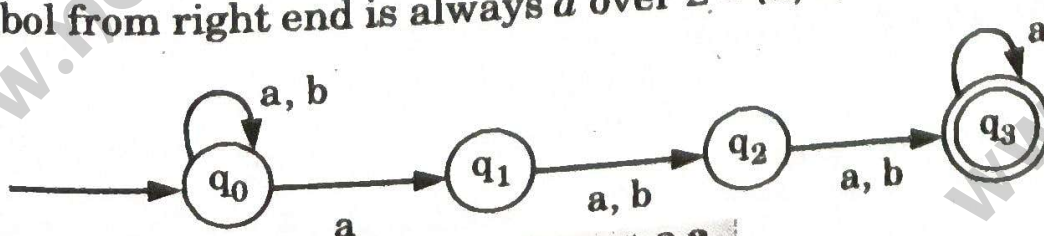


Fig. 1.8.3.

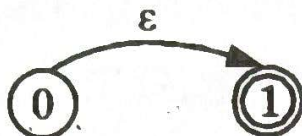
Que 1.9. Explain Thompson's construction with example.

Answer

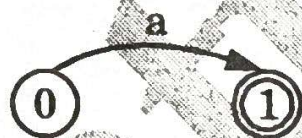
Thompson's construction :

1. It is an algorithm for transforming a regular expression to equivalent NFA.
2. Following rules are defined for a regular expression as a basis for the construction :

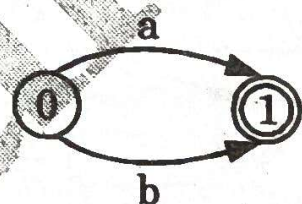
i. The NFA representing the empty string is :



ii. If the regular expression is just a character, thus a can be represented as :



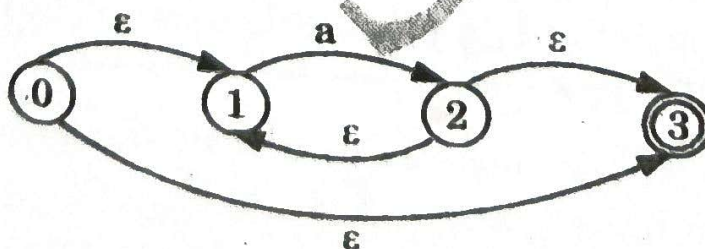
iii. The union operator is represented by a choice of transitions from a node thus $a | b$ can be represented as :



iv. Concatenation simply involves connecting one NFA to the other thus ab can be represented as :



v. The Kleene closure must allow for taking zero or more instances of the letter from the input; thus a^* looks like :



For example :

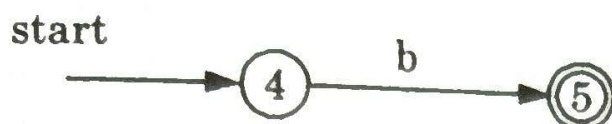
Construct NFA for $r = (a | b)^* a$

For $r_1 = a$,

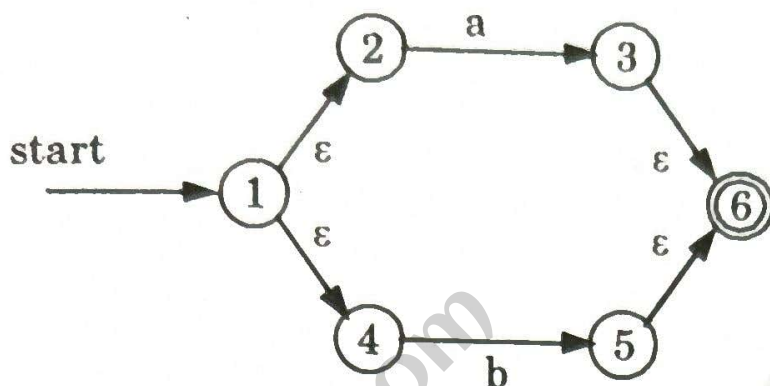
start



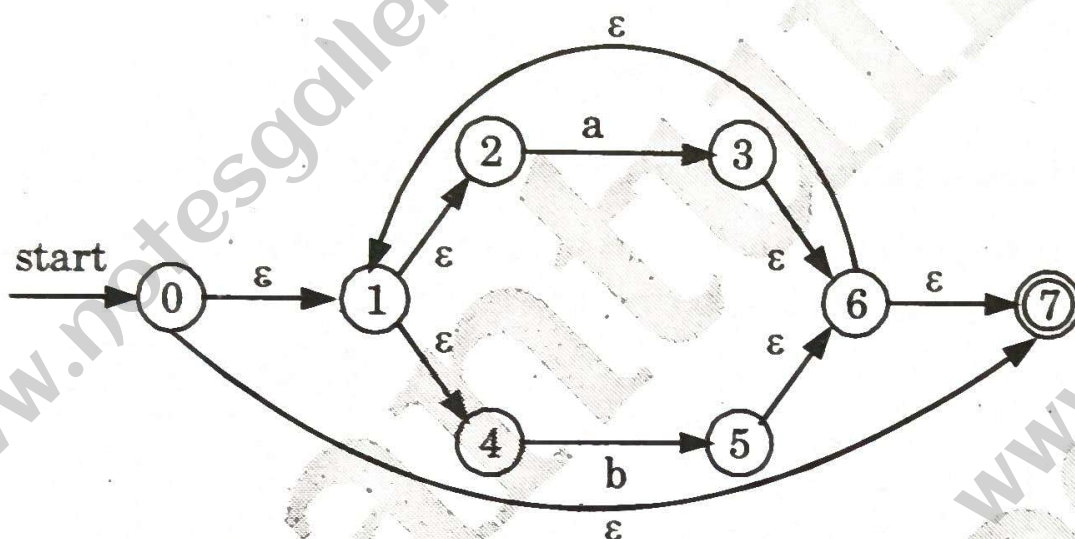
For $r_2 = b$,



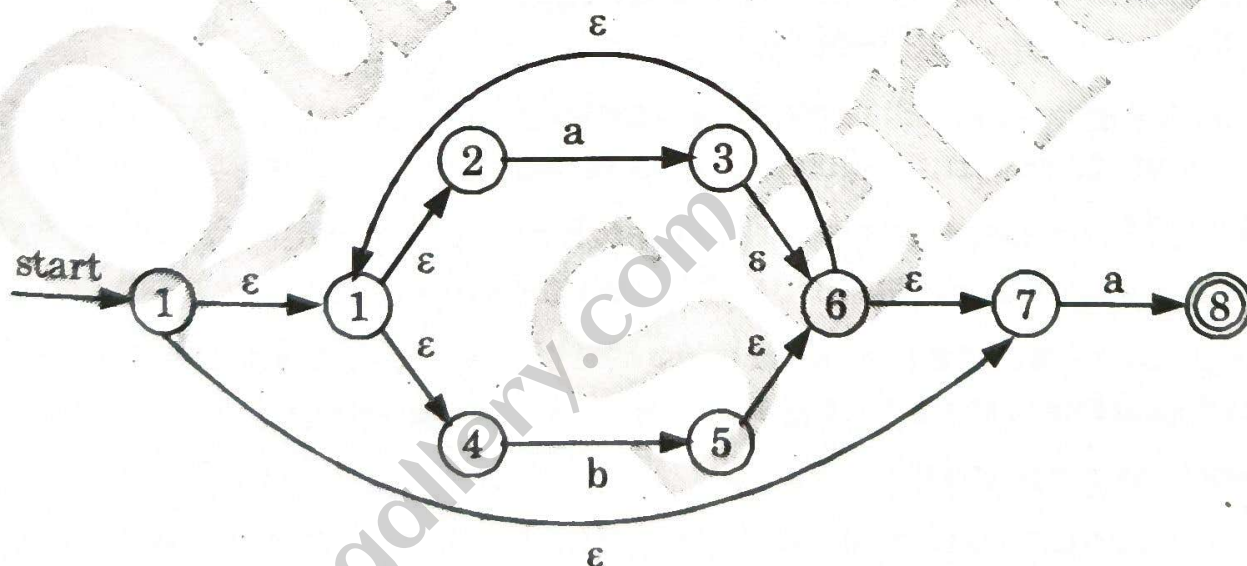
For $r_3 = a | b$



The NFA for $r_4 = (r_3)^*$



Finally, NFA for $r_5 = r_4 \cdot r_1 = (a | b)^* a$

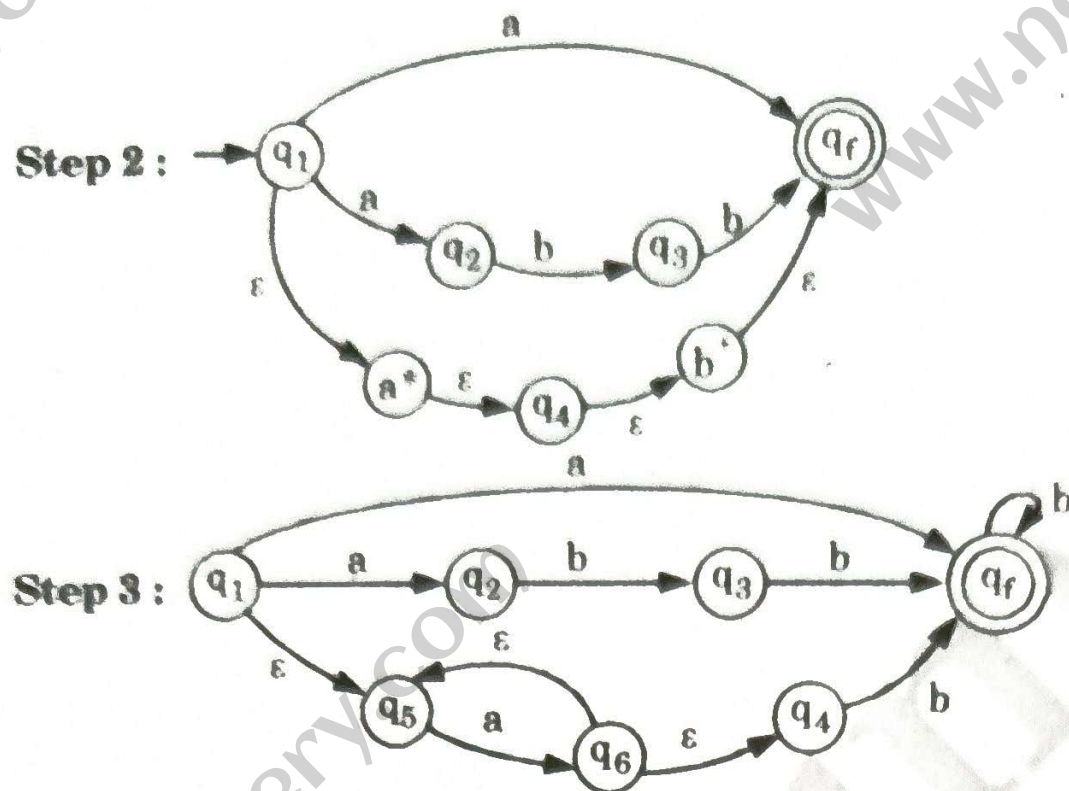


Que 1.10. Construct the NFA for the regular expression $a | abb | a^*b^+$ by using Thompson's construction methodology.

Answer

Given regular expression : $a + abb + a^*b^+$

Step 1 : $\rightarrow (q_1) \xrightarrow{a + abb + a^*b^+} (q_f)$



Que 1.11. Describe the relationship between finite state machines and regular expressions. Discuss how regular expressions are used in lexical analysis and pattern matching.

AKTU 2023-24, Marks 10

Answer

Relationship :

1. Finite state machines (FSMs) and regular expressions are closely related in formal language theory.
2. Regular expressions provide a way to describe patterns in strings, while finite state machines are used to recognize these patterns.
3. An FSM can be constructed to match any pattern described by a regular expression, demonstrating their equivalence in expressive power.

Regular expressions are extensively used in lexical analysis and pattern matching in the following ways :

1. Lexical analysis :

- a. Regular expressions define the patterns of tokens in the input stream of a programming language or any textual data.
- b. Tokenizers or lexical analyzers use regular expressions to recognize and extract tokens such as keywords, identifiers, literals, and symbols.
- c. By defining regular expressions for each token type, lexical analyzers can efficiently segment the input stream into meaningful units.

2. Pattern matching :

- a. Regular expressions provide a powerful mechanism for searching and matching patterns within strings or documents.

- b. Applications like text editors, search engines, and data validation tools use regular expressions to locate specific substrings or patterns within large datasets.
- c. Regular expressions are used in pattern matching operations such as finding all occurrences of a pattern, searching for patterns with variations, and extracting specific portions of text that match a pattern.

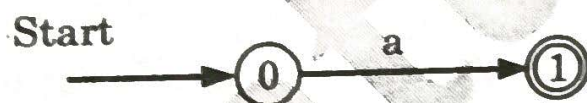
Que 1.12. Given a regular expression $a(b|c)^*$, construct the corresponding Non-deterministic Finite Automaton (NFA) that recognizes the language described by the regular expression.

AKTU 2023-24, Marks 10

Answer

Construct NFA for $r = a(b|c)^*$

For $r_1 = a$,



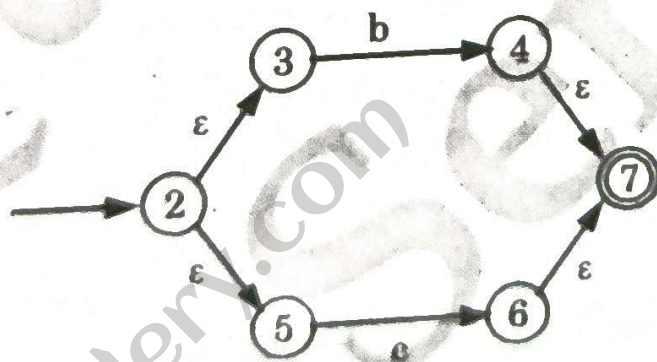
For $r_2 = b$,



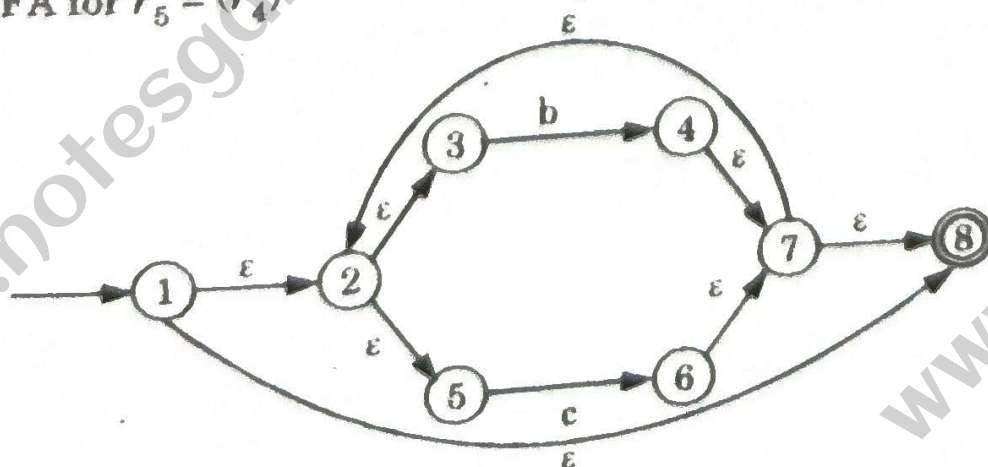
For $r_3 =$



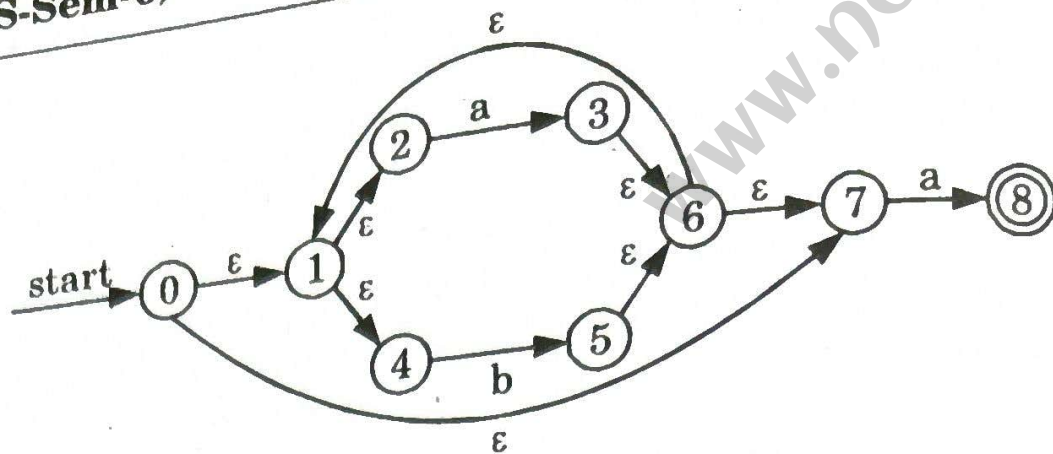
For $r_4 = b|c$,



The NFA for $r_5 = (r_4)^*$



Finally, NFA for $r_6 = r_{10} \cdot r_5 = a(b|c)^*$



Que 1.13. Discuss conversion of NFA into a DFA. Also give the algorithm used in this conversion.

Answer

Conversion from NFA to DFA :

Suppose there is an NFA $N < Q, \Sigma, q_0, \delta, F >$ which recognizes a language L . Then the DFA $D < Q', \Sigma, q_0, \delta', F' >$ can be constructed for language L as:

Step 1 : Initially $Q' = \phi$.

Step 2 : Add q_0 to Q' .

Step 3 : For each state in Q' , find the possible set of states for each input symbol using transition function of NFA. If this set of states is not in Q' , add it to Q' .

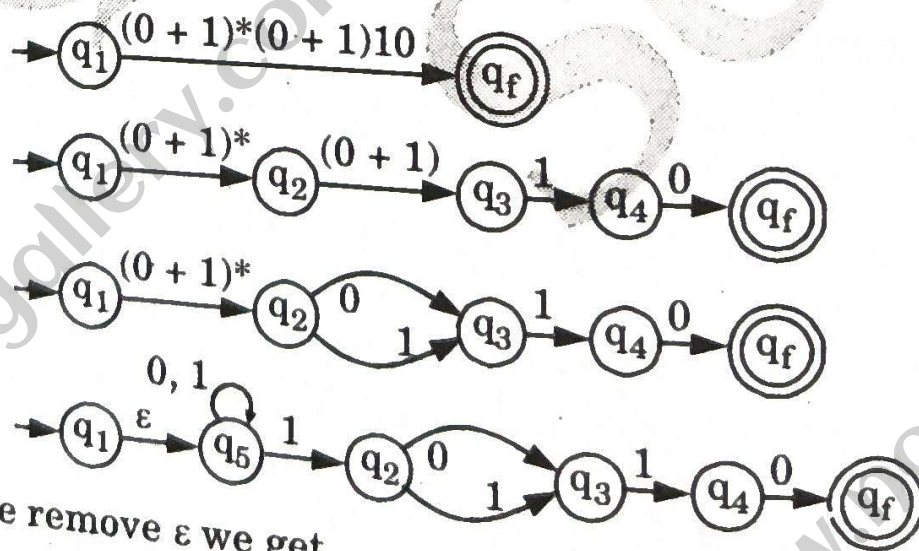
Step 4 : Final state of DFA will be all states which contain F (final states of NFA).

Que 1.14. Construct the minimized DFA for the regular expression $(0 + 1)^*(0 + 1)10$.

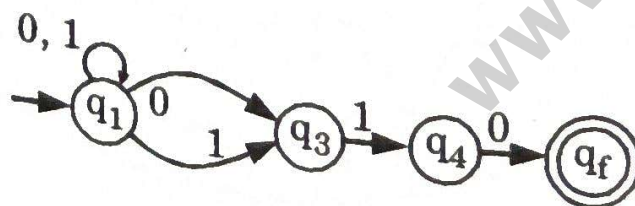
Answer

Given regular expression : $(0 + 1)^*(0 + 1)10$

NFA for given regular expression :



If we remove ϵ we get



$\because \epsilon$ can be neglected so $q_1 = q_5$

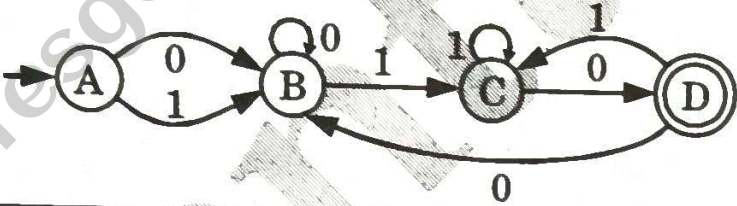
Now, we convert above NFA into DFA :
Transition table for NFA :

δ/Σ	0	1
$\rightarrow q_1$	$q_1 q_3$	$q_1 q_3$
q_3	ϕ	q_4
q_4	q_f	ϕ
$* q_f$	ϕ	ϕ

Transition table for DFA :

δ/Σ	0	1	Let
$\rightarrow q_1$	$q_1 q_3$	$q_1 q_3$	q_1 as A
$q_1 q_3$	$q_1 q_3$	$q_1 q_3 q_4$	$q_1 q_3$ as B
$q_1 q_3 q_4$	$q_1 q_3 q_f$	$q_1 q_3 q_4$	$q_1 q_3 q_4$ as C
$* q_1 q_3 q_f$	$q_1 q_3$	$q_1 q_3 q_4$	$q_1 q_3 q_f$ as D

Transition diagram for DFA :



δ/Σ	0	1
$\rightarrow A$	B	B
B	B	C
C	D	C
*D	B	C

For minimization divide the rows of transition table into 2 sets, as
Set-1 : It consists of non-final state rows.

A	B	B
B	B	C
C	D	C

Set-2 : It consists of final state rows.

*D	B	C
----	---	---

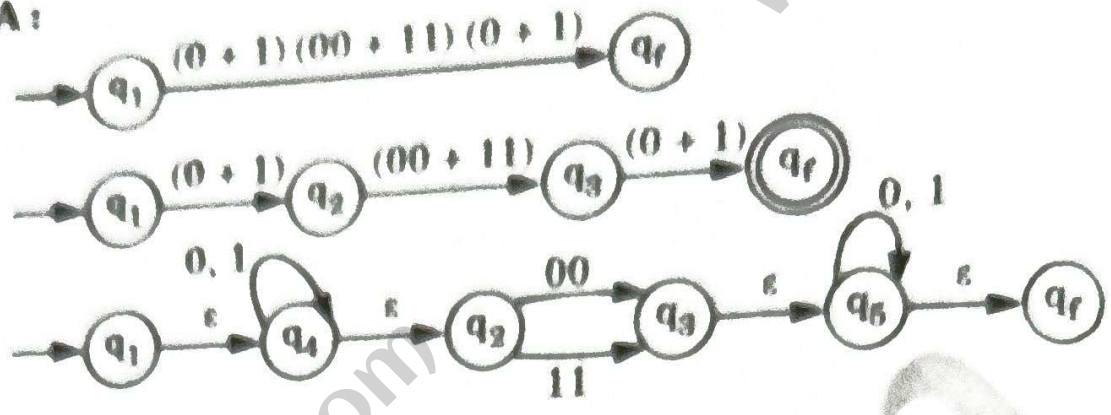
No two rows are similar.
So, the DFA is already minimized.

Que 1.15. Construct the NFA and DFA for the following regular expression.

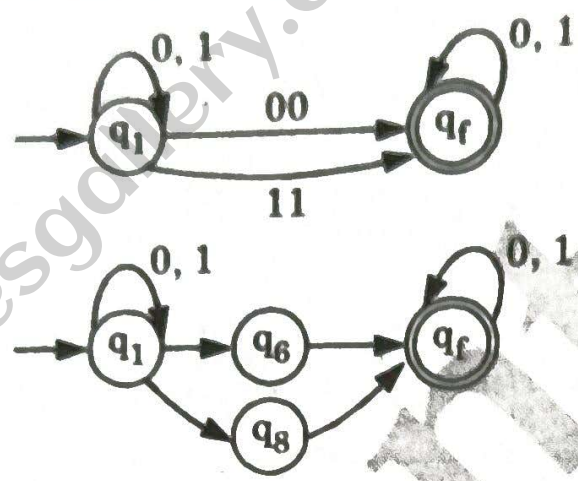
$(0 + 1)^* (00 + 11)(0 + 1)^*$

Answer

NFA :



If we remove ϵ we get



[$\because \epsilon$ can be neglected, so, $q_4 = q_2 = q_3 = q_5$]

Now, we convert above NFA to DFA

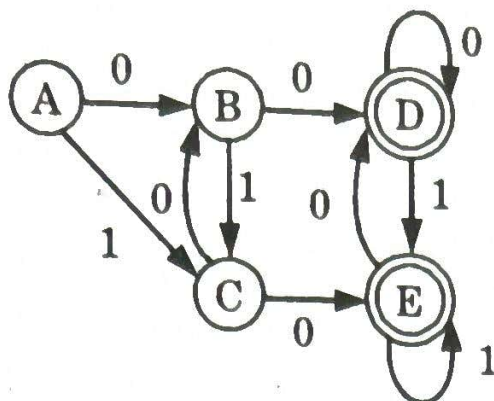
Transition table for NFA :

δ/Σ	0	1
$\rightarrow q_1$	q_1, q_6	q_1, q_8
q_6	q_f	ϕ
q_8	ϕ	q'_f
q_f	q_f	q_f

Transition table for DFA :

δ/Σ	0	1	Let
$\rightarrow q_1$	q_1, q_6	q_1, q_8	q_1 as A
q_1, q_6	q_1, q_6, q_f	q_1, q_8	$q_1 q_6$ as B
q_1, q_8	q_1, q_6, q_f	q_1, q_8, q_f	$q_1 q_8$ as C
q_1, q_6, q_f	q_1, q_6, q_f	q_1, q_8, q_f	q_1, q_6, q_f as D
q_1, q_8, q_f	q_1, q_6, q_f	q_1, q_8, q_f	q_1, q_8, q_f as E

Transition diagram for DFA :



Que 1.16. How does finite automata useful for lexical analysis ?

Answer

1. Lexical analysis is the process of reading the source text of a program and converting it into a sequence of tokens.
2. The lexical structure of every programming language can be specified by a regular language, a common way to implement a lexical analyzer is to :
 - a. Specify regular expressions for all of the kinds of tokens in the language.
 - b. The disjunction of all of the regular expressions thus describes any possible token in the language.
 - c. Convert the overall regular expression specifying all possible tokens into a Deterministic Finite Automaton (DFA).
 - d. Translate the DFA into a program that simulates the DFA. This program is the lexical analyzer.
3. This approach is so useful that programs called lexical analyzer generators exist to automate the entire process.

Que 1.17. Write down the regular expression for

1. The set of all string over $\{a, b\}$ such that fifth symbol from right is a .
2. The set of all string over $\{0, 1\}$ such that every block of four consecutive symbol contain at least two zero.

Answer

1. DFA for all strings over $\{a, b\}$ such that fifth symbol from right is a :
Regular expression : $(a + b)^* a (a + b) (a + b) (a + b) (a + b)$
2. **Regular expression :**
 $[00(0 + 1) (0 + 1) 0(0 + 1) 0(0 + 1) + 0(0 + 1) (0 + 1) 0 + (0 + 1) 00(0 + 1) + (0 + 1) 0(0 + 1) 0 + (0 + 1) (0 + 1) 00]$

Que 1.18. Convert following NFA to equivalent DFA and hence minimize the number of states in the DFA.

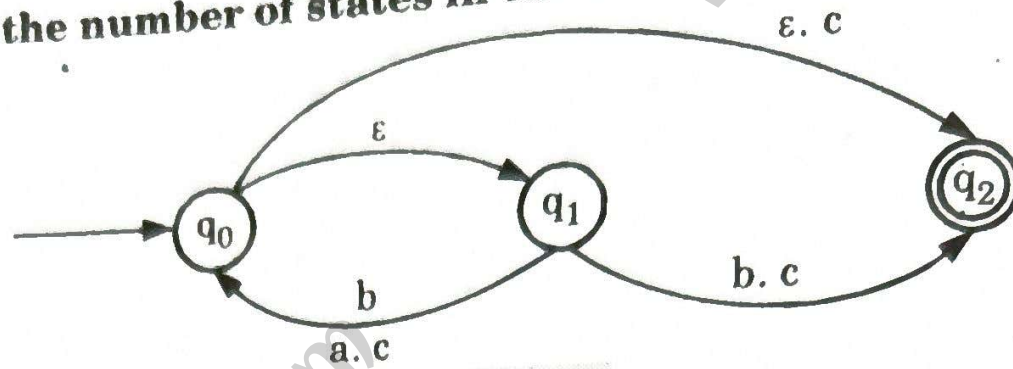


Fig. 1.18.1.

Answer

Transition table for ϵ -NFA :

δ/Σ	a	b	c	ϵ
q_0	ϕ	q_1	q_2	$\{q_1, q_2\}$
q_1	q_0	q_2	$\{q_0, q_2\}$	ϕ
q_2	ϕ	ϕ	ϕ	ϕ

ϵ -closure of $\{q_0\} = \{q_0, q_1, q_2\}$

ϵ -closure of $\{q_1\} = \{q_1\}$

ϵ -closure of $\{q_2\} = \{q_2\}$

Transition table for NFA :

δ/Σ	a	b	c
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$
$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_2\}$	$\{q_0, q_1, q_2\}$
$\{q_2\}$	ϕ	ϕ	ϕ

Let

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

$\{q_1, q_2\} = B$

$\{q_2\} = C$

Transition table for DFA :

δ/Σ	a	b	c
A	A	B	A
B	A	C	A
C	ϕ	ϕ	ϕ

Transition table for DFA :

δ/Σ	a	b	c
A	A	B	A
B	A	C	A
C	ϕ	ϕ	ϕ

So, DFA is given by

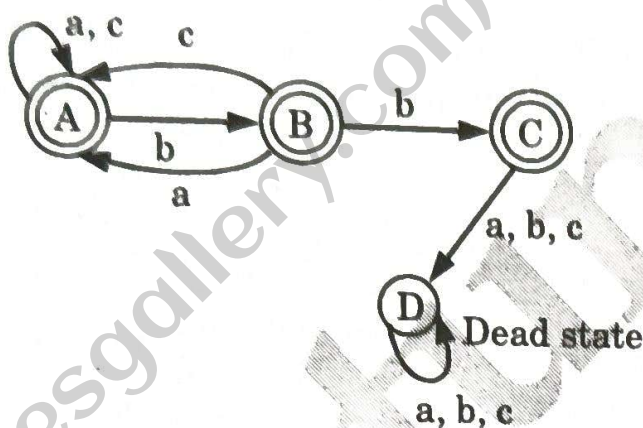


Fig. 1.18.2.

Que 1.19. Write a regular expression to represent a language consisting of strings made up of odd number of *a* and odd number of *b*.

AKTU 2020-21, Marks 05

Answer

Regular expression with odd number of *a* and odd number of *b* will be

$$R.E. = (aa + bb)^* (ab + ba) ((aa + bb)^*$$

$$(ab + ba (aa + bb)^* (ab + ba))^* (aa + bb)^*$$

PART-4

Implementation of Lexical Analyzers, Lexical Analyzer Generator, LEX Compiler.

Que 1.20. Explain the implementation of lexical analyzer.

Answer

Lexical analyzer can be implemented in following step :

1. Input to the lexical analyzer is a source program.

1-22 C (CS-Sem-6)

2. By using input buffering scheme, it scans the source program.
3. Regular expressions are used to represent the input patterns.
4. Now this input pattern is converted into NFA by using finite automation machine.
5. This NFA are then converted into DFA and DFA are minimized by using different method of minimization.
6. The minimized DFA are used to recognize the pattern and broken into lexemes.
7. Each minimized DFA is associated with a phase in a programming language which will evaluate the lexemes that match the regular expression.

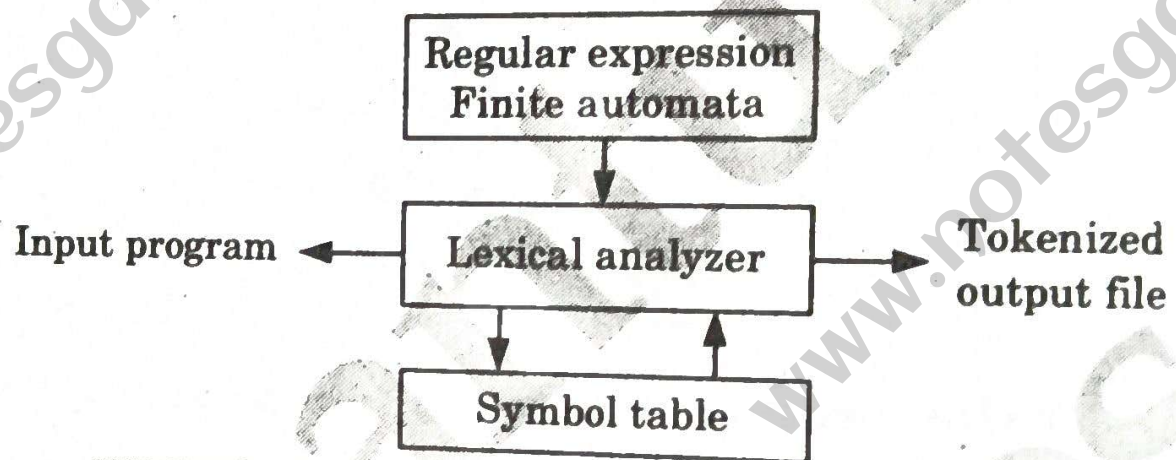


Fig. 1.20.1. Implementation of lexical analyzer.

8. The tool then constructs a state table for the appropriate finite state machine and creates program code which contains the table, the evaluation phases, and a routine which uses them appropriately.

Que 1.21. Write short notes on lexical analyzer generator.

Answer

1. For efficient design of compiler, various tools are used to automate the phases of compiler. The lexical analysis phase can be automated using a tool called LEX.
2. LEX is a Unix utility which generates lexical analyzer.
3. The lexical analyzer is generated with the help of regular expressions.
4. LEX lexer is very fast in finding the tokens as compared to handwritten LEX program in C.
5. LEX scans the source program in order to get the stream of tokens and these tokens can be related together so that various programming structure such as expression, block statement, control structures, procedures can be recognized.

Que 1.22. Explain the automatic generation of lexical analyzer.

Answer

1. Automatic generation of lexical analyzer is done using LEX programming language.
2. The LEX specification file can be denoted using the extension .l (often pronounced as dot L).
3. For example, let us consider specification file as x.l.
4. This x.l file is then given to LEX compiler to produce lex.yy.c as shown in Fig. 1.22.1. This lex.yy.c is a C program which is actually a lexical analyzer program.



Fig. 1.22.1.

5. The LEX specification file stores the regular expressions for the token and the lex.yy.c file consists of the tabular representation of the transition diagrams constructed for the regular expression.
6. In specification file, LEX actions are associated with every regular expression.
7. These actions are simply the pieces of C code that are directly carried over to the lex.yy.c.
8. Finally, the C compiler compiles this generated lex.yy.c and produces an object program a.out as shown in Fig. 1.22.2.
9. When some input stream is given to a.out then sequence of tokens gets generated. The described scenario is shown in Fig. 1.22.2.

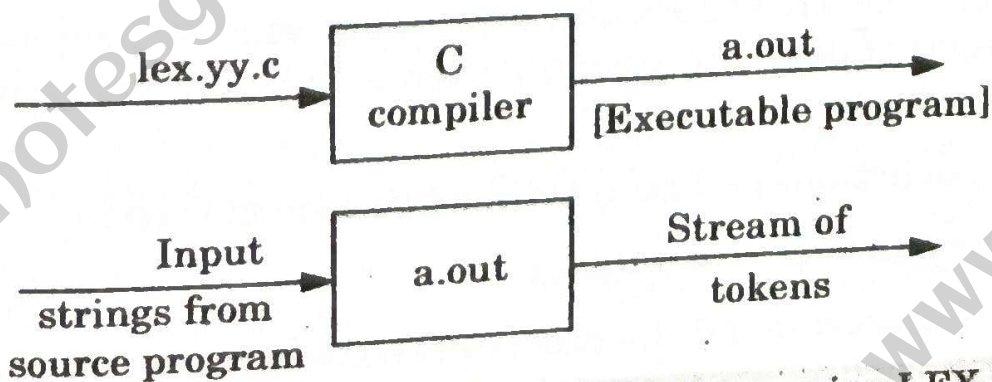


Fig. 1.22.2. Generation of lexical analyzer using LEX.

Que 1.23. Explain different parts of LEX program.

Answer

The LEX program consists of three parts :

% {

Declaration section

% }

% %

Rule section

% %

Auxiliary procedure section

1. Declaration section :

- In the declaration section, declaration of variable constants can be done.
- Some regular definitions can also be written in this section.
- The regular definitions are basically components of regular expressions.

2. Rule section :

- The rule section consists of regular expressions with associated actions. These translation rules can be given in the form as :

R_1 {action₁}

R_2 {action₂}

.

.

R_n {action_n}

Where each R_i is a regular expression and each action_i is a program fragment describing what action is to be taken for corresponding regular expression.

- These actions can be specified by piece of C code.

3. Auxiliary procedure section :

- In this section, all the procedures are defined which are required by the actions in the rule section.
- This section consists of two functions :
 - main() function

ii. yywrap() function

Que 1.24. Write a LEX program to identify few reserved words of C language.

Answer

```
%{
int count;
/*program to recognize the keywords*/
%}
%%
[%\t] +      /* "+" indicates zero or more and this pattern is use for
               ignoring the white spaces*/
auto | double | if | static | break | else | int | struct | case |
enum | long | switch | char | extern | near | typedef | const | float |
register | union | unsigned | void | while | default
printf("C keyword(%d) : \t %s",count,yytext);
[a-zA-Z]+ { printf("%s: is not the keyword\n", yytext);
%%
main()
{
yylex();
}
```

Que 1.25. What are the various LEX actions that are used in LEX programming ?

Answer

There are following LEX actions that can be used for ease of programming using LEX tool :

1. **BEGIN** : It indicates the start state. The lexical analyzer starts at state 0.
2. **ECHO** : It emits the input as it is.
3. **yytext()** :
 - a. yytext is a null terminated string that stores the lexemes when lexer recognizes the token from input token.

- b. When new token is found the contents of `yytext` are replaced by new token.
4. **yylex()** : This is an important function. The function `yylex()` is called when scanner starts scanning the source program.
5. **yywrap()** :
- The function `yywrap()` is called when scanner encounter end of file.
 - If `yywrap()` returns 0 then scanner continues scanning.
 - If `yywrap()` returns 1 that means end of file is encountered.
6. **yyin** : It is the standard input file that stores input source program.
7. **yylen** : `yylen` stores the length or number of characters in the input string.

Que 1.26. Explain the term token, lexeme and pattern.

Answer

Token :

1. A token is a pair consisting of a token name and an optional attribute value.
2. The token name is an abstract symbol representing a kind of lexical unit.
3. Tokens can be identifiers, keywords, constants, operators and punctuation symbols such as commas and parenthesis.

Lexeme :

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

Pattern :

1. A pattern is a description of the form that the lexemes of a token may take.
2. Regular expressions play an important role for specifying patterns.
3. If a keyword is considered as token, pattern is just sequence of characters.

PART-5*Formal Grammars and their Application to Syntax Analysis,
BNF Notation.***Que 1.27. Describe grammar.****Answer**

A grammar or phrase structured grammar is combination of four tuples and can be represented as $G(V, T, P, S)$. Where,

1. V is finite non-empty set of variables/non-terminals. Generally non-terminals are represented by capital letters like $A, B, C, \dots, X, Y, Z$.
2. T is finite non-empty set of terminals, sometimes also represented by Σ or V_T . Generally terminals are represented by $a, b, c, x, y, z, \alpha, \beta, \gamma$ etc.
3. P is finite set whose elements are in the form $\alpha \rightarrow \beta$. Where α and β are strings, made up by combination of V and T i.e., $(V \cup T)$. α has at least one symbol from V . Elements of P are called productions or production rule or rewriting rules.
4. S is special variable/non-terminal known as starting symbol.

While writing a grammar, it should be noted that $V \cap T = \phi$, i.e., no terminal can belong to set of non-terminals and no non-terminal can belong to set of terminals.

Que 1.28. What is Context Free Grammar (CFG) ? Explain.**Answer**

Context free grammar :

1. A CFG describes a language by recursive rules called productions.
2. A CFG can be described as a combination of four tuples and represented by $G(V, T, P, S)$.

where,

$V \rightarrow$ set of variables or non-terminal represented by $A, B, \dots, Y, Z$.

$T \rightarrow$ set of terminals represented by $a, b, c, \dots, x, y, z, +, -, *, (,)$ etc.

$S \rightarrow$ starting symbol.

1-28 C (CS-Sem-6)

- $P \rightarrow$ set of productions.
3. The production used in CFG must be in the form of $A \rightarrow \alpha$, where A is a variable and α is string of symbols $(V \cup T)^*$.
4. The example of CFG is :

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

where $V = \{E\}, T = \{+, *, (,), id\}$

$S = \{E\}$ and production P is given as :

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow id$$

Que 1.29. Explain formal grammar and its application to syntax analyzer.

Answer

1. Formal grammar represents the specification of programming language with the use of production rules.
2. The syntax analyzer basically checks the syntax of the language.
3. A syntax analyzer takes the tokens from the lexical analyzer and groups them in such a way that some programming structure can be recognized.
4. After grouping the tokens if at all any syntax cannot be recognized then syntactic error will be generated.
5. This overall process is called syntax checking of the language.
6. This syntax can be checked in the compiler by writing the specifications.
7. Specification tells the compiler how the syntax of the programming language should be.

Que 1.30. Write a CFG to represent the language
 $L = \{a^{n+m} b^n \mid m, n \geq 1\}.$

Answer

Given language

Let

$$L = \{a^{n+m} b^n \mid m, n \geq 1\}$$

$$n + m = k. \text{ Since } m, n \geq 1 \text{ so } k \geq 2$$

AKTU 2020-21, Marks 05

$$L = \{a^k b^n \mid k \geq 2, n \geq 1\}$$

CFG is given by

$$S \rightarrow AS_1$$

$$S_1 \rightarrow aa S_1 b \mid \epsilon$$

$$A \rightarrow aA \mid a$$

Que 1.31. Write short note on BNF notation.

Answer

BNF notation :

1. The BNF (Backus-Naur Form) is a notation technique for context free grammar. This notation is useful for specifying the syntax of the language.

2. The BNF specification is as :

$$\langle \text{symbol} \rangle := \text{Exp1} \mid \text{Exp2} \mid \text{Exp3} \dots$$

Where $\langle \text{symbol} \rangle$ is a non terminal, and Exp1, Exp2 is a sequence of symbols. These symbols can be combination of terminal or non terminals.

3. For example :

$$\langle \text{Address} \rangle := \langle \text{fullname} \rangle : ", \langle \text{street} \rangle ", \langle \text{zip code} \rangle$$

$$\langle \text{fullname} \rangle := \langle \text{firstname} \rangle "- \langle \text{middle name} \rangle "- \langle \text{surname} \rangle$$

$$\langle \text{street} \rangle := \langle \text{street name} \rangle ", \langle \text{city} \rangle$$

We can specify first name, middle name, surname, street name, city and zip code by valid strings.

4. The BNF notation is more often non-formal and in human readable form. But commonly used notations in BNF are :

a. Optional symbols are written with square brackets.

b. For repeating the symbol for 0 or more number of times asterisk can be used.

For example : $\{ \text{name} \}^*$

c. For repeating the symbols for at least one or more number of times + is used.

For example : $\{ \text{name} \}^+$

d. The alternative rules are separated by vertical bar.

e. The group of items must be enclosed within brackets.

PART-6

Ambiguity, YACC.

Que 1.32. What is an ambiguous grammar? Is the following grammar is ambiguous? Prove $EE + | E(E) | id$. The grammar should be moved to the next line, centered.

Answer

Ambiguous grammar: A context free grammar G is ambiguous if there is at least one string in $L(G)$ having two or more distinct derivation tree.

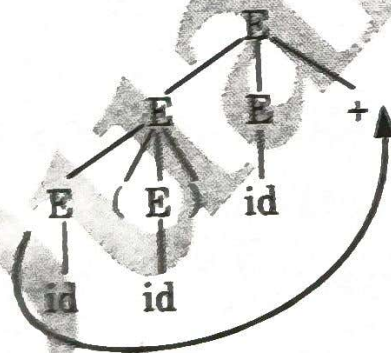
Proof: Let production rule is given as :

$$E \rightarrow EE+$$

$$E \rightarrow E(E)$$

$$E \rightarrow id$$

Parse tree for $id(id)id +$ is



Only one parse tree is possible for $id(id)id+$ so, the given grammar is unambiguous.

Que 1.33. Consider the grammar G given as follows :

$$S \rightarrow AB | aaB$$

$$A \rightarrow a | Aa$$

$$B \rightarrow b$$

Determine whether the grammar G is ambiguous or not. If G is ambiguous then construct an unambiguous grammar equivalent to G .

Answer

Given :

$$S \rightarrow AB | aaB$$

$$A \rightarrow a | Aa$$

$$B \rightarrow b$$

Let us generate string aab from the given grammar. Parse tree for generating string aab are as follows :

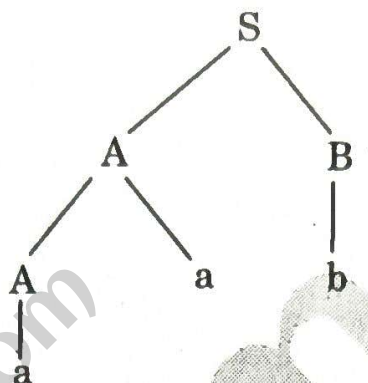
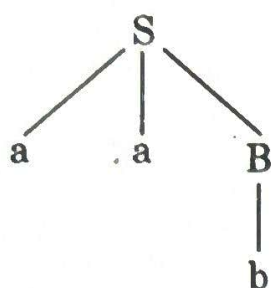


Fig. 1.33.1.

Here for the same string, we are getting more than one parse tree. Hence, grammar is an ambiguous grammar.

The grammar

$$S \rightarrow AB$$

$$A \rightarrow Aa/a$$

$$B \rightarrow b$$

is an unambiguous grammar equivalent to G . Now this grammar has only one parse tree for string aab .

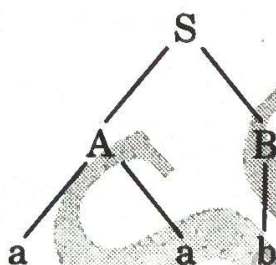


Fig. 1.33.2.

Que 1.34.

Check whether given grammar is ambiguous or not. If ambiguous then convert it into unambiguous grammar :

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

AKTU 2020-21, Marks 05

OR

Check whether given grammar is ambiguous or not. If ambiguous then convert it into unambiguous grammar :

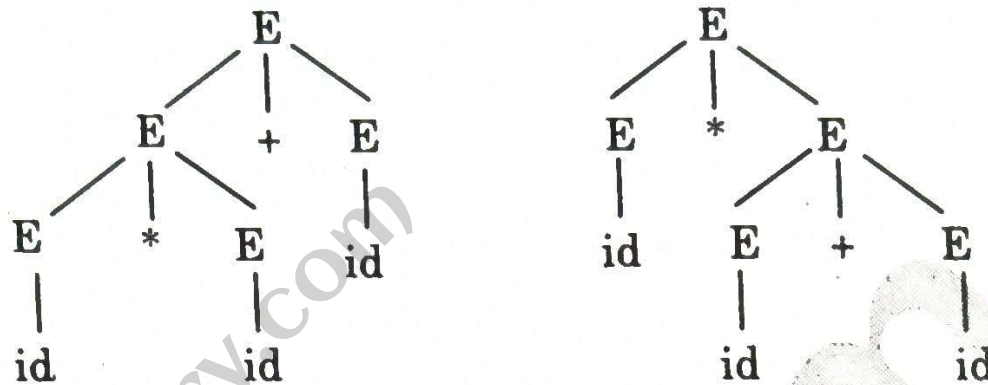
$$E \rightarrow E + E \mid E * E \mid id.$$

AKTU 2023-24, Marks 10

Answer

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

For string $id * id + id$



Here for the same string, we are getting more than one parse tree. Hence, grammar is an ambiguous grammar.

For unambiguous grammar apply right recursion as :

$$E \rightarrow E + T$$

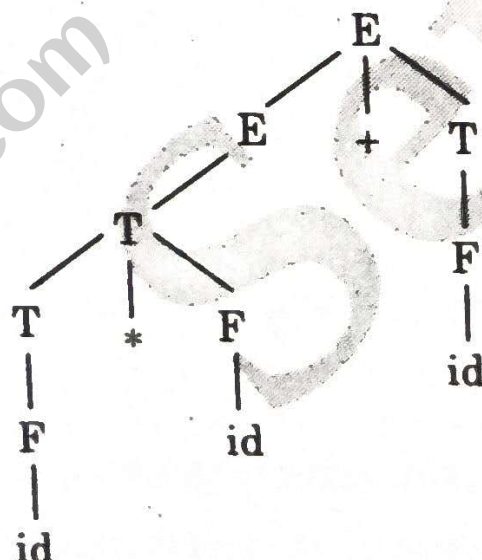
$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow id$$

Now, this grammar has only one parse tree for string $id * id + id$.



Que 1.35. Write short note on :

- i. Context free grammar
- ii. YACC parser generator

OR

Write a short note on YACC parser generator.

Answer

- i. **Context free grammar** : Refer Q. 1.28, Page 1-27C, Unit-1.
- ii. **YACC parser generator** :
 1. YACC (Yet Another Compiler - Compiler) is the standard parser generator for the Unix operating system.
 2. An open source program, YACC generates code for the parser in the C programming language.
 3. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code.

PART-7

The Syntactic Specification of Programming Languages : Context Free Grammar (CFG), Derivation and Parse Trees, Capabilities of CFG.

Que 1.36. Define parse tree. What are the conditions for constructing a parse tree from a CFG ?

Answer

Parse tree :

1. A parse tree is an ordered tree in which left hand side of a production represents a parent node and children nodes are represented by the production's right hand side.
2. Parse tree is the tree representation of deriving a Context Free Language (CFL) from a given Context Free Grammar (CFG). These types of trees are sometimes called derivation trees.

Conditions for constructing a parse tree from a CFG :

- i. Each vertex of the tree must have a label. The label is a non-terminal or terminal or null (ϵ).
- ii. The root of the tree is the start symbol, i.e., S .
- iii. The label of the internal vertices is non-terminal symbols $\in V_N$.
- iv. If there is a production $A \rightarrow X_1 X_2 \dots X_K$. Then for a vertex, label A , the children of that node, will be $X_1 X_2 \dots X_K$.

1-34 C (CS-Sem-6)

- v. A vertex n is called a leaf of the parse tree if its label is a terminal symbol $\in \Sigma$ or null (ϵ).

Que 1.37. How derivation is defined in CFG ?

Answer

1. A derivation is a sequence of tokens that is used to find out whether a sequence of string is generating valid statement or not.
2. We can define the notations to represent a derivation.

3. First, we define two notations $\Rightarrow_G$ and $\xRightarrow{*}_G$.

4. If $\alpha \rightarrow \beta$ is a production of P in CFG and a and b are strings in $(V_n \cup V_t)^*$, then

$$aab \Rightarrow_G a\beta b.$$

5. We say that the production $\alpha \rightarrow \beta$ is applied to the string aab to obtain $a\beta b$ or we say that aab directly drives $a\beta b$.

6. Now suppose $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_m$ are string in $(V_n \cup V_t)^*$, $m \geq 1$ and

$$\alpha_1 \xRightarrow{*}_G \alpha_2, \alpha_2 \xRightarrow{*}_G \alpha_3, \alpha_3 \xRightarrow{*}_G \alpha_4, \dots, \alpha_{m-1} \xRightarrow{*}_G \alpha_m.$$

7. Then we say that $\alpha_1 \xRightarrow{*}_G \alpha_m$, i.e., we say α_1 drives α_m in grammar G . If α drives by exactly i steps, we say $\alpha_1 \xRightarrow{i}_G \beta$.

Que 1.38. What do you mean by left most derivation and right most derivation with example ?

Answer

Left most derivation : The derivation $S \rightarrow s$ is called a left most derivation, if the production is applied only to the left most variable (non-terminal) at every step.

Example : Let us consider a grammar G that consist of production rules $E \rightarrow E + E \mid E * E \mid id$.

Firstly take the production

$$E \rightarrow E + E \rightarrow \underline{E} * E + E$$

(Replace $E \rightarrow E * E$)

$$\begin{aligned} &\rightarrow id * \underline{E} + E && \text{(Replace } E \rightarrow id) \\ &\rightarrow id * id + \underline{E} && \text{(Replace } E \rightarrow id) \\ &\rightarrow id * id + id && \text{(Replace } E \rightarrow id) \end{aligned}$$

Right most derivation : A derivation $S \rightarrow s$ is called a right most derivation, if production is applied only to the right most variable (non-terminal) at every step.

Example : Let us consider a grammar G having production.

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

Start with production

$$\begin{aligned} E &\rightarrow E * \underline{E} \\ &\rightarrow E * E + \underline{E} && \text{(Replace } E \rightarrow E + E) \\ &\rightarrow E * \underline{E} + id && \text{(Replace } E \rightarrow id) \\ &\rightarrow E * id + id && \text{(Replace } E \rightarrow id) \\ &\rightarrow id * id + id && \text{(Replace } E \rightarrow id) \end{aligned}$$

Que 1.39. Describe the capabilities of CFG.

Answer

Various capabilities of CFG are :

1. Context free grammar is useful to describe most of the programming languages.
2. If the grammar is properly designed then an efficient parser can be constructed automatically.
3. Using the features of associativity and precedence information, grammars for expressions can be constructed.
4. Context free grammar is capable of describing nested structures like : balanced parenthesis, matching begin-end, corresponding if-then-else's and so on.

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. Explain in detail the process of compilation. Illustrate the output of each phase of compilation of the input " $a = (b + c) * (b + c) * 2$ ".

Ans. Refer Q. 1.1.

Q. 2. What do you mean by regular expression? Write the formal recursive definition of a regular expression.

Ans. Refer Q. 1.7.

Q. 3. Define and differentiate between DFA and NFA with an example.

Ans. Refer Q. 1.8.

Q. 4. Given a regular expression $a(b|c)^*$, construct the corresponding Non-deterministic Finite Automaton (NFA) that recognizes the language described by the regular expression.

Ans. Refer Q. 1.12.

Q. 5. Construct the minimized DFA for the regular expression $(0 + 1)^*(0 + 1) 10$.

Ans. Refer Q. 1.14.

Q. 6. Describe grammar.

Ans. Refer Q. 1.27.

Q. 7. Write short note on BNF notation.

Ans. Refer Q. 1.31.

Q. 8. Consider the grammar G given as follows :

$S \rightarrow AB | aaB$

$A \rightarrow a | Aa$

$B \rightarrow b$

Determine whether the grammar G is ambiguous or not. If G is ambiguous then construct an unambiguous grammar equivalent to G .

Ans. Refer Q. 1.33.

Q. 9. Write short note on :

- i. Context free grammar
- ii. YACC parser generator

OR

Write a short note on YACC parser generator.

Ans. Refer Q. 1.35.



2 UNIT

Basic Parsing Techniques

CONTENTS

Part-1 :	Basic Parsing Techniques : Parsers Shift Reduce Parsing	2-2C to 2-4C
Part-2 :	Operator Precedence Parsing	2-4C to 2-9C
Part-3 :	Top-down Parsing Predictive Parsers	2-9C to 2-17C
Part-4 :	Automatic Generation of Efficient Parser : LR Parsers The Canonical Collections of LR(0) Items	2-17C to 2-19C
Part-5 :	Constructing SLR Parsing Tables	2-19C to 2-31C
Part-6 :	Constructing canonical LR Parsing Tables	2-31C to 2-32C
Part-7 :	Constructing LALR Parsing Tables Using Ambiguous Grammars An Automatic Parser Generator Implementation of LR Parsing Tables	2-32C to 2-43C

PART-1*Basic Parsing Techniques : Parsers, Shift Reduce Parsing.*

Que 2.1. What is parser ? Write the role of parser. What are the most popular parsing techniques ?

OR

Explain about basic parsing techniques. What is top-down parsing ? Explain in detail.

Answer

A parser for any grammar is a program that takes as input string w and produces as output a parse tree for w .

Role of parser :

1. The role of parsing is to determine the syntactic validity of a source string.
2. Parser helps to report any syntax errors and recover from those errors.
3. Parser helps to construct parse tree and passes it to rest of phases of compiler.

There are basically two type of parsing techniques :

1. Top-down parsing :

- a. Top-down parsing attempts to find the left-most derivation for an input string w , that start from the root (or start symbol), and create the nodes in pre-defined order.
- b. In top-down parsing, the input string w is scanned by the parser from left to right, one symbol/token at a time.
- c. The left-most derivation generates the leaves of parse tree in left to right order, which matches to the input scan order.
- d. In the top-down parsing, parsing decisions are based on the lookahead symbol (or sequence of symbols).

2. Bottom-up parsing :

- a. Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of a grammar by finding out the right-most derivation of w in reverse.
- b. Parsing involves searching for the substring that matches the right side of any of the productions of the grammar.
- c. This substring is replaced by the left hand side non-terminal of the production.

- d. Process of replacing the right side of the production by the left side non-terminal is called "reduction".

Que 2.2. Discuss bottom-up parsing. What are bottom-up parsing techniques ?

Answer

Bottom-up parsing : Refer Q. 2.1, Page 2-2C, Unit-2.

Bottom-up parsing techniques are :

1. Shift-reduce parser :

- Shift-reduce parser attempts to construct parse tree from leaves to root and uses a stack to hold grammar symbols.
- A parser goes on shifting the input symbols onto the stack until a handle comes on the top of the stack.
- When a handle appears on the top of the stack, it performs reduction.

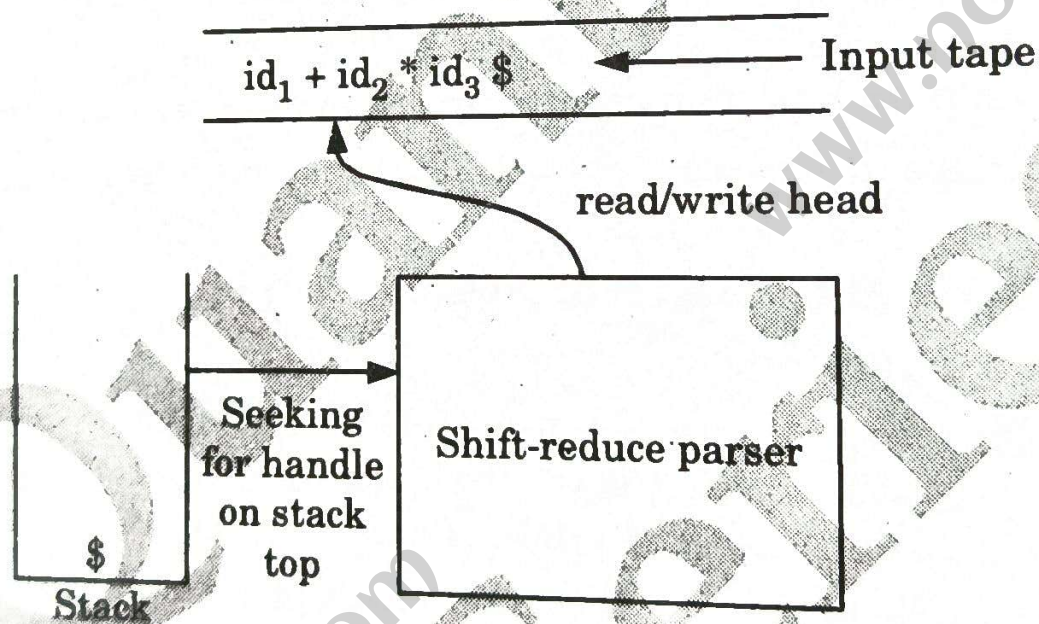


Fig. 2.2.1. Shift-reduce parser.

- d. This parser performs following basic operations :

- Shift
- Reduce
- Accept
- Error

- 2. LR parser :** LR parser is the most efficient method of bottom-up parsing which can be used to parse the large class of context free grammars. This method is called LR(k) parsing. Here
- L stands for left to right scanning.
 - R stands for right-most derivation in reverse.
 - k is number of input symbols. When k is omitted it is assumed to be 1.

Que 2.3. What are the common conflicts that can be encountered in shift-reduce parser ?

Answer

There are two most common conflict encountered in shift-reduce parser :

1. Shift-reduce conflict :

- a. The shift-reduce conflict is the most common type of conflict found in grammars.
- b. This conflict occurs because some production rule in the grammar is shifted and reduced for the particular token at the same time.
- c. This error is often caused by recursive grammar definitions where the system cannot determine when one rule is complete and another is just started.

2. Reduce-reduce conflict :

- a. A reduce-reduce conflict is caused when a grammar allows two or more different rules to be reduced at the same time, for the same token.
- b. When this happens, the grammar becomes ambiguous since a program can be interpreted more than one way.
- c. This error can be caused when the same rule is reached by more than one path.

PART-2

Operator Precedence Parsing.

Que 2.4. Explain operator precedence parsing with example.

Answer

1. A grammar G is said to be operator precedence if it posses following properties :
 - a. No production on the right side is ϵ .
 - b. There should not be any production rule possessing two adjacent non-terminals at the right hand side.
2. In operator precedence parsing, we will first define precedence relations $<\cdot \doteq$ and $\cdot >$ between pair of terminals. The meanings of these relations are :

$$p < \cdot q$$

$$p \doteq q$$

$$p \cdot > q$$

p gives more precedence than q .

p has same precedence as q .

p takes precedence over q .

Compiler Design

For example :

Consider the grammar for arithmetic expressions

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

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

1. Now consider the string $id + id * id$

2. We will insert \$ symbols at the start and end of the input string. We will also insert precedence operator by referring the precedence relation table.

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

3. We will follow following steps to parse the given string :

- Scan the input from left to right until first $\cdot >$ is encountered.
- Scan backwards over $=$ until $< \cdot$ is encountered.
- The handle is a string between $< \cdot$ and $\cdot >$.

4. The parsing can be done as follows :

$\$ < \cdot id \cdot > + < \cdot id \cdot > * < id \cdot > \$$	Handle id is obtained between $< \cdot >$. Reduce this by $E \rightarrow id$.
$E + < \cdot id \cdot > * < id \cdot > \$$	Handle id is obtained between $< \cdot >$. Reduce this by $E \rightarrow id$.
$E + E * < \cdot id \cdot > \$$	Handle id is obtained between $< \cdot >$. Reduce this by $E \rightarrow id$.
$E + E * E$	Remove all the non-terminals.
$+ *$	Insert \$ at the beginning and at the end. Also insert the precedence operators.
$\$ < \cdot + < \cdot * \cdot > \$$	The $*$ operator is surrounded by $< \cdot >$. This indicates that $*$ becomes handle. That means, we have to reduce $E * E$ operation first.
$\$ < \cdot + \cdot > \$$	Now $+$ becomes handle. Hence, we evaluate $E + E$.
$\$ \$$	Parsing is done.

Que 2.5.

Give the algorithm for computing precedence function.

Consider the following operator precedence matrix draw precedence graph and compute the precedence function :

	a	(	)	;	\$
A			>	>	>
(	<	<	=	<	
)			>	>	>
;	<	<	>	>	
\$	<	<			

Answer

Algorithm for computing precedence function :

Input : An operator precedence matrix.

Output : Precedence functions representing the input matrix or an indication that none exist.

Method :

1. Create symbols f_a and g_a for each a that is a terminal or \$.
2. Partition the created symbols into as many groups as possible, in such a way that if ab , then f_a and g_b are in the same group.
3. Create a directed graph whose nodes are the groups found in step 2. For any a and b , if $a < b$, place an edge from the group of g_b to the group of f_a . If $a > b$, place an edge from the group of f_a to that of g_b .
4. If the graph constructed in step 3 has a cycle, then no precedence functions exist. If there are no cycles, let $f(a)$ be the length of the longest path from the group of f_a ; let $g(b)$ be the length of the longest path from the group of g_b . Then there exists a precedence function.

Precedence graph for above matrix is :

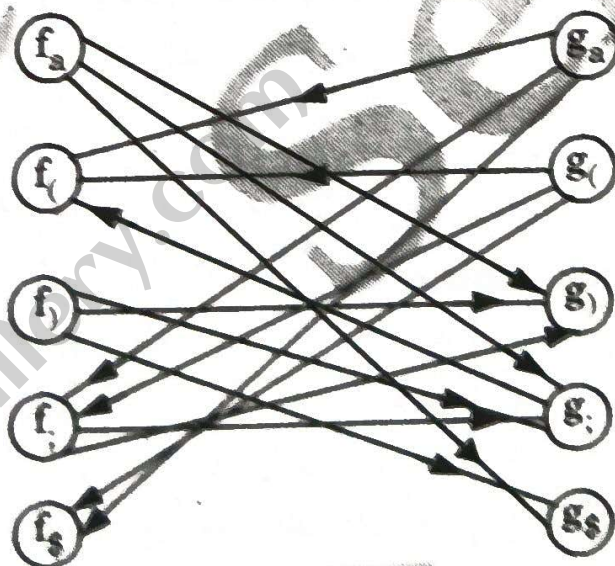


Fig. 2.5.1.

From the precedence graph, the precedence function using algorithm calculated as follows :

	(	(	)	;	\$
f	1	0	2	2	0
g	3	3	0	1	0

Que 2.6.

Give operator precedence parsing algorithm. Consider the following grammar and build up operator precedence table. Also parse the input string $(id + (id * id))$

$E \rightarrow E + T \mid T, T \rightarrow T * F \mid F, F \rightarrow (E) \mid id$

Answer

Operator precedence parsing algorithm :

Let the input string be $a_1, a_2, \dots, a_n \$$. Initially, the stack contains \$.

1. Set p to point to the first symbol of $w\$$.
2. Repeat: Let a be the topmost terminal symbol on the stack and let b be the current input symbol.
 - i. If only \$ is on the stack and only \$ is the input then accept and break.
 - else
 - begin
 - ii. If $a > b$ or $a = b$ then shift a onto the stack and increment p to next input symbol.
 - iii. else if $a < b$ then reduce b from the stack
 - iv. Repeat:
 - $c \leftarrow \text{pop the stack}$
 - v. Until the top stack terminal is related by $>$ to the terminal most recently popped.
 - else
 - vi. Call the error correcting routine
 - end

Operator precedence table :

	+	*	(	)	id	\$
+	$\cdot >$	$<$	$<$	$\cdot >$	$<$	$\cdot >$
*	$\cdot >$	$\cdot >$	$<$	$\cdot >$	$<$	$\cdot >$
(	$<$	$<$	$<$	$\cdot$	$<$	
)	$\cdot >$	$\cdot >$		$\cdot >$		$\cdot >$
id	$\cdot >$	$\cdot >$		$\cdot >$		$\cdot >$
\$	$<$	$<$	$<$		$<$	

Parsing :

$\$(\langle \cdot id \rangle + (\langle \cdot id \cdot \rangle * \langle \cdot id \cdot \rangle))\$$	Handle id is obtained between $\langle \cdot \cdot \rangle$ Reduce this by $F \rightarrow id$
$(F + (\langle \cdot id \cdot \rangle * \langle \cdot id \cdot \rangle))\$$	Handle id is obtained between $\langle \cdot \cdot \rangle$ Reduce this by $F \rightarrow id$
$(F + (F * \langle \cdot id \cdot \rangle))\$$	Handle id is obtained between $\langle \cdot \cdot \rangle$. Reduce this by $F \rightarrow id$
$(F + (F * F))$	Remove all the non-terminals.
$(+(*))$	Insert \$ at the beginning and at the end.
	Also insert the precedence operators.
$\$(\langle \cdot + \cdot \rangle (\langle \cdot * \cdot \rangle)) \$$	The $*$ operator is surrounded by $\langle \cdot \cdot \rangle$. This indicates that $*$ becomes handle. That means we have to reduce $T * F$ operation first.
$\$ \langle \cdot + \cdot \rangle \$$	Now $+$ becomes handle. Hence we evaluate $E + T$.
$\$ \$$	Parsing is done.

Que 2.7. Find the precedence and function table of the following grammar by using operator precedence technique.

$P \rightarrow SR \mid S, R \rightarrow bSR \mid bS, S \rightarrow WbS \mid W, W \rightarrow L * W \mid L, L \rightarrow id.$

AKTU 2020-21, Marks 10

Answer

Given grammar :

$P \rightarrow SR \mid S$

$R \rightarrow bSR \mid bS$

$S \rightarrow WbS \mid W$

$W \rightarrow L * W \mid L$

$L \rightarrow id$

The given grammar is not a operator grammar. So, first we convert the given grammar into operator grammar.

$P \rightarrow SbSbS \mid S$

$R \rightarrow bSbS \mid bS$

$S \rightarrow WbS \mid W$

$W \rightarrow L * W \mid L$

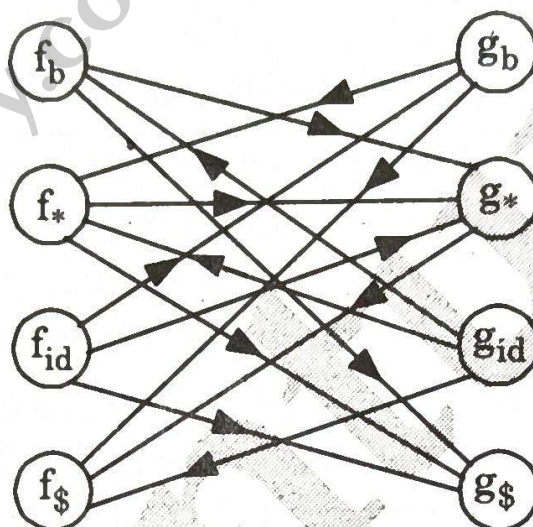
$L \rightarrow id$

Operator precedence table :

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

Precedence function table : To construct the table, we first have to create precedence graph for operator precedence table.

Precedence graph :



Function table :

	b	*	id	\$
f	2	2	2	0
g	3	1	2	0

PART-3

Top-down Parsing, Predictive Parsers.

Que 2.8.

What are the problems with top-down parsing ?

OR

What is top down parsing ? What are the problems in top down parsing ? Explain each with suitable example.

AKTU 2022-23, Marks 10

Answer

Top-down parsing : Refer Q. 2.1, Page 2-2C, Unit-2.

Problems with top-down parsing are :

1. **Backtracking :**

- Backtracking is a technique in which for expansion of non-terminal symbol, we choose alternative and if some mismatch occurs then we try another alternative if any.

- b. If for a non-terminal, there are multiple production rules beginning with the same input symbol then to get the correct derivation, we need to try all these alternatives.
- c. Secondly, in backtracking, we need to move some levels upward in order to check the possibilities. This increases lot of overhead in implementation of parsing.
- d. Hence, it becomes necessary to eliminate the backtracking by modifying the grammar.

2. Left recursion :

- a. The left recursive grammar is represented as :

$$A \Rightarrow^+ A\alpha$$

- b. Here $\Rightarrow^+$ means deriving the input in one or more steps.
- c. Here, A is a non-terminal and α denotes some input string.
- d. If left recursion is present in the grammar then top-down parser can enter into infinite loop.

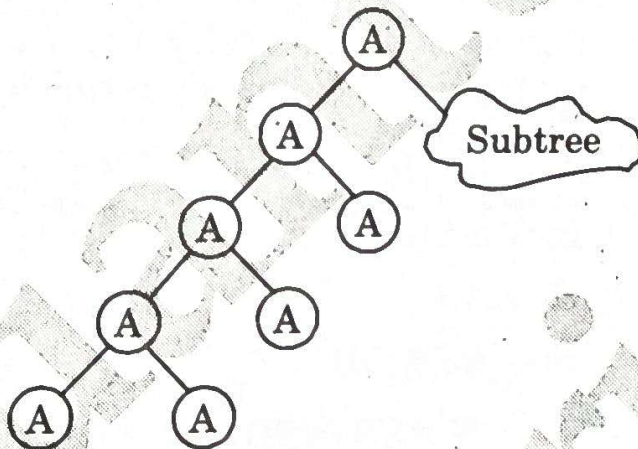


Fig. 2.8.1. Left recursion.

- e. This causes major problem in top-down parsing and therefore elimination of left recursion is must.

3. Left factoring :

- a. Left factoring is occurred when it is not clear that which of the two alternatives is used to expand the non-terminal.
- b. If the grammar is not left factored then it becomes difficult for the parser to make decisions.

Que 2.9. What do you understand by left factoring and left recursion and how it is eliminated ?

Answer

Left factoring and left recursion : Refer Q. 2.8, Page 2-9C, Unit-2.

Left factoring can be eliminated by the following scheme :

- a. In general if

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n | \gamma_1 | \dots | \gamma_m$$

is a production then it is not possible for parser to take a decision whether to choose first rule or second.

- b. In such situation, the given grammar can be left factored as

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \dots \mid \gamma_m \mid$$

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

Left recursion can be eliminated by following scheme :

- a. In general if

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \dots \mid \beta_m$$

where no β_i begin with an A.

- b. In such situation we replace the A-productions by

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

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

Que 2.10. Eliminate left recursion from the following grammar

$$S \rightarrow AB, A \rightarrow BS \mid b, B \rightarrow SA \mid a$$

Answer

$$S \rightarrow AB$$

$$A \rightarrow BS \mid b$$

$$B \rightarrow SA \mid a$$

$$S \rightarrow AB$$

$$S \rightarrow BSB \mid bB$$

$$S \rightarrow \underbrace{S}_{\alpha} \underbrace{ASB}_{\beta_1} \mid \underbrace{aSB}_{\beta_2} \mid \underbrace{bB}_{\beta_2}$$

$$S \rightarrow aSBS' \mid bBS'$$

$$S' \rightarrow ASBS' \mid \epsilon$$

$$B \rightarrow ABA \mid a$$

$$B \rightarrow \underbrace{B}_{\alpha} \underbrace{ABBA}_{\beta_1} \mid \underbrace{bBA}_{\beta_2} \mid \underbrace{a}_{\beta_2}$$

$$B \rightarrow bBA B' \mid aB'$$

$$B' \rightarrow ABBA B' \mid \epsilon$$

$$A \rightarrow BS \mid a$$

$$A \rightarrow SAS \mid aS \mid a$$

$$A \rightarrow \underbrace{A}_{\alpha} \underbrace{BAAB}_{\beta_1} \mid \underbrace{aAB}_{\beta_2} \mid \underbrace{a}_{\beta_2}$$

$$A \rightarrow aABA' \mid aA'$$

$$A' \rightarrow BAAB A' \mid \epsilon$$

The production after left recursion is

$$\begin{aligned}
 S &\rightarrow aSB S' \mid bBS' \\
 S' &\rightarrow ASB S' \mid \epsilon \\
 A &\rightarrow aABA' \mid aA' \\
 A' &\rightarrow BAABA' \mid \epsilon \\
 B &\rightarrow bBA B' \mid aB' \\
 B' &\rightarrow ABBA B' \mid \epsilon
 \end{aligned}$$

Que 2.11. Write short notes on top-down parsing. What are top-down parsing techniques ?

Answer

Top-down parsing : Refer Q. 2.1, Page 2-2C, Unit-2.

Top-down parsing techniques are :

1. Recursive-descent parsing :

- i. A top-down parser that executes a set of recursive procedures to process the input without backtracking is called recursive-descent parser and parsing is called recursive-descent parsing.
- ii. The recursive procedures can be easy to write and fairly efficient if written in a language that implements the procedure call efficiently.

2. Predictive parsing :

- i. A predictive parsing is an efficient way of implementing recursive-descent parsing by handling the stack of activation records explicitly.
- ii. The predictive parser has an input, a stack, a parsing table, and an output. The input contains the string to be parsed, followed by \$, the right end-marker.

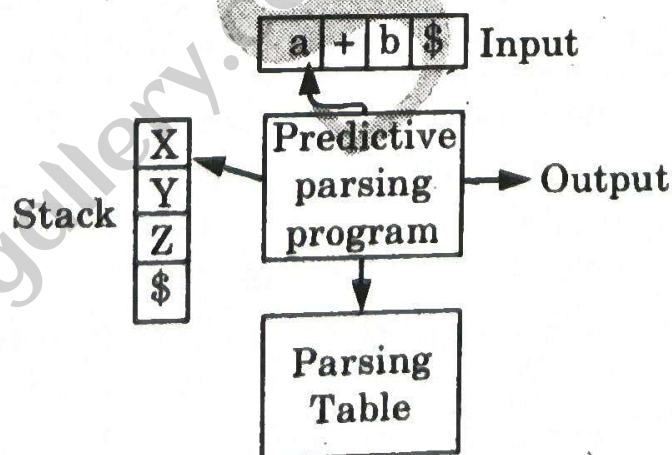


Fig. 2.11.1. Model of a predictive parser.

- iii. The stack contains a sequence of grammar symbols with \$ indicating bottom of the stack. Initially, the stack contains the start symbol of the grammar preceded by \$.

Compiler Design

- iv. The parsing table is a two-dimensional array $M[A, a]$ where 'A' is a non-terminal and 'a' is a terminal or the symbol \$.
- v. The parser is controlled by a program that behaves as follows :
The program determines X symbol on top of the stack, and 'a' the current input symbol. These two symbols determine the action of the parser.
- vi. Following are the possibilities :
- If $X = a = \$$, the parser halts and announces successful completion of parsing.
 - If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
 - If X is a non-terminal, the program consults entry $M[X, a]$ of the parsing table M . This entry will be either an X -production of the grammar or an error entry.

Que 2.12. Differentiate between top-down and bottom-up parser.

Under which conditions predictive parsing can be constructed for a grammar ?

Answer

S.No.	Top-down parser	Bottom-up parser
1.	In top-down parser left recursion is done.	In bottom-up parser right-most derivation is done.
2.	Backtracking is possible.	Backtracking is not possible.
3.	In this, input token are popped off the stack.	In this, input token are pushed on the stack.
4.	First and follow are defined in top-down parser.	First and follow are used in bottom-up parser.
5.	Predictive parser and recursive descent parser are top-down parsing techniques.	Shift-reduce parser, operator precedence parser, and LR parser are bottom-up parsing technique.

Predictive parsing can be constructed if the following condition holds :

- Every grammar must be recursive in nature.
- Each grammar must be left factored.

Que 2.13. What are the problems with top-down parsing ? Write the algorithm for FIRST and FOLLOW.

Answer

Problems with top-down parsing : Refer Q. 2.8, Page 2-9C, Unit-2.

Algorithm for FIRST and FOLLOW :

1. FIRST function :

- i. FIRST (X) is a set of terminal symbols that are first symbols appearing at R.H.S. in derivation of X.
- ii. Following are the rules used to compute the FIRST functions.
 - a. If X determines terminal symbol 'a' then the $\text{FIRST}(X) = \{a\}$.
 - b. If there is a rule $X \rightarrow \epsilon$ then $\text{FIRST}(X)$ contain $\{\epsilon\}$.
 - c. If X is non-terminal and $X \rightarrow Y_1 Y_2 Y_3 \dots Y_k$ is a production and if ϵ is in all of $\text{FIRST}(Y_1) \dots \text{FIRST}(Y_k)$ then

$$\text{FIRST}(X) = \{\text{FIRST}(Y_1) \cup \text{FIRST}(Y_2) \cup \text{FIRST}(Y_3) \dots \cup \text{FIRST}(Y_k)\}.$$

2. FOLLOW function :

- i. FOLLOW(A) is defined as the set of terminal symbols that appear immediately to the right of A.
- ii. $\text{FOLLOW}(A) = \{a \mid S \Rightarrow^* \alpha A a \beta \text{ where } \alpha \text{ and } \beta \text{ are some grammar symbols may be terminal or non-terminal}\}.$
- iii. The rules for computing FOLLOW function are as follows :
 - a. For the start symbol S place \$ in FOLLOW(S).
 - b. If there is a production $A \rightarrow \alpha B \beta$ then everything in $\text{FIRST}(\beta)$ without ϵ is to be placed in FOLLOW(B).
 - c. If there is a production $A \rightarrow \alpha B \beta$ or $A \rightarrow \alpha B$ and $\text{FIRST}(B)$ contain ϵ then $\text{FOLLOW}(B) = \text{FOLLOW}(A)$. That means everything in FOLLOW(A) is in FOLLOW(B).

Que 2.14. Differentiate between recursive descent parsing and predictive parsing.

Answer

S. No.	Recursive descent parsing	Predictive parsing
1.	CFG is used to build recursive routine.	Recursive routine is not build.
2.	RHS of production rule is converted into program.	Production rule is not converted into program.
3.	Parsing table is not constructed.	Parsing table is constructed.
4.	First and follow is not used.	First and follow is used to construct parsing table.

Que 2.15. Construct predictive parse table for the following grammar.

$$E \rightarrow E + T / T$$

$$T \rightarrow T * F / F$$

$$F \rightarrow F / a / b$$

AKTU 2021-22, Marks 10

Answer

$$E \rightarrow E + T / T$$

$$T \rightarrow T * F / F$$

$$F \rightarrow F / a / b$$

First we remove left recursion

$$E \rightarrow \underbrace{E}_A + \underbrace{T}_\alpha / \underbrace{T}_\beta$$

$$E \rightarrow TE'$$

$$E' \rightarrow TE' / \epsilon$$

$$T \rightarrow \underbrace{T}_A * \underbrace{F}_\alpha / \underbrace{F}_\beta$$

$$T \rightarrow FT'$$

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

$$F \rightarrow \underbrace{F}_A / \underbrace{a}_{\beta_1} / \underbrace{b}_{\beta_2}$$

$$F \rightarrow aF' \mid bF'$$

$$F' \rightarrow *FF' / \epsilon$$

After removing left recursion, we get

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow aF' / bF'$$

$$F' \rightarrow FF' / \epsilon$$

$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{a, b\}$$

$$\text{FIRST}(E') = \{+, \epsilon\}$$

$$\text{FIRST}(T') = \{*, \epsilon\}$$

$$\text{FIRST}(F') = \{\text{FIRST}(F) \cup \epsilon\} = \{a, b, \epsilon\}$$

$$\text{FIRST}(F) = \{a, b\}$$

$$\text{FOLLOW}(E) = \{\$ \}$$

$$\text{FOLLOW}(E') = \text{FOLLOW}(E) = \{\$ \}$$

$$\text{FOLLOW}(T) = \text{FIRST}(E') = \{+, \$ \}$$

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

$$\text{FOLLOW}(F) = \text{FIRST}(T') = \{*, +, \$, a, b\}$$

$$\text{FOLLOW}(F') = \text{FIRST}(F) = \{*, +, a, b, \$ \}$$

Predictive parsing table :

Non-terminal	Input symbol				
	+	*	a	b	\$
E			$E \rightarrow TE'$	$E \rightarrow TE'$	
E'	$E' \rightarrow + TE'$				$E' \rightarrow \epsilon$
T			$T \rightarrow FT'$	$T \rightarrow FT'$	
T'	$T' \rightarrow \epsilon$	$T' \rightarrow * FT'$			$T' \rightarrow \epsilon$
F			$F \rightarrow aF'$	$F \rightarrow bF'$	
F'	$F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$	$F' \rightarrow FF'$ F'	$F' \rightarrow FF'$ $F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$

Que 2.16. Test whether the grammar is LL (1) or not, and construct parsing table for it.

$$S \rightarrow 1AB/\epsilon$$

$$A \rightarrow 1AC/0C$$

$$B \rightarrow 0S$$

$$C \rightarrow 1$$

AKTU 2021-22, Marks 10

Answer

Given grammar :

$$S \rightarrow 1AB/\epsilon$$

$$A \rightarrow 1AC/0C$$

$$B \rightarrow 0S$$

$$C \rightarrow 1$$

$$\text{FIRST}(S) = \{1, \epsilon\}$$

$$\text{FIRST}(A) = \{1, 0\}$$

$$\text{FIRST}(B) = \{0\}$$

$$\text{FIRST}(C) = \{1\}$$

$$\text{FOLLOW}(S) = \{\$ \}$$

$$\text{FOLLOW}(A) = \{1, 0\}$$

$$\text{FOLLOW}(B) = \{\$ \}$$

$$\text{FOLLOW}(C) = \{0, 1\}$$

Compiler Design

Parsing table :

Non-Terminal	Input Symbol		
	0	1	\$
S		$S \rightarrow 1AB$	$S \rightarrow \epsilon$
A	$A \rightarrow 0C$	$A \rightarrow 1AC$	
B	$B \rightarrow 0S$		
C		$C \rightarrow 1$	

Since, no column has more than one value. So the given grammar is LL(1) grammar.

PART-4

Automatic Generation of Efficient Parsers : LR Parsers, The Canonical Collections of LR(0) Items

Que 2.17. Discuss working of LR parser with its block diagram. Why it is most commonly used parser ?

Answer

Working of LR parser :

1. The working of LR parser can be understood by using block diagram as shown in Fig. 2.17.1.

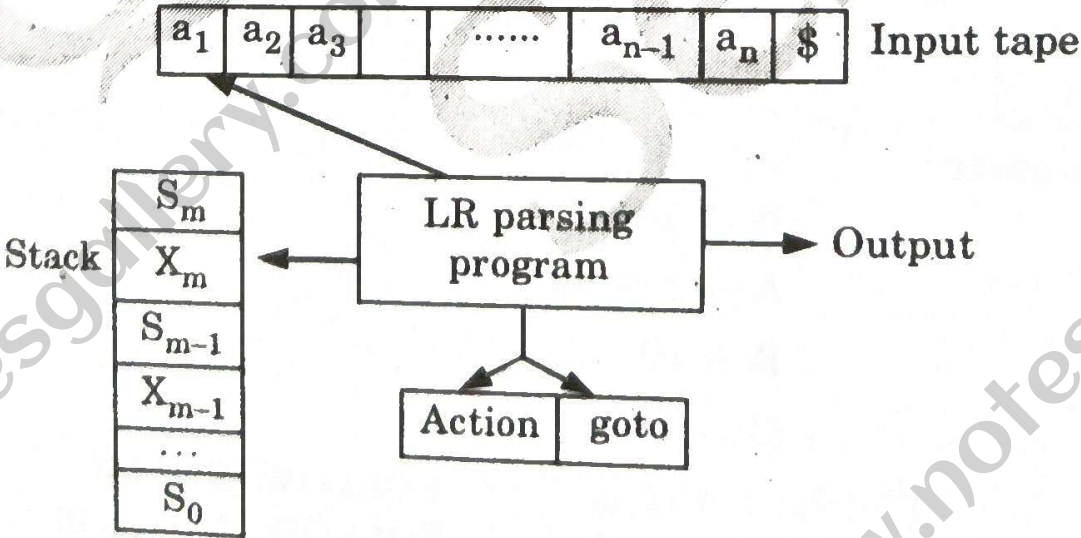


Fig. 2.17.1. Model of an LR parser.

2. In LR parser, it has input buffer for storing the input string, a stack for storing the grammar symbols, output and a parsing table comprised of two parts, namely action and goto.

3. There is a driver program and reads the input symbol one at a time from the input buffer. This program is same for all LR parser.
4. It reads the input string one symbol at a time and maintains a stack.
5. The stack always maintains the following form :

$$S_0 X_1 S_1 X_2 S_2 \dots \dots X_{m-1} S_{m-1} X_m S_m$$
 where X_i is a grammar symbol, each S_i is the state and S_m state is top of the stack.
6. The action of the driver program depends on action $[S_m, a_i]$ where a_i is the current input symbol.
7. Following action are possible for input $a_i a_{i+1} \dots a_n$:
 - a. **Shift** : If action $[S_m, a_i] = \text{shift } S$, the parser shift the input symbol, a_i onto the stack and then stack state S . Now current input symbol becomes a_{i+1} .

Stack

$$S_0 X_1 S_1 X_2 \dots \dots X_{m-r} S_{m-r} X_m$$
Input

$$S_m a_i S, a_{i+1}, a_{i+2} \dots \dots a_n \$$$

- b. **Reduce** : If action $[S_m, a_i] = \text{reduce } A \rightarrow \beta$ the parser executes a reduce move using the $A \rightarrow \beta$ production of the grammar. If $A \rightarrow \beta$ has r grammar symbols, first $2r$ symbols are popped off the stack (r state symbol and r grammar symbol). So, the top of the stack now becomes S_{m-r} then A is pushed on the stack, and then state goto $[S_{m-r} A]$ is pushed on the stack. The current input symbol is still a_i .

Stack

$$S_0 X_1 S_1 X_2 \dots \dots X_{m-r} S_{m-r} AS$$
Input

$$a_i a_{i+1} \dots \dots a_n \$$$

where, $S = \text{Goto } [S_{m-r} A]$

- i. If action $[S_m, a_i] = \text{accept}$, parsing is completed.
- ii. If action $[S_m, a_i] = \text{error}$, the parser has discovered a syntax error.

LR parser is widely used for following reasons :

1. LR parsers can be constructed to recognize most of the programming language for which context free grammar can be written.
2. The class of grammar that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsers.
3. LR parser works using non-backtracking shift-reduce technique.
4. LR parser is an efficient parser as it detects syntactic error very quickly.

Que 2.18. Write short note on the following :

1. LR (0) items
2. Augmented grammar
3. Kernel and non-kernel items
4. Functions closure and goto

Answer

1. **LR(0) items** : The LR (0) item for grammar G is production rule in which symbol $\bullet$ is inserted at some position in R.H.S. of the rule. For example

$$S \rightarrow \bullet ABC$$

$$S \rightarrow A \bullet BC$$

$$S \rightarrow AB \bullet C$$

$$S \rightarrow ABC \bullet$$

The production $S \rightarrow \epsilon$ generates only one item $S \rightarrow \bullet$.

2. **Augmented grammar** : If a grammar G is having start symbol S then augmented grammar G' in which S' is a new start symbol such that $S' \rightarrow S$. The purpose of this grammar is to indicate the acceptance of input. That is when parser is about to reduce $S' \rightarrow S$, it reaches to acceptance state.

3. **Kernel items** : It is collection of items $S' \rightarrow \bullet S$ and all the items whose dots are not at the left most end of R.H.S. of the rule.

Non-kernel items : The collection of all the items in which $\bullet$ are at the left end of R.H.S. of the rule.

4. **Functions closure and goto** : These are two important functions required to create collection of canonical set of LR (0) items.

PART-5

Constructing SLR Parsing Tables.

Que 2.19. Explain SLR parsing techniques. Also write the algorithm to construct SLR parsing table.

Answer

The SLR parsing can be done as :

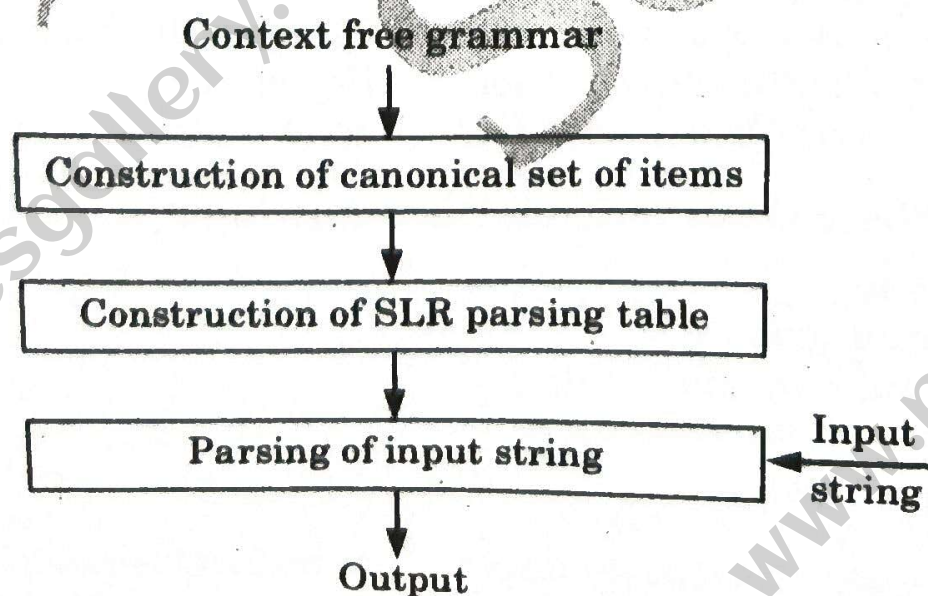


Fig. 2.19.1. Working of SLR parser.

Algorithm for construction of an SLR parsing table :

Input : C the canonical collection of sets of items for an augmented grammar G' .

Output : If possible, an LR parsing table consisting of parsing action function ACTION and a goto function GOTO.

Method :

Let $C = [I_0, I_1, \dots, I_n]$. The states of the parser are $0, 1, \dots, n$ state i being constructed from I_i .

The parsing action for state is determined as follows :

1. If $[A \rightarrow \alpha \bullet a \beta]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$ then set ACTION $[i, a]$ to "shift j ". Here a is a terminal.
2. If $[A \rightarrow \alpha \bullet]$ is in I_i the set ACTION $[i, a]$ to "reduce $A \rightarrow \alpha$ " for all ' a ' in FOLLOW(A).
3. If $[S' \rightarrow S \bullet]$ is in I_i , then set ACTION $[i, \$]$ to "accept".

The goto transitions for state i are constructed using the rule.

4. If $\text{GOTO}(I_i, A) = I_j$ then $\text{GOTO}[i, A] = j$.
5. All entries not defined by rules (1) through rule (4) are made "error".
6. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \bullet S]$.

The parsing table consisting of the parsing ACTION and GOTO function determined by this algorithm is called the SLR parsing table for G . An LR parser using the SLR parsing table for G is called the SLR parser for G and a grammar having an SLR parsing table is said to be SLR(1).

Que 2.20. Check whether the given grammar is LR(0) or not :

$S \rightarrow PQy, P \rightarrow Sy|x, Q \rightarrow yS.$

AKTU 2020-21, 2023-24; Marks 10

Answer

The augmented grammar is :

$S' \rightarrow S$

$S \rightarrow PQy$

$P \rightarrow Sy|x$

$Q \rightarrow yS$

The canonical collection of LR(0) items are

$I_0: S' \rightarrow \bullet S$

$S \rightarrow \bullet PQy$

$P \rightarrow \bullet Sy | \bullet x$

$Q \rightarrow \bullet yS$

$I_1 = \text{GOTO}(I_0, S)$

$I_1: S' \rightarrow S \bullet$

$P \rightarrow S \bullet y$

$I_2 = \text{GOTO}(I_0, P)$

$I_7 = \text{GOTO}(I_2, y) = I_4$

$I_8 = \text{GOTO}(I_4, S)$

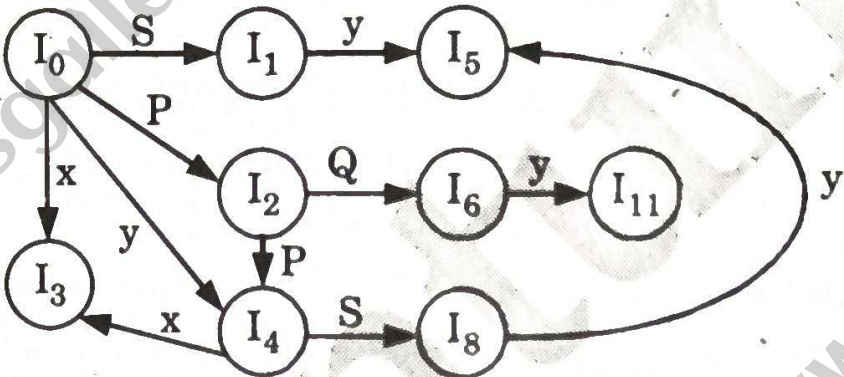
$I_8: Q \rightarrow yS \bullet$

$P \rightarrow S \bullet y$

$I_2: S \rightarrow P \bullet Qy$
 $Q \rightarrow \bullet yS$
 $I_3 = \text{GOTO}(I_0, x)$
 $I_3: P \rightarrow x \bullet$
 $I_4 = \text{GOTO}(I_0, y)$
 $I_4: Q \rightarrow y \bullet S$
 $S \rightarrow \bullet PQy$
 $P \rightarrow \bullet Sy \mid \bullet x$
 $I_5 = \text{GOTO}(I_1, y)$
 $I_5: P \rightarrow Sy \bullet$
 $I_6 = \text{GOTO}(I_2, Q)$
 $I_6: S \rightarrow PQ \bullet y$

$I_9 = \text{GOTO}(I_4, P) = I_2$
 $I_{10} = \text{GOTO}(I_4, x) = I_3$
 $I_{11} = \text{GOTO}(I_6, y)$
 $I_{11}: S \rightarrow PQy \bullet$
 $I_{12} = \text{GOTO}(I_8, y) = I_5$

DFA for set of items :



Let us number the production rules in the grammar as.

- 1. $S \rightarrow PQy$
- 2. $P \rightarrow Sy$
- 3. $S \rightarrow x$
- 4. $Q \rightarrow yS$

Action				GOTO		
States	x	y	\$	P	Q	S
I_0	S_3	S_4		2		1
I_1		S_5	accept			
I_2		S_4			6	
I_3	r_3	r_3	$\bullet r_3$			
I_4	S_3			2		8
I_5	r_2	r_2	r_2			
I_6		S_{11}				
I_8	r_4	S_5/r_4	r_4			
I_{11}	r_1	r_1	r_1			

In the parsing table there is Shift-Reduce conflict in I_8 . So the given grammar is not LR(0).

Que 2.21.

Construct the SLR parse table for the following

Grammar

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

AKTU 2021-22, Marks 10

Answer

The augmented grammar is :

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

The canonical collection of sets of LR(0) items for grammar are as follows :

 $I_0: E' \rightarrow \bullet E$ $E \rightarrow \bullet E + E$ $E \rightarrow \bullet E * E$ $E \rightarrow \bullet id$ $I_1 = \text{GOTO}(I_0, E)$ $I_1: E' \rightarrow E \bullet$ $E \rightarrow E \bullet + E$ $E \rightarrow E \bullet * E$ $I_2 = \text{GOTO}(I_0, id)$ $I_2: E' \rightarrow id \bullet$ $I_3 = \text{GOTO}(I_1, +)$ $I_3: E \rightarrow E + \bullet E$ $E \rightarrow \bullet E + E$ $E \rightarrow \bullet E * E$ $E' \rightarrow \bullet id$ $I_4 = \text{GOTO}(I_1, *)$ $I_4: E \rightarrow E * \bullet E$ $E \rightarrow \bullet E + E$ $E \rightarrow \bullet E * E$ $E' \rightarrow \bullet id$ $I_5 = \text{GOTO}(I_3, E)$ $I_5: E \rightarrow E + E \bullet$ $E \rightarrow E \bullet + E$ $E \rightarrow E \bullet * E$

$I_6 = \text{GOTO}(I_4, E)$
 $I_6 : E \rightarrow E * E \bullet$
 $E \rightarrow E \bullet + E$
 $E \rightarrow E \bullet * E$

Let us numbered the production rules is the grammar as

- 1. $E \rightarrow E + E$
- 2. $E \rightarrow E * E$
- 3. $E \rightarrow id$

$\text{FIRST}(E) = \{id\}$
 $\text{FOLLOW}(E) = \{\$, +, *\}$

SLR Parsing table :

States	Action				Go to
	+	*	id	\$	
I_0			S_2		1
I_1	S_3	S_4		accept	
I_2	r_3	r_3		r_3	
I_3	r_1	r_1		r_1	5
I_4	r_2	r_2		r_2	6
I_5	S_3	S_4			
I_6	S_3	S_4			

DFA for set of items :

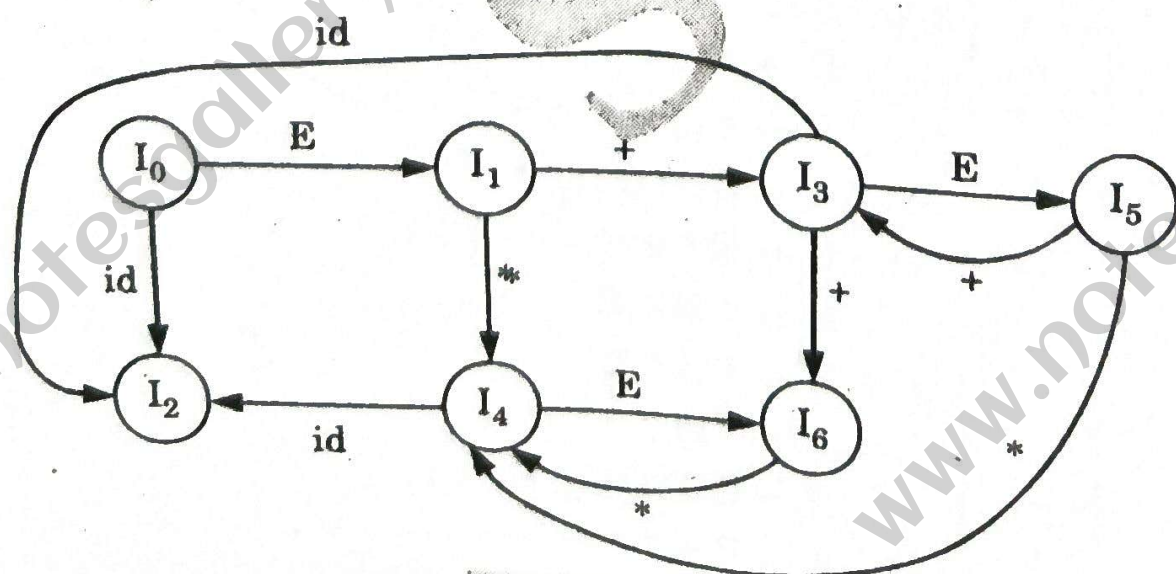


Fig. 2.21.1.

Que 2.22. Construct the SLR parsing table for the following grammar.

$$S \rightarrow 0S0$$

$$S \rightarrow 1S1$$

$$S \rightarrow 10$$

AKTU 2022-23, Marks 10

Answer

The augmented grammar will be :

$$S' \rightarrow S$$

$$S \rightarrow 0S0$$

$$S \rightarrow 1S1$$

$$S \rightarrow 10$$

The set of LR (0) items are as follows :

$$I_0: S' \rightarrow \bullet S$$

$$S \rightarrow \bullet 0S0$$

$$S \rightarrow \bullet 1S1$$

$$S \rightarrow \bullet 10$$

$$I_1 = \text{GOTO}(I_0, S)$$

$$I_1: S' \rightarrow S \bullet$$

$$I_2 = \text{GOTO}(I_0, 0)$$

$$I_2: S \rightarrow 0 \bullet S0$$

$$S \rightarrow \bullet 0S0$$

$$S \rightarrow \bullet 1S1$$

$$I_3 = \text{GOTO}(I_0, 1)$$

$$I_3: S \rightarrow 1 \bullet S1$$

$$S \rightarrow \bullet 0S0$$

$$S \rightarrow \bullet 1S1$$

$$S \rightarrow 1 \bullet 0$$

$$I_4 = \text{GOTO}(I_2, 5)$$

$$I_4: S \rightarrow 0S \bullet 0$$

$$I_5 = \text{GOTO}(I_3, S)$$

$$I_5: S \rightarrow 1S \bullet 1$$

$$I_6 = \text{GOTO}(I_3, 0)$$

$$I_6: S \rightarrow 0 \bullet S0$$

$$S \rightarrow \bullet 1S1$$

$$S \rightarrow \bullet 0S0$$

$S \rightarrow \bullet 10$

$S \rightarrow 10 \bullet$

$I_7 = \text{GOTO } (I_4, 0)$

$I_7: S \rightarrow 0S0 \bullet$

$I_8 = \text{GOTO } (I_5, 1)$

$I_8: S \rightarrow 1S1 \bullet$

Let us numbered the production rules in the grammar as

1.
- $S \rightarrow 0S0$
2.
- $S \rightarrow 1S1$
3.
- $S \rightarrow 10$

State	Action			Goto	
	10	0	1	\$	S
0		S_2	S_3		1
1				accept	
2		S_2	S_3		4
3		S_6	S_3		5
4		S_7			
5			S_8		
6		S_2, r_3	S_3, r_3	r_3	
7		r_1	r_1	r_1	
8		r_2	r_2	r_2	

Que 2.23. Consider the following grammar for a simple expression language :

$E \rightarrow E + T | T$

$T \rightarrow T * F | F$

$F \rightarrow (E) | id$

Consider expression $id * (id + id)$. Apply shift reduce parsing to construct parse tree.

Answer

Given

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow E$

AKTU 2023-24, Marks 10

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

S.No.	Stack	i/p string	Action
1.	\$	id * (id + id) \$	
2.	\$ id	* (id + id) \$	shift id
3.	\$F	* (id + id) \$	reduce $F \rightarrow id$
4.	\$T	* (id + id) \$	reduce $T \rightarrow F$
5.	\$T *	(id + id) \$	shift *
6.	\$T * (	id + id) \$	shift (
7.	\$T * (id	+ id) \$	shift id
8.	\$T * (F	+ id) \$	reduce $F \rightarrow id$
9.	\$T * (T	+ id) \$	reduce $T \rightarrow F$
10.	\$T * (E	+ id) \$	reduce $E \rightarrow T$
11.	\$T * (E +	id) \$	shift +
12.	\$T * (E + id	) \$	shift id
13.	\$T * (E + F	) \$	reduce $F \rightarrow id$
14.	\$T * (E + T	) \$	reduce $T \rightarrow F$
15.	\$T * (E + T)	\$	shift)
16.	\$T * (E)	\$	reduce $E \rightarrow E + T$
17.	\$T * F	\$	reduce $F \rightarrow (E)$
18.	\$T	\$	reduce $T \rightarrow T * F$
19.	\$E	\$	reduce $E \rightarrow T$ accepted

Que 2.24. Consider the following grammar $E \rightarrow E + E \mid E * E \mid (E) \mid id$. Construct the SLR parsing table and suggest your final parsing table.

Answer

The augmented grammar is as :

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

The set of LR(0) items is as follows :

I_0 :
 $E' \rightarrow \bullet E$
 $E \rightarrow \bullet E + E$
 $E \rightarrow \bullet E * E$
 $E \rightarrow \bullet (E)$
 $E \rightarrow \bullet id$

$I_1 = \text{GOTO}(I_0, E)$

$I_1:$ $E' \rightarrow E \bullet$
 $E \rightarrow E \bullet + E$
 $E \rightarrow E \bullet * E$

$I_2 = \text{GOTO}(I_0, ($

$I_2:$ $E \rightarrow (\bullet E)$
 $E \rightarrow \bullet E + E$
 $E \rightarrow \bullet E * E$
 $E \rightarrow \bullet (E)$
 $E \rightarrow \bullet id$

$I_3 = \text{GOTO}(I_0, id)$

$I_3:$ $E \rightarrow id \bullet$

$I_4 = \text{GOTO}(I_1, +)$

$I_4:$ $E \rightarrow E + \bullet E$
 $E \rightarrow \bullet E + E$
 $E \rightarrow \bullet E * E$
 $E \rightarrow \bullet (E)$
 $E \rightarrow \bullet id$

$I_5 = \text{GOTO}(I_1, *)$

$I_5:$ $E \rightarrow E * \bullet E$
 $E \rightarrow \bullet E + E$
 $E \rightarrow \bullet E * E$
 $E \rightarrow \bullet (E)$
 $E \rightarrow \bullet id$

$I_6 = \text{GOTO}(I_2, E)$

$I_6:$ $E \rightarrow (E \bullet)$
 $E \rightarrow E \bullet + E$
 $E \rightarrow E \bullet * E$

$I_7 = \text{GOTO}(I_4, E)$

$I_7:$ $E \rightarrow E + E \bullet$
 $E \rightarrow E \bullet + E$
 $E \rightarrow E \bullet * E$

$I_8 = \text{GOTO}(I_5, E)$

$I_8:$ $E \rightarrow E * E \bullet$
 $E \rightarrow E \bullet + E$
 $E \rightarrow E \bullet * E$

$I_9 = \text{GOTO}(I_6,))$

$I_9:$ $E \rightarrow (E) \bullet$

State	Action						Goto
	<i>id</i>	+	*	(	)	\$	
0	S_3			S_2			1
1		S_4	S_5			accept	
2	S_3	.		S_2			6
3		r_4	r_4		r_4	r_4	
4	S_3			S_2			8
5	S_3			S_2			8
6		S_4	S_5		S_3		
7		r_1	S_5		r_1	r_1	
8		r_2	r_2		r_2	r_2	
9		r_3	r_3		r_3	r_3	

Que 2.25. Perform shift reduce parsing for the given input strings

using the grammar $S \rightarrow (L) \mid a \mid L, S \mid S$

- i. $(a, (a, a))$
- ii. (a, a)

Answer

i.

Stack contents	Input string	Actions
\$	$(a, (a, a))\$$	Shift (
$\$($	$(a, (a, a))\$$	Shift a
$\$(a$	$,(a, a))\$$	Reduce $S \rightarrow a$
$\$(S$	$,(a, a))\$$	Reduce $L \rightarrow S$
$\$(L$	$,(a, a))\$$	Shift)
$\$(L,$	$(a, a))\$$	Shift (
$\$(L, ($	$a, a))\$$	Shift a
$\$(L, (a$	$,a))\$$	Reduce $S \rightarrow a$
$\$(L, (S$	$,a))\$$	Reduce $L \rightarrow S$
$\$(L, (L$	$,a))\$$	Shift,
$\$(L, (L,$	$a))\$$	Shift a
$\$(L, (L, a$	$))\$$	Reduce $S \rightarrow a$
$\$(L, (L, S$	$))\$$	Reduce $L \rightarrow L, S$
$\$(L, (L$	$))\$$	Shift)
$\$(L, (L)$	$) \$$	Reduce $S \rightarrow (L)$
$\$(L, S$	$) \$$	Reduce $L \rightarrow L, S$
$\$(L$	$) \$$	Shift)
$\$(L)$	$\$$	Reduce $S \rightarrow (L)$
$\$S$	$\$$	Accept

ii.

Stack contents	Input string	Actions
	(a, a)\$	Shift (
\$	a, a)\$	Shift a
\$(	, a)\$	Reduce $S \rightarrow a$
\$(a	, a)\$	Reduce $L \rightarrow S$
\$(S	, a)\$	Shift ,
\$(L	a)\$	Shift a
\$(L,	)\$	Reduce $S \rightarrow a$
\$(L, a	)\$	Reduce $L \rightarrow L, S$
\$(L, S	)\$	Shift)
\$(L	\$	Reduce $S \rightarrow L$
\$(L)	\$	Accept
\$S		

Que 2.26. Construct LR(0) parsing table for the following grammar

$$\begin{aligned} S &\rightarrow cB \mid ccA \\ A &\rightarrow cA \mid a \\ B &\rightarrow ccB \mid b \end{aligned}$$

Answer

The augmented grammar is :

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow cB \mid ccA \\ A &\rightarrow cA \mid a \\ B &\rightarrow ccB \mid b \end{aligned}$$

The canonical collection of LR (0) items are :

$$\begin{aligned} I_0 : S' &\rightarrow \bullet S \\ S &\rightarrow \bullet cB \mid \bullet ccA \\ A &\rightarrow \bullet cA \mid \bullet a \\ B &\rightarrow \bullet ccB \mid \bullet b \\ I_1 &= \text{GOTO}(I_0, S) \\ I_1 : S' &\rightarrow S \bullet \\ I_2 &= \text{GOTO}(I_0, c) \\ I_2 : S &\rightarrow c \bullet B \mid c \bullet cA \\ A &\rightarrow c \bullet A \\ B &\rightarrow c \bullet cB \\ A &\rightarrow \bullet cA \mid \bullet a \\ B &\rightarrow \bullet ccB \mid \bullet b \end{aligned}$$

$$I_3 = \text{GOTO}(I_0, a)$$

$$I_3: A \rightarrow a \bullet$$

$$I_4 = \text{GOTO}(I_0, b)$$

$$I_4: B \rightarrow b \bullet$$

$$I_5 = \text{GOTO}(I_2, B)$$

$$I_5: S \rightarrow cB \bullet$$

$$I_6 = \text{GOTO}(I_2, A)$$

$$I_6: A \rightarrow cA \bullet$$

$$I_7 = \text{GOTO}(I_2, c)$$

$$I_7: S \rightarrow cc \bullet A$$

$$B \rightarrow cc \bullet B$$

$$A \rightarrow c \bullet A$$

$$B \rightarrow c \bullet cB$$

$$A \rightarrow \bullet cA / \bullet a$$

$$B \rightarrow \bullet ccB / \bullet b$$

$$I_8 = \text{GOTO}(I_7, A)$$

$$I_8: S \rightarrow ccA \bullet$$

$$A \rightarrow cA \bullet$$

$$I_9 = \text{GOTO}(I_7, B)$$

$$I_9: B \rightarrow ccB \bullet$$

$$I_{10} = \text{GOTO}(I_7, c)$$

$$I_{10}: B \rightarrow cc \bullet B$$

$$A \rightarrow c \bullet A$$

$$B \rightarrow c \bullet cB$$

$$B \rightarrow \bullet ccB / \bullet b$$

$$A \rightarrow \bullet cA / \bullet a$$

$$I_{11} = \text{GOTO}(I_{10}, A)$$

$$I_{11}: A \rightarrow cA \bullet$$

DFA for set of items :

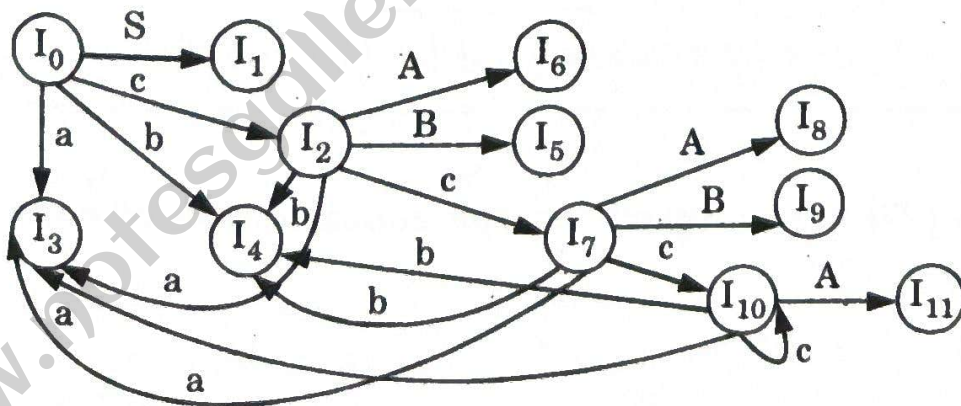


Fig. 2.26.1.

Let us numbered the production rules in the grammar as

1. $S \rightarrow cB$
2. $S \rightarrow ccA$
3. $A \rightarrow cA$
4. $A \rightarrow a$
5. $B \rightarrow ccB$
6. $B \rightarrow b$

Action					GOTO		
States	a	b	c	\$	A	B	S
I_0	S_3	S_4	S_2				1
I_1				Accept			
I_2	S_3	S_4	S_7		6	5	
I_3	r_4	r_4	r_4	r_4			
I_4	r_6	r_6	r_6	r_6			
I_5	r_1	r_1	r_1	r_1			
I_6	r_3	r_3	r_3	r_3			
I_7	S_3	S_4	S_{10}		8	9	
I_8	r_2, r_3	r_2, r_3	r_2, r_3	r_2, r_3			
I_9	r_5	r_5	r_5	r_5			
I_{10}	S_3	S_4	S_{10}		11		
I_{11}	r_3	r_3	r_3	r_3			

PART-6

Constructing Canonical LR Parsing Tables.

Que 2.27. Give the algorithm for construction of canonical LR parsing table.

Answer

Algorithm for construction of canonical LR parsing table :

Input : An augmented grammar G' .

Output : The canonical parsing table function ACTION and GOTO for G' .

Method : Construct $C' = \{I_0, I_1, \dots, I_n\}$ the collection of sets of LR (1) items of G' . State i of the parser is constructed from I_i .

1. The parsing actions for state i are determined as follows :
 - a. If $[A \rightarrow \alpha \bullet a \beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set ACTION $[i, a]$ to "shift j ". Here, a is required to be a terminal.
 - b. If $[A \rightarrow \alpha \bullet, a]$ is in I_i , $A \neq S'$, then set ACTION $[i, a]$ to "reduce $A \rightarrow \alpha \bullet$ ".
 - c. If $[S' \rightarrow S \bullet, \$]$ is in I_i , then set ACTION $[i, \$]$ to "accept".

The goto transitions for state i are determined as follows :

2. If $\text{GOTO}(I_i, A) = I_j$ then $\text{GOTO}[i, A] = j$.
3. All entries not defined by rules (1) and (2) are made "error".
4. The initial state of parser is the one constructed from the set containing items $[S' \rightarrow \bullet S, \$]$.

If the parsing action function has no multiple entries then grammar is said to be LR (1) or LR.

PART-7

Constructing LALR Parsing Tables Using Ambiguous Grammars, An Automatic Parser Generator, Implementation of LR Parsing Tables.

Que 2.28. Give the algorithm for construction of LALR parsing table.

Answer

The algorithm for construction of LALR parsing table is as :

Input : An augmented grammar G' .

Output : The LALR parsing table function ACTION and GOTO for G' .

Method :

1. Construct $C = \{I_0, I_1, \dots, I_n\}$ the collection of sets of LR (1) items.
2. For each core present among the LR (1) items, find all sets having that core, and replace these sets by their union.

Compiler Design

3. Let $C' = \{J_0, J_1, \dots, J_m\}$ be the resulting sets of LR (1) items. The parsing actions for state i are constructed from J_i . If there is a parsing action conflicts, the algorithm fails to produce a parser and the grammar is said not to be LALR (1).
4. The goto table constructed as follows. If ' J ' is the union of one or more sets of LR (1) items, i.e., $J = I_1 \cup I_2 \cup I_3 \dots \cup I_k$, then the cores of the $\text{GOTO}(I_1, X), \text{GOTO}(I_2, X), \dots, \text{GOTO}(I_k, X)$ are the same. Since $I_1, I_2, \dots, I_k$ all have the same core. Let k be the union of all sets of the items having the same core as $\text{GOTO}(I_1, X)$. Then $\text{GOTO}(J, X) = k$.

The table produced by this algorithm is called LALR parsing table for grammar G . If there are no parsing action conflicts, then the given grammar is said to be LALR(1) grammar.

The collection of sets of items constructed in step '3' of this algorithm is called LALR(1) collections.

Que 2.29. Construct the CLR(1) parsing table for the following

grammar $S \rightarrow AA, A \rightarrow aA | b$.

AKTU 2020-21, Marks 10

Answer

The given grammar is :

$$S \rightarrow AA$$

$$A \rightarrow aA | b$$

The augmented grammar will be :

$$S' \rightarrow S$$

$$S \rightarrow AA$$

$$A \rightarrow aA | b$$

The LR (1) items will be :

$$I_0: S' \rightarrow \bullet S, \$$$

$$S \rightarrow \bullet AA, \$$$

$$A \rightarrow \bullet aA, a/b$$

$$A \rightarrow \bullet b, a/b$$

$$I_1 = \text{GOTO}(I_0, S)$$

$$I_1: S' \rightarrow S \bullet, \$$$

$$I_2 = \text{GOTO}(I_0, A)$$

$$I_2: S \rightarrow A \bullet A, \$$$

$$A \rightarrow \bullet aA, \$$$

$$A \rightarrow \bullet b, \$$$

$$I_3 = \text{GOTO}(I_0, a)$$

$$I_3: A \rightarrow a \bullet A, a/b$$

$$A \rightarrow \bullet aA, a/b$$

$$A \rightarrow \bullet b, a/b$$

$$I_4 = \text{GOTO}(I_0, b)$$

$$I_4: A \rightarrow b \bullet, a/b$$

$$I_5 = \text{GOTO}(I_2, A)$$

$$I_5: S \rightarrow AA \bullet, \$$$

$$I_6 = \text{GOTO}(I_2, a)$$

$$I_6: A \rightarrow a \bullet A, \$$$

$$A \rightarrow \bullet aA, \$$$

$$A \rightarrow \bullet b, \$$$

$$I_7 = \text{GOTO}(I_2, b)$$

$$I_7: A \rightarrow b \bullet, \$$$

$$I_8 = \text{GOTO}(I_3, A)$$

$$I_8: A \rightarrow aA \bullet, a/b$$

$$I_9 = \text{GOTO}(I_6, A)$$

$$I_9: A \rightarrow aA \bullet, \$$$

Table 2.29.1.

State	Action			Goto	
	a	b	\$	S	A
0	S_3	S_4		1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

Since table does not contain any conflict. So it is LR(1).

The goto table will be for LALR I_3 and I_6 will be unioned, I_4 and I_7 will be unioned, and I_8 and I_9 will be unioned.

So,

$$I_{36}: A \rightarrow a \bullet A, a / b / \$$$

$$A \rightarrow \bullet a A, a / b / \$$$

$$A \rightarrow \bullet b, a / b / \$$$

$$I_{47}: A \rightarrow b \bullet, a / b / \$$$

$$I_{89}: A \rightarrow a A \bullet, a / b / \$ \text{ and LALR table will be :}$$

Table 2.29.2.

State	Action			Goto	
	a	b	\$	S	A
0	S_{36}	S_{47}		1	2
1			accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Since, LALR table does not contain any conflict. So, it is also LALR(1).

DFA:

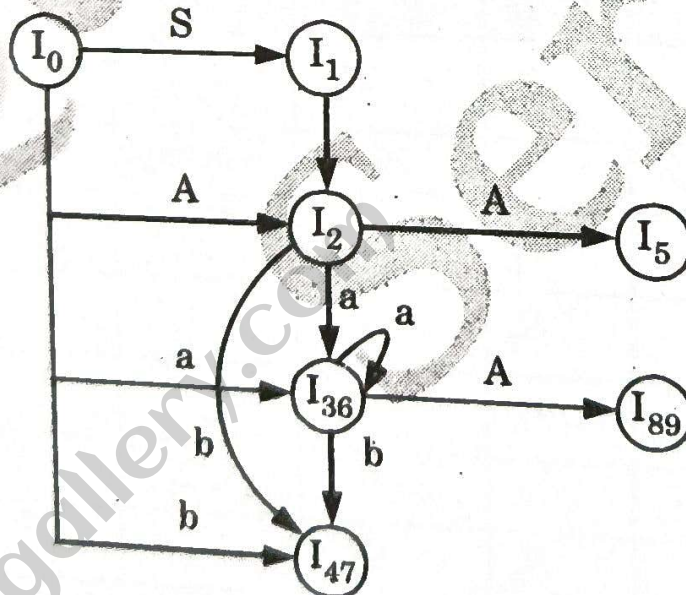


Fig. 2.29.1.

Que 2.30.

Construct the LALR parsing table for the given grammar

$$S \rightarrow BB$$

$$B \rightarrow aB/b$$

Answer

The given grammar is :

$$S \rightarrow BB$$

$$B \rightarrow aB \mid b$$

The augmented grammar will be :

$$S' \rightarrow S$$

$$S \rightarrow BB$$

$$B \rightarrow aB \mid b$$

The LR (1) items will be :

$$I_0: S' \rightarrow \bullet S, \$$$

$$S \rightarrow \bullet BB, \$$$

$$B \rightarrow \bullet aB, a/b$$

$$B \rightarrow \bullet b, a/b$$

$$I_1 = \text{GOTO}(I_0, S)$$

$$I_1: S' \rightarrow S \bullet, \$$$

$$I_2 = \text{GOTO}(I_0, B)$$

$$I_2: S \rightarrow B \bullet B, \$$$

$$B \rightarrow \bullet aB, \$$$

$$B \rightarrow \bullet b, \$$$

$$I_3 = \text{GOTO}(I_0, a)$$

$$I_3: B \rightarrow a \bullet B, a/b$$

$$B \rightarrow \bullet aB, a/b$$

$$B \rightarrow \bullet b, a/b$$

$$I_4 = \text{GOTO}(I_0, b)$$

$$I_4: B \rightarrow b \bullet, a/b$$

$$I_5 = \text{GOTO}(I_2, B)$$

$$I_5: S \rightarrow BB \bullet, \$$$

$$I_6 = \text{GOTO}(I_2, a)$$

$$I_6: B \rightarrow a \bullet B, \$$$

$$B \rightarrow \bullet aB, \$$$

$$B \rightarrow \bullet b, \$$$

$$I_7 = \text{GOTO}(I_2, b)$$

$$I_7: B \rightarrow b \bullet, \$$$

$$I_8 = \text{GOTO}(I_3, B)$$

$$I_8: B \rightarrow aB \bullet, a/b$$

$$I_9 = \text{GOTO}(I_6, B)$$

$$I_9: B \rightarrow aB \bullet, \$$$

Table 2.30.1.

State	Action			Goto	
	a	b	\$	S	A
0	S ₃	S ₄		1	2
1			accept		
2	S ₆	S ₇			5
3	S ₃	S ₄			8
4	r ₃	r ₃			
5			r ₁		
6	S ₆	S ₇			9
7			r ₃		
8	r ₂	r ₂			
9			r ₂		

Since table does not contain any conflict. So it is LR(1).
The goto table will be for LALR I_3 and I_6 will be unioned, I_4 and I_7 will be unioned, and I_8 and I_9 will be unioned.

So,

$I_{36}: B \rightarrow a \bullet B, a / b / \$$
 $B \rightarrow \bullet a B, a / b / \$$
 $B \rightarrow \bullet b, a / b / \$$

$I_{47}: B \rightarrow b \bullet, a / b / \$$

$I_{89}: B \rightarrow a B \bullet, a / b / \$$ and LALR table will be :

Table 2.30.2.

State	Action			Goto	
	a	b	\$	S	A
0	S ₃₆	S ₄₇		1	2
1			accept		
2	S ₃₆	S ₄₇			5
36	S ₃₆	S ₄₇			89
47	r ₃	r ₃	r ₃		
5			r ₁		
89	r ₂	r ₂	r ₂		

Since, LALR table does not contain any conflict. So, it is also LALR(1).

DFA:

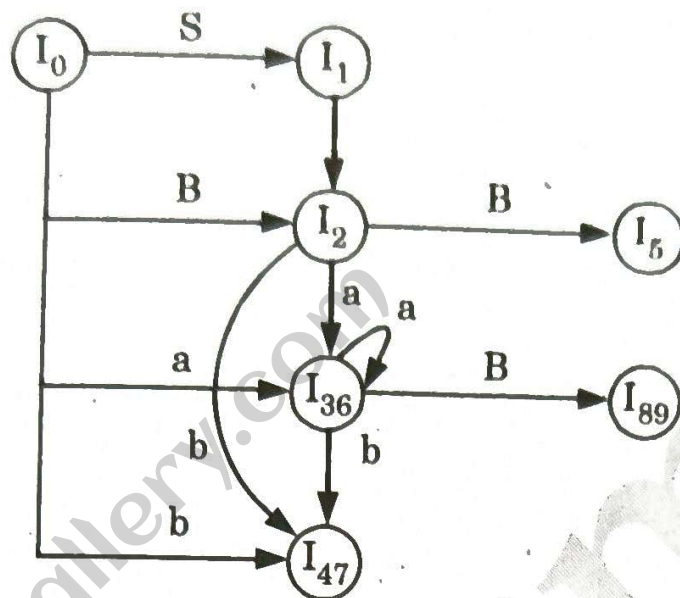


Fig. 2.30.1.

Que 2.31. Construct the CLR parse table for the following Grammar :

$A \rightarrow BB$

$B \rightarrow cB$

$B \rightarrow d$

AKTU 2022-23, Marks 10

Answer

The given grammar is :

$A \rightarrow BB$

$B \rightarrow cB \mid d$

The augmented grammar will be :

$A' \rightarrow A$

$A \rightarrow BB$

$B \rightarrow cB \mid d$

The LR (1) items will be :

$I_0: A' \rightarrow \bullet A, \$$

$A \rightarrow \bullet BB, \$$

$B \rightarrow \bullet cB, c/d$

$B \rightarrow \bullet d, c/d$

$I_1 = \text{GOTO}(I_0, A)$

$I_1: A' \rightarrow A \bullet, \$$

$I_2 = \text{GOTO}(I_0, B)$

$I_2: A \rightarrow B \bullet B, \$$
 $B \rightarrow \bullet cB, \$$
 $B \rightarrow \bullet d, \$$
 $I_3 = \text{GOTO } (I_0, c)$
 $I_3: B \rightarrow c \bullet B, c/d$
 $B \rightarrow \bullet cB, c/d$
 $B \rightarrow \bullet d, c/d$
 $I_4 = \text{GOTO } (I_0, c)$
 $I_4: B \rightarrow d \bullet, c/d$
 $I_5 = \text{GOTO } (I_2, B)$
 $I_5: A \rightarrow BB \bullet, \$$
 $I_6 = \text{GOTO } (I_2, c)$
 $I_6: B \rightarrow c \bullet B, \$$
 $B \rightarrow \bullet cB, \$$
 $B \rightarrow \bullet d, \$$
 $I_7 = \text{GOTO } (I_2, d)$
 $I_7: B \rightarrow d \bullet, \$$
 $I_8 = \text{GOTO } (I_3, B)$
 $I_8: B \rightarrow cB \bullet, c/d$
 $I_9 = \text{GOTO } (I_6, B)$
 $I_9: B \rightarrow cB \bullet, \$$

Table 2.31.1.

State	Action			Goto	
	c	d	\$	A	B
0	A ₃	A ₄		1	2
1			accept		
2	A ₆	A ₇			5
3	A ₃	A ₄			8
4	r ₃	r ₃			
5			r ₁		
6	A ₆	A ₇			9
7			r ₃		
8	r ₂	r ₂			
9			r ₂		

Since table does not contain any conflict. So it is LR(1).

The goto table will be for LALR I_3 and I_6 will be unioned, I_4 and I_7 will be unioned, and I_8 and I_9 will be unioned.

So,

$$I_{36}: B \rightarrow c \bullet B, c / d / \$$$

$$B \rightarrow \bullet cB, c / d / \$$$

$$B \rightarrow \bullet d, c / d / \$$$

$$I_{47}: B \rightarrow d \bullet, c / d / \$$$

$$I_{89}: B \rightarrow cB \bullet, c / d / \$ \text{ and LALR table will be :}$$

Table 2.31.2.

State	Action			Goto	
	<i>c</i>	<i>d</i>	<i>\$</i>	<i>A</i>	<i>B</i>
0	A_{36}	A_{47}		1	2
1			accept		
2	A_{36}	A_{47}			5
36	A_{36}	A_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Since, LALR table does not contain any conflict. So, it is also LALR(1).

DFA:

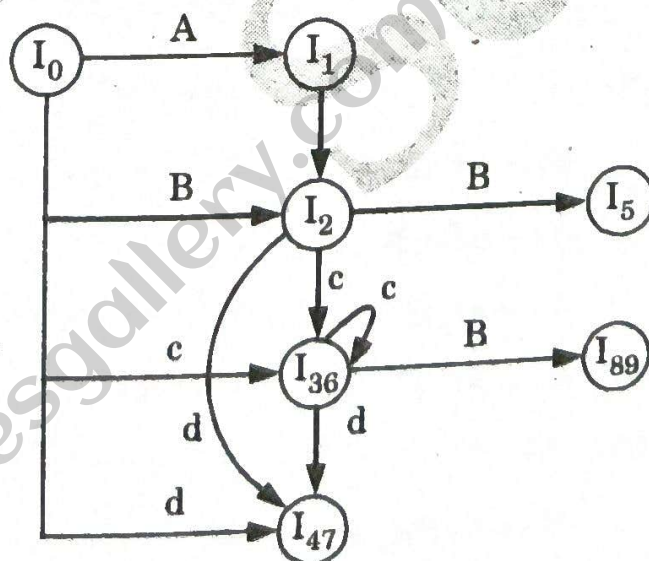


Fig. 2.31.1.

Que 2.32. For the grammar

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$, $A \rightarrow f$, $B \rightarrow f$

Construct LR(1) Parsing table. Also draw the LALR table.

AKTU 2023-24, Marks 10

Answer

Augmented grammar :

$S' \rightarrow S$

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

$A \rightarrow f$

$B \rightarrow f$

Canonical collection of LR(1) grammar :

$I_0:$

$S' \rightarrow \bullet S, \$$

$S \rightarrow \bullet aAd, \$$

$S \rightarrow \bullet bBd, \$$

$S \rightarrow \bullet aBe, \$$

$S \rightarrow \bullet bAe, \$$

$A \rightarrow \bullet f, d/e$

$B \rightarrow \bullet f, d/e$

$I_1 := \text{GOTO}(I_0, S)$

$I_1:$

$S' \rightarrow \bullet S, \$$

$I_2 := \text{GOTO}(I_0, a)$

$I_2:$

$S \rightarrow a \bullet Ad, \$$

$S \rightarrow aBe, \$$

$A \rightarrow \bullet f, d$

$B \rightarrow \bullet f, e$

$I_3 := \text{GOTO}(I_0, b)$

$I_3:$

$S \rightarrow b \bullet Bd, \$$

$$S \rightarrow b \bullet Ae, \$$$

$$A \rightarrow \bullet f, d$$

$$B \rightarrow \bullet f, e$$

$$I_4 := \text{GOTO}(I_2, A)$$

$$I_4: \quad S \rightarrow aA \bullet d, \$$$

$$I_5 := \text{GOTO}(I_2, B)$$

$$I_5: \quad S \rightarrow aB \bullet d, \$$$

$$I_6 := \text{GOTO}(I_2, f)$$

$$I_6: \quad A \rightarrow f \bullet, d$$

$$B \rightarrow f \bullet, e$$

$$I_7 := \text{GOTO}(I_3, B)$$

$$I_7: \quad S \rightarrow bB \bullet d, \$$$

$$I_8 := \text{GOTO}(I_3, A)$$

$$I_8: \quad S \rightarrow bA \bullet e, \$$$

$$I_9 := \text{GOTO}(I_3, f)$$

$$I_9: \quad A \rightarrow f \bullet, d$$

$$B \rightarrow f \bullet, e$$

$$I_{10} := \text{GOTO}(I_4, d)$$

$$I_{10}: \quad S \rightarrow aAd \bullet, \$$$

$$I_{11} := \text{GOTO}(I_5, d)$$

$$I_{11}: \quad S \rightarrow aBd \bullet, \$$$

$$I_{12} := \text{GOTO}(I_7, d)$$

$$I_{12}: \quad S \rightarrow bBd \bullet, \$$$

$$I_{13} := \text{GOTO}(I_8, e)$$

$$I_{13}: \quad S \rightarrow bAe \bullet, \$$$

State	Action						Goto		
	a	b	d	e	f	\$	A	B	S
I ₀	S ₂	S ₃							1
I ₁						accept			
I ₂					S ₆		4	5	
I ₃					S ₉		7	8	
I ₄			r ₁₀						
I ₅			r ₁₁						
I ₆					r ₆				
I ₇			r ₁₂						
I ₈				r ₁₃					
I ₉									
I ₁₀						r ₁			
I ₁₁						r ₃			
I ₁₂						r ₂			
I ₁₃						r ₄			

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. What is parser ? Write the role of parser. What are the most popular parsing techniques ?

Ans. Refer Q. 2.1.

Q. 2. Find the precedence and function table of the following grammar by using operator precedence technique.

$P \rightarrow SR \mid S, R \rightarrow bSR \mid bS, S \rightarrow WbS \mid W, W \rightarrow L^*W \mid L, L \rightarrow id.$

Ans. Refer Q. 2.7.

Q. 3. Construct predictive parse table for the following grammar.

$$E \rightarrow E + T/T$$

$$T \rightarrow T * F/F$$

$$F \rightarrow F / a/b$$

Ans. Refer Q. 2.15.

Q. 4. Construct the SLR parse table for the following Grammar

$$E' \rightarrow E$$

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow id$$

Ans. Refer Q. 2.21.

Q. 5. Construct the LALR parsing table for the given grammar

$$S \rightarrow BB$$

$$B \rightarrow aB/b$$

Ans. Refer Q. 2.30.

Q. 6. Construct the CLR parse table for the following Grammar :

$$A \rightarrow BB$$

$$B \rightarrow cB$$

$$B \rightarrow d$$

Ans. Refer Q. 2.31.



3

UNIT

Syntax-Directed Translations

CONTENTS

- Part-1** : Syntax-Directed Translation : 3-2C to 3-6C
Syntax-Directed Translation Scheme,
Implementation of Syntax-Directed
Translators
- Part-2** : Intermediate Code, Post Fix 3-6C to 3-9C
Notation, Parse Trees and
Syntax Trees
- Part-3** : Three Address Code, 3-10C to 3-17C
Quadruple and Triples
- Part-4** : Translation of Assignment 3-17C to 3-22C
Statements
- Part-5** : Boolean Expressions 3-22C to 3-26C
Statements that Alter
the Flow of Control
- Part-6** : Postfix Translation : Array 3-26C to 3-29C
Reference in Arithmetic
Expressions
- Part-7** : Procedures Call 3-29C to 3-32C
- Part-8** : Declarations Statements 3-32C to 3-33C

PART-1

Syntax-Directed Translation : Syntax-Directed Translation Schemes, Implementation of Syntax-Directed Translators.

Que 3.1. Define syntax directed translation. Construct an annotated parse tree for the expression $(4 * 7 + 1) * 2$, using the simple desk calculator grammar.

Answer

1. Syntax directed definition/translation is a generalization of context free grammar in which each grammar production $X \rightarrow \alpha$ is associated with a set of semantic rules of the form $a := f(b_1, b_2, \dots, b_k)$, where a is an attribute obtained from the function f .
2. Syntax directed translation is a kind of abstract specification.
3. It is done for static analysis of the language.
4. It allows subroutines or semantic actions to be attached to the productions of a context free grammar. These subroutines generate intermediate code when called at appropriate time by a parser for that grammar.
5. The syntax directed translation is partitioned into two subsets called the synthesized and inherited attributes of grammar.

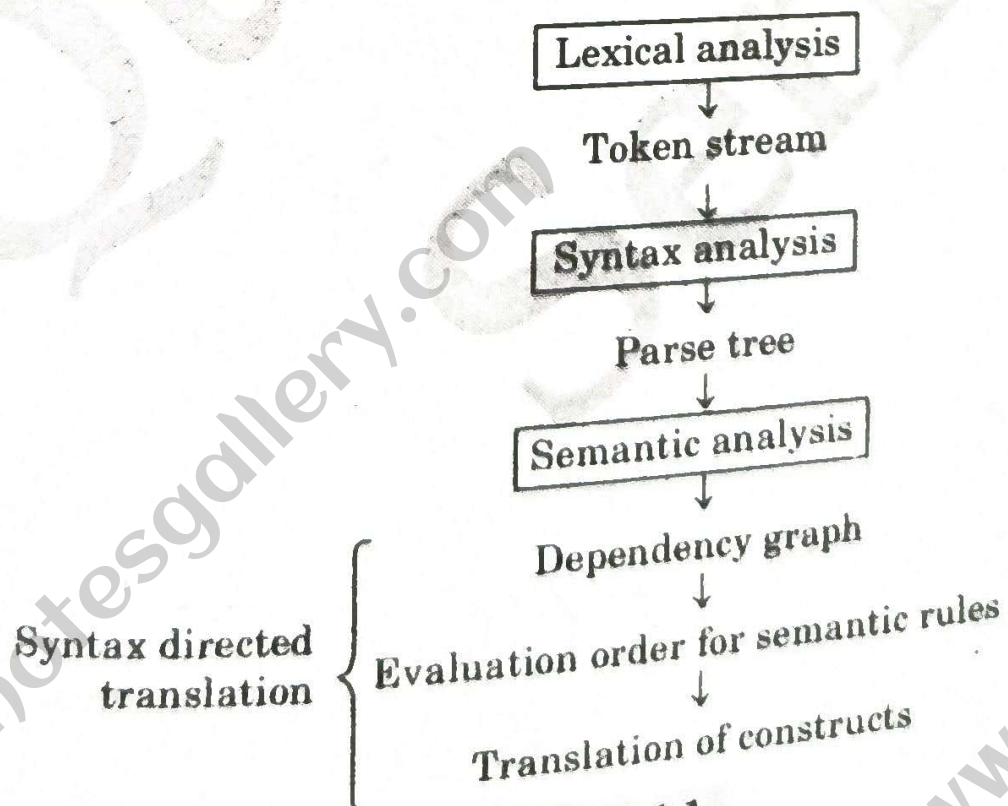


Fig. 3.1.1.

Annotated tree for the expression $(4 * 7 + 1) * 2$:

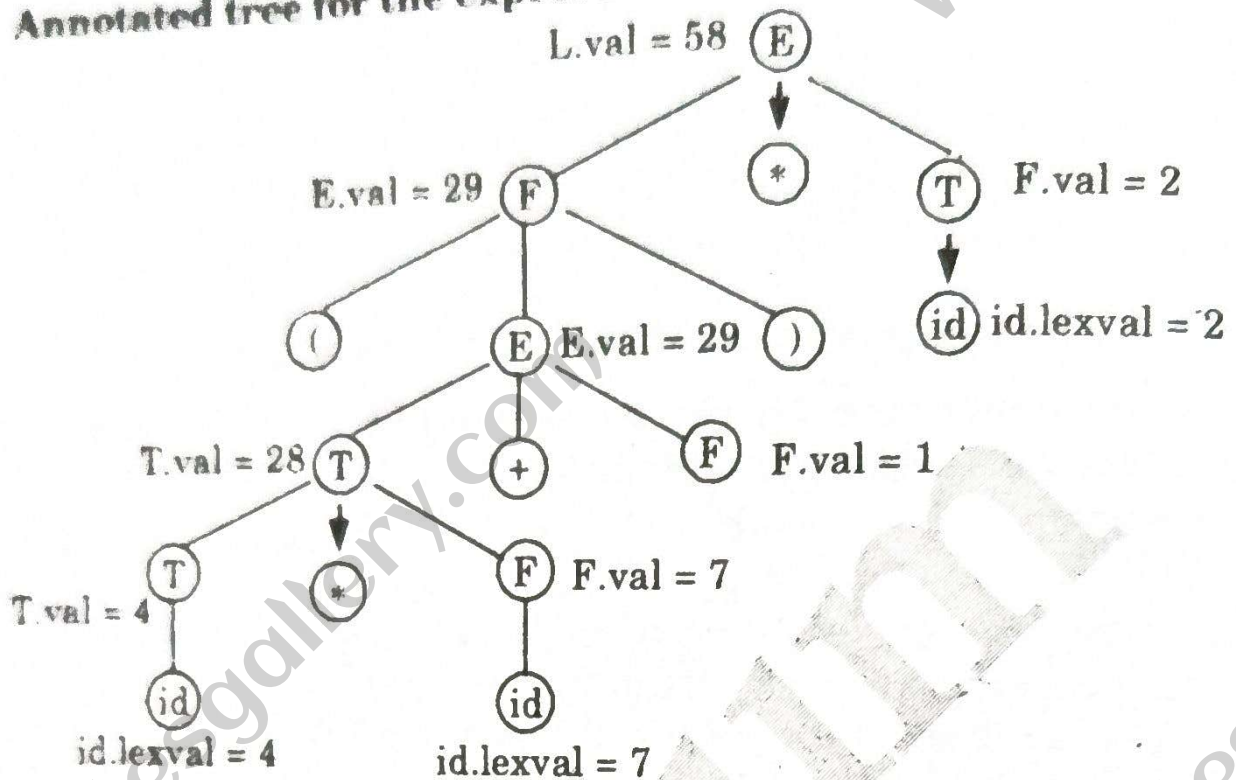


Fig. 3.1.2.

Que 3.2. What is syntax directed translation ? How are semantic actions attached to the production ? Explain with an example.

Answer

Syntax directed translation : Refer Q. 3.1, Page 3-2C, Unit-3.

Semantic actions are attached with every node of annotated parse tree.

Example : A parse tree along with the values of the attributes at nodes (called an "annotated parse tree") for an expression $2 + 3 * 5$ with synthesized attributes is shown in the Fig. 3.2.1.

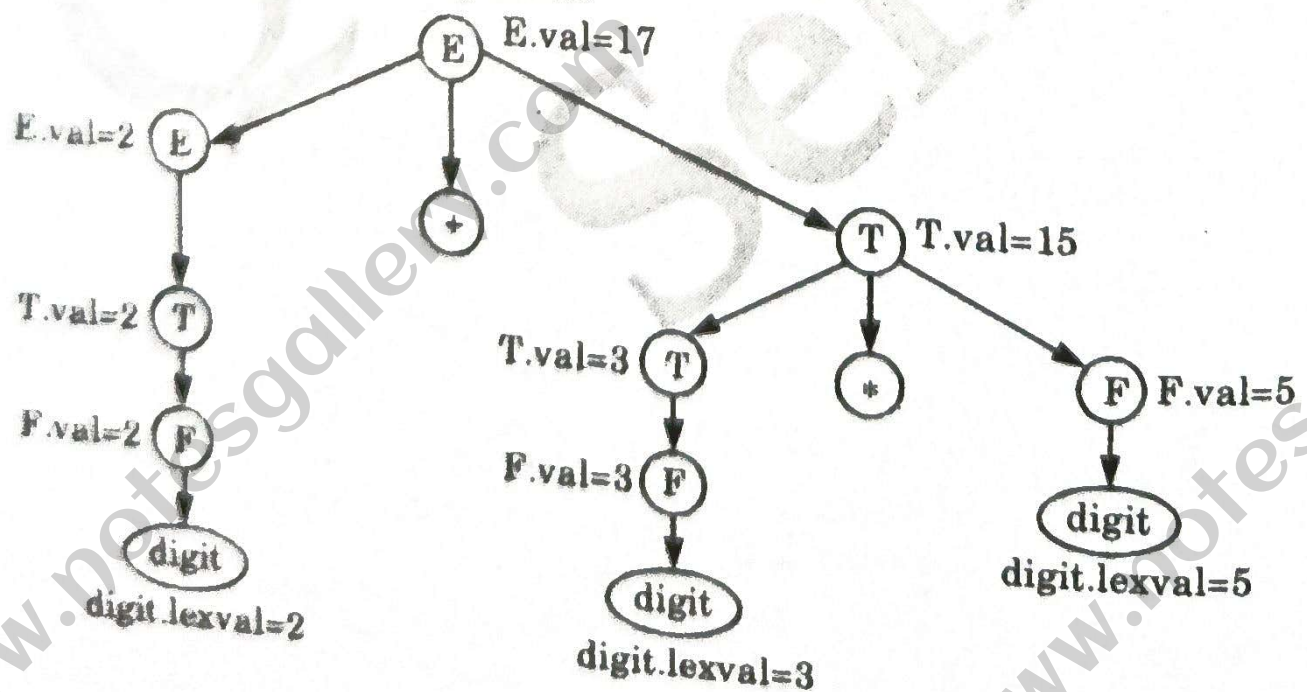


Fig. 3.2.1.

Que 3.3. Explain attributes. What are synthesized and inherited attribute ?

Answer**Attributes :**

1. Attributes are associated information with language construct by attaching them to grammar symbols representing that construct.
2. Attributes are associated with the grammar symbols that are the labels of parse tree node.
3. An attribute can represent anything (reasonable) such as string, a number, a type, a memory location, a code fragment etc.
4. The value of an attribute at parse tree node is defined by a semantic rule associated with the production used at that node.

Synthesized attribute :

1. An attribute at a node is said to be synthesized if its value is computed from the attributed values of the children of that node in the parse tree.
2. A syntax directed definition that uses the synthesized attributes is exclusively said to be S-attributed definition.
3. Thus, a parse tree for S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node from leaves to root.
4. If the translations are specified using S-attributed definitions, then the semantic rules can be conveniently evaluated by the parser itself during the parsing.

For example : A parse tree along with the values of the attributes at nodes (called an "annotated parse tree") for an expression $2 + 3 * 5$ with synthesized attributes is shown in the Fig. 3.3.1.

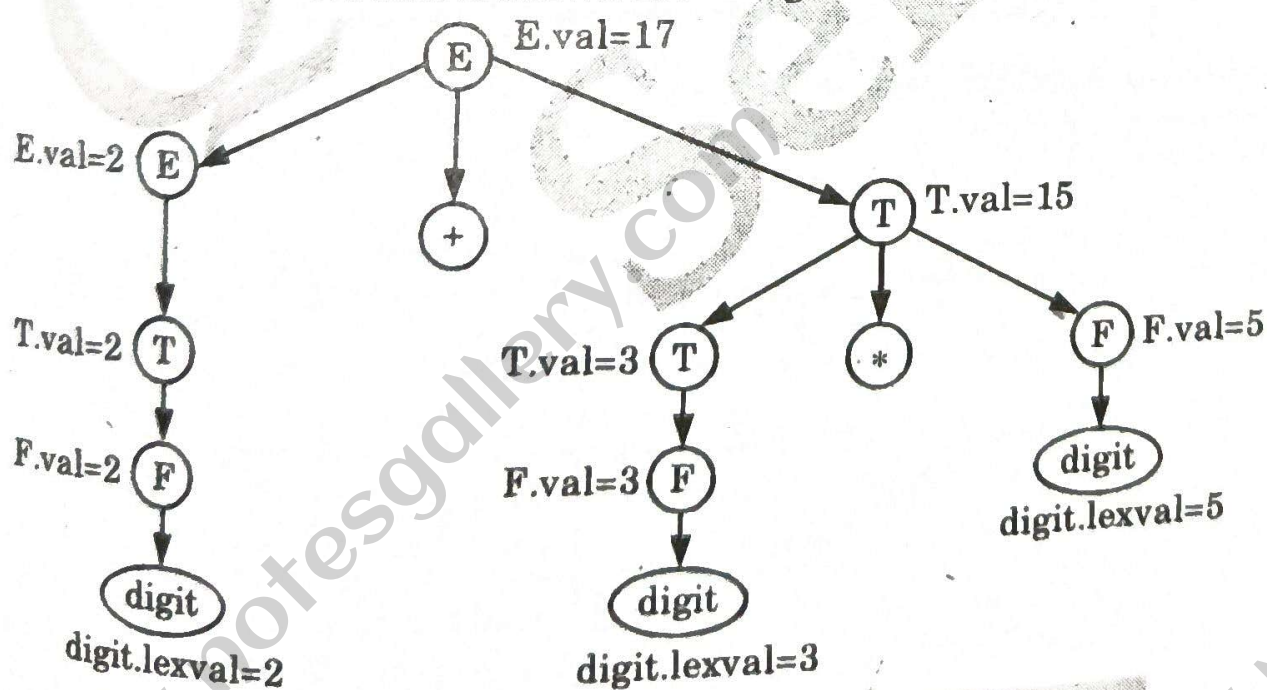


Fig. 3.3.1. An annotated parse tree for expression $2 + 3 * 5$.

Inherited attribute :

1. An inherited attribute is one whose value at a node in a parse tree is defined in terms of attributes at the parent and/or sibling of that node.

Compiler Design

2. Inherited attributes are convenient for expressing the dependence of a programming language construct.

For example : Syntax directed definitions that uses inherited attribute are given as :

$$D \rightarrow TL$$

$$T \rightarrow \text{int}$$

$$T \rightarrow \text{real}$$

$$L \rightarrow L_1, id$$

$$L \rightarrow id$$

$$L.type := T.Type$$

$$T.type := \text{integer}$$

$$T.type := \text{real}$$

$$L_1.type := L.type$$

$$\text{enter}(id.prt, L.type)$$

$$\text{enter}(id.prt, L.type)$$

The parse tree, along with the attribute values at the parse tree nodes, for an input string $\text{int } id_1, id_2 \text{ and } id_3$ is shown in the Fig. 3.3.2.

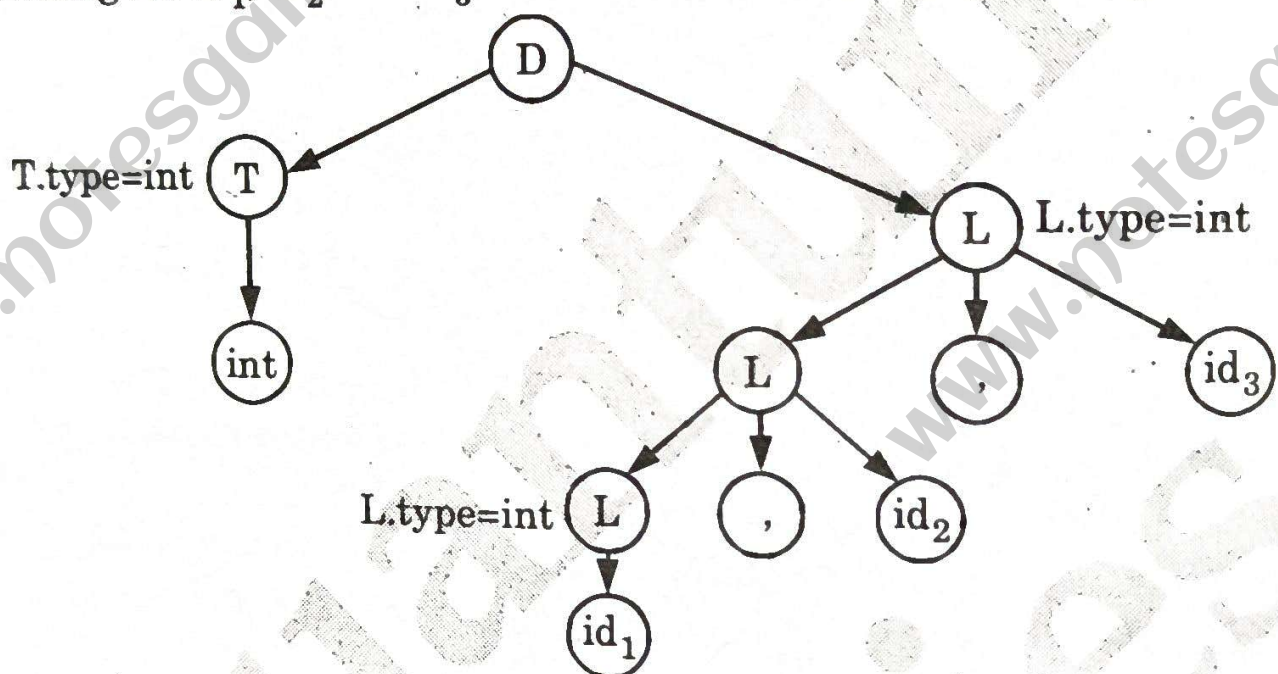


Fig. 3.3.2. Parse tree with inherited attributes for the string $\text{int } id_1, id_2, id_3$.

Que 3.4. What is the difference between S-attributed and L-attributed definitions?

Answer

S.No.	S-attributed definition	L-attributed definition
1.	It uses synthesized attributes.	It uses synthesized and inherited attributes.
2.	Semantics actions are placed at right end of production.	Semantics actions are placed at anywhere on RHS.
3.	S-attributes can be evaluated during parsing.	L-attributes are evaluated by traversing the parse tree in depth first, left to right.

Que 3.5. Consider the following grammar and give the syntax directed definition to construct parse tree for the input expression $4*7 + 3*9$. Also construct an annotated parse tree according to your syntax directed definition.

$E \rightarrow E + T \mid T, T \rightarrow T * F \mid F, F \rightarrow \text{num.}$

AKTU 2020-21, Marks 10

Answer

Production :

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (F)$

$F \rightarrow \text{num}$

Syntax directed definition :

$E.\text{code} = E \text{ code} \parallel T \text{ code} \parallel '+'$

$E \text{ code} = T.\text{code}$

$T \text{ code} = T \text{ code} \parallel F.\text{code} \parallel 'x'$

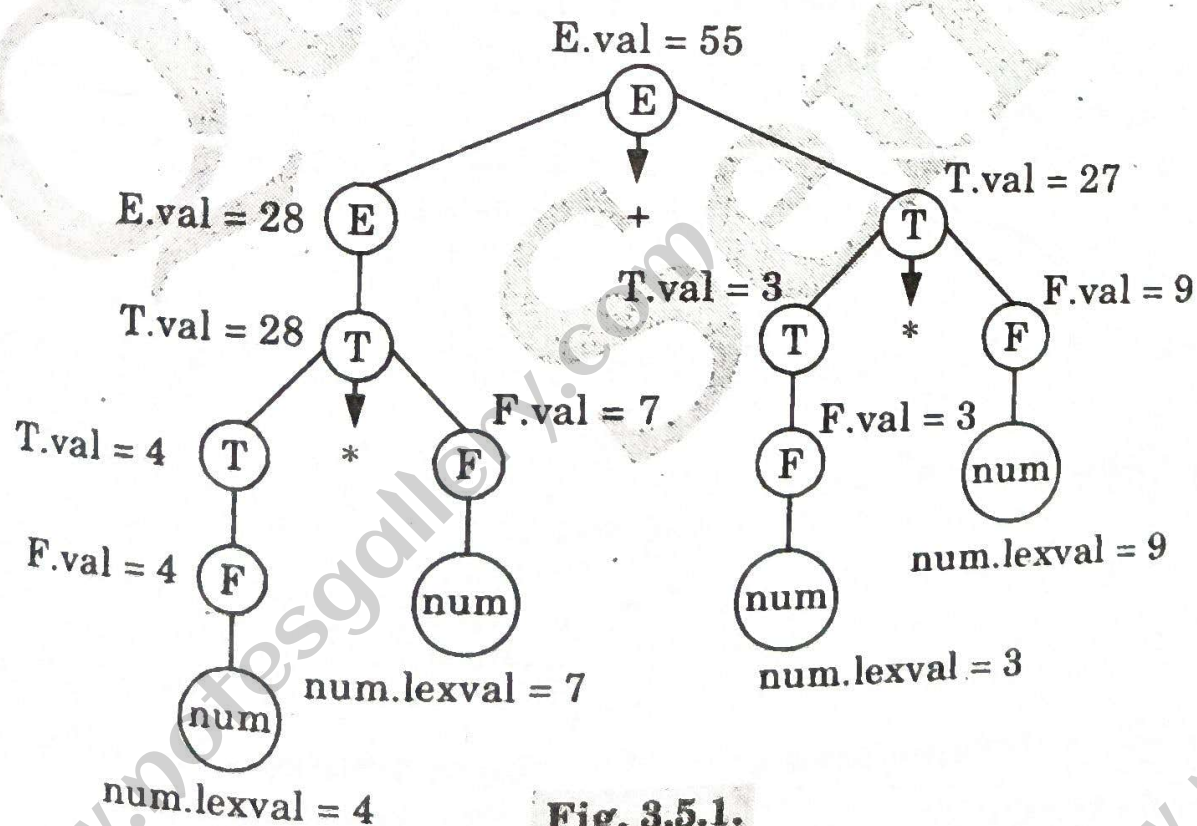
$T.\text{code} = F.\text{code}$

$F.\text{code} = E.\text{code}$

$F.\text{code} = \text{num.}\text{code}$

where "||" sign is used for concatenation

Annotated parse tree :



Que 3.6. What is intermediate code generation and discuss benefits of intermediate code ?

Answer

Intermediate code generation is the fourth phase of compiler which takes parse tree as an input from semantic phase and generates an intermediate code as output.

The benefits of intermediate code are :

1. Intermediate code is machine independent, which makes it easy to retarget the compiler to generate code for newer and different processors.
2. Intermediate code is nearer to the target machine as compared to the source language so it is easier to generate the object code.
3. The intermediate code allows the machine independent optimization of the code by using specialized techniques.
4. Syntax directed translation implements the intermediate code generation, thus by augmenting the parser, it can be folded into the parsing.

Que 3.7. What is postfix translation ? Explain it with suitable example.

Answer

Postfix (reverse polish) translation : It is the type of translation in which the operator symbol is placed after its two operands.

For example :

Consider the expression : $(20 + (-5) * 6 + 12)$

Postfix for above expression can be calculated as :

$$\begin{array}{ll}
 (20 + t_1 * 6 + 12) & t_1 = 5 - \\
 20 + t_2 + 12 & t_2 = t_1 6 * \\
 t_3 + 12 & t_3 = 20 t_2 + \\
 t_4 & t_4 = t_3 12 +
 \end{array}$$

Now putting values of t_4, t_3, t_2, t_1

$$\begin{aligned}
 t_4 &= t_3 12 + \\
 &= 20 t_2 + 12 + \\
 &= 20 t_1 6 * + 12 + \\
 &= (20) 5 - 6 * + 12 +
 \end{aligned}$$

Que 3.8. Define parse tree. Why parse tree construction is only possible for CFG ?

Answer

Parse tree : A parse tree is an ordered tree in which left hand side of a production represents a parent node and children nodes are represented by the production's right hand side.

Conditions for constructing a parse tree from a CFG are :

- Each vertex of the tree must have a label. The label is a non-terminal or terminal or null (ϵ).
- The root of the tree is the start symbol, i.e., S .
- The label of the internal vertices is non-terminal symbols $\in V_N$.
- If there is a production $A \rightarrow X_1 X_2 \dots X_K$. Then for a vertex, label A , the children node, will be $X_1 X_2 \dots X_K$.
- A vertex n is called a leaf of the parse tree if its label is a terminal symbol $\in \Sigma$ or null (ϵ).

Parse tree construction is only possible for CFG. This is because the properties of a tree match with the properties of CFG.

Que 3.9. What is syntax tree ? What are the rules to construct syntax tree for an expression ?

Answer

- A syntax tree is a tree that shows the syntactic structure of a program while omitting irrelevant details present in a parse tree.
- Syntax tree is condensed form of the parse tree.
- The operator and keyword nodes of a parse tree are moved to their parent and a chain of single production is replaced by single link.

Rules for constructing a syntax tree for an expression :

- Each node in a syntax tree can be implemented as a record with several fields.
- In the node for an operator, one field identifies the operator and the remaining field contains pointer to the nodes for the operands.
- The operator often is called the label of the node.
- The following functions are used to create the nodes of syntax trees for expressions with binary operators. Each function returns a pointer to newly created node.
 - Mknode(op, left, right)** : It creates an operator node with label op and two field containing pointers to left and right.
 - Mkleaf(id, entry)** : It creates an identifier node with label id and the field containing entry, a pointer to the symbol table entry for the identifier.
 - Mkleaf(num, val)** : It creates a number node with label num and a field containing val , the value of the number.

For example : Construct a syntax tree for an expression $a - 4 + c$. In this sequence, $p_1, p_2, \dots, p_5$ are pointers to nodes, and entry a and entry c are pointers to the symbol table entries for identifier 'a' and 'c' respectively.

$p_1 := \text{mkleaf}(id, \text{entry } a);$

$p_2 := \text{mkleaf}(\text{num}, 4);$
 $p_3 := \text{mknnode}('-', p_1, p_2);$
 $p_4 := \text{mkleaf}(\text{id}, \text{entry } c);$
 $p_5 := \text{mknnode}('+', p_3, p_4);$

The tree is constructed in bottom-up fashion. The function calls `mkleaf` (`id`, entry `a`) and `mkleaf` (`num`, 4) construct the leaves for `a` and 4. The pointers to these nodes are saved using p_1 and p_2 . Call `mknnode` (`'-'`, p_1, p_2) then constructs the interior node with the leaves for `a` and 4 as children. The syntax tree will be :

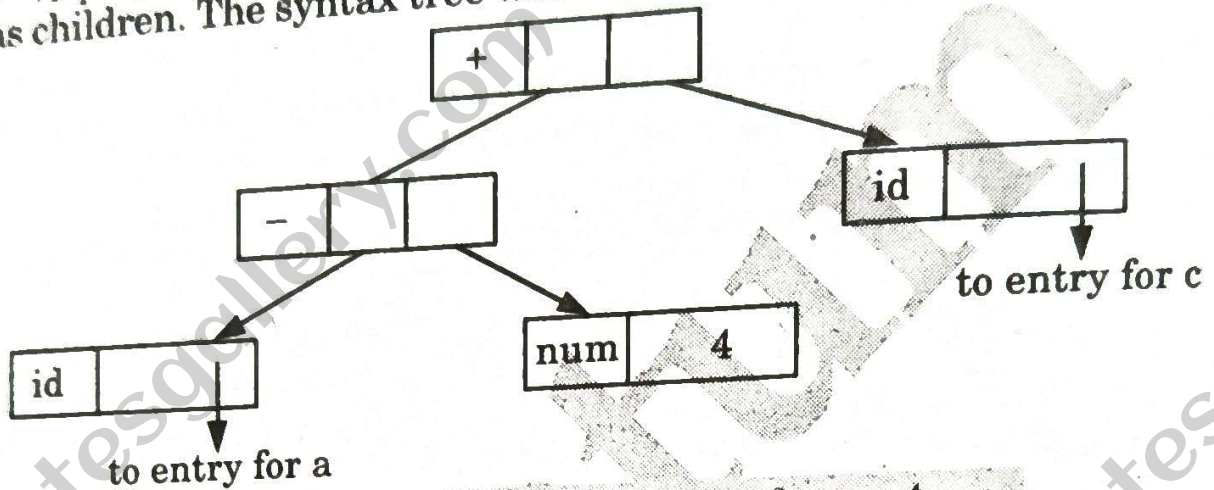


Fig. 3.9.1. The syntax tree for $a - 4 + c$.

Que 3.10. Draw syntax tree for the arithmetic expressions : $a * (b + c) - d/2$. Also write the given expression in postfix notation.

Answer

Syntax tree for given expression : $a * (b + c) - d/2$

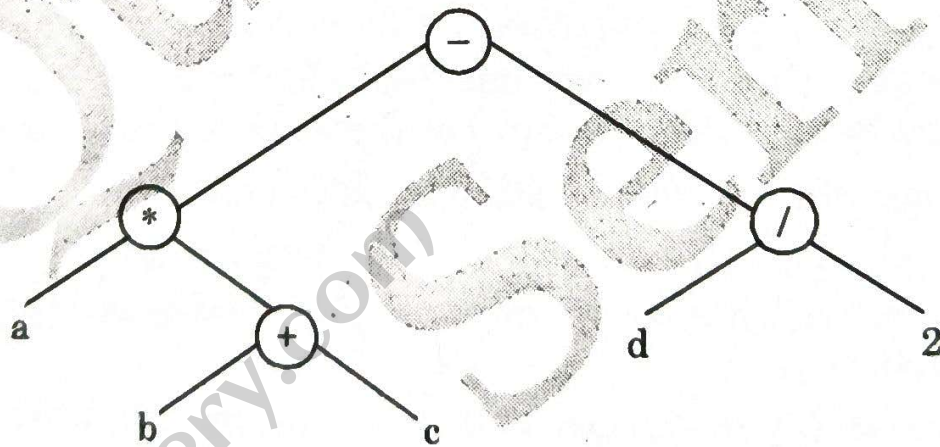


Fig. 3.10.1.

Postfix notation for $a * (b + c) - d/2$

$$t_1 = bc +$$

$$t_2 = a t_1 *$$

$$t_3 = d 2 /$$

$$t_4 = t_2 t_3 -$$

Put value of t_1, t_2, t_3

$$(a * t_1 - d/2)$$

$$(t_2 - d/2)$$

$$(t_2 - t_3)$$

$$(t_4)$$

$$t_4 = t_2 t_3 -$$

$$= t_2 d 2 / -$$

$$= a t_1 * d 2 / -$$

$$= abc + * d 2 / -$$

PART-3*Three Address Code, Quadruples and Triples.*

Que 3.11. Explain three address code with examples.

Answer

1. Three address code is an abstract form of intermediate code that can be implemented as a record with the address fields.
2. The general form of three address code representation is :

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

where a , b and c are operands that can be names, constants and op represents the operator.

3. The operator can be fixed or floating point arithmetic operator or logical operators or boolean valued data. Only single operation at right side of the expression is allowed at a time.
4. There are at most three addresses are allowed (two for operands and one for result). Hence, the name of this representation is three address code.

For example : The three address code for the expression $a = b + c + d$ will be :

$$t_1 := b + c$$

$$t_2 := t_1 + d$$

$$a := t_2$$

Here t_1 and t_2 are the temporary names generated by the compiler.

Que 3.12. What are different ways to write three address code ?

Answer

Different ways to write three address code are :

1. **Quadruple representation :**
 - a. The quadruple is a structure with at most four fields such as op , $arg1$, $arg2$, $result$.
 - b. The op field is used to represent the internal code for operator, the $arg1$ and $arg2$ represent the two operands used and $result$ field is used to store the result of an expression.

For example : Consider the input statement $x := -a * b + -a * b$.

The three address code is

$t_1 := \text{uminus } a$
 $t_2 := t_1 * b$
 $t_3 := -a$
 $t_4 := t_3 * b$
 $t_5 := t_2 + t_4$
 $x := t_5$

Op	Arg1	Arg2	Result
uminus	a		t_1
*	t_1	b	t_2
uminus	a		t_3
*	t_3	b	t_4
+	t_2	t_4	t_5
:=	t_5		x

2. **Triples representation** : In the triple representation, the use of temporary variables is avoided by referring the pointers in the symbol table.

For example : $x := -a * b + -a * b$

The triple representation is

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	x	(4)

3. **Indirect triples representation** : In the indirect triple representation, the listing of triples is done and listing pointers are used instead of using statement.

For example : $x = -a * b + -a + b$

The indirect triples representation is

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(11)	b
(2)	uminus	a	
(3)	*	(13)	b
(4)	+	(12)	(14)
(5)	:=	x	(15)

Location	Statement
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)

Three address code of given statement is :

1. If $A > C$ and $B < D$ goto 2
2. If $A = 1$ goto 6
3. If $A \leq D$ goto 6
4. $t_1 = A + 2$
5. $A = t_1$

6. $t_2 = C + 1$

7. $C = t_2$

Que 3.13. Write the quadruples, triple and indirect triple for the following expression :

$$(x + y) * (y + z) + (x + y + z)$$

AKTU 2021-22, Marks 10

Answer

The three address code for given expression :

$$t_1 := x + y$$

$$t_2 := y + z$$

$$t_3 := t_1 * t_2$$

$$t_4 := t_1 + z$$

$$t_5 := t_3 + t_4$$

i. The quadruple representation :

Location	Operator	Operand 1	Operand 2	Result
(1)	+	x	y	t_1
(2)	+	y	z	t_2
(3)	*	t_1	t_2	t_3
(4)	+	t_1	z	t_4
(5)	+	t_3	t_4	t_5

ii. The triple representation :

Location	Operator	Operand 1	Operand 2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

iii. The indirect triple representation :

Location	Operator	Operand 1	Operand 2	Location	Statement
(1)	+	x	y	(1)	(11)
(2)	+	y	z	(2)	(12)
(3)	*	(11)	(12)	(3)	(13)
(4)	+	(11)	z	(4)	(14)
(5)	+	(13)	(14)	(5)	(15)

Que 3.14. Generate three address code for the following code :

```
switch a + b
{
    case 1 : x = x + 1
    case 2 : y = y + 2
    case 3 : z = z + 3
    default : c = c - 1
}
```

AKTU 2020-21, Marks 10

Answer

```
101:  $t_1 = a + b$  goto 103
102: goto 115
103:  $t = 1$  goto 105
104: goto 107
105:  $t_2 = x + 1$ 
106:  $x = t_2$ 
107: if  $t = 2$  goto 109
108: goto 111
109:  $t_3 = y + 2$ 
110:  $y = t_3$ 
111: if  $t = 3$  goto 113
112: goto 115
113:  $t_4 = z + 3$ 
114:  $z = t_4$ 
115:  $t_5 = c - 1$ 
116:  $c = t_5$ 
117: Next statement
```

Que 3.15. Translate the following arithmetic expression into quadruples and triples :

- i. $x = y * z + y * -z$
- ii. $a = -b * (c + d) + b$

AKTU 2020-21, Marks 10

Answer

i. Three address code for given expression :

```
 $x = y * z + y * -z$ 
 $t_1 = -z$ 
 $t_2 = y * t_1$ 
 $t_3 = y * z$ 
```


$$t_4 = t_3 + t_2$$

$$x = t_4$$

Quadruples representation :

Location	Operator	Operand 1	Operand 2	Result
(1)	Uminus	z	$-$	t_1
(2)	*	y	t_1	t_2
(3)	*	y	z	t_3
(4)	+	t_3	t_2	t_4
(5)	=	t_4	$-$	x

Triples representation :

Location	Operator	Operand 1	Operand 2
(1)	Uminus	z	$-$
(2)	*	y	(1)
(3)	*	y	z
(4)	+	(3)	(2)
(5)	=	(4)	$-$

ii

More address code for given expression :

$$t_1 = -b$$

$$t_2 = c + d$$

$$t_3 = t_1 * t_2$$

$$t_4 = t_3 + b$$

$$a = t_4$$

Quadruple representation :

Location	Operator	Operand 1	Operand 2	Result
(1)	Uminus	b	$-$	t_1
(2)	+	c	d	t_2
(3)	*	t_1	t_2	t_3
(4)	+	t_3	b	t_4
(5)	=	t_4	$-$	a

Triples representation :

Location	Operator	Operand 1	Operand 2
(1)	Uminus	b	d
(2)	+	c	(2)
(3)	*	(1)	b
(4)	+	(3)	$-$
(5)	=	(4)	

Que 3.16.

Generate the three address code for the following code fragment.

```

while(a > b)
{
  if(c < d)
    x = y + z;
  else
    x = y - z;
}

```

AKTU 2021-22, Marks 10**Answer**

Three address code for the given code :

1. If ($a > b$) goto (3)
2. goto
3. If ($c < d$) goto
4. goto (7)
5. $t_1 = y + z$
6. $x = t_1$
7. $t_2 = y - z$
8. $x = t_2$

Que 3.17.

Write quadruple, triples and indirect triples for following expression :

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

AKTU 2022-23, Marks 10**Answer**

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

The three address code

$$t_1 = \text{uminus } c$$

$$t_2 = t_1 * b$$

$$t_3 = t_1 + t_2$$

$$a = t_3$$

1. Quadruple representation :

op	Arg 1	Arg 2	Result
uminus	c		t_1
*	t_1	b	t_2
+	t_1	t_2	t_2
=	t_1		a

2. Triples representation :

Number	op	Arg 1	Arg 2
(0)	uminus	c	
(1)	*	t ₁	b
(2)	+	(1)	(1)
(3)	=	a	(2)

3. Indirect triples representation :

Number	op	Arg 1	Arg 2
(0)	uminus	c	
(1)	*	(11)	b
(2)	+	(12)	(12)
(3)	=	a	(13)

Location	Statement
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)

Que 3.18. Explain the concepts of quadruples and triples in the context of syntax-directed translation. Discuss how they represent intermediate code and support optimization.

AKTU 2023-24, Marks 10

Answer

Quadruples and triples : Refer Q. 3.12, Page 3-10C, Unit-3.

Representation of intermediate code :

- Both quadruples and triples serve as intermediate representations of code generated during compilation or translation.
- They abstract away from the source language's syntax and semantics, providing a platform-independent representation that can be optimized and translated into target machine code.

Support for optimization :

- Optimizations are performed on intermediate code before translating it into machine code, leading to improved performance and efficiency of the resulting executable.
- Quadruples and triples play a crucial role in supporting optimization techniques during compilation. Some ways they facilitate optimization include :

Compiler Design

- a. **Common sub expression elimination** : Detecting and eliminating redundant computations by analyzing intermediate code.
 - b. **Dead code elimination** : Identifying and removing code that does not contribute to the final output.
 - c. **Code motion** : Moving computations outside of loops or repetitive structures to reduce redundant calculations.
 - d. **Register allocation** : Optimizing the usage of processor registers to minimize memory access operations.
3. Quadruples and triples provide a structured representation of code that facilitates the implementation of optimization techniques.

PART-4*Translation of Assignment Statements.*

Que 3.19. How would you convert the following into intermediate code? Give a suitable example.

- i. Assignment statements
- ii. Case statements

Answer

- i. **Assignment statements :**

Production rule	Semantic actions
$S \rightarrow id := E$	{ $id_entry := look_up(id.name);$ if $id_entry \neq nil$ then append ($id_entry := E.place$) else error; /* id not declared */ }
$E \rightarrow E_1 + E_2$	{ $E.place := newtemp();$ append ($E.place := E_1.place + E_2.place$) }
$E \rightarrow E_1 * E_2$	{ $E.place := newtemp();$ append ($E.place := E_1.place * E_2.place$) }
$E \rightarrow -E_1$	{ $E.place := newtemp();$ append ($E.place := 'minus' E_1.place$) }

$E \rightarrow (E_1)$	
$E \rightarrow id$	<pre> { E.place := E₁.place } { id_entry := look_up(id.name); if id_entry ≠ nil then append(id_entry := E.place) else error; /* id not declared */ } </pre>

1. The look_up returns the entry for *id.name* in the symbol table if it exists there.
2. The function append is used for appending the three address code to the output file. Otherwise, an error will be reported.
3. Newtemp() is the function used for generating new temporary variables.
4. *E.place* is used to hold the value of *E*.

Example : $x := (a + b) * (c + d)$

We will assume all these identifiers are of the same type. Let us have bottom-up parsing method :

Production rule	Semantic action attribute evaluation	Output
$E \rightarrow id$	$E.place := a$	
$E \rightarrow id$	$E.place := b$	
$E \rightarrow E_1 + E_2$	$E.place := t_1$	$t_1 := a + b$
$E \rightarrow id$	$E.place := c$	
$E \rightarrow id$	$E.place := d$	
$E \rightarrow E_1 + E_2$	$E.place := t_2$	$t_2 := c + d$
$E \rightarrow E_1 * E_2$	$E.place := t_3$	$t_3 := (a + b) * (c + d)$
$S \rightarrow id := E$		$x := t_3$

ii. Case statements:

Production rule	Semantic action
Switch E { case $v_1 : s_1$ case $v_2 : s_2$... case $v_{n-1} : s_{n-1}$ default : s_n }	Evaluate E into t such that $t = E$ goto check L_1 : code for s_1 goto last L_2 : code for s_2 goto last L_n : code for s_n goto last check : if $t = v_1$ goto L_1 if $t = v_2$ goto L_2 ... if $t = v_{n-1}$ goto L_{n-1} goto L_n last

```

switch expression
{
case value : statement
case value : statement
...
case value : statement
default : statement
}

```

Example :

```

switch(ch)
{
    case 1 : c = a + b;
    break;
    case 2 : c = a - b;
    break;
}

```

The three address code can be

```

if ch = 1 goto  $L_1$ 
if ch = 2 goto  $L_2$ 
 $L_1$ :  $t_1 := a + b$ 
      $c := t_1$ 

```


goto last

$L_2: t_2 := a - b$

$c := t_2$

goto last

last :

Que 3.20. Write down the translation procedure for control statement and switch statement.

Answer

1. Boolean expression are used along with if-then, if-then-else, while-do, do-while statement constructs.
2. $S \rightarrow \text{If } E \text{ then } S_1 \mid \text{If } E \text{ then } S_1 \text{ else } S_2 \mid \text{while } E \text{ do } S_1 \mid \text{do } E_1 \text{ while } E.$
3. All these statements ' E ' correspond to a boolean expression evaluation.
4. This expression ' E ' should be converted to three address code.
5. This is then integrated in the context of control statement.

Translation procedure for if-then and if-then-else statement :

1. Consider a grammar for if-else

$$S \rightarrow \text{if } E \text{ then } S_1 \mid \text{if } E \text{ then } S_1 \text{ else } S_2$$

2. Syntax directed translation scheme for if-then is given as follows :

$$S \rightarrow \text{if } E \text{ then } S_1$$

$$E.\text{true} := \text{new_label}()$$

$$E.\text{false} := S.\text{next}$$

$$S_1.\text{next} := S.\text{next}$$

$$S.\text{code} := E.\text{code} \parallel \text{gen_code}(E.\text{true} ':') \parallel S_1.\text{code}$$

3. In the given translation scheme $\parallel$ is used to concatenate the strings.
4. The function `gen_code` is used to evaluate the non-quoted arguments passed to it and to concatenate complete string.
5. The $S.\text{code}$ is the important rule which ultimately generates the three address code.

$$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$$

$$E.\text{true} := \text{new_label}()$$

$$E.\text{false} := \text{new_label}()$$

$$S_1.\text{next} := S.\text{next}$$

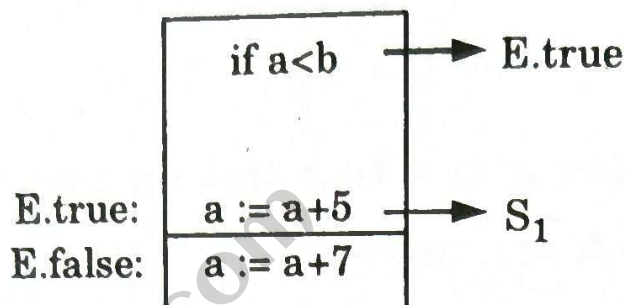
$$S_2.\text{next} := S.\text{next}$$

$$S.\text{code} := E.\text{code} \parallel \text{gen_code}(E.\text{true} ':') \parallel$$

$$S_1.\text{code} := \text{gen_code}(\text{'goto'}, S.\text{next}) \parallel$$

$\text{gen_code}(E.\text{false} ':') \parallel S_2.\text{code}$

For example : Consider the statement if $a < b$ then $a = a + 5$ else $a = a + 7$



The three address code for if-else is

100 : if $a < b$ goto 102

101 : goto 103

102 : $L1 \ a := a + 5 \ /*E.\text{true}*/$

103 : $L2 \ a := a + 7$

Hence, $E.\text{code}$ is "if $a < b$ " $L1$ denotes $E.\text{true}$ and $L2$ denotes $E.\text{false}$ is shown by jumping to line 103 (i.e., $S.\text{next}$).

Translation procedure for while-do statement :

Production

$S \rightarrow \text{while } E \text{ do } S1$

Semantic rules

$S.\text{begin} := \text{newlable}$
 $E.\text{true} := \text{newlable}$
 $E.\text{false} := S.\text{next}$
 $S1.\text{next} := S.\text{begin}$
 $S.\text{code} = \text{gen}(S.\text{begin} ':') \parallel$
 $E.\text{code} \parallel$
 $\text{gen}(E.\text{true} ':') \parallel S1.\text{code} \parallel$
 $\text{gen}(\text{'goto' } S.\text{being})$

Translation procedure for do-while statement :

Production

$S \rightarrow \text{do } S1 \text{ while } E$

Semantic rules

$S.\text{begin} := \text{newlable}$
 $E.\text{true} := S.\text{begin}$
 $E.\text{false} := S.\text{next}$
 $S.\text{code} = S1.\text{code} \parallel E.\text{code} \parallel$
 $\text{gen}(E.\text{true} ':') \parallel$
 $\text{gen}(\text{'goto' } S.\text{being})$

Que 3.21.

Write syntax directed definition for a given assignment statement :

$S \rightarrow \text{id} = E$

$E \rightarrow E + E$

$E \rightarrow E * E$

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

AKTU 2021-22, 2023-24; Marks 10

Answer

The syntax directed definition of given grammar is :

S.No.	Production rule	Semantic actions
1.	$S \rightarrow id := E$	{p = look_up(id.name); If p ≠ nil then Emit (p = E.place) Else Error; }
2.	$E \rightarrow E + E$	E.place = newtemp(); Emit (E.place = E.place '+' E.place) }
3.	$E \rightarrow E * E$	{E.place = newtemp(); Emit (E.place = E.place '*' E.place) }
4.	$E \rightarrow -E$	{E.place = -E.place }
5.	$E \rightarrow id$	{p = look_up(id.name); If p ≠ nil then Emit (p = E.place) Else Error; }

PART-5

Boolean Expressions, Statements that alter the Flow of Control.

Que 3.22. Define backpatching and semantic rules for boolean expression. Derive the three address code for the following expression : $P < Q$ or $R < S$ and $T < U$.

OR

Write short notes on backpatching.

Answer

1. Backpatching is the activity of filling up unspecified information of labels using appropriate semantic actions during the code generation process.

2. Backpatching refers to the process of resolving forward branches that have been used in the code, when the value of the target becomes known.
3. Backpatching is done to overcome the problem of processing the incomplete information in one pass.
4. Backpatching can be used to generate code for boolean expressions and flow of control statements in one pass.

To generate code using backpatching following functions are used :

1. **Makelist(i)** : Makelist is a function which creates a new list from one item where i is an index into the array of instructions.
2. **Merge(p_1, p_2)** : Merge is a function which concatenates the lists pointed by p_1 and p_2 , and returns a pointer to the concatenated list.
3. **Backpatch(p, i)** : Inserts i as the target label for each of the instructions on the list pointed by p .

Backpatching in boolean expressions :

1. The solution is to generate a sequence of branching statements where the addresses of the jumps are temporarily left unspecified.
2. For each boolean expression E we maintain two lists :
 - a. E .truelist which is the list of the (addresses of the) jump statements appearing in the translation of E and forwarding to E .true.
 - b. E .falselist which is the list of the (addresses of the) jump statements appearing in the translation of E and forwarding to E .false.
3. When the label E .true (resp. E .false) is eventually defined we can walk down the list, patching in the value of its address.
4. In the translation scheme below :
 - a. We use emit to generate code that contains place holders to be filled in later by the backpatch procedure.
 - b. The attributes E .truelist, E .falselist are synthesized.
 - c. When the code for E is generated, addresses of jumps corresponding to the values true and false are left unspecified and put on the lists E .truelist and E .falselist, respectively.
5. A marker non-terminal M is used to capture the numerical address of a statement.
6. nextinstr is a global variable that stores the number of the next statement to be generated.

The grammar is as follows :

$$B \rightarrow B_1 \parallel MB_2 \mid B_1 \text{ AND } MB_2 \mid !B_1 \mid (B_1) \mid E_1 \text{ rel } E_2 \mid \text{True} \mid \text{False}$$

$$M \rightarrow \epsilon$$

The translation scheme is as follows :

- i. $B \rightarrow B_1 \parallel MB_2$ {backpatch (B_1 .falselist, M .instr);

- $B.\text{truelist} = \text{merge}(B_1.\text{truelist}, B_2.\text{truelist});$
 $B.\text{falselist} = B_2.\text{falselist};$
- ii. $B \rightarrow B_1 \text{ AND } MB_2$
- $\{\text{backpatch}(B_1.\text{truelist}, M.\text{instr});$
 $B.\text{truelist} = B_2.\text{truelist};$
 $B.\text{falselist} = \text{merge}(B_1.\text{falselist}, B_2.\text{falselist});\}$
- iii. $B \rightarrow !B_1 \{B.\text{truelist} = B_1.\text{falselist};$
 $B.\text{falselist} = B_1.\text{truelist};\}$
- iv. $B \rightarrow (B_1) \{B.\text{truelist} = B_1.\text{truelist};$
 $B.\text{falselist} = B_1.\text{falselist};\}$
- v. $B \rightarrow E_1 \text{ rel } E_2 \{B.\text{truelist} = \text{makelist}(\text{nextinstr});$
 $B.\text{falselist} = \text{makelist}(\text{nextinstr} + 1);$
 $\text{append}('if E_1.\text{addr relop } E_2.\text{addr goto_}');$
 $\text{append}('goto_');\}$
- vi. $B \rightarrow \text{true} \{B.\text{truelist} = \text{makelist}(\text{nextinstr});$
 $\text{append}('goto_');\}$
- vii. $B \rightarrow \text{false} \{B.\text{falselist} = \text{makelist}(\text{nextinstr});$
 $\text{append}('goto_');\}$
- viii. $M \rightarrow e \{M.\text{instr} = \text{nextinstr};\}$

Three address code :

100 : if $P < Q$ goto_
 101 : goto 102
 102 : if $R < S$ goto 104
 103 : goto_
 104 : if $T < U$ goto_
 105 : goto_

Que 3.23. Explain translation scheme for boolean expression.

Answer

Translation scheme for boolean expression can be understood by following example.

Consider the boolean expression generated by the following grammar :

$E \rightarrow E \text{ OR } E$
 $E \rightarrow E \text{ AND } E$

$E \rightarrow \text{NOT } E$

$E \rightarrow (E)$

$E \rightarrow id \text{ relop } id$

$E \rightarrow \text{TRUE}$

$E \rightarrow \text{FALSE}$

Here the relop is denoted by $\leq, \geq, \neq, <, >$. The OR and AND are left associate. The highest precedence is NOT then AND and lastly OR.

The translation scheme for boolean expressions having numerical representation is as given below :

Production rule	Semantic rule
$E \rightarrow E_1 \text{ OR } E_2$	{ $E.\text{place} := \text{newtemp}()$ append($E.\text{place} := E_1.\text{place} \text{ 'OR' } E_2.\text{place}$) }
$E \rightarrow E_1 \text{ AND } E_2$	{ $E.\text{place} := \text{newtemp}()$ append($E.\text{place} := E_1.\text{place} \text{ 'AND' } E_2.\text{place}$) }
$E \rightarrow \text{NOT } E_1$	{ $E.\text{place} := \text{newtemp}()$ append($E.\text{place} := \text{'NOT' } E_1.\text{place}$) }
$E \rightarrow (E_1)$	{ $E.\text{place} := E_1.\text{place}$ }
$E \rightarrow id_1 \text{ relop } id_2$	{ $E.\text{place} := \text{newtemp}()$ append('if $id_1.\text{place}$ relop.op $id_2.\text{place}$ 'goto' next_state + 3); append($E.\text{place} := \text{'0'}$); append('goto' next_state + 2); append($E.\text{place} := \text{'1'}$) }
$E \rightarrow \text{TRUE}$	{ $E.\text{place} := \text{newtemp}()$; append($E.\text{place} := \text{'1'}$) }
$E \rightarrow \text{FALSE}$	{ $E.\text{place} := \text{newtemp}()$ append($E.\text{place} := \text{'0'}$) }

Que 3.24. Write SDD to produce three-address code for Boolean expressions and obtain the three-address code for the statement given below :

```
while  $a < b$  do
  if  $c < d$  then
     $x = y * z$ 
  else
```

```
 $x = y + z$ 
```

AKTU 2022-23, Marks 10

Answer

SDD for Boolean expression : Refer Q. 3.23, Page 3-24C, Unit-3.

Three address code :

1. if $(a < b)$ goto (3)
2. goto (9)
3. if $(c < d)$ goto (7)
4. goto (5)
5. $t_1 = y + z$
6. $x = t_1$
7. $t_2 = y * z$
8. $x = t_2$
9. goto

Que 3.25. What is back patching. Generate three address code for the following Boolean expression using back patching :

$a < b$ or $c > d$ and $e < f$

AKTU 2022-23, Marks 10

Answer

Back patching : Refer Q. 3.22, Page 3-22C, Unit-3.

Numerical :

Three address code

```
100 : if  $a < b$  goto _
101 : goto 102
102 : if  $c < d$  goto 104
103 : goto _
104 : if  $e < f$  goto _
105 : goto _
```

PART-6

Postfix Translation : Array References in Arithmetic Expressions.

Que 3.26. Write a short note on postfix translation.

Answer

1. In a production $A \rightarrow \alpha$, the translation rule of A.CODE consists of the concatenation of the CODE translations of the non-terminals in α in the same order as the non-terminals appear in α .
2. Production can be factored to achieve postfix form.

Postfix translation of while statement :

Production : $S \rightarrow \text{while } M1 \ E \ \text{do } M2 \ S1$

Can be factored as :

1. $S \rightarrow C \ S1$
2. $C \rightarrow W \ E \ \text{do}$
3. $W \rightarrow \text{while}$

A suitable transition scheme is given as :

Production rule	Semantic action
$W \rightarrow \text{while}$	$W.QUAD = \text{NEXTQUAD}$
$C \rightarrow W \ E \ \text{do}$	$C \ W \ E \ \text{do}$
$S \rightarrow C \ S1$	$\text{BACKPATCH}(S1.NEXT,$ $C.QUAD)$ $S.NEXT = C.FALSE$ $\text{GEN}(\text{goto } C.QUAD)$

Que 3.27. What is postfix notations ? Translate $(C + D) * (E + Y)$ into postfix using Syntax Directed Translation Scheme (SDTS).

Answer

Postfix notation : Refer Q. 3.7, Page 3-7C, Unit-3.

Numerical : Syntax directed translation scheme to specify the translation of an expression into postfix notation are as follow :

Production :

$$E \rightarrow E_1 + T$$

$$E_1 \rightarrow T$$

$$T \rightarrow T_1 \times F$$

$$T_1 \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

Schemes :

$$E.code = E_1.code \parallel T_1.code \parallel '+'$$

$$E_1.code = T.code$$

$$T_1.code = T_1.code \parallel F.code \parallel 'x'$$

$$T_1.code = F.code$$

$$F.code = E.code$$

$$F.code = id.code$$

where '||' sign is used for concatenation.

Que 3.28. What do you mean by attributed grammars ? Discuss the translation scheme for converting an infix expression to its equivalent postfix form.

AKTU 2022-23, Marks 10

Answer

Attributed grammars :

- i. Attributed 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.
- ii. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.
- iii. Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.
- iv. Attribute grammar can pass values or information among the nodes of a tree.

Infix expression to its equivalent postfix form : Following is the grammar :

$$E \rightarrow E + T \{E.val = E1.val + T.val; E.code = E1.code \parallel T.code \parallel "ADD \backslash n";\}$$

$$| E - T \{E.val = E1.val - T.val; E.code = E1.code \parallel T.code \parallel "SUB \backslash n";\}$$

$$| T \{E.val = T.val; E.code = T.code;\}$$

$$T \rightarrow T * F \{T.val = T1.val * F.val; T.code = T1.code \parallel F.code \parallel "MUL \backslash n";\}$$


```

| T / F {T.val = T1.val / F.val; T.code = T1.code || F.code || "DIV\n";}
| F    {T.val = F.val; T.code = F.code;}
F -> ( E ) {F.val = E.val; F.code = E.code;}

| num  {F.val = num.val; F.code = "PUSH " || num.val || "\n";}
    
```

1. Here, the attribute 'val' represents the value of the expression, and 'code' represents the three-address code that produces the value of the expression.
2. We use synthesized attributes to propagate the values and code from the right-hand side of a production rule to the left-hand side.
3. To convert an infix expression to postfix form, we can use a stack to store the operators and operands as we scan the input from left to right.
4. We can use the following algorithm :
 - i. Initialize an empty stack.
 - ii. For each token in the input :
 - a. If the token is a number, output it.
 - b. If the token is an operator, pop all operators with higher or equal precedence from the stack and output them. Then push the current operator onto the stack.
 - c. If the token is a left parenthesis, push it onto the stack.
 - d. If the token is a right parenthesis, pop all operators from the stack until the matching left parenthesis is found. Output each operator as it is popped from the stack.
 - iii. Pop any remaining operators from the stack and output them.

PART-7

Procedures Call.

Que 3.29. Explain procedure call with example.

Answer

Procedures call :

1. Procedure is an important and frequently used programming construct for a compiler.
2. It is used to generate code for procedure calls and returns.

3. Queue is used to store the list of parameters in the procedure call.
4. The translation for a call includes a sequence of actions taken on entry and exit from each procedure. Following actions take place in a calling sequence :
 - a. When a procedure call occurs then space is allocated for activation record.
 - b. Evaluate the argument of the called procedure.
 - c. Establish the environment pointers to enable the called procedure to access data in enclosing blocks.
 - d. Save the state of the calling procedure so that it can resume execution after the call.
 - e. Also save the return address. It is the address of the location to which the called routine must transfer after it is finished.
 - f. Finally generate a jump to the beginning of the code for the called procedure.

For example : Let us consider a grammar for a simple procedure call statement :

1. $S \rightarrow \text{call } id(\text{Elist})$
2. $\text{Elist} \rightarrow \text{Elist}, E$
3. $\text{Elist} \rightarrow E$

A suitable transition scheme for procedure call would be :

Production rule	Semantic action
$S \rightarrow \text{call } id(\text{Elist})$	for each item p on QUEUE do GEN (param p) GEN (call $id.PLACE$)
$\text{Elist} \rightarrow \text{Elist}, E$	append $E.PLACE$ to the end of QUEUE
$\text{Elist} \rightarrow E$	initialize QUEUE to contain only $E.PLACE$

Que 3.30. Explain the concept of array references in arithmetic expressions.

Answer

1. An array is a collection of elements of similar data type. Here, we assume the static allocation of array, whose subscripts ranges from one to some limit known at compile time.
2. If width of each array element is ' w ' then the i^{th} element of array A begins in location,

$$\text{base} + (i - \text{low}) * d$$

where low is the lower bound on the subscript and base is the relative address of the storage allocated for an array i.e., base is the relative address of A[low].

3. A two dimensional array is normally stored in one of two forms, either row-major (row by row) or column-major (column by column).
4. The Fig. 3.30.1 for row-major and column-major are given as:

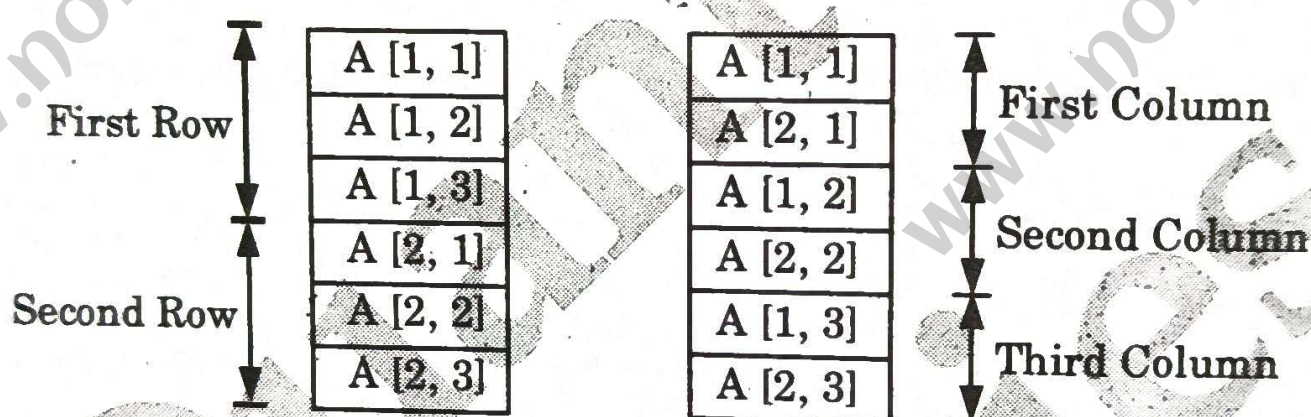


Fig. 3.22.1.

5. In case of a two dimensional array stored in row-major form, the relative address of $A[i_1, i_2]$ can be calculated by formula,

$$(\text{base} + (i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * w$$

where low_1 and low_2 are lower bounds on the values of i_1 and i_2 and n_2 is the number of values that i_2 can take.

6. That is, if high_2 is the upper bound on the value of i_2 then $n_2 = [\text{high}_2 - \text{low}_2 + 1]$.
7. Assuming that i_1 and i_2 are only values that are not known at compile time, we can rewrite above expression as :

$$((i_1 * n_2) + i_2) * w + (\text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w)$$

8. The generalize form of row-major will be,

$$(((...((i_1 n_2 + i_2) n_3 + i_3)...) n_k + I_i) * w + \text{base} - (((...(\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3)...) n_k + \text{low}_k) * w$$

Que 3.31. Explain how syntax-directed translation handles array references within arithmetic expressions.

AKTU 2023-24, Marks 10

Answer

1. Syntax-directed translation (SDT) uses semantic rules attached to grammar productions to translate high-level language constructs into intermediate code.
2. For array references within arithmetic expressions, SDT manages the translation by associating attributes with non-terminal symbols in the grammar.
3. These attributes typically include :
 - i. **Index computation** : When an array element is referenced, the index expression is evaluated first. Semantic rules compute the index's value, ensuring it is within bounds.
 - ii. **Address calculation** : The base address of the array is combined with the computed index to derive the effective memory address of the array element.
 - iii. **Intermediate code generation** : SDT generates intermediate code for the arithmetic expression, including the computed address.
4. By systematically applying these steps, SDT ensures accurate and efficient translation of array references within arithmetic expressions.

PART-8

Declarations Statements.

Que 3.32. Explain declarative statements with example.

Answer

In the declarative statements the data items along with their data types are declared.

For example :

$S \rightarrow D$	{offset:= 0}
$D \rightarrow id : T$	{enter_tab(<i>id.name</i> , <i>T.type</i> ,offset); offset:= offset + <i>T.width</i>)}
$T \rightarrow \text{integer}$	{ <i>T.type</i> := integer; <i>T.width</i> := 8}
$T \rightarrow \text{real}$	{ <i>T.type</i> := real; <i>T.width</i> := 8}
$T \rightarrow \text{array[num] of } T_1$	{ <i>T.type</i> := array(num.val, <i>T₁.type</i>) <i>T.width</i> := num.val $\times$ <i>T₁.width</i> }
$T \rightarrow *T_1$	{ <i>T.type</i> := pointer(<i>T.type</i>) <i>T.width</i> := 4}

- Initially, the value of offset is set to zero. The computation of offset can be done by using the formula $\text{offset} = \text{offset} + \text{width}$.
- In the above translation scheme, *T.type*, *T.width* are the synthesized attributes. The type indicates the data type of corresponding identifier and width is used to indicate the memory units associated with an identifier of corresponding type. For instance integer has width 4 and real has 8.
- The rule $D \rightarrow id : T$ is a declarative statements for *id* declaration. The enter_tab is a function used for creating the symbol table entry for identifier along with its type and offset.
- The width of array is obtained by multiplying the width of each element by number of elements in the array.
- The width of pointer types of supposed to be 4.

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. Define syntax directed translation. Construct an annotated parse tree for the expression $(4 * 7 + 1) * 2$, using the simple desk calculator grammar.

Ans. Refer Q. 3.1.

Q. 2. Explain attributes. What are synthesized and inherited attribute ?

Ans. Refer Q. 3.3.

Q. 3. Consider the following grammar and give the syntax directed definition to construct parse tree for the input expression $4*7 + 3*9$. Also construct an annotated parse tree according to your syntax directed definition.

$E \rightarrow E + T \mid T, T \rightarrow T * F \mid F, F \rightarrow \text{num}.$

Ans. Refer Q. 3.5.

Q. 4. Define parse tree. Why parse tree construction is only possible for CFG ?

Ans. Refer Q. 3.8.

Q. 5. Explain three address code with examples.

Ans. Refer Q. 3.11.

Q. 6. What are different ways to write three address code ?

Ans. Refer Q. 3.12.

Q. 7. Translate the following arithmetic expression into quadruples and triples :

i. $x = y * z + y * -z$

ii. $a = -b * (c + d) + b$

Ans. Refer Q. 3.15.

Q. 8. Define backpatching and semantic rules for boolean expression. Derive the three address code for the following expression : $P < Q$ or $R < S$ and $T < U$.

Ans. Refer Q. 3.22.

Q. 9. Explain translation scheme for boolean expression.

Ans. Refer Q. 3.23.

Q. 10. Write SDD to produce three-address code for Boolean expressions and obtain the three-address code for the statement given below :

while $a < b$ **do**

if $c < d$ **then**

$x = y * z$

else

$x = y + z$

Ans. Refer Q. 3.24.



4

UNIT

Symbol Tables

CONTENTS

- Part-1** : Symbol Tables : 4-2C to 4-7C
Data Structure for
Symbol Tables
- Part-2** : Representing Scope Information 4-7C to 4-10C
- Part-3** : Run-Time Administration : 4-10C to 4-17C
Implementation of Simple Stack
Allocation Scheme
- Part-4** : Storage Allocation in 4-17C to 4-18C
Block Structured Language
- Part-5** : Error Detection and Recovery : 4-18C to 4-24C
Lexical Phase Errors
Syntactic Phase Errors
Semantic Errors

PART-1*Symbol Tables : Data Structure for Symbol Tables.*

Que 4.1. Discuss symbol table with its capabilities ?

Answer

1. A symbol table is a data structure used by a compiler to keep track of scope, life and binding information about names.
2. These information are used in the source program to identify the various program elements, like variables, constants, procedures, and the labels of statements.
3. A symbol table must have the following capabilities :
 - a. **Lookup** : To determine whether a given name is in the table.
 - b. **Insert** : To add a new name (a new entry) to the table.
 - c. **Access** : To access the information related with the given name.
 - d. **Modify** : To add new information about a known name.
 - e. **Delete** : To delete a name or group of names from the table.

Que 4.2. What are the symbol table requirements ? What are the demerits in the uniform structure of symbol table ?

Answer

The basic requirements of a symbol table are as follows :

1. **Structural flexibility** : Based on the usage of identifier, the symbol table entries must contain all the necessary information.
2. **Fast lookup/search** : The table lookup/search depends on the implementation of the symbol table and the speed of the search should be as fast as possible.
3. **Efficient utilization of space** : The symbol table must be able to grow or shrink dynamically for an efficient usage of space.
4. **Ability to handle language characteristics** : The characteristic of a language such as scoping and implicit declaration needs to be handled.

Demerits in uniform structure of symbol table :

1. The uniform structure cannot handle a name whose length exceed upper bound or limit or name field.
2. If the length of a name is small, then the remaining space is wasted.

Que 4.3. How names can be looked up in the symbol table ?

Answer

1. The symbol table is searched (looked up) every time a name is encountered in the source text.
2. When a new name or new information about an existing name is discovered, the content of the symbol table changes.
3. Therefore, a symbol table must have an efficient mechanism for accessing the information held in the table as well as for adding new entries to the symbol table.
4. In any case, the symbol table is a useful abstraction to aid the compiler to ascertain and verify the semantics, or meaning of a piece of code.
5. It makes the compiler more efficient, since the file does not need to be re-parsed to discover previously processed information.

For example : Consider the following outline of a C function :

```
void scopes ( )
```

```
{
  int a, b, c;                /* level 1 */
  ....
```

```
    {
      int a, b;                /* level 2 */
      ....
```

```
    }
    {
      float c, d;              /* level 3 */
      {
```

```
        int m;                /* level 4 */
        ....
      }
    }
  }
}
```

The symbol table could be represented by an upwards growing stack as :

- i. Initially the symbol table is empty.

--	--

- ii. After the first three declarations, the symbol table will be

c	int
b	int
a	int

- iii. After the second declaration of Level 2.

b	int
a	int
c	int
b	int
a	int

iv. As the control come out from Level 2.

c	int
b	int
a	int

v. When control will enter into Level 3.

d	float
e	float
c	int
b	int
a	int

vi. After entering into Level 4.

m	int
d	float
c	float
e	int
b	int
a	int

vii. On leaving the control from Level 4.

d	float
e	float
c	int
b	int
a	int

viii. On leaving the control from Level 3.

c	int
b	int
a	int

ix. On leaving the function entirely, the symbol table will be again empty.

--	--

Que 4.4.

What is the role of symbol table ? Discuss different data structures used for symbol table.

OR

Discuss the various data structures used for symbol table with suitable example.

OR

Define symbol table ? Explain about the data structures used for symbol table.

OR

What is symbol table ? Explain various data structures used for symbol table.

AKTU 2022-23, 2023-24; Marks 10

AKTU 2020-21, Marks 10

Answer**Role of symbol table :**

1. It keeps the track of semantics of variables.
2. It stores information about scope.
3. It helps to achieve compile time efficiency.

Different data structures used in implementing symbol table are :**1. Unordered list :**

- a. Simple to implement symbol table.
- b. It is implemented as an array or a linked list.
- c. Linked list can grow dynamically that eliminate the problem of a fixed size array.
- d. Insertion of variable take $O(1)$ time , but lookup is slow for large tables *i.e.*, $O(n)$.

2. Ordered list :

- a. If an array is sorted, it can be searched using binary search in $O(\log_2 n)$.
- b. Insertion into a sorted array is expensive that it takes $O(n)$ time on average.
- c. Ordered list is useful when set of names is known *i.e.*, table of reserved words.

3. Search tree :

- a. Search tree operation and lookup is done in logarithmic time.
- b. Search tree is balanced by using algorithm of AVL and Red-black tree.

4. Hash tables and hash functions :

- a. Hash table translate the elements in the fixed range of value called hash value and this value is used by hash function.
- b. Hash table can be used to minimize the movement of elements in the symbol table.
- c. The hash function helps in uniform distribution of names in symbol table.

For example : Consider a part of C program

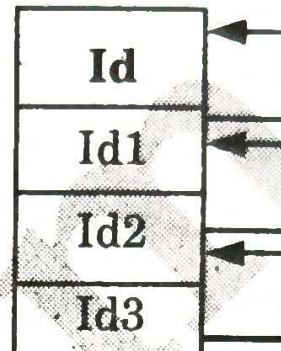
```
int x, y;  
msg ( );
```


1. Unordered list :

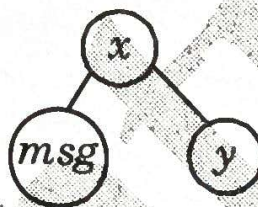
S. No.	Name	Type
1	<i>x</i>	int
2	<i>msg</i>	function
3	<i>y</i>	int

2. Ordered list :

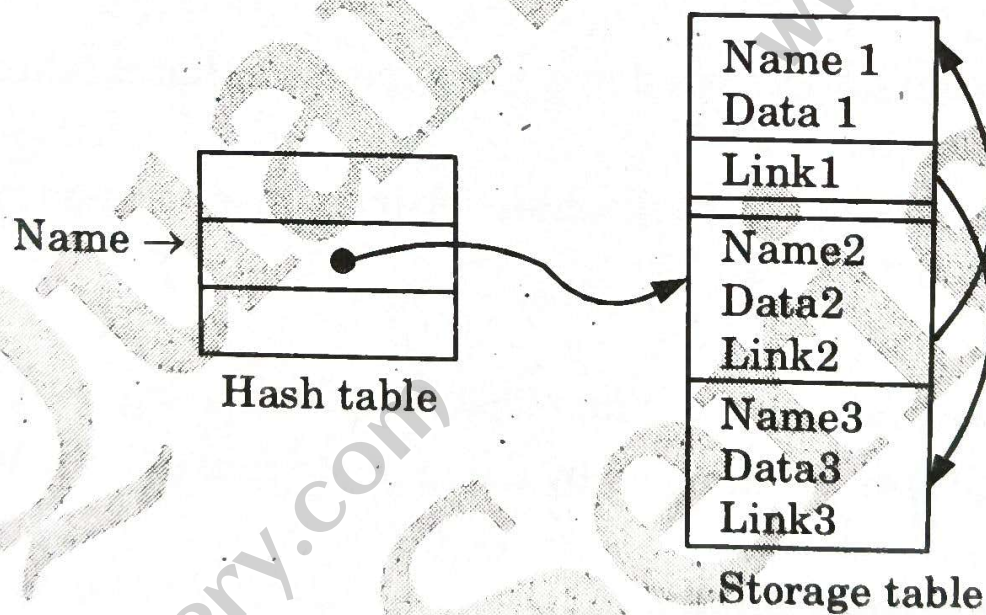
Id	Name	Type
Id1	<i>x</i>	int
Id2	<i>y</i>	int
Id3	<i>msg</i>	function



3. Search tree :



4. Hash table :



Que 4.5.

Describe symbol table and its entries. Also, discuss various data structure used for symbol table.

Answer

Symbol table : Refer Q. 4.1, Page 4-2C, Unit-4.

Entries in the symbol table are as follows :

1. **Variables :**

- Variables are identifiers whose value may change between executions and during a single execution of a program.
- They represent the contents of some memory location.
- The symbol table needs to record both the variable name as well as its allocated storage space at runtime.

2. Constants :

- Constants are identifiers that represent a fixed value that can never be changed.
- Unlike variables or procedures, no runtime location needs to be stored for constants.
- These are typically placed right into the code stream by the compiler at compilation time.

3. Types (user defined) :

- A user defined type is combination of one or more existing types.
- Types are accessed by name and reference a type definition structure.

4. Classes :

- Classes are abstract data types which restrict access to its members and provide convenient language level polymorphism.
- This includes the location of the default constructor and destructor, and the address of the virtual function table.

5. Records :

- Records represent a collection of possibly heterogeneous members which can be accessed by name.
- The symbol table probably needs to record each of the record's members.

Various data structure used for symbol table : Refer Q. 4.4, Page 4-4C, Unit-4.

PART-2

Representing Scope Information.

Que 4.6.

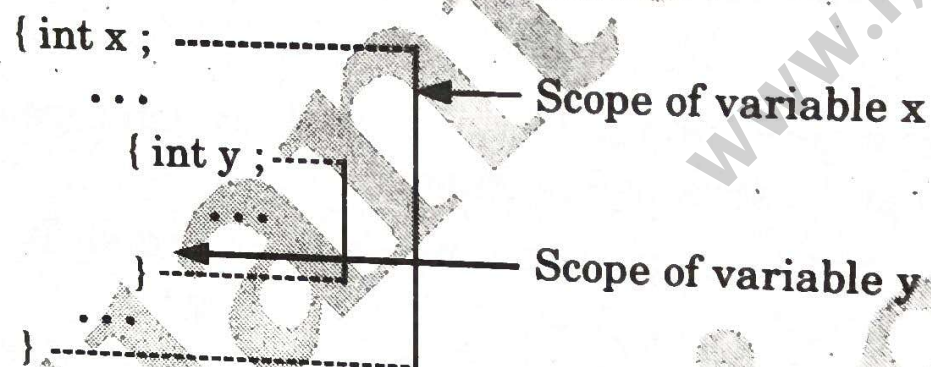
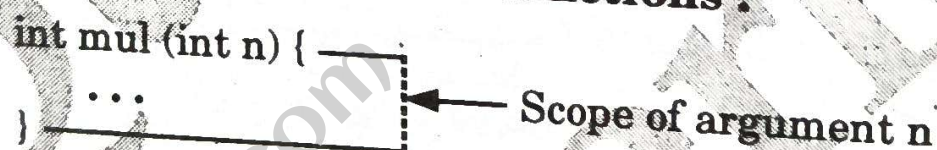
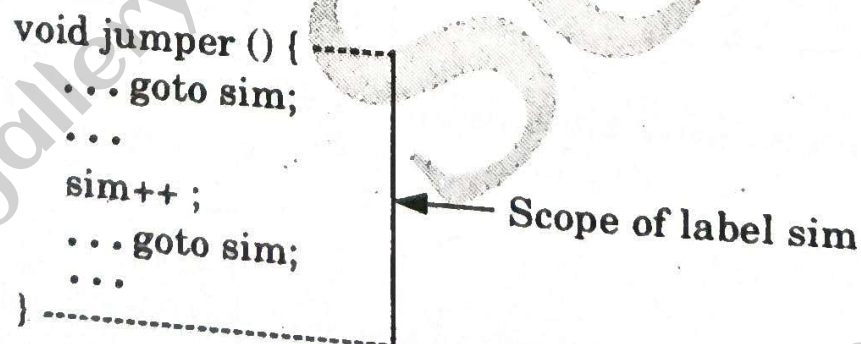
Discuss how the scope information is represented in a symbol table.

Answer

- Scope information characterizes the declaration of identifiers and the portions of the program where it is allowed to use each identifier.
- Different languages have different scopes for declarations. For example, in FORTRAN, the scope of a name is a single subroutine, whereas in ALGOL, the scope of a name is the section or procedure in which it is declared.
- Thus, the same identifier may be declared several times as distinct names, with different attributes, and with different intended storage locations.

4-8 C (CS-Sem-6)

4. The symbol table is thus responsible for keeping different declaration of the same identifier distinct.
5. To make distinction among the declarations, a unique number is assigned to each program element that in return may have its own local data.
6. Semantic rules associated with productions that can recognize the beginning and ending of a subprogram are used to compute the number of currently active subprograms.
7. There are mainly two semantic rules regarding the scope of an identifier :
 - a. Each identifier can only be used within its scope.
 - b. Two or more identifiers with same name and are of same kind cannot be declared within the same lexical scope.
8. The scope declaration of variables, functions, labels and objects within a program is shown below :

Scope of variables in statement blocks :**Scope of formal arguments of functions :****Scope of labels :****Que 4.7.****Write a short note on scoping.****AKTU 2020-21, Marks 05****Answer**

1. Scoping is method of keeping variables in different parts of program distinct from one another.
2. Scoping is generally divided into two classes :

- a. **Static scoping :** Static scoping is also called lexical scoping. In this scoping a variable always refers to its top level environment.
- b. **Dynamic scoping :** In dynamic scoping, a global identifier refers to the identifier associated with the most recent environment.

Que 4.8. | Differentiate between lexical (or static) scope and dynamic scope.

Answer

S. No.	Lexical scope	Dynamic scope
1.	The binding of name occurrences to declarations is done statistically at compile time.	The binding of name occurrences to declarations is done dynamically at run-time.
2.	The structure of the program defines the binding of variables.	The binding of variables is defined by the flow of control at the run time.
3.	A free variable in a procedure gets its value from the environment in which the procedure is defined.	A free variable gets its value from where the procedure is called.

Que 4.9. | Distinguish between static scope and dynamic scope. Briefly explain access to non-local names in static scope.

AKTU 2021-22, Marks 10

Answer

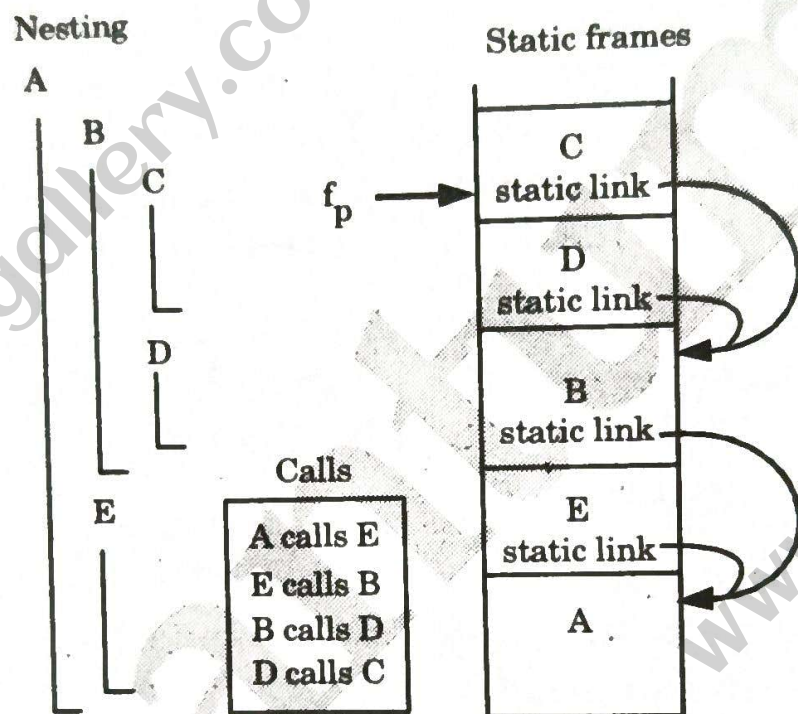
Difference : Refer Q. 4.8, Page 4-9C, Unit-4.

Access to non-local names in static scope :

1. Static chain is the mechanism to implement non-local names (variable) access in static scope.
2. A static chain is a chain of static links that connects certain activation record instances in the stack.
3. The static link, static scope pointer, in an activation record instance for subprogram A points to one of the activation record instances of A's static parent.

4-10 C (CS-Sem-6)

4. When a subroutine at nesting level j has a reference to an object declared in a static parent at the surrounding scope nested at level k , then $j-k$ static links forms a static chain that is traversed to get to the frame containing the object.
5. The compiler generates code to make these traversals over frames to reach non-local names.
- For example :** Subroutine A is at nesting level 1 and C at nesting level 3. When C accesses an object of A, 2 static links are traversed to get to A's frame that contains that object

**PART-3**

Run-Time Administration : Implementation of Simple Stack, Allocation Scheme.

Que 4.10. Draw the format of activation record in stack allocation and explain each field in it.

OR

What is an activation record ? Draw diagram of general activation record and explain the purpose of different fields of an activation record.

AKTU 2022-23, Marks 10

Answer

1. Activation record is used to manage the information needed by a single execution of a procedure.
2. An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.

Format of activation records in stack allocation :

Return value
Actual parameters
Control link
Access link
Saved machine status
Local data
Temporaries

Fields of activation record are :

1. **Return value** : It is used by calling procedure to return a value to calling procedure.
2. **Actual parameter** : It is used by calling procedures to supply parameters to the called procedures.
3. **Control link** : It points to activation record of the caller.
4. **Access link** : It is used to refer to non-local data held in other activation records.
5. **Saved machine status** : It holds the information about status of machine before the procedure is called.
6. **Local data** : It holds the data that is local to the execution of the procedure.
7. **Temporaries** : It stores the value that arises in the evaluation of an expression.

Que 4.11. How to sub-divide a run-time memory into code and data areas ? Explain.

OR

Discuss the stack allocation and heap allocation strategies of the runtime environment with an example.

AKTU 2022-23, Marks 10

Answer

Sub-division of run-time memory into codes and data areas is shown in Fig. 4.11.1.

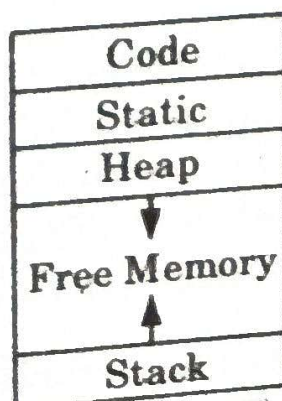


Fig. 4.11.1.

1. **Code** : It stores the executable target code which is of fixed size and do not change during compilation.

2. **Static allocation** :

- The static allocation is for all the data objects at compile time.
- The size of the data objects is known at compile time.
- The names of these objects are bound to storage at compile time only and such an allocation of data objects is done by static allocation.
- In static allocation, the compiler can determine amount of storage required by each data object. Therefore, it becomes easy for a compiler to find the address of these data in the activation record.
- At compile time, compiler can fill the addresses at which the target code can find the data on which it operates.

3. **Heap allocation** : There are two methods used for heap management :

a. **Garbage collection method** :

- When all access path to a object are destroyed but data object continue to exist, such type of objects are said to be garbaged.
- The garbage collection is a technique which is used to reuse that object space.
- In garbage collection, all the elements whose garbage collection bit is 'on' are garbaged and returned to the free space list.

b. **Reference counter** :

- Reference counter attempt to reclaim each element of heap storage immediately after it can no longer be accessed.
- Each memory cell on the heap has a reference counter associated with it that contains a count of number of values that point to it.
- The count is incremented each time a new value point to the cell and decremented each time a value ceases to point to it.

4. **Stack allocation** :

- Stack allocation is used to store data structure called activation record.
- The activation records are pushed and popped as activations begins and ends respectively.
- Storage for the locals in each call of the procedure is contained in the activation record for that call. Thus, locals are bound to fresh storage in each activation, because a new activation record is pushed onto the stack when call is made.
- These values of locals are deleted when the activation ends.

Que 4.12. Why run-time storage management is required ? How simple stack implementation is implemented ?

Answer

Run-time storage management is required because :

1. A program needs memory resources to execute instructions.
2. The storage management must connect to the data objects of programs.
3. It takes care of memory allocation and deallocation while the program is being executed.

Simple stack implementation is implemented as :

1. In stack allocation strategy, the storage is organized as stack. This stack is also called control stack.
2. As activation begins the activation records are pushed onto the stack and on completion of this activation the corresponding activation records can be popped.
3. The locals are stored in the each activation record. Hence, locals are bound to corresponding activation record on each fresh activation.
4. The data structures can be created dynamically for stack allocation.

Que 4.13. Discuss the following parameter passing techniques with suitable example.

- i. Call by name
- ii. Call by reference

OR

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

Answer

i. **Call by name :**

1. In call by name, the actual parameters are substituted for formals in all the places where formals occur in the procedure.
2. It is also referred as lazy evaluation because evaluation is done on parameters only when needed.

For example :

```
main () {
    int n1 = 10; n2 = 20;
    printf("n1: %d, n2: %d\n", n1, n2);
    swap(n1, n2);
    printf("n1: %d, n2: %d\n", n1, n2);
    swap(int c, int d) {
        int t;
        t = c;
```



```

c = d;
d = t;
printf("n1: %d, n2: %d\n", n1, n2);
}

```

Output : 10 20

20 10

20 10

ii. Call by reference :

1. In call by reference, the location (address) of actual arguments is passed to formal arguments of the called function. This means by accessing the addresses of actual arguments we can alter them within the called function.
2. In call by reference, alteration to actual arguments is possible within called function; therefore the code must handle arguments carefully else we get unexpected results.

For example :

```
#include <stdio.h>
```

```
void swapByReference(int*, int*); /* Prototype */
```

```
int main() /* Main function */
```

```
{
```

```
int n1 = 10; n2 = 20;
```

```
/* actual arguments will be altered */
```

```
swapByReference(&n1, &n2);
```

```
printf("n1: %d, n2: %d\n", n1, n2);
```

```
}
```

```
void swapByReference(int *a, int *b)
```

```
{
```

```
int t;
```

```
t = *a; *a = *b; *b = t;
```

```
}
```

Output : n1: 20, n2: 10

Que 4.14.

Differentiate between stack allocation and heap allocation.

AKTU 2021-22, Marks 10

Answer

S.No.	Static allocation	Heap allocation
1.	Static allocation allocates memory on the basis of size of data objects.	Heap allocation makes use of heap for managing the allocation of memory at run time.
2.	In static allocation, there is no possibility of creation of dynamic data structures and objects.	In heap allocation, dynamic data structures and objects are created.
3.	In static allocation, the names of the data objects are fixed with storage for addressing.	Heap allocation allocates contiguous block of memory to data objects.
4.	Static allocation is simple, but not efficient memory management technique.	Heap allocation does memory management in efficient way.
5.	Static allocation strategy is faster in accessing data as compared to heap allocation.	Heap allocation is slow in accessing as there is chance of creation of holes in reusing the free space.
6.	It is easy to implement.	It is comparatively expensive to implement.

Que 4.15. Describe a simple stack allocation scheme for managing memory during program execution. Explain how it is used for storing local variables and managing function calls.

AKTU 2023-24, Marks 10

Answer

Stack allocation scheme :

1. A simple stack allocation scheme for managing memory during program execution involves using a stack data structure to allocate and deallocate memory dynamically.
2. In this scheme, memory is allocated from the top of the stack, and deallocated by simply adjusting the stack pointer.

Stack allocation scheme can be used for :

1. Storing local variables :

- a. When a function is called, space for its local variables is allocated on the stack.
- b. Each local variable is assigned a memory location within this allocated stack frame.
- c. As the function executes, it accesses its local variables relative to the base pointer of stack frame or stack pointer.
- d. When the function exits, the stack frame is deallocated by adjusting the stack pointer, freeing up memory for reuse.

2. Managing function calls :

- a. When a function is called, its parameters, return address, and sometimes other information are pushed onto the stack.
- b. This information forms the function's activation record or stack frame.
- c. The function's code then executes using the allocated stack frame for accessing parameters and storing local variables.
- d. Upon function completion, the stack frame is deallocated by popping the stack, restoring the previous stack state.

Que 4.16. How do we represent the scope information ? Explain scope by number and scope by location. **AKTU 2022-23, Marks 10**

Answer

Representation of scope information : Refer Q.4.6, Page 4-7C, Unit-4

Scope by number :

1. It stores all values in one Symbol Table.
2. The same name can be declared many times as a distinct name in different blocks or procedures.
3. We can recognize the particular identifier in a procedure by matching not only the name of the identifier but also match the procedure number to which it belongs.
4. Each block or procedure will be assigned a number.

Scope by location :

1. It creates a separate table for each scope block or procedure.

2. Multiple identifiers with the same name can easily be inserted because the same name variable will be shared in a separate block.
3. We cannot have two variables with a similar name in the same block or procedure.

PART-4

Storage Allocation in Block Structured Language.

Que 4.17. Explain symbol table organization using hash tables. With an example show the symbol table organization for block structured language.

Answer

1. Hashing is an important technique used to search the records of symbol table. This method is superior to list organization.
2. In hashing scheme, a hash table and symbol table are maintained.
3. The hash table consists of k entries from 0, 1 to $k - 1$. These entries are basically pointers to symbol table pointing to the names of symbol table.
4. To determine whether the 'Name' is in symbol table, we used a hash function ' h ' such that $h(\text{name})$ will result any integer between 0 to $k - 1$. We can search any name by position = $h(\text{name})$.
5. Using this position, we can obtain the exact locations of name in symbol table.
6. The hash table and symbol table are shown in Fig. 4.17.1.

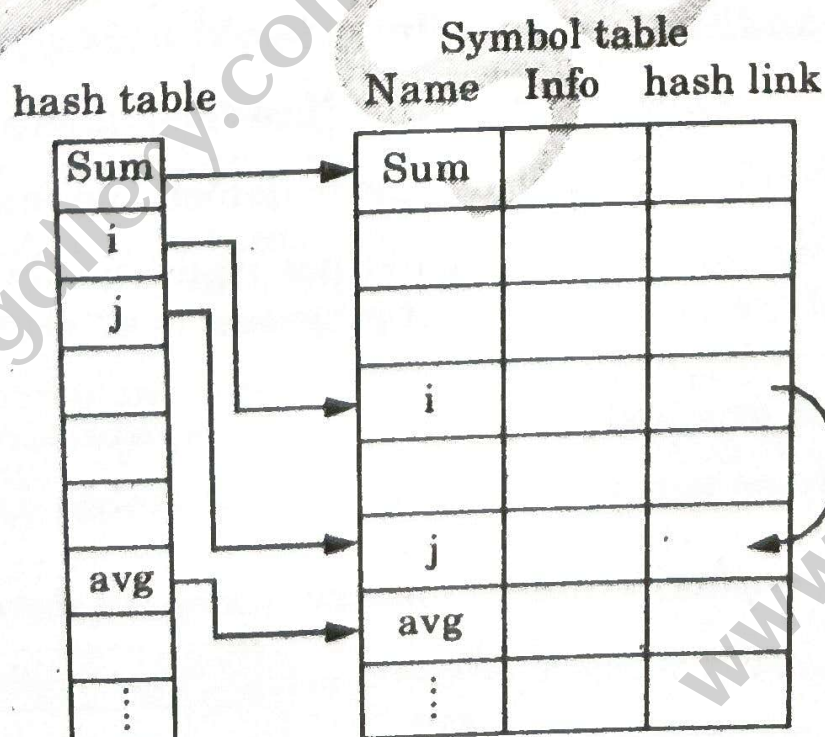


Fig. 4.17.1.

7. The hash function should result in uniform distribution of names in symbol table.
8. The hash function should have minimum number of collision.

PART-5

Error Detection and Recovery : Lexical Phase Errors, Syntactic Phase Errors, Semantic Errors.

Que 4.18. Define error recovery. What are the properties of error message ? Discuss the goals of error handling.

Answer

Error recovery : Error recovery is an important feature of any compiler, through which compiler can read and execute the complete program even it have some errors.

Properties of error message are as follows :

1. Message should report the errors in original source program rather than in terms of some internal representation of source program.
2. Error message should not be complicated.
3. Error message should be very specific and should fix the errors at correct positions.
4. There should be no duplicacy of error messages, i.e., same error should not be reported again and again.

Goals of error handling are as follows :

1. Detect the presence of errors and produce "meaningful" diagnostics.
2. To recover quickly enough to be able to detect subsequent errors.
3. Error handling components should not significantly slow down the compilation of syntactically correct programs.

Que 4.19. What are lexical phase errors, syntactic phase errors and semantic phase errors ? Explain with suitable example.

OR

Explain lexical, syntax, semantic phase errors in detail.

AKTU 2020-21, Marks 10

OR

What do you understand by lexical phase error and syntactic error? Also suggest methods for recovery of errors.

AKTU 2021-22, Marks 10

OR

Explain the lexical analysis and syntax analysis phases of the compiler with a suitable example. Explain the reporting errors in these two phases as well.

AKTU 2022-23, Marks 10

Answer

1. Lexical phase error :

- a. A lexical phase error is a sequence of character that does not match the pattern of token *i.e.*, while scanning the source program, the compiler may not generate a valid token from the source program.
- b. Reasons due to which errors are found in lexical phase are :
 - i. The addition of an extraneous character.
 - ii. The removal of character that should be presented.
 - iii. The replacement of a character with an incorrect character.
 - iv. The transposition of two characters.

For example :

- i. In Fortran, an identifier with more than 7 characters long is a lexical error.
- ii. In Pascal program, the character ~, & and @ if occurred is a lexical error.

2. Syntactic phase errors (syntax error) :

- a. Syntactic errors are those errors which occur due to the mistake done by the programmer during coding process.
- b. Reasons due to which errors are found in syntactic phase are :
 - i. Missing of semicolon
 - ii. Unbalanced parenthesis and punctuation

For example : Let us consider the following piece of code :

```
int x;
```

```
int y //Syntax error
```

In example, syntactic error occurred because of absence of semicolon.

3. Semantic phase errors :

- a. Semantic phase errors are those errors which occur in declaration and scope in a program.
- b. Reason due to which errors are found :
 - i. Undeclared names
 - ii. Type incompatibilities
 - iii. Mismatching of actual arguments with the formal arguments.

For example : Let us consider the following piece of code :

```
scanf("%f%f", a, b);
```

In example, a and b are semantic error because `scanf` uses address of the variables as `&a` and `&b`.

4. Logical errors :

- a. Logical errors are the logical mistakes founded in the program which is not handled by the compiler.
- b. In these types of errors, program is syntactically correct but does not operate as desired.

For example :

Let consider following piece of code :

```
x = 4;
```

```
y = 5;
```

```
average = x + y/2;
```

The given code do not give the average of x and y because BODMAS property is not used properly.

Que 4.20. What do you understand by lexical error and syntactic error ? Also, suggest methods for recovery of errors.

OR

Explain logical phase error and syntactic phase error. Also suggest methods for recovery of error.

Answer

Lexical and syntactic error : Refer Q. 4.19, Page 4-18C, Unit-4.

Various error recovery methods are :

1. Panic mode recovery :

- a. This is the simplest method to implement and used by most of the parsing methods.

- b. When parser detect an error, the parser discards the input symbols one at a time until one of the designated set of synchronizing token is found.
- c. Panic mode correction often skips a considerable amount of input without checking it for additional errors. It gives guarantee not to go in infinite loop.

For example :

Let consider a piece of code :

$a = b + c;$

$d = e + f;$

By using panic mode it skips $a = b + c$ without checking the error in the code.

2. Phrase-level recovery :

- a. When parser detects an error the parser may perform local correction on remaining input.
- b. It may replace a prefix of the remaining input by some string that allows parser to continue.
- c. A typical local correction would replace a comma by a semicolon, delete an extraneous semicolon or insert a missing semicolon.

For example :

Let consider a piece of code

$\text{while } (x > 0) \quad y = a + b;$

In this code local correction is done by phrase-level recovery by adding 'do' and parsing is continued.

- 3. **Error production :** If error production is used by the parser, we can generate appropriate error message and parsing is continued.

For example :

Let consider a grammar

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

$A \rightarrow E$

When error production encounters $* A$, it sends an error message to the user asking to use $**$ as unary or not.

4. Global correction :

- a. Global correction is a theoretical concept.
- b. This method increases time and space requirement during parsing.

Que 4.21. Explain in detail the error recovery process in operator precedence parsing method.

Answer

Error recovery in operator precedence parsing :

1. There are two points in the parsing process at which an operator-precedence parser can discover syntactic error :
 - a. If no precedence relation holds between the terminal on top of the stack and the current input.
 - b. If a handle has been found, but there is no production with this handle as a right side.
2. The error checker does the following errors :
 - a. Missing operand
 - b. Missing operator
 - c. No expression between parentheses
 - d. These error diagnostic issued at handling of errors during reduction.
3. During handling of shift/reduce errors, the diagnostic's issues are :
 - a. Missing operand
 - b. Unbalanced right parenthesis
 - c. Missing right parenthesis
 - d. Missing operators

Que 4.22. Explain the function of error handling phase of a compiler.

AKTU 2020-21, Marks 05

Answer

1. The tasks of the error handling process are to detect each error, report it to the user, and then make some recovery strategy and implement them to handle the error.
2. An error is the blank entries in the symbol table.
3. Errors in the program should be detected and reported by the parser.

Functions of error handler :**1. Error detection :**

- a. Viable-prefix is the property of a parser that allows early detection of syntax errors.
- b. Detection of an error as soon as possible without further consuming unnecessary input.
- c. Detect an error as soon as the prefix of the input does not match a prefix of any string in the language.

For example : for(;), this will report an error as for having two semicolons inside braces.

2. Error recovery : Refer Q. 4.20, Page 4-19C, Unit-4.

Que 4.23. Define semantic errors in a compiler. Discuss the challenges associated with detecting and handling semantic errors. Provide example to illustrate semantic issues.

AKTU 2023-24, Marks 10

Answer

Semantic errors : Semantic errors in a compiler are logical errors in the code that occur despite syntactically correct statements. They involve incorrect meaning or usage, such as type mismatches, undeclared variables, and incorrect function calls.

Challenges associated with detecting and handling semantic errors :

1. **Complexity of detection :** Semantic errors involve logic and meaning, making them harder to identify than syntax errors.
2. **Context dependency :** Errors depend on specific application context and data, requiring deep understanding of the domain.
3. **Testing limitations :** Traditional testing may miss logical flaws that only occur in specific scenarios.
4. **Dynamic behavior :** Errors may emerge during runtime, complicating detection and debugging.
5. **Data quality :** Inconsistent or incorrect data can lead to semantic errors.

Example to illustrate semantic issue :

1. In a payroll system, mistakenly using gross salary instead of net salary for tax calculations is a semantic error.
2. The code runs without errors, but the logic is flawed.
3. Detecting and handling such errors requires thorough domain knowledge, comprehensive testing, and rigorous validation techniques.

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. Discuss symbol table with its capabilities ?

Ans. Refer Q. 4.1.

Q. 2. What are the symbol table requirements ? What are the demerits in the uniform structure of symbol table ?

Ans. Refer Q. 4.2.

Q. 3. What is the role of symbol table ? Discuss different data structures used for symbol table.

Ans. Refer Q. 4.4.

Q. 4. Describe symbol table and its entries. Also, discuss various data structure used for symbol table.

Ans. Refer Q. 4.5.

Q. 5. Draw the format of activation record in stack allocation and explain each field in it.

Ans. Refer Q. 4.10.

Q. 6. How to sub-divide a run-time memory into code and data areas ? Explain.

Ans. Refer Q. 4.11.

Q. 7. How do we represent the scope information ? Explain scope by number and scope by location.

Ans. Refer Q. 4.16.

Q. 8. Explain symbol table organization using hash tables. With an example show the symbol table organization for block structured language.

Ans. Refer Q. 4.17.

Q. 9. What are lexical phase errors, syntactic phase errors and semantic phase errors ? Explain with suitable example.

Ans. Refer Q. 4.19.

Q. 10. What do you understand by lexical error and syntactic error ? Also, suggest methods for recovery of errors.

Ans. Refer Q. 4.20.



5

UNIT

Code Generation

CONTENTS

-
- Part-1** : Code Generation : 5-2C to 5-4C
Design Issues
- Part-2** : The Target Language 5-4C to 5-5C
Address in Target Code
- Part-3** : Basic Blocks and Flow Graphs 5-5C to 5-10C
Optimization of Basic Blocks
Code Generator
- Part-4** : Machine Independent 5-11C to 5-19C
Optimizations
Loop Optimization
- Part-5** : DAG Representation 5-20C to 5-25C
of Basic Blocks
- Part-6** : Value Numbers and 5-25C to 5-27C
Algebraic Laws
Global Data Flow Analysis

PART-1*Code Generation : Design Issues.*

Que 5.1. What is code generation ? Discuss the design issues of code generation.

Answer

1. Code generation is the final phase of compiler.
2. It takes as input the Intermediate Representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program as shown in Fig. 5.1.1.

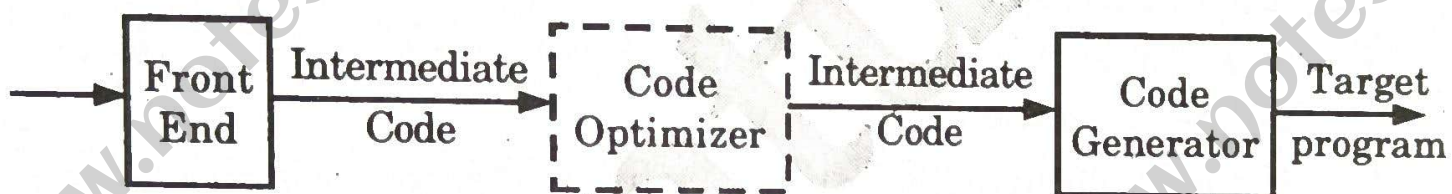


Fig. 5.1.1. Position of code generator.

Design issues of code generator are :

1. **Input to the code generator :**
 - a. The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table.
 - b. IR includes three address representations and graphical representations.
2. **The target program :**
 - a. The instruction set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high quality machine code.
 - b. The most common target machine architectures are RISC (Reduced Instruction Set Computer), CISC (Complex Instruction Set Computer), and stack based.
3. **Instruction selection :**
 - a. The code generator must map the IR program into a code sequence that can be executed by the target machine.
 - b. If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates.

4. Register allocation :

- a. A key problem in code generation is deciding what values to hold in which registers on the target machine do not have enough space to hold all values.
- b. Values that are not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.
- c. The use of registers is often subdivided into two subproblems :
 - i. Register allocation, during which we select the set of variables that will reside in registers at each point in the program.
 - ii. Register assignment, during which we pick the specific register that a variable will reside in.

5. Evaluation order :

- a. The order in which computations are performed can affect the efficiency of the target code.
- b. Some computation orders require fewer registers to hold intermediate results than others.

Que 5.2. Explain the role of a code generator in a compiler. Discuss its responsibilities and how it translates intermediate code into the target code.

AKTU 2023-24, Marks 10

Answer

Role of code generator in a compiler :

1. It translates the intermediate code into a low-level machine language that can be directly executed by the machine.
2. It must ensure that the generated code is correct, efficient, optimised, and compatible with the target architecture of the machine.
3. It also manages system resources such as memory and registers.

Responsibilities of code generator are :

1. **Intermediate code translation :** Translate intermediate code into target machine instructions or another form of executable code.
2. **Optimization :** Apply optimization techniques to improve code efficiency and performance, such as removing redundant computations and rearranging instructions.
3. **Register allocation :** Assign program variables and intermediate results to registers or memory locations efficiently.
4. **Instruction selection :** Map high-level operations from intermediate code to suitable machine instructions supported by the target architecture.

The code generator in a compiler translates intermediate code into target code through these steps :

1. **IR analysis** : The compiler parses source code and generates an intermediate representation (IR).
2. **Optimization** : Various optimizations are applied to enhance code efficiency and performance.
3. **Target architecture considerations** : Characteristics of the target hardware are accounted for.
4. **Instruction selection** : High-level language constructs are mapped to machine instructions.
5. **Register allocation** : Efficient usage of processor registers is ensured for speed.
6. **Memory management** : Memory allocation and usage are managed for variables and data.
7. **Control flow handling** : Constructs like loops and conditionals are translated into machine instructions.
8. **Error handling** : Detection and reporting of code generation errors are facilitated.
9. **Code emission** : Translated target code, typically machine code or assembly, is produced.

PART-2

The Target Language, Address in Target Code.

Que 5.3. Discuss addresses in the target code.

Answer

1. Addresses in the target code show how names in the IR can be converted into addresses in the target code by looking at code generation for simple procedure calls and returns using static and stack allocation.
2. Addresses in the target code represent executing program runs in its own logical address space that was partitioned into four code and data areas :
 - a. A statically determined area code that holds the executable target code. The size of the target code can be determined at compile time.
 - b. A statically determined data area static for holding global constants and other data generated by the compiler. The size of the global constants and compiler data can also be determined at compile time.

- c. A dynamically managed area heap for holding data objects that are allocated and freed during program execution. The size of the heap cannot be determined at compile time.
- d. A dynamically managed area stack for holding activation records as they are created and destroyed during procedure calls and returns. Like the heap, the size of the stack cannot be determined at compile time.

PART-3

Basic Blocks and Flow Graphs, Optimization of Basic Blocks, Code Generator.

Que 5.4. Write an algorithm to partition a sequence of three address statements into basic blocks.

Answer

The algorithm for construction of basic block is as follows :

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, the first statement of basic block. The rules we use are given as :
 - a. The first statement is a leader.
 - b. Any statement which is the target of a conditional or unconditional goto is a leader.
 - c. Any statement which immediately follows a conditional goto is a leader.
2. For each leader construct its basic block, which consist of leader and all statements up to the end of program but not including the next leader. Any statement not placed in the block can never be executed and may now be removed, if desired.

Que 5.5. Explain flow graph with example.

Answer

1. A flow graph is a directed graph in which the flow control information is added to the basic blocks.
2. The nodes to the flow graph are represented by basic blocks.
3. The block whose leader is the first statement is called initial blocks.

4. There is a directed edge from block B_{i-1} to block B_i if B_i immediately follows B_{i-1} in the given sequence. We can say that B_{i-1} is a predecessor of B_i .

For example : Consider the three address code as :

1. $prod := 0$
2. $i := 1$
3. $t_1 := 4 * i$
4. $t_2 := a[t_1]$ /* computation of $a[i]$ */
5. $t_3 := 4 * i$
6. $t_4 := b[t_3]$ /* computation of $b[i]$ */
7. $t_5 := t_2 * t_4$
8. $t_6 := prod + t_5$
9. $prod := t_6$
10. $t_7 := i + 1$
11. $i := t_7$
12. if $i \leq 10$ goto (3)

The flow graph for the given code can be drawn as follows :

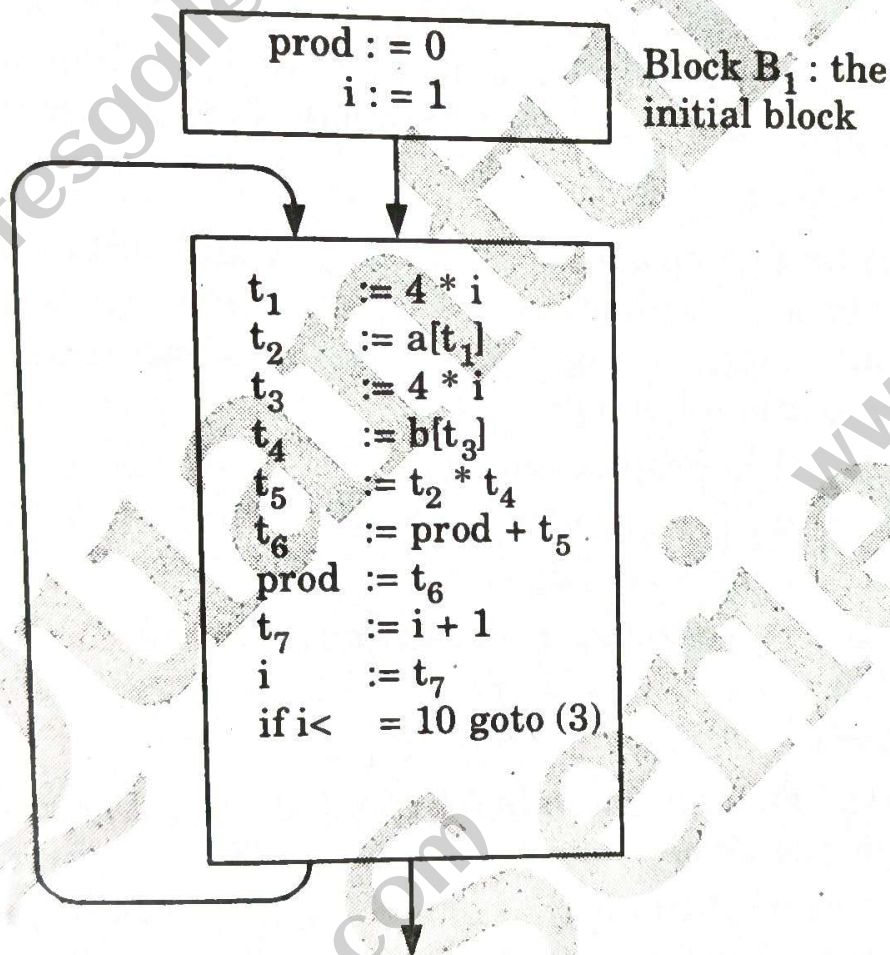


Fig. 5.5.1. Flow graph.

Que 5.6. What is loop ? Explain what constitute a loop in a flow graph.

Answer

Loop is a collection of nodes in the flow graph such that :

1. All such nodes are strongly connected i.e., there is always a path from any node to any other node within that loop.
2. The collection of nodes has unique entry. That means there is only one path from a node outside the loop to the node inside the loop.
3. The loop that contains no other loop is called inner loop.

Following term constitute a loop in flow graph :

1. **Dominators :**

- In control flow graphs, a node d dominates a node n if every path from the entry node to n must go through d . This is denoted as $d \text{ dom } n$.
- By definition, every node dominates itself.
- A node d strictly dominates a node n if d dominates n and d is not equal to n .
- The immediate dominator (or *idom*) of a node n is the unique node that strictly dominates n but does not strictly dominate any other node that strictly dominates n . Every node, except the entry node, has an immediate dominator.
- A dominator tree is a tree where each node's children are those nodes it immediately dominates. Because the immediate dominator is unique, it is a tree. The start node is the root of the tree.

2. **Natural loops :**

- The natural loop can be defined by a back edge $n \rightarrow d$ such that there exists a collection of all the nodes that can reach to n without going through d and at the same time d can also be added to this collection.
- Loop in a flow graph can be denoted by $n \rightarrow d$ such that $d \text{ dom } n$.
- These edges are called back edges and for a loop there can be more than one back edge.
- If there is $p \rightarrow q$ then q is a head and p is a tail and head dominates tail.

3. **Pre-header :**

- The pre-header is a new block created such that successor of this block is the header block.
- All the computations that can be made before the header block can be made before the pre-header block.

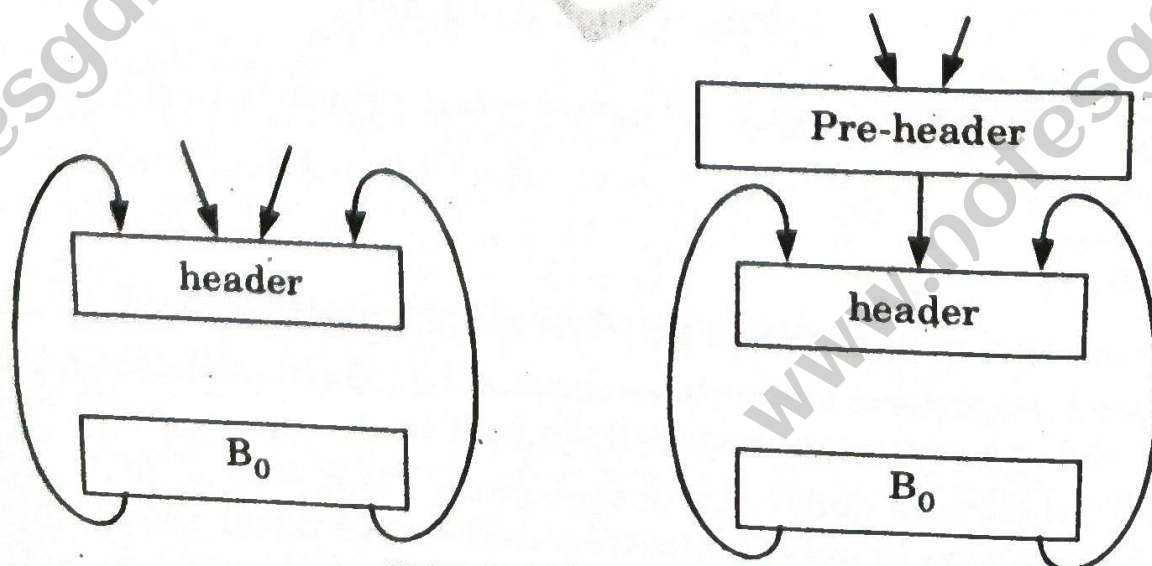


Fig. 5.6.1. Pre-header

4. Reducible flow graph :

- a. A flow graph G is reducible graph if and only if we can partition the edges into two disjointed groups i.e., forward edges and backward edges.
- b. These edges have following properties :
 - i. The forward edge forms an acyclic graph.
 - ii. The backward edges are such edges whose heads dominate their tails.
- c. The program structure in which there is exclusive use of if-then, while-do or goto statements generates a flow graph which is always reducible.

Que 5.7. Discuss in detail the process of optimization of basic blocks. Give an example.

OR

What are different issues in code optimization ? Explain it with proper example.

Answer

Different issues in code optimization are :

1. **Function preserving transformation :** The function preserving transformations are basically divided into following types :

- a. **Common sub-expression elimination :**

- i. A common sub-expression is nothing but the expression which is already computed and the same expression is used again and again in the program.
- ii. If the result of the expression not changed then we eliminate computation of same expression again and again.

For example :

Before common sub-expression elimination :

$a = t * 4 - b + c;$

.....

$m = t * 4 - b + c;$

.....

$n = t * 4 - b + c;$

After common sub-expression elimination :

$temp = t * 4 - b + c;$

$a = temp;$

.....

$m = temp;$

-

 n = temp;
 iii. In given example, the equation $a = t * 4 - b + c$ is occurred most of the times. So it is eliminated by storing the equation into temp variable.

b. Dead code elimination :

- i. Dead code means the code which can be emitted from program and still there will be no change in result.
- ii. A variable is live only when it is used in the program again and again. Otherwise, it is declared as dead, because we cannot use that variable in the program so it is useless.
- iii. The dead code occurred during the program is not introduced intentionally by the programmer.

For example :

Define False = 0

!False = 1

If(!False)

{

.....

.....

}

- iv. If false becomes zero, is guaranteed then code in 'IF' statement will never be executed. So, there is no need to generate or write code for this statement because it is dead code.

c. Copy propagation :

- i. Copy propagation is the concept where we can copy the result of common sub-expression and use it in the program.
- ii. In this technique the value of variable is replaced and computation of an expression is done at the compilation time.

For example :

$pi = 3.14;$

$r = 5;$

$Area = pi * r * r;$

Here at the compilation time the value of pi is replaced by 3.14 and r by 5.

d. Constant folding (compile time evaluation) :

- i. Constant folding is defined as replacement of the value of one constant in an expression by equivalent constant value at the compile time.

- ii. In constant folding all operands in an operation are constant. Original evaluation can also be replaced by result which is also a constant.

For example : $a = 3.14157/2$ can be replaced by $a = 1.570785$ thereby eliminating a division operation.

2. Algebraic simplification :

- a. Peephole optimization is an effective technique for algebraic simplification.

- b. The statements such as

$$x := x + 0$$

or

$$x := x * 1$$

can be eliminated by peephole optimization.

Que 5.8.

Write a short note on transformation of basic blocks.

Answer

Transformation :

1. A number of transformations can be applied to basic block without changing set of expression computed by the block.
2. Transformation helps us in improving quality of code and act as optimizer.
3. There are two important classes as local transformation that can be applied to the basic block :

- a. **Structure preserving transformation :** They are as follows :

- i. **Common sub-expression elimination :** Refer Q. 5.7, Page 5-8C, Unit-5.
- ii. **Dead code elimination :** Refer Q. 5.7, Page 5-8C, Unit-5.
- iii. **Interchange of statement :** Suppose we have a block with the two adjacent statements,

$$\text{temp1} = a + b$$

$$\text{temp2} = m + n$$

Then we can interchange the two statements without affecting the value of the block if and only if neither 'm' nor 'n' is temporary variable temp1 and neither 'a' nor 'b' is temporary variable temp2. From the given statements we can conclude that a normal form basic block allow us for interchanging all the statements if they are possible.

- b. **Algebraic transformation :** Refer Q. 5.7, Page 5-8C, Unit-5.

PART-4*Machine Independent Optimizations, Loop Optimization.*

Que 5.9. What is code optimization ? Discuss the classification of code optimization.

Answer**Code optimization :**

1. The code optimization refers to the techniques used by the compiler to improve the execution efficiency of generated object code.
2. It involves a complex analysis of intermediate code and performs various transformations but every optimizing transformation must also preserve the semantic of the program.

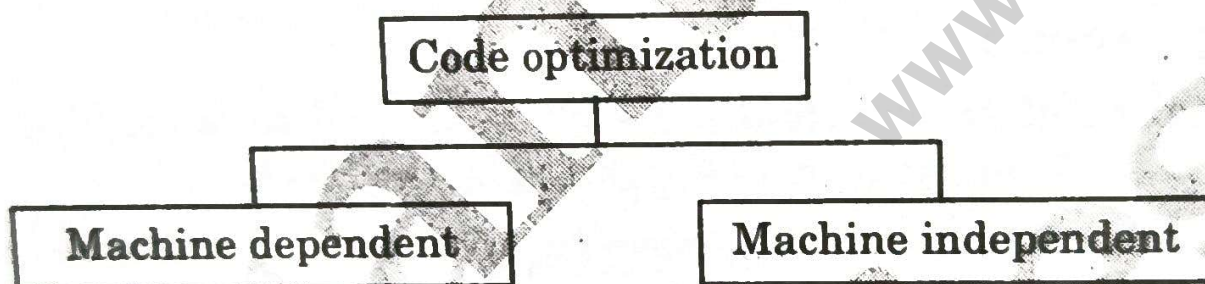
Classification of code optimization :

Fig. 5.9.1. Classification of code optimization.

1. **Machine dependent :** The machine dependent optimization can be achieved using following criteria :
 - a. Allocation of sufficient number of resources to improve the execution efficiency of the program.
 - b. Using immediate instructions wherever necessary.
 - c. The use of intermix instructions along with the data increases the speed of execution.
2. **Machine independent :** The machine independent optimization can be achieved using following criteria :
 - a. The code should be analyzed completely and use alternative equivalent sequence of source code that will produce a minimum amount of target code.
 - b. Use appropriate program structure in order to improve the efficiency of target code.
 - c. By eliminating the unreachable code from the source program.
 - d. Move two or more identical computations at one place and make use of the result instead of each time computing the expressions.

Que 5.10. Explain local optimization.

Answer

1. Local optimization is a kind of optimization in which both the analysis and the transformations are localized to a basic block.
2. The transformations in local optimization are called as local transformations.
3. The name of transformation is usually prefixed with 'local' while referring to the local transformation.
4. There are "local" transformations that can be applied to program to attempt an improvement.

For example : The elimination of common sub-expression, provided A is not an alias for B or C, the assignments :

$$A := B + C + D$$

$$E := B + C + F$$

might be evaluated as

$$T_1 := B + C$$

$$A := T_1 + D$$

$$E := T_1 + F$$

In the given example, $B + C$ is stored in T_1 which act as local optimization of common sub-expression.

Que 5.11. Explain what constitute a loop in a flow graph and how will you do loop optimizations in code optimization of a compiler.

OR

Write a short note on loop optimization.

OR

Explain in detail about loop optimization.

AKTU 2020-21, Marks 10

Answer

Following term constitute a loop in flow graph : Refer Q. 5.6, Page 5-6C, Unit-5.

Loop optimization is a process of increasing execution time and reducing the overhead associated with loops.

The loop optimization is carried out by following methods :

1. Code motion :

- a. Code motion is a technique which moves the code outside the loop.
- b. If some expression in the loop whose result remains unchanged even after executing the loop for several times, then such an expression should be placed just before the loop (i.e., outside the loop).

- c. Code motion is done to reduce the execution time of the program.

2. Induction variables :

- A variable x is called an induction variable of loop L if the value of variable gets changed every time.
- It is either decremented or incremented by some constant.

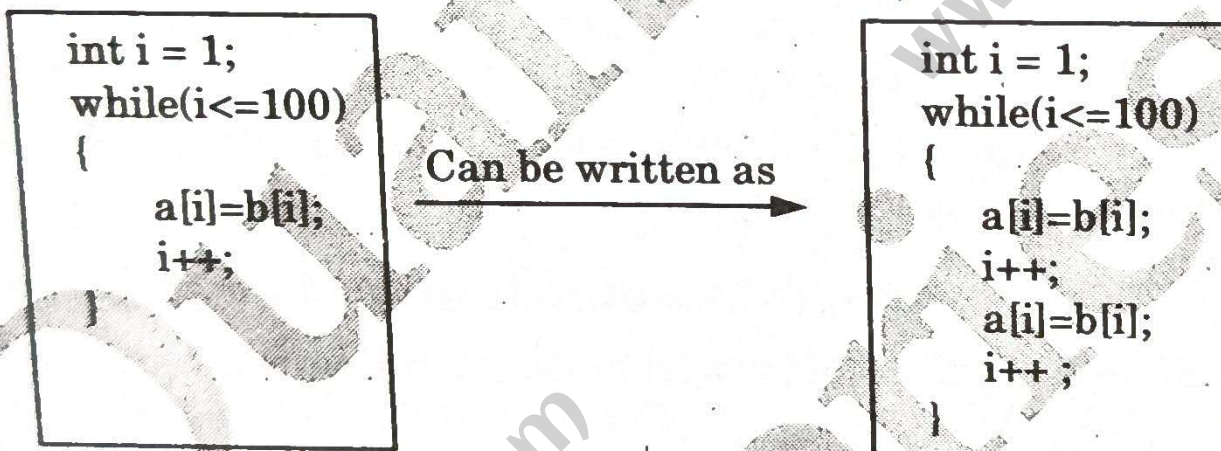
3. Reduction in strength :

- In strength reduction technique the higher strength operators can be replaced by lower strength operators.
- The strength of certain operator is higher than other.
- The strength reduction is not applied to the floating point expressions because it may yield different results.

4. **Loop invariant method :** In loop invariant method, the computation inside the loop is avoided and thereby the computation overhead on compiler is avoided.

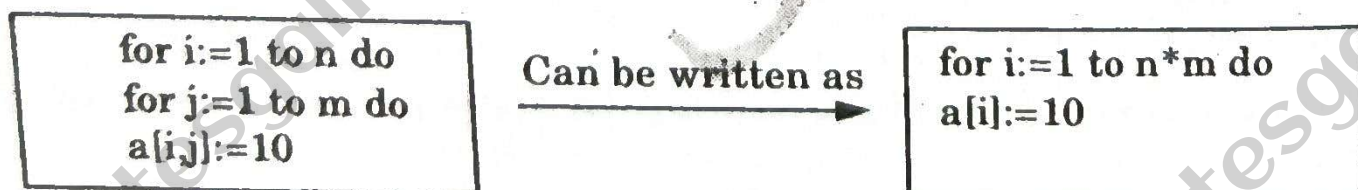
5. **Loop unrolling :** In this method, the number of jumps and tests can be reduced by writing the code two times.

For example :



6. **Loop fusion or loop jamming :** In loop fusion method, several loops are merged to one loop.

For example :



Que 5.12. Consider the following three address code segments :

PROD := 0

- $l := 1$
- $T1 := 4 * l$
- $T2 := \text{addr}(A) - 4$
- $T3 := T2[T1]$
- $T4 := \text{addr}(B) - 4$

6. $T_5 := T_4[T_1]$
7. $T_6 := T_3 * T_5$
8. $PROD := PROD + T_6$
9. $I := I + 1$
10. If $I \leq 20$ goto (3)

- a. Find the basic blocks and flow graph of above sequence.
- b. Optimize the code sequence by applying function preserving transformation optimization technique.

OR

Consider the following sequence of three address codes :

1. $Prod := 0$
 2. $I := 1$
 3. $T_1 := 4 * I$
 4. $T_2 := \text{addr}(A) - 4$
 5. $T_3 := T_2[T_1]$
 6. $T_4 := \text{addr}(B) - 4$
 7. $T_5 := T_4[T_1]$
 8. $T_6 := T_3 * T_5$
 9. $Prod := Prod + T_6$
 10. $I = I + 1$
 11. If $I \leq 20$ goto (3)
- Perform loop optimization.

Answer

- a. Basic blocks and flow graph :

1. As first statement of program is leader statement.
 $\therefore PROD = 0$ is a leader.
2. Fragmented code represented by two blocks is shown below :

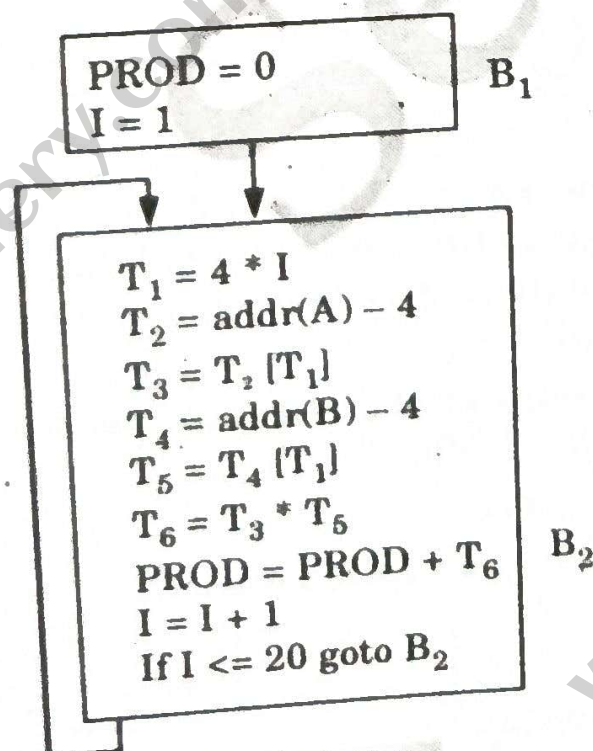


Fig. 5.12.1.

b. Function preserving transformation :

1. **Common sub-expression elimination :** No any block has any sub expression which is used two times. So, no change in flow graphs.
2. **Copy propagation :** No any instruction in the block B_2 is direct assignment *i.e.*, in the form of $x = y$. So, no change in flow graph and basic block.
3. **Dead code elimination :** No any instruction in the block B_2 is dead. So, no change in flow graph and basic block.
4. **Constant folding :** No any constant expression is present in basic block. So, no change in flow graph and basic block.

Loop optimization :

1. **Code motion :** In block B_2 we can see that value of T_2 and T_4 is calculated every time when loop is executed. So, we can move these two instructions outside the loop and put in block B_1 as shown in Fig. 5.12.2.

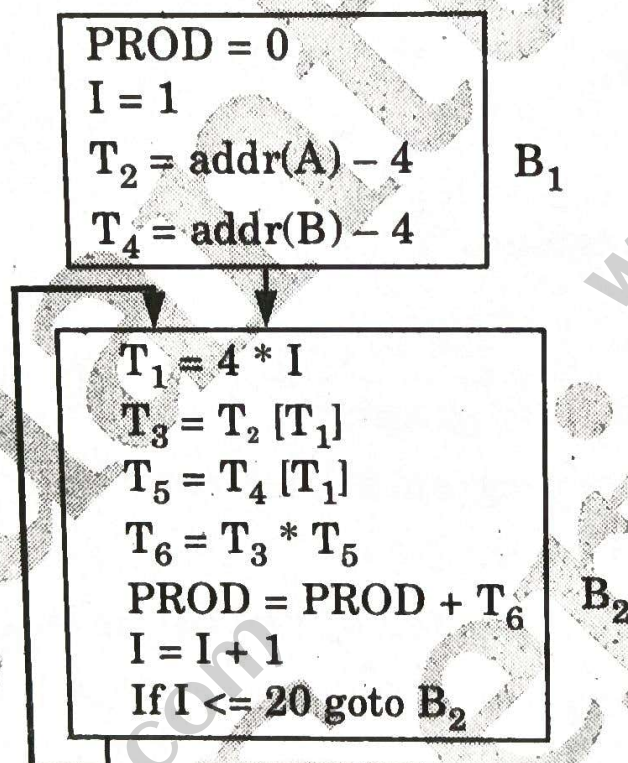


Fig. 5.12.2.

2. **Induction variable :** A variable I and T_1 are called an induction variable of loop L because every time the variable I change the value of T_1 is also change. To remove these variables we use other method that is called reduction in strength.

3. **Reduction in strength :** The values of I varies from 1 to 20 and value T_1 varies from (4, 8, ..., 80).

Block B_2 is now given as

$$T_1 = 4 * I \leftarrow \text{In block } B_1$$

$$T_1 = T_1 + 4 \quad B_2$$

Now final flow graph is given as

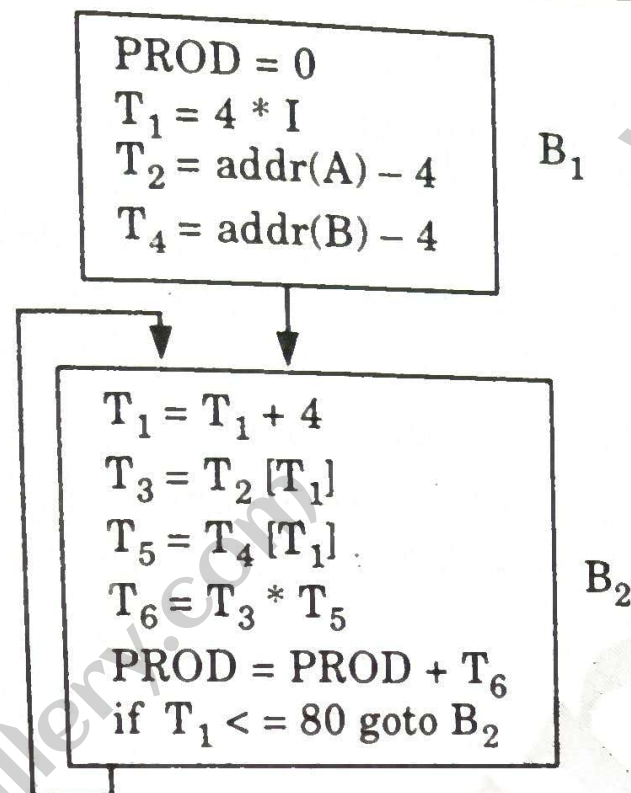


Fig. 5.12.3.

Que 5.13. Write short notes on the following with the help of example :

- i. Loop unrolling
- ii. Loop jamming
- iii. Dominators
- iv. Viable prefix

Answer

- i. **Loop unrolling** : Refer Q. 5.11, Page 5-12C, Unit-5.
- ii. **Loop jamming** : Refer Q. 5.11, Page 5-12C, Unit-5.
- iii. **Dominators** : Refer Q. 5.6, Page 5-6C, Unit-5.

For example : In the flow graph,

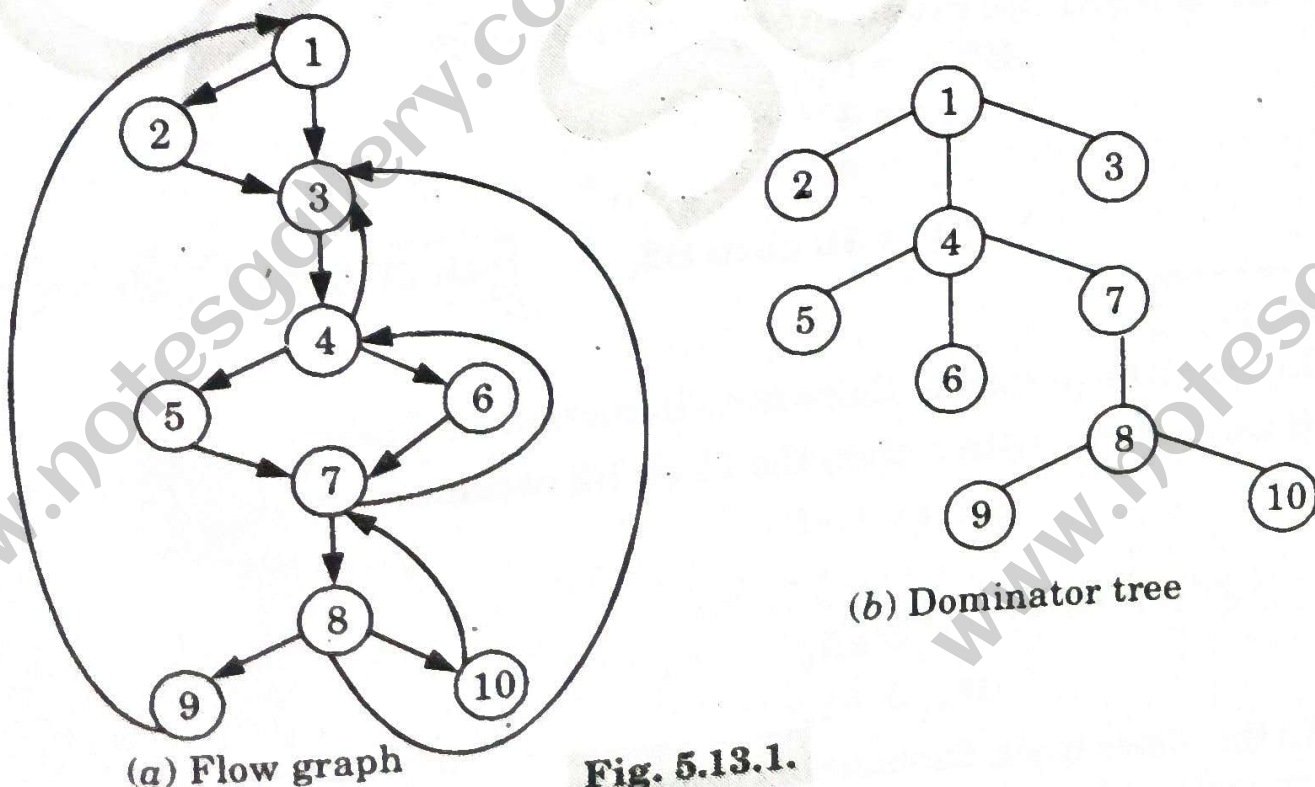


Fig. 5.13.1.

Initial Node, Node 1 dominates every Node.

Node 2 dominates itself. Node 3 dominates all but 1 and 2. Node 4 dominates all but 1, 2 and 3.

Node 5 and 6 dominates only themselves, since flow of control can skip around either by go in through the other. Node 7 dominates 7, 8, 9 and 10. Node 8 dominates 8, 9 and 10.

Node 9 and 10 dominates only themselves.

iv. **Viable prefix** : Viable prefixes are the prefixes of right sentential forms that can appear on the stack of a shift-reduce parser.

For example :

Let : $S \rightarrow x_1 x_2 x_3 x_4$

$A \rightarrow x_1 x_2$

Let $w = x_1 x_2 x_3$

SLR parse trace :

STACK	INPUT
\$	$x_1 x_2 x_3$
$\$ x_1$	$x_2 x_3$
$\$ x_1 x_2$	x_3
$\$ A$	x_3
$\$ A x_3$	\$
⋮	

As we see, $x_1 x_2 x_3$ will never appear on the stack. So, it is not a viable prefix.

Que 5.14. Discuss how induction variables can be detected and eliminated from the given intermediate code

B2: $i = i + 1$
 $t_1 = 4 * j$
 $t_2 = a[t_1]$
 if $t_2 < 10$ goto B2

AKTU 2021-22, Marks 10

Answer

In the given question, there is no induction variable.

If we replace j with i . then the block B2 becomes

B2 : $i = i + 1$
 $t_1 = 4 * i$
 $t_2 = a[t_1]$
 if $t_2 < 10$ goto B2

In the above block B2 value of i is incremented by every time in the loop and value of t_1 is incremented 4 times the value of i .

So, in the given block i and t_1 are two induction variables.

In this case, if we want to remove the induction variables we use reduction is strength technique.

In this technique, we remove same line of code out of the block or modify the computation code in the block.

Modification :

B1 : $t_3 = 4 * i$

B2 : $t_1 = t_3 + 4$

$t_2 = a[t_1]$

if $t_2 < 10$ goto B2

Que 5.15.

Consider a basic block with the following three

instructions :

1. $x = a + b$
2. $y = x * c$
3. $z = y - d$

Apply common sub expression elimination to optimize the basic block.

AKTU 2023-24, Marks 10

Answer

Code before optimization

$x = a + b$

$y = x * c$

$z = y - d$

Code after optimization

$x = a + b$

$y = x * c$

$z = y - d$

Que 5.16.

Construct the flow graph for the following code segment :

fact(n)

{

int f = 1;

for($i = 2; i \leq n; i++$)

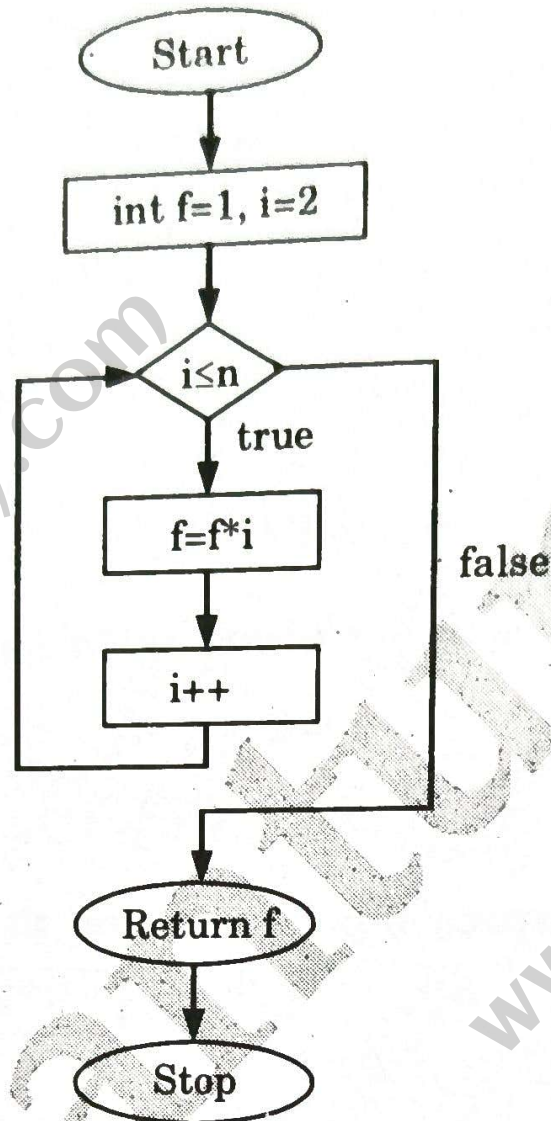
$f = f * i;$

return f;

}

AKTU 2020-21, Marks 10

Answer



Que 5.17. Discuss the following terms :

- i. Basic block
- ii. Next use information
- iii. Flow graph

AKTU 2021-22, Marks 10

Answer

- i. **Basic block :**
 - a. A basic block is a sequence of consecutive instructions with exactly one entry point and one exit point.
 - b. The exit points may be more than one if we consider branch instructions.
- ii. **Next use information :**
 - a. Next-use information is needed for dead-code elimination and register allocation.
 - b. Next-use is computed by a backward scan of a basic block.
 - c. The next-use information will indicate the statement number at which a particular variable that is defined in the current position will be reused.
- iii. **Flow graph :** Refer Q.5.5, Page 5-5C, Unit-5.

PART-5*DAG Representation of Basic Blocks.*

Que 5.18. What is DAG ? What are its advantages in context of optimization ?

OR

Write a short note on direct acyclic graph.

Answer

DAG :

1. The abbreviation DAG stands for Directed Acyclic Graph.
2. DAGs are useful data structure for implementing transformations on basic blocks.
3. A DAG gives picture of how the value computed by each statement in the basic block is used in the subsequent statement of the block.
4. Constructing a DAG from three address statement is a good way of determining common sub-expressions within a block.
5. A DAG for a basic block has following properties :
 - a. Leaves are labeled by unique identifier, either a variable name or constants.
 - b. Interior nodes are labeled by an operator symbol.
 - c. Nodes are also optionally given a sequence of identifiers for labels.
6. Since, DAG is used in code optimization and output of code optimization is machine code and machine code uses register to store variable used in the source program.

Advantage of DAG :

1. We automatically detect common sub-expressions with the help of DAG algorithm.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values which could be used outside the block.

Que 5.19. What is DAG ? How DAG is created from three address code ? Write algorithm for it and explain it with a relevant example.

OR

Explain in the DAG representation of the basic block with example.

AKTU 2022-23, Marks 10

Answer

DAG : Refer Q. 5.18, Page 5-20C, Unit-5.

Algorithm :

Input : A basic block.

Output : A DAG with label for each node (identifier).

Method :

1. Create nodes with one or two left and right children.
2. Create linked list of attached identifiers for each node.
3. Maintain all identifiers for which a node is associated.
4. Node (identifier) represents value that identifier has the current point in DAG construction process. Symbol table store the value of node (identifier).
5. If there is an expression of the form $x = y \text{ op } z$ then DAG contain "op" as a parent node and node(y) as a left child and node(z) as a right child.

For example :

Given expression : $a * (b - c) + (b - c) * d$

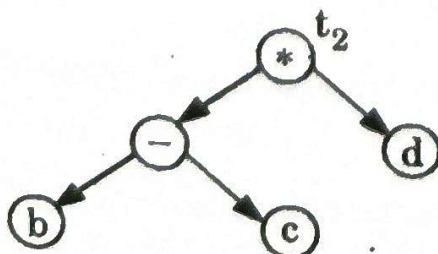
The construction of DAG with three address code will be as follows :

Step 1 :



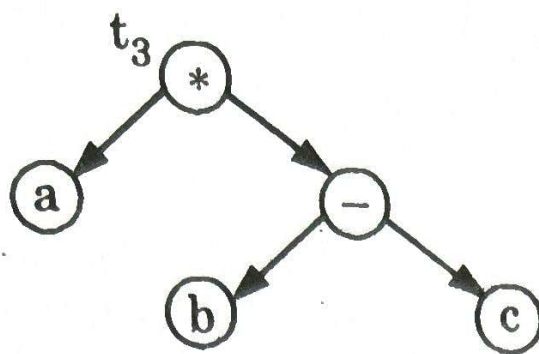
$$t_1 = b - c$$

Step 2 :



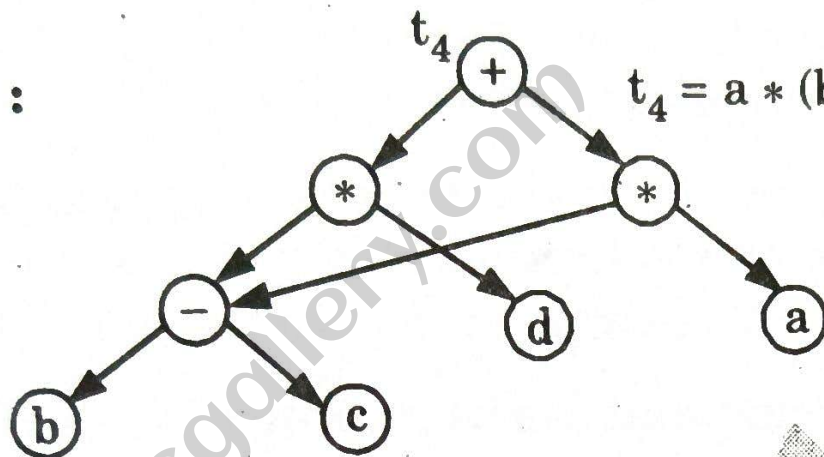
$$t_2 = (b - c) * d$$

Step 3 :



$$t_3 = a * (b - c)$$

Step 4 :



$$t_4 = a * (b - c) + (b - c) * d$$

Que 5.20. How DAG is different from syntax tree ? Construct the

DAG for the following basic blocks :

$a := b + c$

$b := b - d$

$c := c + d$

$e = b + c$

Also, explain the key application of DAG.

Answer

DAG v/s Syntax tree :

1. Directed Acyclic Graph is a data structure for transformations on the basic block. While syntax tree is an abstract representation of the language constructs.
2. DAG is constructed from three address statement while syntax tree is constructed directly from the expression.

DAG for the given code is :

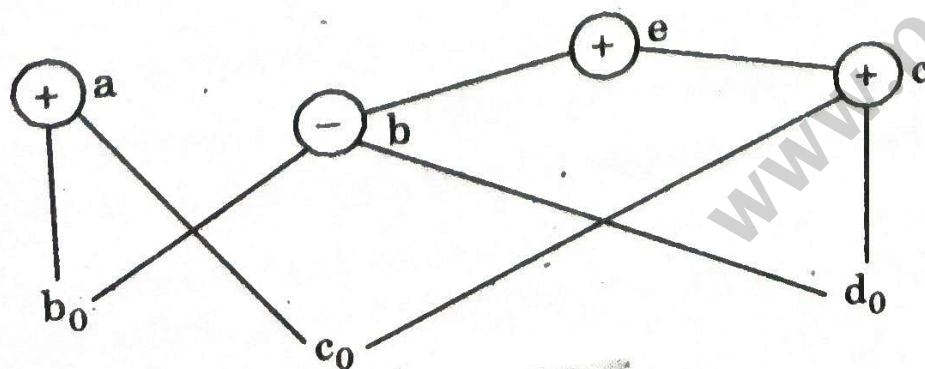


Fig. 5.20.1.

Compiler Design

1. The two occurrences of sub-expressions $b + c$ compute the same value.
2. Value computed by a and e are same.

Applications of DAG :

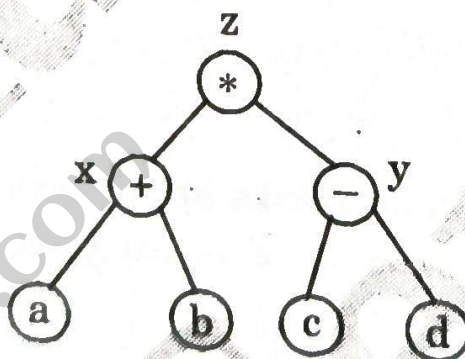
1. **Scheduling** : Directed acyclic graphs representations of partial orderings have many applications in scheduling for systems of tasks.
2. **Data processing networks** : A directed acyclic graph may be used to represent a network of processing elements.
3. **Data compression** : Directed acyclic graphs may also be used as a compact representation of a collection of sequences. In this type of application, one finds a DAG in which the paths form the sequences.
4. It helps in finding statement that can be recorded.

Que 5.21. Construct a Directed Acyclic Graph (DAG) to represent the following basic block :

1. $x = a + b$
2. $y = c - d$
3. $z = x * y$

AKTU 2023-24, Marks 10

Answer



Que 5.22. Define a DAG. Construct a DAG for the expression:

$$p + p * (q - r) + (q - r) * s$$

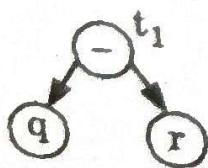
AKTU 2020-21, Marks 10

Answer

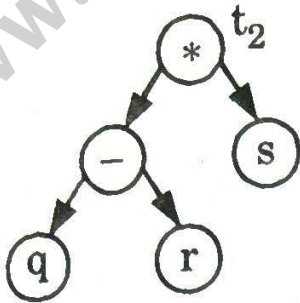
DAG : Refer Q. 5.18, Page 5-20C, Unit-5.

Given expression : $p + q * (q - r) + (q - r) * s$

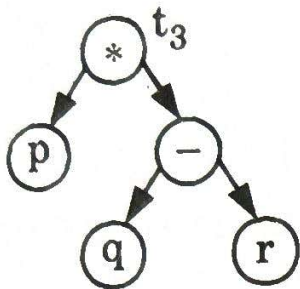
Step 1 :



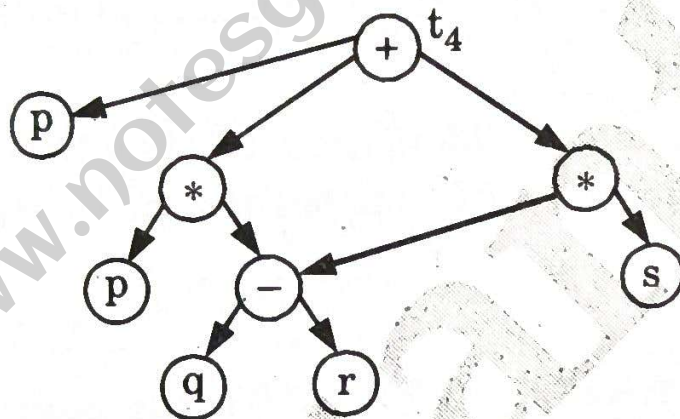
$$t_1 = q - r$$

Step 2:

$$t_2 = (q - r) * s$$

Step 3:

$$t_3 = p * (q - r)$$

Step 4:

$$t_4 = p + p * (q - r) + (q - r) * s$$

Que 5.23. Give the algorithm for the elimination of local and global common sub-expressions algorithm with the help of example.

Answer

Algorithm for elimination of local common sub-expression : DAG algorithm is used to eliminate local common sub-expression.

DAG : Refer Q. 5.18, Page 5-20C, Unit-5.

Algorithm for elimination of global common sub-expressions :

1. An expression is defined at the point where it is assigned a value and killed when one of its operands is subsequently assigned a new value.
2. An expression is available at some point p in a flow graph if every path leading to p contains a prior definition of that expression which is not subsequently killed.
3. Following expressions are used :
 - a. $avail[B]$ = set of expressions available on entry to block B

- b. $\text{exit}[B] = \text{set of expressions available on exit from } B$
- c. $\text{killed}[B] = \text{set of expressions killed in } B$
- d. $\text{defined}[B] = \text{set of expressions defined in } B$
- e. $\text{exit}[B] = \text{avail}[B] - \text{killed}[B] + \text{defined}[B]$

Algorithm :

1. First, compute defined and killed sets for each basic block
2. Iteratively compute the avail and exit sets for each block by running the following algorithm until we get a fixed point:
 - a. Identify each statement s of the form $a = b \text{ op } c$ in some block B such that $b \text{ op } c$ is available at the entry to B and neither b nor c is redefined in B prior to s .
 - b. Follow flow of control backwards in the graph passing back to but not through each block that defines $b \text{ op } c$. the last computation of $b \text{ op } c$ in such a block reaches s .
 - c. After each computation $d = b \text{ op } c$ identified in step 2(a), add statement $t = d$ to that block (where t is a new temp d).
 - d. Replace s by $a = t$

Que 5.24. What are the advantages of DAG ? Explain the peephole optimization.

AKTU 2021-22, Marks 10

Answer

Advantage of DAG: Refer Q. 5.18, Page 5-20C, Unit-5.

Peephole optimization :

1. Peephole optimization is a type of Code Optimization performed on a small part of the code. It is performed on the very small set of instructions in a segment of code.
2. It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without change in output.
3. Peephole is the machine dependent optimization.

PART-6

Value Numbers and Algebraic Laws, Global Data Flow Analysis.

Que 5.25. Write a short note on data flow analysis.

OR

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

Answer

1. Data flow analysis is a process in which the values are computed using data flow properties.
2. In this analysis, the analysis is made on data flow.
3. A program's Control Flow Graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate.
4. A simple way to perform data flow analysis of programs is to set up data flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fix point.
5. Reaching definitions is used by data flow analysis in code optimization.

Reaching definitions :

1. A definition D reaches at point p if there is a path from D to p along which D is not killed.

d1 : $y := 2$ B1



d2 : $x := y + 2$ B2

2. A definition D of variable x is killed when there is a redefinition of x .

d1 : $y := 2$ B1



d2 : $y := y + 2$ B2



d3 : $x := y + 2$ B3

Compiler Design

3. The definition $d1$ is said to a reaching definition for block $B2$. But the definition $d1$ is not a reaching definition in block $B3$, because it is killed by definition $d2$ in block $B2$.

Que 5.26. Write short notes (any two) :

- i. Global data flow analysis
- ii. Loop unrolling
- iii. Loop jamming

OR

Write short note on global data analysis.

Answer

- i. Global data flow analysis :

1. Global data flow analysis collects the information about the entire program and distributed it to each block in the flow graph.
2. Data flow can be collected in various block by setting up and solving a system of equation.
3. A data flow equation is given as :

$$\text{OUT}(s) = \{\text{IN}(s) - \text{KILL}(s)\} \cup \text{GEN}(s)$$

OUT(s) : Definitions that reach exist of block B.

GEN(s) : Definitions within block B that reach the end of B.

IN(s) : Definitions that reaches entry of block B.

KILL(s) : Definitions that never reaches the end of block B.

- ii. Loop unrolling : Refer Q. 5.11, Page 5-12C, Unit-5.
- iii. Loop fusion or loop jamming : Refer Q. 5.11, Page 5-12C, Unit-5.

Que 5.27. Discuss the role of macros in programming language.

Answer

Role of macros in programming language are :

1. It is use to define word that are used most of the time in program.
2. It automates complex task.
3. It helps to reduce the use of complex statement in a program.
4. It makes the program run faster.

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. What is code generation ? Discuss the design issues of code generation.

Ans. Refer Q. 5.1.

Q. 2. Explain flow graph with example.

Ans. Refer Q. 5.5.

Q. 3. What is loop ? Explain what constitute a loop in a flow graph.

Ans. Refer Q. 5.6.

Q. 4. Discuss in detail the process of optimization of basic blocks. Give an example.

Ans. Refer Q. 5.7.

Q. 5. Explain what constitute a loop in a flow graph and how will you do loop optimizations in code optimization of a compiler.

Ans. Refer Q. 5.11.

Q. 6. Consider the following three address code segments :

PROD := 0

1. $l := 1$
2. $T1 := 4 * l$
3. $T2 := \text{addr}(A) - 4$
4. $T3 := T2[T1]$
5. $T4 := \text{addr}(B) - 4$
6. $T5 := T4[T1]$
7. $T6 := T3 * T5$
8. $\text{PROD} := \text{PROD} + T6$
9. $l := l + 1$
10. If $l \leq 20$ goto (3)

- a. Find the basic blocks and flow graph of above sequence.
- b. Optimize the code sequence by applying function preserving transformation optimization technique.

Ans. Refer Q. 5.12.

Q. 7. Discuss how induction variables can be detected and eliminated from the given intermediate code

```
B2: i := i+1
    t1 := 4*j
    0t2 := a[t1]
    if t2 < 10 goto B2
```

Ans. Refer Q. 5.14.

Q. 8. What is DAG ? What are its advantages in context of optimization ?

Ans. Refer Q. 5.18.

Q. 9. What is DAG ? How DAG is created from three address code ? Write algorithm for it and explain it with a relevant example.

Ans. Refer Q. 5.19.



Quantum Series

1

UNIT

Introduction to Compiler (2 Marks Questions)

1.1. State any two reasons as a why phases of compiler should be grouped.

Ans. Two reasons for phases of compiler to be grouped are :

1. It helps in producing compiler for different source language.
2. Front end phases of many compilers are generally same.

1.2. Discuss the challenges in compiler design.

Ans. Challenges in compiler design are :

- i. The compiler should be error free.
- ii. It must generate correct machine code which should run fast.
- iii. Compiler itself must run fast i.e., compilation time must be proportional to program size.
- iv. It must be portable (i.e., modular, supporting separate compilation).
- v. It should give diagnostic and error messages.

1.3. What is translator ?

Ans. A translator takes a program written in a source language as input and converts it into a program in target language as output.

1.4. Differentiate between compiler and assembler.

Ans.

S.No.	Compiler	Assembler
1.	It converts high level language into machine language.	It converts assembly language into machine language.
2.	Debugging is slow.	Debugging is fast.
3.	It is used by C, C++.	It is used by assembly language.

1.5. What do you mean by regular expression ?

Ans. Regular expressions are mathematical symbolisms which describe the set of strings of specific language. It provides convenient and useful notation for representing tokens.

1.6. Differentiate between compilers and interpreters.

Ans.

S. No.	Compiler	Interpreter
1.	It scans all the lines of source program and list out all syntax errors at a time.	It scans one line at a time, if there is any syntax error, the execution of program terminates immediately.
2.	Object produced by compiler gets saved in a file. Hence, file does not need to compile again and again.	Machine code produced by interpreter is not saved in any file. Hence, we need to interpret the file each time.
3.	It takes less time to execute.	It takes more time to execute.

1.7. What is cross compiler ?

Ans. A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.

1.8. How YACC can be used to generate parser ?

Ans.

1. YACC is a tool which will produce a parser for a given grammar.
2. YACC is a program designed to compile a LALR(1) grammar and produce the source code. Hence, it is used to generate a parser.

1.9. Write regular expression to describe a language consist of strings made of even number *a* and *b*.

Ans. Language of DFA which accept even number of *a* and *b* is given by :

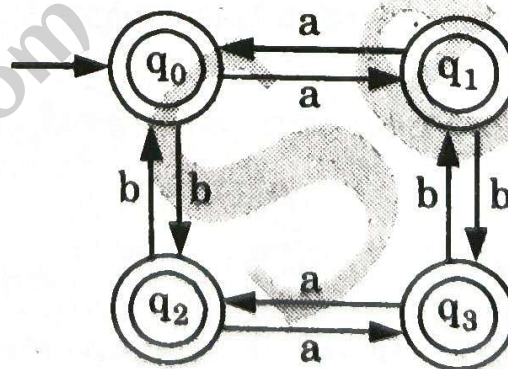


Fig. 1.9.1.

Regular expression for above DFA :

$(aa + bb + (ab + aa)(aa + bb)^*(ab + ba))^*$

1.10. Write a CF grammar to represent palindrome.

Ans. Let the grammar $G = \{V_n, V_t, P, S\}$ and the tuples are :

$V_n = \{S\}$

$V_t = \{0, 1\}$

S = start symbol

P is defined as :

$$S \rightarrow 0 \mid 1 \mid 0S0 \mid 1S1 \mid \epsilon$$

1.11. List the features of good compiler.

Ans. Features of good compiler are :

- i. It compiles the large amount of code in less time.
- ii. It requires less amount of memory space to compile the source language.
- iii. It can compile only the modified code segment if frequent modifications are required in the source program.
- iv. While handling the hardware interrupts good compilers interact closely with operating system.

1.12. What are the two parts of a compilation ? Explain briefly.

Ans. Two parts of a compilation are :

1. **Analysis part** : It breaks up the source program into constituent pieces and creates an intermediate representation of source program.
2. **Synthesis part** : It constructs the desire target program from the intermediate representation.

1.13. What are the classifications of a compiler ?

Ans. Classification of a compiler :

1. Single pass compiler
2. Two pass compiler
1. Multiple pass compiler

1.14. Design a DFA for the following regular expression :

$$(x + y)^*xyy.$$

AKTU 2020-21, Marks 02

Ans.

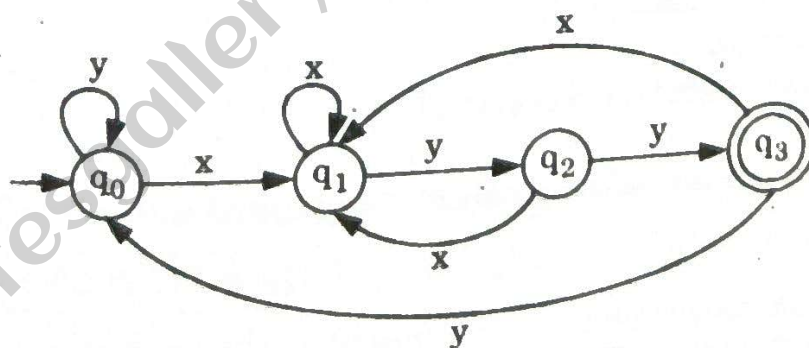


Fig. 1.14.1.

1.15. What is YACC ? Discuss about it.

AKTU 2020-21, Marks 02

Ans. YACC parser generator :

1. YACC (Yet Another Compiler - Compiler) is the standard parser generator for the Unix operating system.
2. An open source program, YACC generates code for the parser in the C programming language.
3. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code.

1.16. Define the terms Language Translator and compiler.

AKTU 2021-22, Marks 02

Ans. **Language translator :** Language translator is a program which is used to translate instructions that are written in the source code to object code *i.e.*, from high-level language or assembly language into machine language.

Compiler : A compiler is a translator used to convert high-level programming language to low-level programming language.

1.17. How will you group the phases of compiler ?

AKTU 2022-23, Marks 02

Ans. Phases of compiler can be grouped into two parts :

1. **Front end phase :** It consists of Lexical Analysis, Syntactic Analysis, Semantic analysis and Intermediate Code generation.
2. **Back end phase :** It consists of code optimization and code generation.

1.18. What are the various parts in LEX program ?

AKTU 2022-23, Marks 02

Ans. **Various parts in LEX program :**

- i. Declaration section
- ii. Rule section
- iii. Auxiliary procedure section

1.19. Which phase of compiler is optional and why ?

AKTU 2023-24, Marks 02

Ans. Code optimization is an optional phase because it is used to improve the performance and efficiency of the generated code but does not affect the correctness of the program.

1.20. Discuss the capabilities of CFG.

AKTU 2022-23, Marks 02

Ans. Capabilities of CFG :

1. Capable of capturing the structure of programming languages.
2. Easy to read and understand.
3. Can generate infinite number of sentences.
4. Can be parsed efficiently.

1.21. What is an ambiguous grammar ? Give example.

AKTU 2021-22, Marks 02

Ans. Ambiguous grammar : A context free grammar G is ambiguous if there is at least one string in $L(G)$ having two or more distinct derivation tree.

Example :

Given :

$$S \rightarrow AB \mid aaB$$

$$A \rightarrow a \mid Aa$$

$$B \rightarrow b$$

Let us generate string aab from the given grammar. Parse tree for generating string aab are as follows :

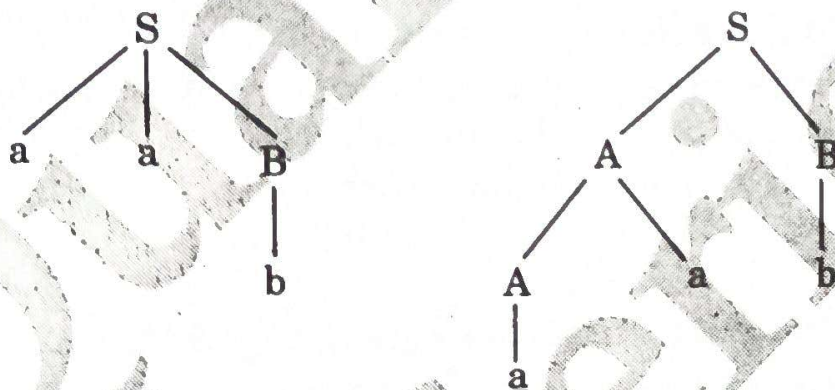


Fig. 1.21.1.

1.22. Define bootstrapping in the context of compilers.

AKTU 2023-24, Marks 02

Ans.

1. Bootstrapping is the process of writing a compiler (or assembler) in the source programming language that it intends to compile.
2. Bootstrapping leads to a self-hosting compiler.
3. An initial minimal core version of the compiler is generated in a different language.



2**UNIT**

Basic Parsing Techniques (2 Marks Questions)

2.1. State the problems associated with the top-down parsing.

OR

Explain the problems associated with top-down Parser.

AKTU 2021-22, Marks 02

Ans. Problems associated with top-down parsing are :

1. Backtracking
2. Left recursion
3. Left factoring
4. Ambiguity

2.2. What is the role of left recursion ?

Ans. Left recursion is a special case of recursion where a string is recognized as part of a language by the fact that it decomposes into a string from that same language (on the left) and a suffix (on the right).

2.3. Name the data structures used by LL(1) parser.

Ans. Data structures used by LL(1) parser are :

- i. Input buffer
- ii. Stack
- iii. Parsing table

2.4. Give the data structures for shift reduce parser.

Ans. Data structures for shift reduce parser are :

- i. The input buffer storing the input string.
- ii. A stack for storing and accessing the L.H.S and R.H.S of rules.

2.5. What are various types of LR parser ?

Ans. Various types of LR parser are :

- i. SLR parser
- ii. LALR parser
- iii. Canonical LR parser

2.6. Why do we need LR parsing table ?

Ans. We need LR parsing tables to parse the input string using shift reduce method.

2.7. State limitation of SLR parser.

Ans. Limitations of SLR parser are :

- SLR is less powerful than LR parser since SLR grammars constitute a small subset of CFG.
- Problem of multiple entries arises when reduction may not lead to the generation of previous rightmost derivations.

2.8. Define left factoring.

OR

What is do you mean by left factoring the grammars ?

Explain.

AKTU 2021-22, Marks 02

Ans. A grammar G is said to be left factored if any production of it is in the form of :

$$A \rightarrow \alpha\beta \mid \alpha\gamma$$

2.9. Define left recursion.

Ans. A grammar $G(V, T, P, S)$ is said to be left recursive if it has a production in the form of :

$$A \rightarrow A\alpha \mid \beta$$

2.10. Consider the following grammar : $S \rightarrow B \mid SabS, B \rightarrow bB \mid \epsilon$

Compute FOLLOW(B).

AKTU 2020-21, Marks 02

Ans. $S \rightarrow B \mid SabS, B \rightarrow bB \mid \epsilon$
 $\text{Follow}(B) = \{\text{Follow}(S)\}$
 $= \{\$, a\}$

2.11. Discuss about shift-reduce parsing.

AKTU 2020-21, Marks 02

OR

Explain the concept of shift-reduce parsing.

AKTU 2023-24, Marks 02

Ans. **Shift-reduce parsing :**

- Shift-reduce parser attempts to construct parse tree from leaves to root and uses a stack to hold grammar symbols.
- A parser goes on shifting the input symbols onto the stack until a handle comes on the top of the stack.
- When a handle appears on the top of the stack, it performs reduction.

2.12. Define left recursion. Is the following grammar left recursive ?

$$E \rightarrow E + E \mid E * E \mid a \mid b$$

AKTU 2021-22, Marks 02

Ans. Left recursion : Refer Q. 2.9, Page SQ-7C, Unit-2, Two Marks Questions.

Numerical :

$$E \rightarrow E + E \mid E * E \mid a \mid b$$

if we compare above grammar with

$$A \rightarrow A \alpha \mid \beta$$

$$\underbrace{E}_A \rightarrow \underbrace{E}_A + \underbrace{E}_\alpha$$

$$\underbrace{E}_A \rightarrow \underbrace{E}_A * \underbrace{E}_\alpha$$

Yes, the given grammar is left recursive.

2.13. List down the conflicts during shift-reduce parsing.

AKTU 2021-22, Marks 02

Ans. Two most common conflicts during shift-reduce parsing are as follows :

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

2.14. Differentiate between LR and LL parsers.

AKTU 2022-23, Marks 02

Ans.

S. No.	LL Parser	LR Parser
1.	Scans the input sequence from left to right and constructs a leftmost derivation.	Scans the input sequence from left to right and constructs a rightmost derivation.
2.	Uses a top-down parsing technique.	Uses a bottom-up parsing technique.





Syntax Directed Translation (2 Marks Questions)

3.1. What do you mean by Syntax Directed Definition (SDD) ?

Ans. SDD is a generalization of CFG in which each grammar production $X \rightarrow \alpha$ is associated with a set of semantic rules of the form $a = f(b_1, b_2, \dots, b_n)$ where a is an attribute obtained from the function f .

3.2. Define intermediate code.

Ans. Intermediate code is a code which is very close to machine code generated by intermediate code generator during compilation.

3.3. What are the various types of intermediate code representation ?

Ans. Different forms of intermediate code are :

- i. Abstract syntax tree
- ii. Polish (prefix/postfix) notation
- iii. Three address code

3.4. Define three address code.

Ans. Three address code is an abstract form of intermediate code that can be implemented as a record with the address fields. The general form of three address code representation is :

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

3.5. What is postfix notations ?

Ans. Postfix notation is the type of notation in which operator are placed at the right end of the expression. For example, to add A and B, we can write as $AB+$ or $BA+$.

3.6. Why are quadruples preferred over triples in an optimizing compiler ?

Ans.

1. Quadruples are preferred over triples in an optimizing compiler as instructions are after found to move around in it.

2. In the triples notation, the result of any given operations is referred by its position and therefore if one instruction is moved then it is required to make changes in all the references that lead to that result.

3.7. Give syntax directed translation for case statement.

Ans. Syntax directed translation scheme for case statement :

Production rule	Semantic action
Switch E { case $v_1 : s_1$ case $v_2 : s_2$... case $v_{n-1} : s_{n-1}$ default : s_n }	Evaluate E into t such that $t = E$ goto check L_1 : code for s_1 goto last L_2 : code for s_2 goto last L_n : code for s_n goto last check : if $t = v_1$ goto L_1 if $t = v_2$ goto L_2 ... if $t = v_{n-1}$ goto L_{n-1} goto L_n last :

3.8. What is a syntax tree ? Draw the syntax tree for the following statement : $c\ b\ c\ b\ a\ -\ * \ + \ - \ * \ =$

Ans.

1. A syntax tree is a tree that shows the syntactic structure of a program while omitting irrelevant details present in a parse tree.
2. Syntax tree is condensed form of the parse tree.

Syntax tree of $c\ b\ c\ b\ a\ -\ * \ + \ - \ * \ =$:

In the given statement, number of alphabets is less than symbols.
So, the syntax tree drawn will be incomplete.

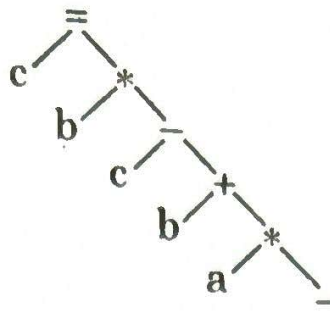


Fig. 3.8.1.

3.9. Define backpatching.

Ans. Backpatching is an activity of filling up unspecified information of labels using appropriate semantic actions during the code generation process. Backpatching is done for :

- Boolean expressions
- Flow of control statements

3.10. Give advantages of SDD.

Ans. The main advantage of SDD is that it helps in deciding evaluation order. The evaluation of semantic actions associated with SDD may generate code, save information in symbol table, or may issue error messages.

3.11. What are quadruples ?

Ans. Quadruples are structure with at most four fields such as op, arg1, arg2, result.

3.12. Differentiate between quadruples and triples.

Ans.

S. No.	Quadruples	Triples
1.	Quadruple represent three address code by using four fields : OP, ARG1, ARG2, RESULT.	Triples represent three address code by using three fields : OP, ARG1, ARG 2.
2.	Quadruple can perform code immovability during optimization.	Triple faces the problem of code immovability during optimization.

3.13. Find the postfix notation for the following expression :

$$(a + b + c) * (c + q).$$

AKTU 2020-21, Marks 02

Ans. Given expression : $(a + b + c) * (c + q)$
 $(a + b + c) * t_1$
 $(t_2 + c) * t_1$
 $t_1 = cq +$
 $t_2 = ab +$

$$t_3 * t_1$$
$$t_4$$

$$t_3 = t_2^c +$$
$$t_4 = t_3 t_1^*$$

Now putting values of t_1, t_2, t_3, t_4

$$t_4 = t_3 t_1^*$$
$$= t_3 c q + *$$
$$= t_2^c + c q + *$$
$$= a b + c + c q + *$$

3.14. What is the difference between parse tree and abstract syntax tree ?

AKTU 2021-22, Marks 02

OR

Differentiate Parse tree and Syntax tree with an example.

AKTU 2022-23, 2023-24; Marks 02

Ans.

S.No.	Parse tree	Abstract syntax tree
1.	An ordered, rooted tree that represents the syntactic structure of string according to some context free grammar.	A tree representation of the abstract syntactic structure of source code written in programming language.
2.	It contains records of the rules (tokens) to match input texts.	It contains records of the syntax of programming language.

3.15. What are the two types of attributes that are associated with a grammar symbol ?

AKTU 2021-22, 2023-24; Marks 02

- Ans. Two types of attributes that are associated with a grammar symbol are :
- a. Synthesized attributes
 - b. Inherited attributes

3.16. Mention the role of semantic analysis.

AKTU 2022-23, Marks 02

Ans. Role of semantic analysis is to ensure that the declarations and statements of a program are semantically correct.

3.17. Define syntax-directed translation schemes.

AKTU 2023-24, Marks 02

Ans. Syntax directed definition/translation is a generalization of context free grammar in which each grammar production $X \rightarrow \alpha$ is associated with a set of semantic rules of the form $a := f(b_1, b_2, \dots, b_k)$, where a is an attribute obtained from the function f .

3.18. Discuss about non-linear type intermediate code.

AKTU 2020-21, Marks 02

Ans. Syntax tree is a non-linear type intermediate code.

Syntax tree :

1. A syntax tree is a tree that shows the syntactic structure of a program while omitting irrelevant details present in a parse tree.
2. Syntax tree is condensed form of the parse tree.
3. The operator and keyword nodes of a parse tree are moved to their parent and a chain of single production is replaced by single link.





Symbol Tables (2 Marks Questions)

4.1. Write down the short note on symbol table.

Ans. A symbol table is a data structure used by a compiler to keep track of scope, life and binding information about names. These names are used in the source program to identify the various program elements, like variables, constants, procedures, and the labels of statements.

4.2. Describe data structure for symbol table.

Ans. Data structures for symbol table are :

1. Unordered list
2. Ordered list
3. Search tree
4. Hash tables and hash functions

4.3. What is mean by activation record ?

AKTU 2023-24, Marks 02

OR

Write short note on "Activation Record".

AKTU 2020-21, Marks 02

Ans. Activation record is a data structure that contains the important state information for a particular instance of a function call.

4.4. What is phase level error recovery ?

AKTU 2022-23, Marks 02

Ans. Phase level error recovery is implemented by filling the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack.

4.5. List the various error recovery strategies for a lexical analysis.

Ans. Error recovery strategies are :

1. Panic mode recovery
2. Phrase level recovery
3. Error production
4. Global correction

4.6. What are different storage allocation strategies ?

Ans. Different storage allocation strategies are :

- i. Static allocation
- ii. Stack allocation
- iii. Heap allocation

4.7. What do you mean by stack allocation ?

Ans. Stack allocation is a strategy in which the storage is organized as stack. This stack is also called control stack. This stack helps to manage the storage space of local variable by using push and pop operation.

4.8. Discuss heap allocation strategy.

Ans. The heap allocation allocates and deallocates the continuous block of memory when required for storage data object at run time.

4.9. Discuss demerit of heap allocation.

Ans. Demerit of heap allocation is that it introduces holes in the memory during allocation.

4.10. Discuss how scope information is represented in a symbol table.

AKTU 2023-24, Marks 02

Ans. Scope information in a symbol table is typically represented using a hierarchical structure or through unique identifiers.

4.11. Discuss about hash table.

AKTU 2020-21, Marks 02

Ans.

1. Hashing is an important technique used to search the records of symbol table. This method is superior to list organization.
2. In hashing scheme, a hash table and symbol table are maintained.
3. The hash table consists of k entries from 0, 1 to $k - 1$. These entries are basically pointers to symbol table pointing to the names of symbol table.

4.12. What are the various errors that may appear in compilation process ?

AKTU 2021-22, Marks 02

Ans. Types of errors that may appear in compilation process are:

- i. Lexical phase error
- ii. Syntactic phase error
- iii. Semantic phase error

AKTU 2021-22, Marks 02

4.13. What is hashing ? Explain.

Ans. Hashing is the process of transforming any given key or a string of characters into another value.



5**UNIT**

Code Generation (2 Marks Questions)

5.1. What do you mean by code optimization ?

Ans. Code optimization refers to the technique used by the compiler to improve the execution efficiency of the generated object code.

5.2. Define code generation.

Ans. Code generation is a process of creating assembly language/machine language statements which will perform the operations specified by source program when they run. Code generation is the final activity of compiler.

5.3. What are the various loops in flow graph ?

Ans. Various loops in flow graph are :

- i. Dominators
- ii. Natural loops
- iii. Inner loops
- iv. Pre-header
- v. Reducible flow graph

5.4. Define DAG.

Ans.

1. The abbreviation DAG stands for Directed Acyclic Graph.
2. DAGs are useful data structure for implementing transformations on basic blocks.
3. A DAG gives picture of how the value computed by each statement in the basic block is used in the subsequent statement of the block.

5.5. Write applications of DAG.

Ans. The DAG is used in :

- i. Determining the common sub-expressions.
- ii. Determining which names are used inside the block and computed outside the block.
- iii. Determining which statements of block could have their computed value outside the block.
- iv. Simplifying the list of quadruples by eliminating the common sub-expressions.

5.6. What is loop optimization ? What are its methods ?

Ans. Loop optimization is a technique in which code optimization is performed on inner loops. The loop optimization is carried out by following methods :

- i. Code motion
- ii. Induction variable and strength reduction
- iii. Loop invariant method
- iv. Loop unrolling
- v. Loop fusion

5.7. What are various issues in design of code generation ?

OR

Discuss about designing issues of code generator.

AKTU 2020-21, Marks 02

OR

Discuss two design issues in code generation.

AKTU 2023-24, Marks 02

Ans. Various issues in design of code generation are :

- i. Input to the code generator
- ii. Target programs
- iii. Memory management
- iv. Instruction selection
- v. Register allocation
- vi. Choice of evaluation order
- vii. Approaches to code generation

5.8. List out the criteria for code improving transformations.

Ans. Criteria for code improving transformations are :

1. A transformation must preserve meaning of a program.
2. A transformation must improve program by a measurable amount on average.
3. A transformation must worth the effort.

**5.9. Represent the following in flow graph $i = 1$; $sum = 0$;
while ($i \leq 10$) { $sum += i$; $i++$;} }**

Ans. Flow graph is given as :

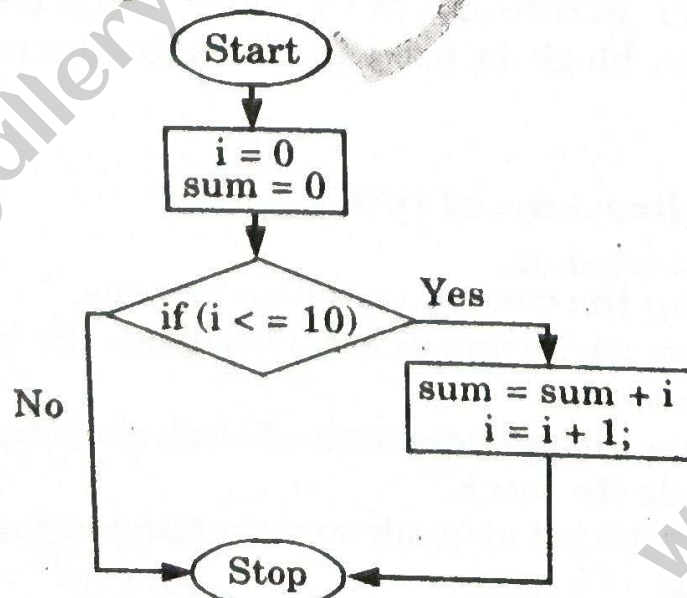


Fig. 5.9.1.

5.10. What is the use of algebraic identities in optimization of basic blocks ?

Ans. Uses of algebraic identities in optimization of basic blocks are :

1. The algebraic transformation can be obtained using the strength reduction technique.
2. The constant folding technique can be applied to achieve the algebraic transformations.
3. The use of common sub-expression elimination, use of associativity and commutativity is to apply algebraic transformations on basic blocks.

5.11. Discuss the subset construction algorithm.

Ans.

1. Subset construction algorithm is the algorithm in which algorithms are partitioned into subset by finding the leader in the algorithm.
2. The target element of conditional or unconditional goto is a leader.
3. The basic block is formed by starting at the leader and ending just before the leader statement.

5.12. How to perform register assignment for outer loops ?

Ans.

Global register allocation allocates registers for variables in inner loops first, since that is generally where a program spends a lot of its time and the same register is used for the variables if it also appears in an outer loop.

5.13. What is meant by viable prefixes ?

Ans.

Viable prefixes are the prefixes of right sentential forms that can appear on the stack of a shift-reduce parser.

5.14. Define peephole optimization.

Ans.

Peephole optimization is a type of code optimization performed on a small part of the code. It is performed on the very small set of instructions in a segment of code. The small set of instructions or small part of code on which peephole optimization is performed is known as peephole.

5.15. What is dangling else problem ?

Ans.

Dangling else problem is the problem in which compiler always associates an 'else' with the immediately preceding 'if' which causes ambiguity. This problem arises in a nested if statement, where number of if's is more than the number of else clause.

5.16. Define loop jamming.

Ans.

Loop jamming is a technique used to improve the performance of loops in a flow graph. It involves merging two or more loops with

similar characteristics into a single loop to reduce the number of iterations and improve cache performance.

5.17. What is induction variable ?

AKTU 2022-23, Marks 02

Ans. Induction variables :

1. A variable x is called an induction variable of loop L if the value of variable gets changed every time.
2. It is either decremented or incremented by some constant.

5.18. Give the properties of intermediate representation.

AKTU 2022-23, Marks 02

Ans. Properties of intermediate representation :

- i. It is platform or machine independent.
- ii. It is used to re-target the compiler to generate code for the given processor.

5.19. Discuss about constant folding.

AKTU 2020-21, Marks 02

Ans. Constant folding (compile time evaluation) :

1. Constant folding is defined as replacement of the value of one constant in an expression by equivalent constant value at the compile time.
2. In constant folding all operands in an operation are constant. Original evaluation can also be replaced by result which is also a constant.

For example : $a = 3.14157/2$ can be replaced by $a = 1.570785$ thereby eliminating a division operation.

5.20. Explain the concept of global data-flow analysis.

AKTU 2023-24, Marks 02

Ans. Global data flow analysis :

1. Global data flow analysis collects the information about the entire program and distributed it to each block in the flow graph.
2. Data flow can be collected in various block by setting up and solving a system of equation.



B. Tech.
(SEM. V) ODD SEMESTER THEORY
EXAMINATION, 2020-21
COMPILER DESIGN

Time : 3 Hours

Max. Marks : 100

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

SECTION-A

1. Attempt all questions in brief. (2 × 10 = 20)

a. What is YACC ? Discuss about it.

Ans. Refer Q. 1.15, Page SQ-4C, Unit-1, Two Marks Questions.

b. Design a DFA for the following regular expression:
 $(x + y)^*xyy$.

Ans. Refer Q. 1.14, Page SQ-3C, Unit-1, Two Marks Questions.

c. Consider the following grammar : $S \rightarrow B | SabS, B \rightarrow bB | \epsilon$
Compute FOLLOW(B).

Ans. Refer Q. 2.10, Page SQ-7C, Unit-2, Two Marks Questions.

d. Discuss about shift-reduce parsing.

Ans. Refer Q. 2.11, Page SQ-7C, Unit-2, Two Marks Questions.

e. Find the postfix notation for the following expression :
 $(a + b + c)^*(c + q)$.

Ans. Refer Q. 3.13, Page SQ-11C, Unit-3, Two Marks Questions.

f. Discuss about non-linear type intermediate code.

Ans. Refer Q. 3.18, Page SQ-13C, Unit-3, Two Marks Questions.

g. Write short note on "Activation Record".

Ans. Refer Q. 4.3, Page SQ-14C, Unit-4, Two Marks Questions.

h. Discuss about hash table.

Ans. Refer Q. 4.11, Page SQ-15C, Unit-4, Two Marks Questions.

i. Discuss about constant folding.

Ans. Refer Q. 5.19, Page SQ-20C, Unit-5, Two Marks Questions.

j. Discuss about designing issues of code generator.

Ans. Refer Q. 5.7, Page SQ-18C, Unit-5, Two Marks Questions.

SECTION-B

2. Attempt any three of the following :

a. Explain in detail the process of compilation for the statement $a = b + c * 70$.

Ans. Refer Q. 1.2, Page 1-5C, Unit-1.

b. Construct the CLR(1) parsing table for the following grammar $S \rightarrow AA, A \rightarrow aA | b$.

Ans. Refer Q. 2.29, Page 2-33C, Unit-2.

c. Consider the following grammar and give the syntax directed definition to construct parse tree for the input expression $4 * 7 + 3 * 9$. Also construct an annotated parse tree according to your syntax directed definition.

$E \rightarrow E + T | T, T \rightarrow T * F | F, F \rightarrow \text{num}.$

Ans. Refer Q. 3.5, Page 3-6C, Unit-3.

d. Explain lexical, syntax, semantic phase errors in detail.

Ans. Refer Q. 4.19, Page 4-18C, Unit-4.

e. Explain in detail about loop optimization.

Ans. Refer Q. 5.11, Page 5-12C, Unit-5.

SECTION-C

3. Attempt any one part of the following : (1 × 10 = 10)

a. i. Write a regular expression to represent a language consisting of strings made up of odd number of a and odd number of b .

Ans. Refer Q. 1.19, Page 1-21C, Unit-1.

ii. Write a CFG to represent the language $L = \{a^n +^m b^n \mid m, n \geq 1\}$.

Ans. Refer Q. 1.30, Page 1-28C, Unit-1.

b. i. Check whether given grammar is ambiguous or not. If ambiguous then convert it into unambiguous grammar : $E \rightarrow E + E | E * E | id.$

Ans. Refer Q. 1.34, Page 1-31C, Unit-1.

ii. Discuss about cross compiler.

Ans. Refer Q. 1.5, Page 1-7C, Unit-1.

4. Attempt any one part of the following : (1 × 10 = 10)

- a. Check whether the given grammar is LR(0) or not :
 $S \rightarrow PQy$, $P \rightarrow Sy \mid x$, $Q \rightarrow yS$.

Ans. Refer Q. 2.20, Page 2-20C, Unit-2.

- b. Find the precedence and function table of the following grammar by using operator precedence technique.

$P \rightarrow SR \mid S$, $R \rightarrow bSR \mid bS$, $S \rightarrow WbS \mid W$, $W \rightarrow L^*W \mid L$, $L \rightarrow id$.

Ans. Refer Q. 2.7, Page 2-8C, Unit-2.

5. Attempt any one part of the following : (1 × 10 = 10)

- a. Translate the following arithmetic expression into quadruples and triples :

i. $x = y * z + y * -z$

ii. $a = -b * (c + d) + b$

Ans. Refer Q. 3.15, Page 3-13C, Unit-3.

- b. Generate three address code for the following code :

Switch $p + q$

```
{
    case 1 :  $x = x + 1$ 
    case 2 :  $y = y + 2$ 
    case 3 :  $z = z + 3$ 
    default:  $c = c - 1$ 
}
```

Ans. Refer Q. 3.14, Page 3-13C, Unit-3.

6. Attempt any one part of the following : (1 × 10 = 10)

- a. What is symbol table ? Explain various data structures used for symbol table.

Ans. Refer Q. 4.4, Page 4-4C, Unit-4.

- b. i. Explain the function of error handling phase of a compiler.

Ans. Refer Q. 4.22, Page 4-22C, Unit-4.

- ii. Write short note on scoping.

Ans. Refer Q. 4.7, Page 4-8C, Unit-4.

7. Attempt any one part of the following : (1 × 10 = 10)

- a. Construct the flow graph for the following code segment:

```
fact(n)
{
    int f = 1;
    for(i = 2; i ≤ n; i++)
        f = f * i;
    return f;
}
```


Ans. Refer Q. 5.16, Page 5-18C, Unit-5.

b. Define a DAG. Construct a DAG for the expression:

$$p + p * (q - r) + (q - r) * s$$

Ans. Refer Q. 5.22, Page 5-23C, Unit-5.



B. Tech.
(SEM. V) ODD SEMESTER THEORY
EXAMINATION, 2021-22
COMPILER DESIGN

Time : 3 Hours

Max. Marks : 100

Note : Attempt **all** sections. If require any missing data; then choose suitably.

SECTION-A

1. Attempt **all** questions in brief. (2 × 10 = 20)

a. What is the difference between parse tree and abstract syntax tree ?

Ans. Refer Q. 3.14, Page SQ-12C, Unit-3, Two Marks Questions.

b. Explain the problems associated with top-down Parser.

Ans. Refer Q. 2.1, Page SQ-6C, Unit-2, Two Marks Questions.

c. What are the various errors that may appear in compilation process ?

Ans. Refer Q. 4.12, Page SQ-15C, Unit-4, Two Marks Questions.

d. What are the two types of attributes that are associated with a grammar symbol ?

Ans. Refer Q. 3.15, Page SQ-12C, Unit-3, Two Marks Questions.

e. Define the terms Language Translator and compiler.

Ans. Refer Q. 1.16, Page SQ-4C, Unit-1, Two Marks Questions.

f. What is hashing ? Explain.

Ans. Refer Q. 4.13, Page SQ-16C, Unit-4, Two Marks Questions.

g. What is do you mean by left factoring the grammars ? Explain.

Ans. Refer Q. 2.8, Page SQ-7C, Unit-2, Two Marks Questions.

h. Define left recursion. Is the following grammar left recursive ?

$$E \rightarrow E + E \mid E * E \mid a \mid b$$

Ans. Refer Q. 2.12, Page SQ-8C, Unit-2, Two Marks Questions.

i. What is an ambiguous grammar ? Give example.

Ans. Refer Q. 1.21, Page SQ-5C, Unit-1, Two Marks Questions.

j. List down the conflicts during shift-reduce parsing.

Ans. Refer Q. 2.13, Page SQ-8C, Unit-2, Two Marks Questions.

SECTION-B

2. Attempt any three of the following :

(10 × 3 = 30)

a. Construct the LALR parsing table for the given grammar

$S \rightarrow BB$

$B \rightarrow aB/b$

Ans. Refer Q. 2.30, Page 2-35C, Unit-2.

b. What is an activation record ? Explain how it is related with runtime -storage organization ?

Ans. Activation record : Refer Q. 4.10, Page 4-10C, Unit-4.
Relation with runtime storage runtime : Refer Q. 4.11, Page 4-11C, Unit-4.

c. Write the quadruple, triple, indirect triple for the following expression $(x + y) * (y + z) + (x + y + z)$

Ans. Refer Q. 3.13, Page 3-12C, Unit-3.

d. Discuss the following terms :

i. Basic block

ii. Next use information

iii. Flow graph

Ans. Refer Q. 5.17, Page 5-19C, Unit-5.

e. Construct predictive parse table for the following grammar.

$E \rightarrow E + T/T$

$T \rightarrow T * F/F$

$F \rightarrow F / a/b$

Ans. Refer Q. 2.15, Page 2-15C, Unit-2.

SECTION-C

3. Attempt any one part of the following :

(10 × 1 = 10)

a. Construct the SLR parse table for the following Grammar

$E' \rightarrow E$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

Ans. Refer Q. 2.21, Page 2-22C, Unit-2.

- b. Differentiate between stack allocation and heap allocation.

Ans. Refer Q. 4.14, Page 4-14C, Unit-4.

4. Attempt any one part of the following : (10 × 1 = 10)

- a. Write syntax directed definition for a given assignment statement :

$S \rightarrow id = E$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow E$

$E \rightarrow (E)$

$E \rightarrow id$

Ans. Refer Q. 3.21, Page 3-21C, Unit-3.

- b. What are the advantages of DAG ? Explain the peephole optimization.

Ans. Refer Q. 5.24, Page 5-25C, Unit-5.

5. Attempt any one part of the following : (10 × 1 = 10)

- a. What do you understand by lexical phase error and syntactic error ? Also suggest methods for recovery of errors.

Ans. Refer Q. 4.19, Page 4-18C, Unit-4.

- b. Discuss how induction variables can be detected and eliminated from the given intermediate code

B2: $i := i + 1$

$t_1 := 4 * j$

$t_2 := a[t_1]$

if $t_2 < 10$ goto B2

Ans. Refer Q. 5.14, Page 5-17C, Unit-5.

6. Attempt any one part of the following : (10 × 1 = 10)

- a. Test whether the grammar is LL (1) or not, and construct parsing table for it.

$S \rightarrow 1AB/\epsilon$

$A \rightarrow 1AC/0C$

$B \rightarrow 0S$

$C \rightarrow 1$

Ans. Refer Q. 2.16, Page 2-16C, Unit-2.

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

Ans. Refer Q. 4.9, Page 4-9C, Unit-4.

7. Attempt any one part of the following : (10 × 1 = 10)

a. What are the various issues in design of code generator and code loop optimization ?

Ans: Various issues of code generator : Refer Q. 5.1, Page 5-2C, Unit-5.

Various issue of code loop optimization : Refer Q. 5.11, Page 5-12C, Unit-5.

b. Generate the three address code for the following code fragment.

```
while(a > b)
{
    if(c < d)
        x = y + z;
    else
        x = y - z;
}
```

Ans: Refer Q. 3.16, Page 3-15C, Unit-3.



B. Tech.
(SEM. V) ODD SEMESTER THEORY
EXAMINATION, 2022-23
COMPILER DESIGN

Time : 3 Hours

Max. Marks : 100

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

Section-A

1. Attempt all questions in brief. (2 × 10 = 20)

a. How will you group the phases of compiler ?

Ans. Refer Q. 1.17, Page SQ-4C, Unit-1, Two Marks Questions.

b. Mention the role of semantic analysis.

Ans. Refer Q. 3.16, Page SQ-12C, Unit-3, Two Marks Questions.

c. What are the various parts in LEX program ?

Ans. Refer Q. 1.18, Page SQ-4C, Unit-1, Two Marks Questions.

d. Differentiate parse tree and syntax tree with an example.

Ans. Refer Q. 3.15, Page SQ-12C, Unit-3, Two Marks Questions.

e. Give the properties of intermediate representation.

Ans. Refer Q. 5.18, Page SQ-20C, Unit-5, Two Marks Questions.

f. Differentiate between LR and LL parsers.

Ans. Refer Q. 2.14, Page SQ-8C, Unit-2, Two Marks Questions.

g. What is phrase level error recovery ?

Ans. Refer Q. 4.4, Page SQ-14C, Unit-4, Two Marks Questions.

h. Discuss the capabilities of CFG.

Ans. Refer Q. 1.20, Page SQ-5C, Unit-1, Two Marks Questions.

i. Define loop jamming.

Ans. Refer Q. 5.16, Page SQ-19C, Unit-5, Two Marks Questions.

j. What is induction variable ?

Ans. Refer Q. 5.17, Page SQ-20C, Unit-5, Two Marks Questions.

Section-B

2. Attempt any three of the following : (10 × 3 = 30)

- a. Write SDD to produce three-address code for Boolean expressions and obtain the three-address code for the statement given below :

```
while a < b do
  if c < d then
    x = y * z
  else
    x = y + z
```

Ans Refer Q. 3.24, Page 3-26C, Unit-3.

- b. Discuss the stack allocation and heap allocation strategies of the runtime environment with an example.

Ans Refer Q. 4.11, Page 4-11C, Unit-4.

- c. What do you mean by attributed grammars ? Discuss the translation scheme for converting an infix expression to its equivalent postfix form.

Ans Refer Q. 3.28, Page 3-28C, Unit-3.

- d. Construct the NFA and DFA for the following regular expression.

$(0 + 1)^* (00 + 11)(0 + 1)^*$

Ans Refer Q. 1.15, Page 1-17C, Unit-1.

- e. Explain the lexical analysis and syntax analysis phases of the compiler with a suitable example. Explain the reporting errors in these two phases as well.

Ans Refer Q. 4.19, Page 4-18C, Unit-4.

Section-C

3. Attempt any one part of the following : (10 × 1 = 10)

- a. Construct the CLR parse table for the following Grammar :

```
A → BB
B → cB
B → d
```

Ans Refer Q. 2.31, Page 2-38C, Unit-2.

- b. Construct the SLR parsing table for the following grammar :

```
S → 0S0
S → 1S1
S → 10
```

Ans Refer Q. 2.22, Page 2-24C, Unit-2.

4. Attempt any one part of the following : (10 × 1 = 10)
- a. What is back patching. Generate three address code for the following Boolean expression using backpatching :
 $a < b$ or $c > d$ and $e < f$

Ans. Refer Q. 3.25, Page 3-26C, Unit-3.

- b. What is top down parsing ? What are the problems in top down parsing ? Explain each with suitable example.

Ans. Refer Q. 2.8, Page 2-9C, Unit-2.

5. Attempt any one part of the following : (10 × 1 = 10)
- a. What is an activation record ? Draw diagram of general activation record and explain the purpose of different fields of an activation record.

Ans. Refer Q. 4.10, Page 4-10C, Unit-4.

- b. How do we represent the scope information ? Explain scope by number and scope by location.

Ans. Refer Q. 4.16, Page 4-16C, Unit-4.

6. Attempt any one part of the following : (10 × 1 = 10)
- a. Define symbol table ? Explain about the data structures used for symbol table.

Ans. Refer Q. 4.4, Page 4-4C, Unit-4.

- b. Explain the following :

- i. Copy propagation
- ii. Dead-Code elimination
- iii. Code motion
- iv. Reduction in strength.

Ans.

- i. Refer Q. 5.7, Page 5-8C, Unit-5.
- ii. Refer Q. 5.7, Page 5-8C, Unit-5.
- iii. Refer Q. 5.11, Page 5-12C, Unit-5.
- iv. Refer Q. 5.11, Page 5-12C, Unit-5.

7. Attempt any one part of the following : (10 × 1 = 10)
- a. Explain in the DAG representation of the basic block with example.

Ans. Refer Q. 5.19, Page 5-21C, Unit-5.

- b. Write quadruple, triples and indirect triples for following expression :

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

Ans. Refer Q. 3.17, Page 3-15C, Unit-3.



B. Tech.
(SEM. V) ODD SEMESTER THEORY
EXAMINATION, 2023-24
COMPILER DESIGN

Time : 3 Hours

Max. Marks : 100

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

Section-A

1. Attempt all questions in brief. (2 × 10 = 20)

a. Define bootstrapping in the context of compilers.

Ans. Refer Q. 1.22, Page SQ-5C, Unit-1, Two Marks Questions.

b. Which phase of compiler is optional and why ?

Ans. Refer Q. 1.19, Page SQ-4C, Unit-1, Two Marks Questions.

c. Explain the concept of shift-reduce parsing.

Ans. Refer Q. 2.11, Page SQ-7C, Unit-2, Two Marks Questions.

d. Differentiate Parse tree and Syntax tree with an example.

Ans. Refer Q. 3.14, Page SQ-12C, Unit-3, Two Marks Questions.

e. Define syntax-directed translation schemes.

Ans. Refer Q. 3.17, Page SQ-12C, Unit-3, Two Marks Questions.

f. What are the two types of attributes that are associated with a grammar symbol ?

Ans. Refer Q. 3.15, Page SQ-12C, Unit-3, Two Marks Questions.

g. Discuss how scope information is represented in a symbol table.

Ans. Refer Q. 4.10, Page SQ-15C, Unit-4, Two Marks Questions.

h. What is mean by activation record ?

Ans. Refer Q. 4.3, Page SQ-14C, Unit-4, Two Marks Questions.

i. Discuss two design issues in code generation.

Ans. Refer Q. 5.7, Page SQ-18C, Unit-5, Two Marks Questions.

j. Explain the concept of global data-flow analysis.

Ans. Refer Q. 5.20, Page SQ-20C, Unit-5, Two Marks Questions.

Section-B

2. Attempt any three of the following : (10 × 3 = 30)

- a. Describe the relationship between finite state machines and regular expressions. Discuss how regular expressions are used in lexical analysis and pattern matching.

Ans. Refer Q. 1.11, Page 1-14C, Unit-1.

- b. For the grammar

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe, A \rightarrow f, B \rightarrow f$

Construct LR(1) Parsing table. Also draw the LALR table.

Ans. Refer Q. 2.32, Page 2-41C, Unit-2.

- c. Explain the concepts of quadruples and triples in the context of syntax-directed translation. Discuss how they represent intermediate code and support optimization.

Ans. Refer Q. 3.18, Page 3-16C, Unit-3.

- d. Describe a simple stack allocation scheme for managing memory during program execution. Explain how it is used for storing local variables and managing function calls.

Ans. Refer Q. 4.15, Page 4-15C, Unit-4.

- e. Explain the role of a code generator in a compiler. Discuss its responsibilities and how it translates intermediate code into the target code.

Ans. Refer Q. 5.2, Page 5-3C, Unit-5.

Section-C

3. Attempt any one of the following : (10 × 1 = 10)

- a. Given a regular expression $a(b|c)^*$, construct the corresponding Non-deterministic Finite Automaton (NFA) that recognizes the language described by the regular expression.

Ans. Refer Q. 1.12, Page 1-15C, Unit-1.

- b. Check whether given grammar is ambiguous or not. If ambiguous then convert it into unambiguous grammar :
 $E \rightarrow E + E \mid E^* E \mid id.$

Ans. Refer Q. 1.34, Page 1-31C, Unit-1.

4. Attempt any one of the following : (10 × 1 = 10)

- a. Check whether the given grammar is LR(0) or not :
 $S \rightarrow PQy, P \rightarrow Sy \mid x, Q \rightarrow yS.$

Ans. Refer Q. 2.20, Page 2-20C, Unit-2.

- b. Consider the following grammar for a simple expression language :

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

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

$$F \rightarrow (E) \mid \text{id}$$

Consider expression $\text{id} * (\text{id} + \text{id})$. Apply shift reduce parsing to construct parse tree.

Ans. Refer Q. 2.23, Page 2-25C, Unit-2.

5. Attempt any one of the following : (10 × 1 = 10)

- a. Write syntax directed definition for a given assignment statement :

$$S \rightarrow \text{id} = E$$

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow -E$$

$$E \rightarrow (E)$$

$$E \rightarrow \text{id}$$

Ans. Refer Q. 3.21, Page 3-21C, Unit-3.

- b. Explain how syntax-directed translation handles array references within arithmetic expressions.

Ans. Refer Q. 3.31, Page 3-32C, Unit-3.

6. Attempt any one of the following : (10 × 1 = 10)

- a. Define symbol table ? Explain about the data structures used for symbol table.

Ans. Refer Q. 4.4, Page 4-4C, Unit-4.

- b. Define semantic errors in a compiler. Discuss the challenges associated with detecting and handling semantic errors. Provide example to illustrate semantic issues.

Ans. Refer Q. 4.23, Page 4-23C, Unit-4.

7. Attempt any one of the following : (10 × 1 = 10)

- a. Consider a basic block with the following three instructions:

$$1. x = a + b \quad 2. y = x * c \quad 3. z = y - d$$

Apply common sub expression elimination to optimize the basic block.

Ans. Refer Q. 5.15, Page 5-18C, Unit-5.

- b. Construct a Directed Acyclic Graph (DAG) to represent the following basic block :

$$1. x = a + b$$

$$2. y = c - d$$

$$3. z = x * y$$

Ans. Refer Q. 5.21, Page 5-23C, Unit-5.



CAUTION NOTICE

**TO STUDENTS, DEALERS, SHOPKEEPERS,
COPIERS AND WHOM IT MAY CONCERN**

ORDER OF THE HON'BLE HIGH COURT OF DELHI REGARDING COPYRIGHT INFRINGEMENT OF QUANTUM BOOKS

This is to inform students, dealers, shopkeepers, copiers and the general public that Quantum Page Private Limited is the owner of copyright in the QUANTUM series of books.

Any unauthorised copying, scanning, reproduction, distribution or circulation of the QUANTUM books (whether in soft copy or hard copy) amounts to infringement of the copyright and trademark rights of Quantum Page Private Limited, which are civil wrongs as well as criminal offences with punishment extending to imprisonment.

Quantum Page Private Limited has initiated a lawsuit titled **Quantum Page Private Limited v. Telegram FZ LLC & Ors., CS(Comm.) 921/2022** against infringers of the QUANTUM books before the Hon'ble High Court of Delhi. By its order dated December 23, 2022, the Hon'ble High Court has held that unauthorised reproduction of the QUANTUM books amounts to copyright infringement and has directed the removal of infringing copies of the QUANTUM books from various sources, including Telegram channels.

Students, dealers, shopkeepers, copiers and the general public are hereby cautioned not to carry out any unauthorised copying, scanning, reproduction, distribution and circulation of the QUANTUM books (whether in soft copy or hard copy).

Any such unauthorised use of the QUANTUM books will lead to initiation of civil and criminal proceedings by Quantum Page Private Limited, at the sole risk of the infringers.

QUANTUM Series

Related titles in Quantum Series

For Semester - 6 (Computer Science & Allied Branches)

- Software Engineering
- Compiler Design
- Computer Networks
- Data Analytics

Departmental Elective-III

- Big Data
- Blockchain Architecture Design
- Data Compression
- Machine Learning Techniques

Open Elective-I

- Idea to Business Model
- Object Oriented Programming
- Understanding the Human Being Comprehensively
- Software Project Management

Common Non Credit Course (NC)

- Constitution of India, Law & Engineering
- Essence of Indian Traditional Knowledge

- Topic-wise coverage in Question-Answer form.
- Clears course fundamentals.
- Includes solved University Questions.

A comprehensive book to get the big picture without spending hours over lengthy text books.

Quantum Series is the complete one-stop solution for engineering student looking for a simple yet effective guidance system for core engineering subject. Based on the needs of students and catering to the requirements of the syllabi, this series uniquely addresses the way in which concepts are tested through university examinations. The easy to comprehend question answer form adhered to by the books in this series is suitable and recommended for student. The students are able to effortlessly grasp the concepts and ideas discussed in their course books with the help of this series. The solved question papers of previous years act as a additional advantage for students to comprehend the paper pattern, and thus anticipate and prepare for examinations accordingly.

The coherent manner in which the books in this series present new ideas and concepts to students makes this series play an essential role in the preparation for university examinations. The detailed and comprehensive discussions, easy to understand examples, objective questions and ample exercises, all aid the students to understand everything in an all-inclusive manner.

- The perfect assistance for scoring good marks.
- Good for brush up before exams.
- Ideal for self-study.