CONTENTS

RCS 602 : Compiler Design

Analysis of previous AKTU Papers

Unit-1 : Introduction to Compiler							
Part	Topics	2017-18	2016-17	2015-16	2014-15	2013-14	Que. No.
1.	Introduction to compiler	0	1	0	1	2	1.1*, 1.2, 1.3
2.	Bootstrapping	0	0	0	1	1	1.4*
3.	Finite state machine and regular expression	3	2	0	1	0	1.8, 1.9, 1.10, 1.11, 1.12, 1.13
4.	Implementation of lexical analyzer	0	0	0	0	1	1.21
5.	Formal grammar and their application to syntax analysis	0	0	0	0	0	0
6.	BNF notation	0	0	0	0	0	0
7.	Ambiguity	1	1	0	1	1	1.26, 1.27, 1.28*
8.	Parse tree, derivation tree	0	0	0	0	0	0
	Total Questions	4	4	0	4	5	

* = Asked in different years

Unit-2 : Basic Parsing Techniques							
Part	Topics	2017-18	2016-17	2015-16	2014-15	2013-14	Que. No.
1.	Basic parsing techniques	0	0	0	0	0	0
2.	Operator precedence parsing	1	0	1	0	1	2.5, 2.6, 2.7
3.	Top-down parsing	2	0	0	2	2	2.10, 2.11, 2.12, 2.17, 2.18, 2.19
4.	LR parser	0	0	0	0	0	0
5.	Constructing SLR parsing tables	1	1	1	0	0	2.23, 2.24, 2.25
6.	Constructing canonical LR parsing table	0	0	0	0	0	0
7.	Constructing LALR parsing tables	1	0	2	1	1	2.28, 2.29, 2.30*, 2.31
	Total Questions	5	1	4	3	4	

* = Asked in different years

Unit-3 : Syntax Directed Translation								
Part	Topics	2017-18	2016-17	2015-16	2014-15	2013-14	Que. No.
1.	Syntax-directed translation	0	0	0	1	1	3.1, 3.2
2.	Intermediate code	0	0	0	0	0	0
3.	Postfix notation and syntax trees	0	0	0	1	0	3.6
4.	Three address code	1	0	1	2	1	3.11, 3.12, 3.13, 3.14 3.15
5.	Assignment statements	0	1	0	0	0	3.16
6.	Boolean expressions	0	0	1	1	1	3.17*
7.	Postfix translation	1	0	0	0	0	3.20
8.	Procedure call	0	0	0	0	0	0
9.	Declarations statements	0	0	0	0	0	0
	Total Questions	2	1	2	5	3	

* = Asked in different years

Unit-4 : Symbol Tables								
Part	Topics	2017-18	2016-17	2015-16	2014-15	2013-14	Que. No.
1.	Symbol tables	0	1	1	1	1	4.3, 4.4*, 4.5
2.	Run-time administration	0	1	0	1	1	4.7, 4.8, 4.9
3.	Storage allocation	0	0	0	1	0	4.12
4.	Error detection and recovery	1	0	1	1	1	4.15*, 4.16*
	Total Questions	1	2	2	4	3	

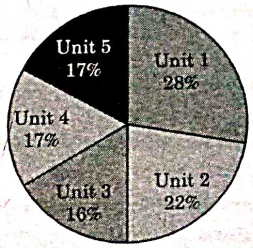
Unit-5 : Code Generation								
Part	Topics	2017-18	2016-17	2015-16	2014-15	2013-14	Que. No.
1.	Code generation	0	0	0	0	0	0
2.	Target language	0	0	0	0	0	0
3.	Basic block and flow graphs	0	2	0	0	1	5.3, 5.6*
4.	Machine independent optimization	2	0	1	1	0	5.10*, 5.11*
5.	DAG representation of basic blocks	2	1	1	1	1	5.12*, 5.13, 5.14, 5.15 5.16
6.	Global data analysis	1	0	1	2	1	5.17, 5.18*, 5.19
	Total Questions	5	3	3	4	3	

* = Asked in different years

2 Marks Questions						
Units \ Year	2017-18	2016-17	2015-16	2014-15	2013-14	Total Questions
Unit-1	4	3	5	0	0	12
Unit-2	0	0	2	0	0	2
Unit-3	3	3	1	0	0	7
Unit-4	3	0	0	0	0	3
Unit-5	0	4	2	0	0	6

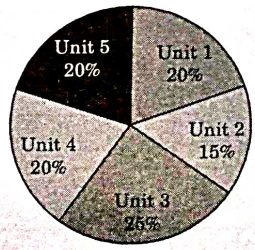
Units	Ques. Asked (2013-14)
Unit-1	5
Unit-2	4
Unit-3	3
Unit-4	3
Unit-5	3

% Weightage of Units (2013-14)



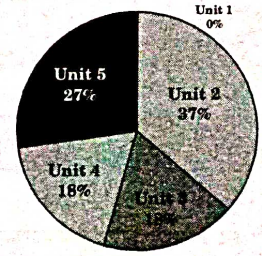
Units	Ques. Asked (2014-15)
Unit-1	4
Unit-2	3
Unit-3	5
Unit-4	4
Unit-5	4

% Weightage of Units (2014-15)



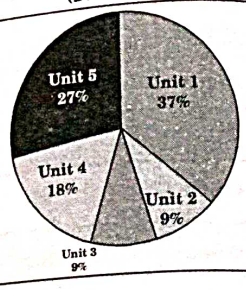
Units	Ques. Asked (2015-16)
Unit-1	0
Unit-2	4
Unit-3	2
Unit-4	2
Unit-5	3

% Weightage of Units (2015-16)



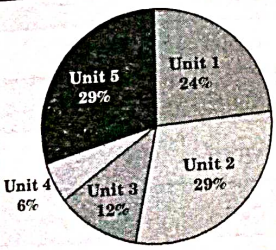
Units	Ques. Asked (2016-17)
Unit-1	4
Unit-2	1
Unit-3	1
Unit-4	2
Unit-5	3

% Weightage of Units (2016-17)



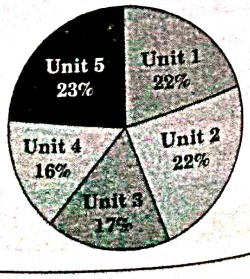
Units	Ques. Asked (2017-18)
Unit-1	4
Unit-2	5
Unit-3	2
Unit-4	1
Unit-5	5

% Weightage of Units (2017-18)



Units	Total Questions (2013-14 to 2017-18)
Unit-1	17
Unit-2	17
Unit-3	13
Unit-4	12
Unit-5	18

% Weightage of Units (2013-14 to 2017-18)



1

UNIT

Introduction to Compiler

CONTENTS

PART-1

Introduction to Compiler, Phases and Passes.

CONCEPT OUTLINE

- Compiler scans all the lines of source program and list out all syntax errors at a time.
- The process of compilation can be carried out in single pass.
- Compiler is a program that converts high level language into machine level language.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.1. **Describe the synthesis-analysis model of compiler.**

AKTU 2013-14, Marks 05

OR

Explain in detail the process of compilation. Illustrate the output of each phase of compilation of the input

"$a = (b + c)*(b + c)* 2$". **AKTU 2016-17, Marks 10**

Answer

A compiler contain six phases which are as follows :

i. **Phase 1 (Lexical analyzer) :**
   a. The lexical analyzer is also called scanner.
   b. The lexical analyzer phase takes source program as an input and separates characters of source language into groups of strings called token.
   c. These tokens may be keywords such as Do, If, While etc., identifiers such as x, num, count etc., operator symbols such as >, >=, +, etc., and punctuation symbols such as parenthesis or commas.

ii. **Phase 2 (Syntax analyzer) :**
   a. The syntax analyzer phase is also called parsing phase.
   b. The syntax analyzer groups tokens together into syntactic structures.
   c. The output of this phase is parse tree.

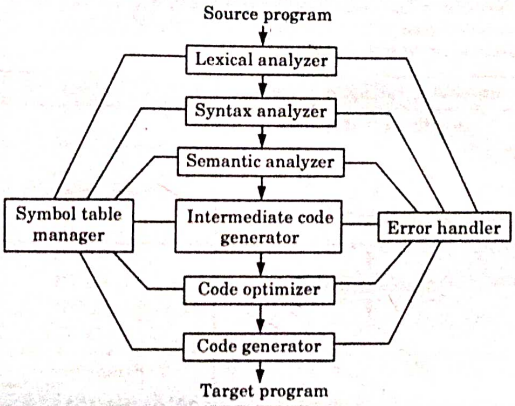


Fig. 1.1.1.

iii. **Phase 3 (Semantic analyzer) :**
   a. The semantic analyzer phase checks the source program for semantic errors and gathers type information for subsequent code generation phase.
   b. It uses the hierarchical structure determined by syntax analyzer phase to identify the operators and operands of expressions and statements.

iv. **Phase 4 (Intermediate code generation) :**
   a. The intermediate code generation takes parse tree as an input from semantic phase and generates intermediate code.
   b. It generates variety of code such as three address code, quadruple, triple.

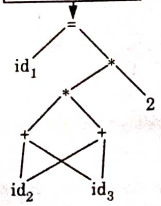
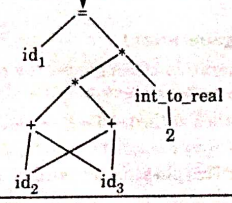
v. **Phase 5 (Code optimization) :** This phase is designed to improve the intermediate code so that the ultimate object program run faster and takes less space.

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.

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.

Compilation of "$a = (b + c)*(b + c)*2$" :

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 ↓ 	Syntax tree
Semantic analyzer ↓ 	Semantic tree
Intermediate code generation ↓ $t_1 = b + c$ $t_2 = t_1 * t_1$ $t_3 = \text{int\_to\_real}(2)$ $t_4 = t_2 * t_3$ $id_1 = t_4$	Intermediate code

Code optimization ↓ $t_1 = b + c$ $t_2 = t_1 * t_1$ $id_1 = t_2 * 2$	Optimized code

Machine code
 ↓
 MOV R_1, b
 ADD R_1, R_1, c
 MUL R_2, R_1, R_1
 MUL $R_2, R_1, \#\ 2.0$
 ST id_1, R_2
	Machine code

Que 1.2. **Explain all the necessary phases and passes of a compiler design.** **AKTU 2014-15, Marks 05**

OR

What are the types of passes ?

Answer

Phases of compiler : Refer Q. 1.1, Page 1–2B, Unit-1.

Types of passes :

1. **Single-pass compiler :**
   a. 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 1.3. **Discuss the role of compiler writing tools. Describe various compiler writing tools.**

OR

Explain some useful compiler construction tools.

OR

What are different compiler tools ? Discuss any two.

AKTU 2013-14, Marks 05

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 takes 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 used to optimized the code involved and gathers the information about how values are transmitted from one part of the program to another.

PART-2

Bootstrapping.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.4. Define bootstrapping with the help of an example.

AKTU 2013-14, Marks 05

OR

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

AKTU 2014-15, Marks 05

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. 1.4.1 is also used to depict the same compiler :

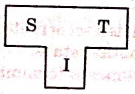


Fig. 1.4.1.

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.)

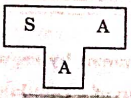


Fig. 1.4.2.

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

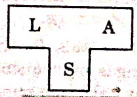


Fig. 1.4.3.

   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 :

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

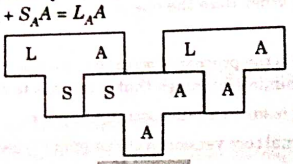


Fig. 1.4.4.

PART-3

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

CONCEPT OUTLINE

- Finite Automata (FA) is a set of finite states and sets of transitions from one state to another state.
- Optimization of DFA means to minimize the number of states in DFA.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.5. **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 1.6. **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

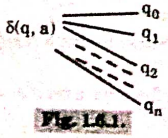


Fig. 1.6.1.

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 :

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

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 : $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$
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 $\Sigma = \{0, 1\}$.

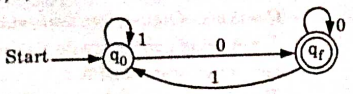


Fig. 1.6.2.

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

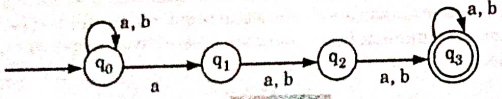


Fig. 1.6.3.

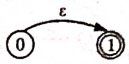
Que 1.7. Explain Thompson's construction with example.

Answer

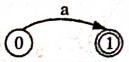
Thompson's construction :

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 :

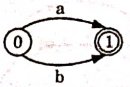
i. The NFA representing the empty string is :



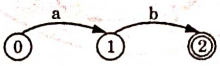
ii. If the regular expression is just a character, thus a can be represented as :



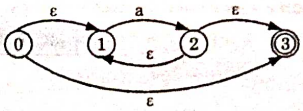
iii. The union operator is represented by a choice of transitions from a node thus $a \mid b$ can be represented as :



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 :

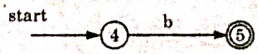


For example :

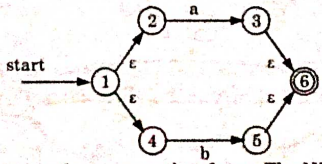
Construct NFA for $r = (a \mid b)^* a$

For $r_1 = a$,

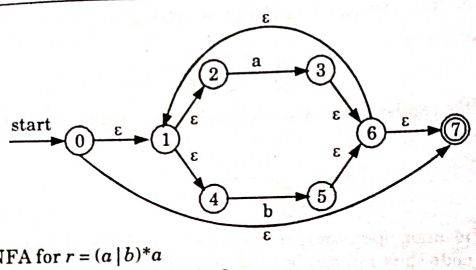
For $r_1 = b$,



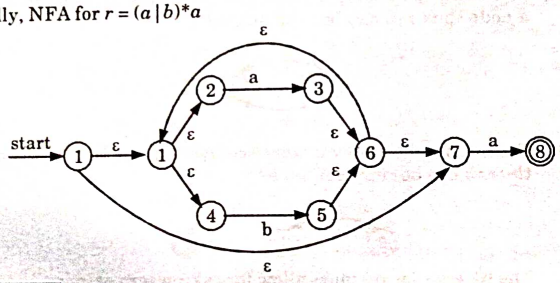
For $r_3 = a \mid b$



The NFA for $r_4 = (r_3)$ is the same as that for r_3. The NFA for $r_5 = (r_3)^*$



Finally, NFA for $r = (a \mid b)^* a$



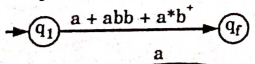
Que 1.8. **Construct the NFA for the regular expression $a \mid abb \mid a^*b^+$ by using Thompson's construction methodology.**

AKTU 2017-18, Marks 10

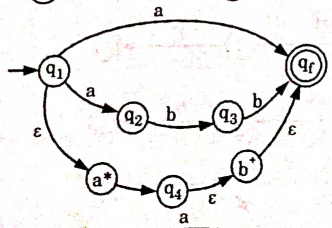
Answer

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

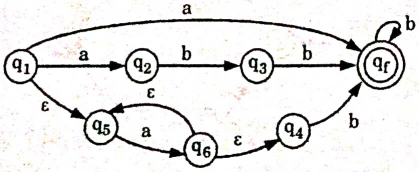
Step 1 :



Step 2 :



Step 3 :



Que 1.9. **Draw NFA for the regular expression $ab^* \mid ab$.**

AKTU 2016-17, Marks 10

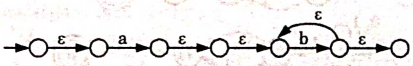
Answer

Step 1 : a

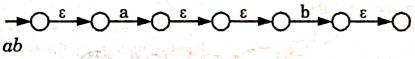
Step 2 : b^*

Step 3 : b

Step 4 : ab^*



Step 5 : ab



Step 6 : $ab^* \mid ab$

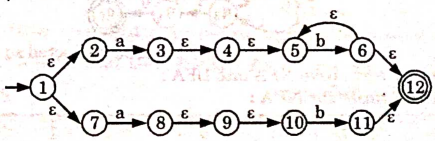


Fig. 1.9.1. NFA of ab* | ab.

Que 1.10. **Discuss conversion of NFA into a DFA. Also give the algorithm used in this conversion.** **AKTU 2017-18, Marks 10**

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'.

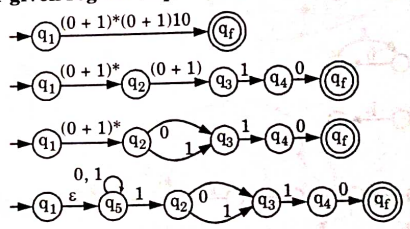
Step 4 : Final state of DFA will be all states which contain F (final states of NFA).

Que 1.11. **Construct the minimized DFA for the regular expression $(0+1)^*(0+1)10$.** **AKTU 2016-17, Marks 10**

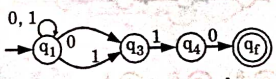
Answer

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

NFA for given regular expression :



If we remove ε we get



[$\because$ ε can be neglected so $q_1 = q_5 = q_2$]

Now, we convert above NFA into DFA :

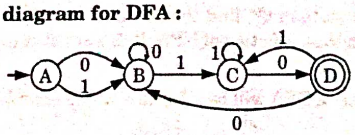
Transition table for NFA :

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 1.12. **How does finite automata useful for lexical analysis ? Construct the NFA and DFA for the following regular expression $(a + b)^* abb$.** **AKTU 2014-15, Marks 05**

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.

NFA for regular expression $(a + b)^* abb$ is :

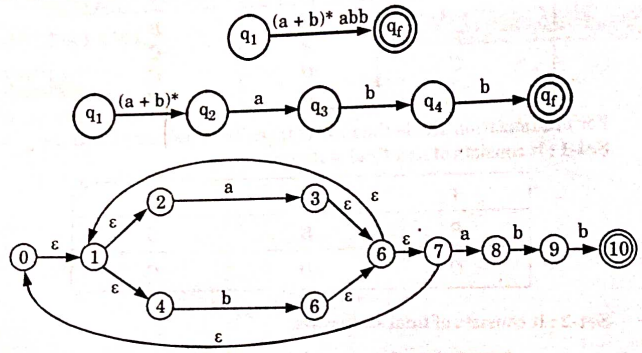


Fig. 1.12.1. NFA accepting (a + b)* abb.

DFA for regular expression $(a + b)^* abb$ is :

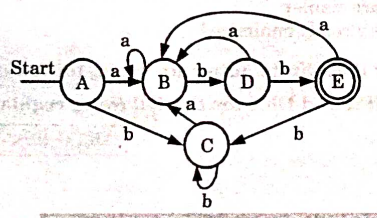


Fig. 1.12.2. DFA accepting (a + b)*abb.

Que 1.13. 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.**

AKTU 2017-18, Marks 10

Answer

1. DFA for all string 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 :**

$$[00(0 + 1) (0 + 1) 0(0 + 1) 0(0 + 1) + 0(0 + 1) (0 + 1)0 + (0 + 1) 00(0 + 1) + (0 + 1)0(0 + 1) 0 + (0 + 1) (0 + 1)00]$$

PART-4

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

CONCEPT OUTLINE

- Lexical analyzer reads the input string from left to right and generates sequence of tokens.
- A LEX source program is a specification of a lexical analyzer.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.14. 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.
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.

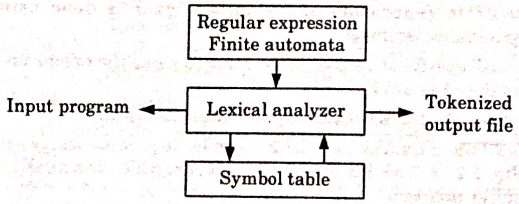


Fig. 1.14.1. Implementation of lexical analyzer.

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 phrase in a programming language which will evaluate the lexemes that match the regular expression.
8. The tool then constructs a state table for the appropriate finite state machine and creates program code which contains the table, the evaluation phrases, and a routine which uses them appropriately.
9. Two more approaches are used to implement lexical analyzer :
   a. Using an automatic generator of lexical analyzers (as LEX or FLEX).
   b. Sometimes it is necessary to write a lexer by hand and lexers are often generated by automated tools.

Que 1.15. 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 much 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 1.16. 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. 1.16.1. This lex.yy.c is a C program which is actually a lexical analyzer program.

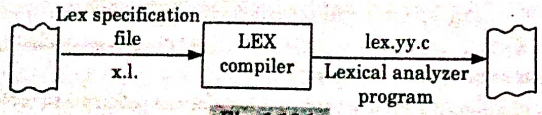


Fig. 1.16.1.

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. The lexemes can be recognized with the help of tabular representation of transition diagram.
7. In specification file, LEX actions are associated with every regular expression.
8. These actions are simply the pieces of C code that are directly carried over to the lex.yy.c.
9. Finally, the C compiler compiles this generated lex.yy.c and produces an object program a.out as shown in Fig. 1.16.2.
10. When some input stream is given to a.out then sequence of tokens gets generated. The described scenario is shown in Fig. 1.16.2.

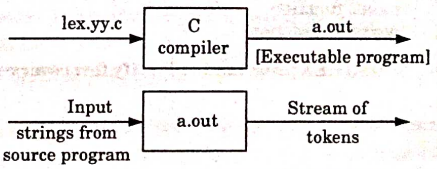


Fig. 1.16.2. Generation of lexical analyzer using LEX.

Que 1.17. What are the three part of LEX program ?

Answer

The LEX program consists of three parts :

```
% {
Declaration section
% }
% %
Rule section
% %
Auxiliary procedure section
```

1. **Declaration section :**
   a. In the declaration section, declaration of variable constants can be done.
   b. Some regular definitions can also be written in this section.
   c. The regular definitions are basically components of regular expressions.
2. **Rule section :**
   a. The rule section consists of regular expressions with associated actions. These translation rules can be given in the form as :

R_1 {action$_1$}
R_2 {action$_2$}
.
.
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.

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

3. **Auxiliary procedure section :**
   a. In this section, all the procedures are defined which are required by the actions in the rule section.
   b. This section consists of two functions :
      i. main() function
      ii. yywrap() function

Que 1.18. **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 1.19. **What are the various LEX action 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() :**
   a. The function yywrap() is called when scanner encounter end of file.
   b. If yywrap() returns 0 then scanner continues scanning.
   c. 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. **yyleng :** yyleng stores the length or number of characters in the input string.

Que 1.20. **Write a LEX program for counting word.**

Answer

LEX program for word counting :

```
% {
int Char_Cnt=0, Word_Cnt=0, Line_Cnt=0;
% }
Word [ ^ \t\n] +
%%
{word} {Word_Cnt++; Char_Cnt+==yyleng;}
\n {Char_Cnt++;Line_Cnt++;}
. {Char_Cnt++;}
%%
main()
{
yylex();
```

```
printf("\nThe character count = %d", Char_Cnt);
printf("\nThe word count = %d", Word_Cnt);
printf("\nThe line count = %d", Line_Cnt);
printf("\n");
}
int yywrap()
{
return 1;}
}
```

OUTPUT :

```
[root@aap root]# lex linecount.l
[root@aap root]# gcc lex.yy.c –ll
[root@app root]# ./a.out
hello Quantum
you are
a newcomer
here
The character count = 31
The word count = 7
The line count = 4
[root@aap root]#
```

Que 1.21. Explain the term token, lexeme and pattern.

AKTU 2013-14, Marks 05

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.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.22. 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 $A, B, C, \ldots\ldots, X, Y, Z$.
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 atleast 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 1.23. 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 tuple and represented by $G(V, T, P, S)$.

   where,

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

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

   $S \rightarrow$ starting symbol.

   $P \rightarrow$ 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 1.24. **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.

PART-6

BNF Notation.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.25. **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 :
   <symbol> : = Exp1 | Exp2 | Exp3...

   Where <symbol> 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

   <Address> : = <fullname> : "," <street> "," <zip code>

   <fullname> : = <firstname> "–" <middle name> "–" <surname>

   <street> : = <street name> "," <city>

   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 : {name}*
   c. For repeating the symbols for atleast one or more number of times + is used.

      For example : {name}+
   d. The alternative rules are separated by vertical bar.
   e. The group of items must be enclosed within brackets.

PART-7

Ambiguity, YACC.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.26. **What do you mean by ambiguous grammar ? Show that the following grammar is ambiguous.**

$S \rightarrow aSbS \mid bSaS \mid \varepsilon$ **AKTU 2013-14, Marks 05**

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.

To prove that G is ambiguous, we have to find $w \in L(G)$ which is ambiguous. Consider a string $w = ababab \in L(G)$.

Then we get two derivation trees for w.

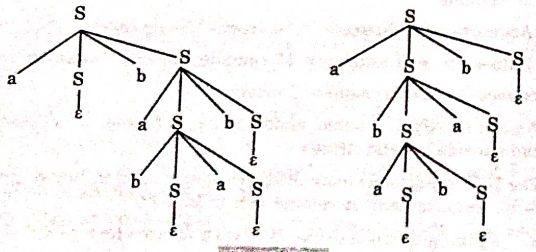


Fig. 1.26.1.

There are two derivation trees for the same string, hence grammar is ambiguous.

Que 1.27. **What is an ambiguous grammar ? Is the following grammar is ambiguous ? Prove $EE+ \mid E(E) \mid id$. The grammar should be moved to the next line, centered.** **AKTU 2016-17, Marks 10**

Answer

Ambiguous grammar : Refer Q. 1.26, Page 1–26B, Unit-1.

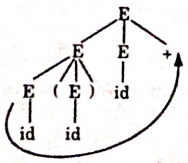
Proof : Let production rule is given as :

$E \rightarrow EE+$

$E \rightarrow E(E)$

$E \rightarrow id$

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



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

Que 1.28. **Write short note on :**

i. Context free grammar

ii. YACC parser generator **AKTU 2014-15, Marks 05**

OR

Write a short note on YACC parser generator.

AKTU 2017-18, Marks 05

Answer

i. Context free grammar : Refer Q. 1.23, Page 1–23B, 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.

Que 1.29. **Consider the grammar G given as follows :**

$S \rightarrow AB \mid aaB$

$A \rightarrow a \mid Aa$

$B \rightarrow b$

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

Answer

Given :

$$S \rightarrow AB \mid aaB$$
$$A \rightarrow a \mid Aa$$
$$B \rightarrow b$$

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

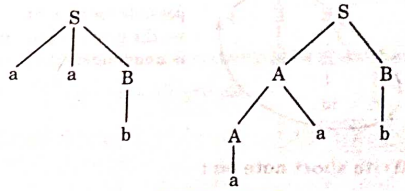


Fig. 1.29.1.

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*.

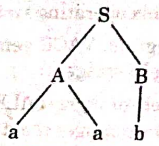


Fig. 1.29.2.

PART-8

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

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.30. **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 $A \rightarrow X_1X_2 \ldots X_K$. Then for a vertex, label *A*, the children of that node, will be $X_1X_2 \ldots X_K$.

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 1.31. **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 $\underset{G}{\Rightarrow}$ and $\overset{*}{\underset{G}{\Rightarrow}}$.
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

$$aab \underset{G}{\Rightarrow} a\beta b.$$

5. We say that the production $\alpha \rightarrow \beta$ is applied to the string $a\alpha b$ to obtain $a\beta b$ or we say that $a\alpha b$ directly drives $a\beta b$.
6. Now suppose $\alpha_1, \alpha_2, \alpha_3, ..., \alpha_m$ are string in $(V_n \cup V_t)^*$, $m \geq 1$ and

   $\alpha_1 \underset{G}{\Rightarrow} \alpha_2, \alpha_2 \underset{G}{\Rightarrow} \alpha_3, \alpha_3 \underset{G}{\Rightarrow} \alpha_4, ..., \alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m$.

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

Que 1.32. **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 $E \rightarrow E * E$)

$\rightarrow id * \underline{E} + E$ (Replace $E \rightarrow id$)

$\rightarrow id * id + \underline{E}$ (Replace $E \rightarrow id$)

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

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

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

$\rightarrow E * E + \underline{E}$ (Replace $E \rightarrow E + E$)

$\rightarrow E * \underline{E} + id$ (Replace $E \rightarrow id$)

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

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

Que 1.33. **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 associatively 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.

2

UNIT

Basic Parsing Techniques

CONTENTS

PART-1

Basic Parsing Techniques : Parsers, Shift Reduce Parsing.

CONCEPT OUTLINE

- **Parsing :** A parser for any grammar is a program that takes as input string w and produces as output a parse tree for w. It is done with the help of two techniques :
  - i. Top-down parsing
  - ii. Bottom-up parsing
- **Bottom-up parsing :** Bottom-up parsing build parse trees from the bottom to the top while parsing the input to the parser is being scanned from left to right, one symbol at a time.

  Shift-Reduce parsing is an example of bottom-up parsing.
- **Shift-reduce parser :** This method is bottom-up as it builds a parse tree start from the leaves and working up towards the root. At each step, a string matching the right side of a production is replaced by the symbol on left side.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

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, which is equivalent to constructing a parse tree for the input string w that start from the root and create the nodes in pre-defined order.
   b. Top-down parsing seeks the left-most derivation for an input string w because 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, every terminal symbol generated by some production of the grammar is matched with the input string symbol pointed by string marker. If the match is successful, the parser can continue.
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".
   e. Bottom-up parsing tries to trace out the right-most derivation and not the left-most derivations because the parser scans the input string w from the left to right, one symbol/token at a time and to trace out right-most derivations of an input string w in reverse, the tokens of w must be made available in a left-to-right order.

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

OR

Write a note on bottom-up parsing.

Answer

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

Bottom-up parsing techniques are :

1. **Shift-reduce parser :**
   a. Shift-reduce parser attempts to construct parse tree from leaves to root and uses a stack to hold grammar symbols.

   b. A parser goes on shifting the input symbols onto the stack until a handle comes on the top of the stack.
   c. When a handle appears on the top of the stack, it performs reduction.

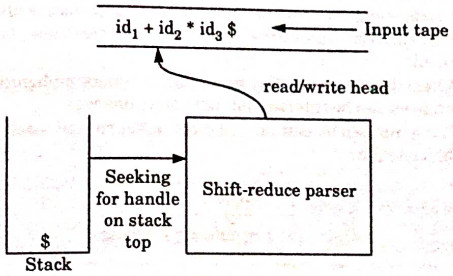


Fig. 2.2.1. Shift-reduce parser.

   d. This parser performs following basic operations :
      i. Shift
      ii. Reduce
      iii. Accept
      iv. 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
   a. L stands for left to right scanning.
   b. R stands for right-most derivation in reverse.
   c. 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.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.4. Explain operator precedence parsing.

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 meaning of these relations is

   $p <\cdot q$ — p gives more precedence than q.
   $p \doteq q$ — p has same precedence as q.
   $p \cdot> q$ — p takes precedence over q.
3. Consider the grammar for arithmetic expressions
   $E \rightarrow EA \mid (E) \mid -E \mid id$
   $A \rightarrow + \mid - \mid * \mid / \mid \wedge$
4. Now consider the string $id + id * id$
5. 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.
   $\$ <\cdot id \cdot> + <\cdot id \cdot> * <\cdot id \cdot> \$$
6. We will follow following steps to parse the given string :

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

   The parsing can be done as follows :

$\$ <\cdot id \cdot> + <\cdot id \cdot> * <\cdot id \cdot> \$$	Handle *id* is obtained between <··>. Reduce this by $E \rightarrow id$.
$E + <\cdot id \cdot> * <\cdot id \cdot> \$$	Handle *id* is obtained between <··>. Reduce this by $E \rightarrow id$.
$E + E * <\cdot id \cdot> \$$	Handle *id* is obtained between <··>. Reduce this by $E \rightarrow id$.
$E + E * E$	Remove all the non-terminals.
$+ *$	Insert \$ at the beginning at the end. Also insert the precedence operators.
$\$ <\cdot + <\cdot * \cdot> \$$	The * operator is surrounded by <··>. 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			>	>	>
(	<	<	=	<	
)			>	>	>
;	<	<	>	>	
$	<	<			

AKTU 2015-16, Marks 10

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 (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.
   An edge or path from f_a to g_b means that $f(a)$ must exceed $g(b)$; a path from g_b to f_a means that $g(b)$ must exceed $f(a)$.
4. If the graph constructed in (3) has a cycle, then no precedence functions exist. If there are no cycles, let $f(a)$ be the length of the longest path beginning at the group of f_a; let $g(b)$ be the length of the longest path from the group of g_b.

Precedence graph for above matrix is :

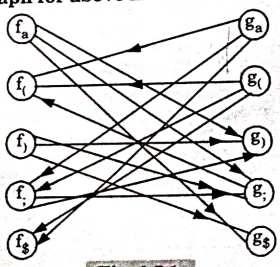


Fig. 2.5.1.

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. **What do you mean by operator precedence grammar ? Compute the operator precedence table for the given grammer :**

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

AKTU 2013-14, Marks 10

Answer

Operator precedence grammar : Refer Q. 2.4, Page 2–5B, Unit-2.

Operator precedence table :

	+	*	(	)	id	$
+	·>	<·	<·	·>	<·	·>
*	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≐	<·	
)	·>	·>		·>		·>
id	·>	·>		·>		·>
$	<·	<·	<·		<·	

Que 2.7. **Give operator precedence parsing algorithm. Consider the following grammar and build up operator precedence table. Also parse the input string (id+(id*id))**

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

AKTU 2017-18, Marks 10

Answer

Operator precedence parsing algorithm :

Let the input string be $a1, a2, \ldots, an$ \$. Initially, the stack contains \$.

1. Set p point to the first symbol of w\$.
2. Repeat forever
3. If only \$ is on the stack and only \$ is the input then accept and break else begin
4. let a be the topmost terminal symbol on the stack and let b be the current input symbol;
5. If $a \lessdot b$ or $a \doteq b$ then shift b onto the stack
6. else if $a \gtrdot b$ then reduce b from the stack
7. Repeat pop stack
8. Until the top stack terminal is related by $\lessdot$ to the terminal most recently popped.
9. else call the error correcting routine end.

Operator precedence table :

	+	*	(	)	id	$
+	·>	<·	<·	·>	<·	·>
*	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≐	<·	
)	·>	·>		·>		·>
id	·>	·>		·>		·>
$	<·	<·	<·		<·	

Parsing :

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

PART-3

Top-down Parsing, Predictive Parsers.

CONCEPT OUTLINE

- FIRST and FOLLOW are used to construct predictive parser.
- Recursive-descent parser executes a set of recursive procedure to process the input without backtracking.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.8. **What are the problems with top-down parsing ?**

Answer

Problems with top-down parsing are :

1. **Backtracking :**
   a. 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 :

$$A \overset{+}{\Rightarrow} A\,\alpha$$

   b. Here $\overset{+}{\Rightarrow}$ 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.

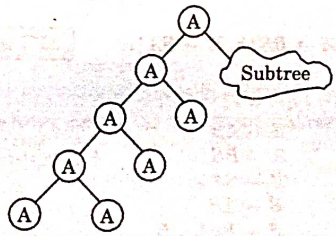


Fig. 2.8.1. Left recursion.

   e. This causes major problem in top-down parsing and therefore elimination of left recursion is must.
3. **Left factoring :**
   a. If the grammar is left factored then it becomes suitable for the use.
   b. Left factoring is used when it is not clear that which of the two alternatives is used to expand the non-terminal.

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

Answer

Left factoring : Refer Q. 2.8, Page 2–9B, Unit-2.

Left factoring can be eliminated by the following scheme :

a. In general if

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_n$$

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

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 | \beta_2 | \ldots | \beta_{n-1} | \beta_n$$

Que 2.10. **Remove left recursion from the grammar**

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

AKTU 2013-14, Marks 05

Answer

Eliminating immediate left recursion among all productions, we get :

$E \rightarrow TE'$
$E' \rightarrow (T)E' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow (F)T' \mid \varepsilon$
$F \rightarrow id$

Que 2.11. **Eliminate left recursion from the following grammar**

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

AKTU 2017-18, Marks 10

Answer

$S \rightarrow AB$
$A \rightarrow BS \mid b$
$B \rightarrow SA \mid a$
$S \rightarrow AB$
$S \rightarrow BSB \mid bB$
$S \rightarrow \underbrace{S}_{A}\underbrace{ASB}_{\alpha} \mid \underbrace{aSB}_{\beta_1} \mid \underbrace{bB}_{\beta_2}$
$S \rightarrow aSBS' \mid bBS'$
$S' \rightarrow ASBS' \mid \varepsilon$
$B \rightarrow ABA \mid a$
$B \rightarrow \underbrace{B}_{\alpha}\underbrace{ABBA}_{\alpha} \mid \underbrace{bBA}_{\beta_1} \mid \underbrace{a}_{\beta_2}$
$B \rightarrow bBA\,B' \mid aB'$
$B' \rightarrow ABBA\,B' \mid \varepsilon$
$A \rightarrow BS \mid a$
$A \rightarrow SAS \mid aS \mid a$

$$A \rightarrow \underbrace{A}_{\alpha}\underbrace{BAAB}_{\alpha} \mid \underbrace{aAB}_{\beta_1} \mid \underbrace{a}_{\beta_2}$$
$$A \rightarrow aAB\,A' \mid aA'$$
$$A' \rightarrow BAAB\,A' \mid \varepsilon$$

The production after left recursion is

$$S \rightarrow aSB\,S' \mid bBS'$$
$$S' \rightarrow ASB\,S' \mid \varepsilon$$
$$A \rightarrow aABA' \mid aA'$$
$$A' \rightarrow BAABA' \mid \varepsilon$$
$$B \rightarrow bBA\,B' \mid aB'$$
$$B' \rightarrow ABBA\,B' \mid \varepsilon$$

Que 2.12. **Check whether left recursion exists for the following grammar :**

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid e$$

AKTU 2014-15, Marks 05

Answer

$$S \rightarrow Aa \mid b$$
$$A \rightarrow \underbrace{\underbrace{A}_{A}\underbrace{c}_{\alpha}}_{\alpha} \mid \underbrace{Sd}_{\beta_1} \mid \underbrace{e}_{\beta_2}$$ ← Left recursion exist
$$A \rightarrow SdA' \mid eA'$$
$$A \rightarrow cA' \mid \varepsilon$$

The production after removing left recursion will be :

$$S \rightarrow Aa \mid b$$
$$A \rightarrow SdA' \mid eA'$$
$$A \rightarrow cA' \mid \varepsilon$$

Que 2.13. **Write short notes on top-down parsing. What are top-down parsing techniques ?**

Answer

Top-down parsing : Refer Q. 2.1, Page 2–2B, 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.

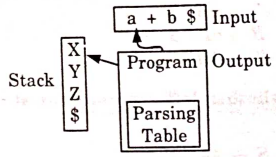


Fig. 2.13.1. Model of a predictive parser.

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

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 :

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

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

ic. If X is a non-terminal, the program consults entry $M[X, a]$ in the parsing table M.

d. If X is a terminal not equal to a then parser announces an error.

e. If $M[X, a]$ = error, the parser calls an error recovery routine.

Que 2.14. 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 because of following condition :

1. Every grammar is recursive in nature.
2. Each grammar must be left factored.

Que 2.15. Write short notes on the following :

1. **FIRST function algorithm**
2. **FOLLOW function algorithm**

Answer

1. **FIRST function :**

i. FIRST(α) is a set of terminal symbols that are first symbols appearing at R.H.S. in derivation of α. If $\alpha \Rightarrow \varepsilon$ then ε is also in FIRST(α).

ii. Following are the rules used to compute the FIRST functions.

a. If the terminal symbol a the FIRST(a) = {a}.

b. If there is a rule $X \rightarrow \varepsilon$ then FIRST(X) = {ε}.

c. For the rule $A \rightarrow X_1X_2X_3 \ldots X_k$,
FIRST(A) = (FIRST(X_1) ∪ FIRST(X_2) ∪ FIRST(X_3)).... ∪ FIRST(X_k)).

2. **FOLLOW function :**

i. FOLLOW(A) is defined as the set of terminal symbols that appear immediately to the right of A.

ii. FOLLOW(A) = {$a \mid S \overset{*}{\Rightarrow} \alpha A a \beta$ where α and β 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 FIRST(β) 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 FIRST(B) = {ε} then FOLLOW(A) = FOLLOW(B) or FOLLOW(B) = FOLLOW(A). That means everything in FOLLOW(A) is in FOLLOW(B).

Que 2.16. **Explain predictive parsing techniques. Also write the algorithm for construction of predictive parsing table.**

Answer

Predictive parsing : Refer Q. 2.13, Page 2–12B, Unit-2.

Algorithm for construction of a predictive parsing table :

Input : Grammar G.

Output : Parsing table M.

Method :

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

Que 2.17. **Differentiate between recursive descent parsing and predictive parsing. Derive the LL(1) parsing table for the following grammar**

bexpr → bexpr or bterm | bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false.

AKTU 2013-14, Marks 10

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.

bexpr → bexpr or bterm | bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false

Remove the left recursion
bexpr → bterm bexpr'
bexpr' → ar bnterm bexpr' | ε
bterm → bfactor bterm'
bterm → and bfactor bterm' | ε
bfactor → not befactor | (bexpr) | true | false
Let denote the given expression in short form as :
bexpr as E
bexper' as E'
bterm as T
bterm' as T'
bfactor as F
or as +
and as *
not as ~
true as 1
false as 0
Now the expression will become as :
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow \sim F \mid (E) \mid 1 \mid 0$
FIRST(E) = FIRST(T) = FIRST(F) = {~, (, 1, 0}
FIRST(E') = {+, E}
FIRST(T') = {*, E}
FOLLOW(E) = {), $}
FOLLOW(E') = {), $}
FOLLOW(T) = {+,), $}
FOLLOW(T') = {+,), $}
FOLLOW(F) = {*,), $, +}
LL(1) parsing table :

	~	+	*	(	)	$	0	1
E				E → TE'			E → TE'	E → TE'
E'	E' → +TE'				E' → ε	E' → ε		
T	T → FT'			T → FT'			T → FT'	T → FT'
T'		T' → ε	T' → *FT'		T' → ε	T' → ε		
F	F → (E)			F → ~F			F → 0	F → 1

Que 2.18. **Explain non-recursive predictive parsing. Consider the following grammar and construct the predictive parsing table**

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

$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow F^* \mid a \mid b$

AKTU 2017-18, Marks 10

Answer

Non-recursive descent parsing (Predictive parsing) : Refer Q. 2.13, Page 2–12B, Unit-2.

Numerical :

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow F^* \mid a \mid b$

First we remove left recursion

$F \rightarrow F\underset{\alpha}{*} \mid \underset{\beta_1}{a} \mid \underset{\beta_2}{b}$

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

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

Predictive parsing table :

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

Que 2.19. **Give algorithm for construction of predictive parsing table. Consider the following grammar and construct predictive parsing table**

$S \rightarrow iEtSS_1 / a$
$S_1 \rightarrow eS / \varepsilon$
$E \rightarrow b$

AKTU 2014-15, Marks 10

Answer

Algorithm for construction of predictive parsing table : Refer Q. 2.16, Page 2–15B, Unit-2.

Numerical : Given grammar :

$S \rightarrow iEtSS_1 / a$
$S_1 \rightarrow eS / \varepsilon$
$E \rightarrow b$

Now we will compute FIRST and FOLLOW for given non-terminals :

FIRST (S) = { i, a }
FIRST (S_1) = { e, ε }
FIRST (E) = { b }
FOLLOW (S) = FOLLOW (S_1) = { e, $ }
FOLLOW (E) = { t }

The predictive parsing table will be

Table 2.19.1.

	a	b	e	i	t	$
S	$S \rightarrow a$			$S \rightarrow iEtSS_1$		
S_1			$S_1 \rightarrow eS$ $S_1 \rightarrow \varepsilon$			$S_1 \rightarrow \varepsilon$
E		$E \rightarrow b$				

Multiple entries show that grammar is ambiguous. So it is not LL(1).

PART-4

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

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.20. **Discuss properties of LR parser and working of LR parser with its block diagram.**

OR

Explain the working of LR parser.

Answer

Properties of LR parser :

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.

Working of LR parser :

1. The working of LR parser can be understood by using block diagram as shown in Fig. 2.20.1.

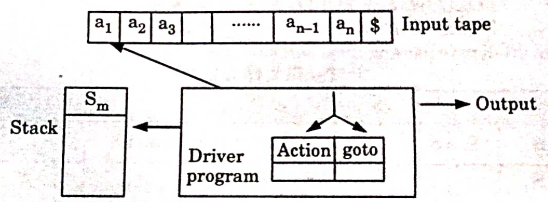


Fig. 2.20.1.

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

   $S_0X_1S_1X_2S_2.....X_{m-1}S_{m-1}X_mS_m$

   where X_i is a grammar symbol, each S_i is the state and S_m state is top of the stack.
7. The action of the driver program depends on action $[S_m, a_i]$ where a_i is the current input symbol.
8. Following action are possible for input $a_ia_{i+1}.....a_n$:

   a. **Shift :** If action $[S_m, a_i]$ = 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 $S_0X_1S_1X_2.....X_{m-r}S_{m-r}X_m$ **Input** $S_ma_iS, a_{i+1}, a_{i+2}.....a_n\$$

   b. **Reduce :** If action $[S_m, a_i]$ = 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 $S_0X_1S_1X_2.....X_{m-r}S_{m-r}AS$ **Input** $a_ia_{i+1}.....a_n\$$

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

   i. If action $[S_m, a_i]$ = accept, parsing is completed.

   ii. If action $[S_m, a_i]$ = error, the parser has discovered a syntax error.

Que 2.21. 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 • 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 $S \rightarrow \varepsilon$ 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 • 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 items.

PART-5

Constructing SLR Parsing Tables.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.22. **Explain SLR parsing techniques. Also write the algorithm to construct SLR parsing table.**

Answer

The SLR parsing can be done as :

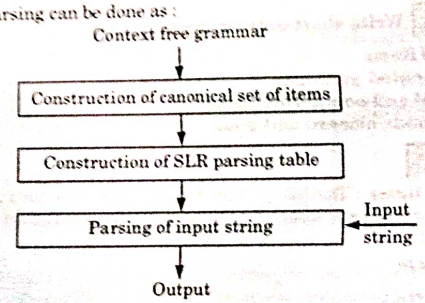


Fig. 2.22.1. Working of SLR parser.

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 $C = \{I_0, I_1, \ldots\ldots, I_n\}$. The states of the parser are 0, 1....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 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".

If any conflicting actions are generated by the given rules then the grammar is not SLR (1) and fails to produce a valid parser in this case.

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

4. If GOTO $(I_i, A) = I_j$ then 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 2.23. **Construct an SLR(1) parsing table for the following grammar :**

$S \rightarrow A)$

$S \rightarrow A, P \mid (P, P$

$P \rightarrow$ {num, num}

AKTU 2015-16, Marks 10

Answer

The augmented grammar G' for the above grammar G is :

$S' \rightarrow S$

$S \rightarrow A)$

$S \rightarrow A, P$

$S \rightarrow (P, P$

$P \rightarrow$ {num, num}

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

I_0 : $S' \rightarrow \bullet S$
$S \rightarrow \bullet A)$
$S \rightarrow \bullet A, P$
$S \rightarrow \bullet (P, P$
$P \rightarrow \bullet$ {num, num}

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

$I_2 = \text{GOTO}(I_0, A)$
I_2 : $S \rightarrow A \bullet)$
$S \rightarrow A \bullet, P$

$I_3 = \text{GOTO}(I_0, ($
I_3 : $S \rightarrow (\bullet P, P$
$P \rightarrow \bullet$ {num, num}

$I_4 = \text{GOTO}(I_0, \{)$
I_4 : $P \rightarrow \{\bullet$ num, num}

$I_5 = \text{GOTO}(I_2,))$
I_5 : $S \rightarrow A) \bullet$

$I_6 = \text{GOTO}(I_2, ,)$
I_6 : $S \rightarrow A, \bullet P$
$P \rightarrow \bullet$ {num, num}

$I_7 = \text{GOTO}(I_3, P)$
I_7 : $S \rightarrow (P \bullet, P$

I_8 = GOTO (I_4, num)
I_8 : $P \rightarrow$ {num •, num}
I_9 = GOTO (I_6, P)
I_9 : $S \rightarrow A, P$•
I_{10} = GOTO (I_7, ,)
I_{10} : $S \rightarrow$ (P, •P
I_{11} = GOTO (I_8, ,)
I_{11} : $P \rightarrow$ {num, • num}
I_{12} = GOTO (I_{10}, P)
$S \rightarrow$ (P , P •
I_{13} = GOTO (I_{11}, num)
I_{13} : $P \rightarrow$ {num, num •}
I_{14} = GOTO (I_{13} ,})
I_{14} : $P \rightarrow$ {num, num} •

	Action							Goto		
Item Set	)	,	(	{	Num	}	$	S	A	P
0			S_3					1	2	
1							accept			
2	S_4	S_5								
3				S_7						6
4	r_1	r_1	r_1	r_1	r_1	r_1	r_1			
5				S_7						8
6		S_9								
7					S_{10}					
8	r_2	r_2	r_2	r_2	r_2	r_2	r_2			
9				S_7						11
10		S_{12}								
11	r_3	r_3	r_3	r_3	r_3	r_3	r_3			
12					S_{13}					
13						S_{14}				
14	r_4	r_4	r_4	r_4	r_4	r_4	r_4			

Que 2.24. **Consider the following grammar**

$S \rightarrow AS \mid b$
$A \rightarrow SA \mid a$

Construct the SLR parse table for the grammar. Show the actions of the parser for the input string "*abab*". **AKTU 2016-17, Marks 15**

Answer

The augmented grammar is :

$S' \rightarrow S$
$S \rightarrow AS \mid b$
$A \rightarrow SA \mid a$

The canonical collection of $LR(0)$ items are

$I_0 : S' \rightarrow \bullet S$
$S \rightarrow \bullet AS \mid \bullet b$
$A \rightarrow \bullet SA \mid \bullet a$
I_1 = GOTO (I_0, S)
$I_1 : S' \rightarrow S \bullet \quad S \rightarrow AS \mid \bullet b$
$A \rightarrow S \bullet A$
$A \rightarrow \bullet SA \mid \bullet a$
I_2 = GOTO (I_0, A)
$I_2 : S \rightarrow A \bullet S$
$S \rightarrow \bullet AS \mid \bullet b$
$A \rightarrow \bullet SA \mid \bullet a$
I_3 = GOTO (I_0,b)
$I_3 : S \rightarrow b \bullet$
I_4 = GOTO (I_0, a)
$I_4 : A \rightarrow a \bullet$
I_5 = GOTO (I_1, A)
$I_5 : A \rightarrow SA \bullet$
I_6 = GOTO (I_1, S) = I_1
I_7 = GOTO (I_1, a) = I_4
I_8 = GOTO (I_2, S)
$I_8 : S \rightarrow AS \bullet$
I_9 = GOTO (I_2, A) = I_2
I_{10} = GOTO (I_2, b) = I_3

Let us numbered the production rules in the grammar as :

1. $S \rightarrow AS$
2. $S \rightarrow b$
3. $A \rightarrow SA$
4. $A \rightarrow a$

FIRST(S) = FIRST(A) = {a, b}
FOLLOW(S) = {$, a, b}
FOLLOW(A) = {a, b}

Table : 2.24.1. SLR parsing table.

States	Action			Goto	
	a	b	$	S	A
I_0	S_4	S_3		1	2
I_1	S_4	S_3	accept		5
I_2	S_4	S_3		8	2
I_3	r_2	r_2	r_2		
I_4	r_4	r_4			
I_5	r_3	r_3	r_3		
I_8	r_1	r_1	r_1		

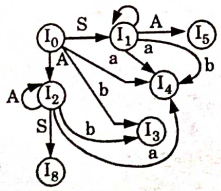


Fig. 2.24.1. DFA for set of items.

Table 2.24.2 : Parse the input *abab* using parse table.

Stack	Input buffer	Action
$0	*abab*$	Shift
$0*a*4	*bab*$	Reduce $A \rightarrow a$
$0A2	*bab*$	Shift
$0A2*b*3	*ab*$	Reduce $S \rightarrow b$
$0A2S8	*ab*$	Shift $S \rightarrow AS$
$0S1	*ab*$	Shift
$0S1*a*4	*b*$	Reduce $A \rightarrow a$
$0S1A5	*b*$	Reduce $A \rightarrow AS$
$0A2	*b*$	Shift
$0A2*b*3	$	Reduce $S \rightarrow b$
$0A2S8	$	Reduce $S \rightarrow AS$
$0S1	$	Accept

Que 2.25. **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.** **AKTU 2017-18, Marks 10**

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 = GOTO (I_0, E)

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

I_2 = 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 = GOTO (I_0, id)

I_3 : $E \rightarrow id\bullet$

I_4 = 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 = 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 = GOTO (I_2, E)
I_6: $E \to (E\bullet)$
$E \to E\bullet + E$
$E \to E\bullet * E$

I_7 = GOTO (I_4, E)
I_7: $E \to E + E\bullet$
$E \to E\bullet + E$
$E \to E\bullet * E$

I_8 = GOTO (I_5, E)
I_8: $E \to E * E\bullet$
$E \to E\bullet + E$
$E \to E\bullet * E$

I_9 = GOTO (I_6,))
I_9: $E \to (E)\bullet$

	Action						Goto
State	id	+	*	(	)	$	E
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	

PART-6

Constructing Canonical LR Parsing Tables.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.26. 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 :

1. Construct $C' = \{I_0, I_1, \ldots, I_n\}$ the collection of sets of LR (1) items of G'.
2. State i of the parser is constructed from I_i. The parsing actions for state i are determined as follows :
   a. If $[A \to \alpha \bullet a\beta, b]$ is in I_i and GOTO $(I_i, a) = I_j$, then set ACTION $[i, a]$ to "shift j". Here, a is required to be a terminal.
   b. If $[A \to \alpha\bullet, a]$ is in I_i, $A \neq S'$, then set ACTION $[i, a]$ to "reduce $A \to \alpha\bullet$".
   c. If $[S' \to S\bullet, \$]$ is in I_i, then set ACTION $[i, \$]$ to "accept".
   If a conflict results from the given rules, the grammar is said not to be LR (1), and algorithm is said to be fail.
3. The goto transitions for state i are determined as follows :
   If GOTO $(I_i, A) = I_j$ then GOTO $[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made "error".
5. The initial state of parser is the one constructed from the set containing items $[S' \to \bullet S, \$]$.
   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.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.27. 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 $C = \{I_0, I_j, \ldots\ldots, I_n\}$ 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.

3. Let $C' = \{J_0, J_1, \ldots\ldots J_m\}$ 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 algorithms 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.*, $J = I_1 \cup I_2 \cup I_3 \ldots\ldots \cup I_k$, then the cores of the GOTO (I_1, X), GOTO (I_2, X),....., GOTO (I_k, X) are the same. Since I_1, I_2,....., I_k all have the same core. Let k be the union of all sets of the items having the same core as GOTO (I_1, X). Then 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 2.28. **For the grammar $S \to aAd \mid bBd \mid aBe \mid bAe\ A \to f, B \to f$ Construct LR(1) parsing table. Also draw the LALR table from the derived LR(1) parsing table.** **AKTU 2017-18, Marks 10**

Answer

Augmented grammar :

$S' \to S$
$S \to aAd \mid bBd \mid aBe \mid bAe$
$A \to f$
$B \to f$

Canonical collection of LR(1) grammar :

$I_0 : S' \to \bullet S, \$$
$S \to \bullet aAd, \$$
$S \to \bullet bBd, \$$
$S \to \bullet aBe, \$$
$S \to \bullet bAe, \$$
$A \to \bullet f, d/e$
$B \to \bullet f, d/e$
$I_1 := \text{GOTO}(I_0, S)$
$I_1 : S' \to \bullet S, \$$
$I_2 := \text{GOTO}(I_0, a)$
$I_2 : S \to a \bullet Ad, \$$
$S \to aBe, \$$
$A \to \bullet f, d$
$B \to \bullet f, e$
$I_3 := \text{GOTO}(I_0, b)$
$I_3 : S \to b \bullet Bd, \$$
$S \to b \bullet Ae, \$$
$A \to \bullet f, d$
$B \to \bullet f, e$
$I_4 := \text{GOTO}(I_2, A)$

$I_4 : S \to aA \bullet d, \$$
$I_5 := \text{GOTO}(I_2, B)$
$I_5 : S \to aB \bullet d, \$$
$I_6 := \text{GOTO}(I_2, f)$
$I_6 : A \to f \bullet, d$
$B \to f \bullet, e$
$I_7 := \text{GOTO}(I_3, B)$
$I_7 : S \to bB \bullet d, \$$
$I_8 := \text{GOTO}(I_3, A)$
$I_8 : S \to bA \bullet e, \$$
$I_9 := \text{GOTO}(I_3, f)$
$I_9 : A \to f \bullet, d$
$B \to f \bullet, e$
$I_{10} := \text{GOTO}(I_4, d)$
$I_{10} : S \to aAd \bullet, \$$
$I_{11} := \text{GOTO}(I_5, d)$
$I_{11} : S \to aBd \bullet, \$$
$I_{12} := \text{GOTO}(I_7, d)$
$I_{12} : S \to bBd \bullet, \$$
$I_{13} := \text{GOTO}(I_8, e)$
$I_{13} : S \to bAe \bullet, \$$

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			

Que 2.29. **Construct the CLR parse table for the following grammar.** **AKTU 2014-15, Marks 10**

$S \rightarrow CC, C \rightarrow cC, C \rightarrow d$

Answer

The given grammar is :

$S \rightarrow CC$

$C \rightarrow cC/d$

The augmented grammar will be :

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC/d$

The LR (1) items will be :

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

$S \rightarrow \bullet CC, \$$

$C \rightarrow \bullet cC, c/d$

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

I_1 = GOTO (I_0, S)

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

I_2 = GOTO (I_0, C)

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

$C \rightarrow \bullet cC, \$$

$C \rightarrow \bullet d, \$$

I_3 = GOTO (I_0, c)

$I_3 : C \rightarrow c\bullet C, c/d$

$C \rightarrow \bullet cC, c/d$

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

I_4 = GOTO (I_0, d)

$I_4 : C \rightarrow d\bullet, c/d$

I_5 = GOTO (I_2, C)

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

I_6 = GOTO (I_2, c)

$I_6 : C \rightarrow c\bullet C, \$$

$C \rightarrow \bullet d, \$$

I_7 = GOTO (I_2, d)

$I_7 : C \rightarrow d\bullet, \$$

I_8 = GOTO (I_3, C)

$I_8 : C \rightarrow cC\bullet, c/d$

I_9 = GOTO (I_6, C)

$I_9 : C \rightarrow cC\bullet, \$$

The goto table will be

Table 2.29.1.

State	Action			Goto	
	c	d	$	S	C
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		

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

$C \rightarrow \bullet cC, c/d/\$$

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

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

$I_{89} : C \rightarrow cC\bullet, c/d/\$$ and LALR table will be :

Table 2.29.2.

State	Action			Goto	
	c	d	$	S	C
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		

DFA :

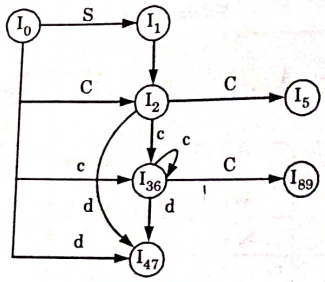


Fig. 2.29.1.

Que 2.30. **Show that the following grammar**

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$
$A \rightarrow d$
$B \rightarrow d$

is *LR*(1) but not *LALR* (1). **AKTU 2013-14, 2015-16; Marks 10**

Answer

Augmented grammar G' for the given grammar :

$S' \rightarrow S$
$S \rightarrow Aa$
$S \rightarrow bAc$
$S \rightarrow Bc$
$S \rightarrow bBa$
$A \rightarrow d$
$B \rightarrow d$

Canonical collection of sets of *LR*(0) items for grammar are as follows :

$I_0 : S' \rightarrow \bullet S, \$$
$S \rightarrow \bullet Aa, \$$
$S \rightarrow \bullet bAc, \$$
$S \rightarrow \bullet Bc, \$$
$S \rightarrow \bullet bBa, \$$
$A \rightarrow \bullet d, a$
$B \rightarrow \bullet d, c$

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

$I_2 = \text{GOTO}(I_0, A)$
$I_2 : S \rightarrow A\bullet a, \$$

$I_3 = \text{GOTO}(I_0, b)$
$I_3 : S \rightarrow b\bullet Ac, \$$
$S \rightarrow b\bullet Ba, \$$
$A \rightarrow \bullet d, c$

$B \rightarrow \bullet d, a$

$I_4 = \text{GOTO}(I_0, B)$
$I_4 : S \rightarrow B\bullet c, \$$

$I_5 = \text{GOTO}(I_0, d)$
$I_5 : A \rightarrow d\bullet, a$
$B \rightarrow d\bullet, c$

$I_6 = \text{GOTO}(I_0, a)$
$I_6 : S \rightarrow Aa\bullet, \$$

$I_7 = \text{GOTO}(I_3, A)$
$I_7 : S \rightarrow bA\bullet c, \$$

$I_8 = \text{GOTO}(I_3, B)$
$I_8 : S \rightarrow bB\bullet a, \$$

$I_9 = \text{GOTO}(I_3, d)$
$I_9 : A \rightarrow d\bullet, c$
$B \rightarrow d\bullet, a$

$I_{10} = \text{GOTO}(I_4, c)$
$I_{10} : S \rightarrow Bc\bullet, \$$

$I_{11} = \text{GOTO}(I_7, c)$
$I_{11} : S \rightarrow bAc\bullet, \$$

$I_{12} = \text{GOTO}(I_8, a)$
$I_{12} : S \rightarrow bBa\bullet, \$$

The action/goto table will be designed as follows :

Table 2.30.1.

State	Action					Goto		
	a	*b*	*c*	*d*	*$*	*S*	*A*	*B*
0		S_3		S_5		1	2	4
1					accept			
2	S_6							
3				S_9			7	8
4			S_{10}					
5	r_5		r_6					
6					r_1			
7			S_{11}					
8	S_{12}							
9	r_6		r_5					
10					r_3			
11					r_2			
12					r_4			

Since the table does not have any conflict, it is LR(1).
For LALR(1) table, item set 5 and item set 9 are same. Thus we merge both the item sets (I_5, I_9) = item set I_{59}. Now, the resultant parsing table becomes :

Table 2.30.2.

State	Action					Goto		
	a	*b*	*c*	*d*	$	*S*	*A*	*B*
0		S_3		S_{59}		1	2	4
1					accept			
2	S_6							
3				S_{59}			7	8
4			S_{10}					
59	r_{59}, r_6		r_6, r_{59}					
6					r_1			
7			S_{11}					
8	S_{12}							
10					r_3			
11					r_2			
12					r_4			

Since the table contains reduce-reduce conflict, it is not LALR(1).

Que 2.31. Construct the LALR parsing table for following grammar :

$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

is LR (1) but not LALR(1). **AKTU 2015-16, Marks 10**

Answer

The given grammar is :

$S \rightarrow AA$
$A \rightarrow aA \mid b$

The augmented grammar will be :

$S' \rightarrow S$
$S \rightarrow AA$

$A \rightarrow aA \mid b$

The LR (1) items will be :

$I_0 : S' \rightarrow \bullet S, \$$
$S \rightarrow \bullet AA, \$$
$A \rightarrow \bullet aA, a/b$
$A \rightarrow \bullet b, a/b$

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

$I_2 = \text{GOTO}(I_0, A)$
$I_2 : S \rightarrow A\bullet A, \$$
$A \rightarrow \bullet aA, \$$
$A \rightarrow \bullet b, \$$

$I_3 = \text{GOTO}(I_0, a)$
$I_3 : A \rightarrow a\bullet A, a/b$
$A \rightarrow \bullet aA, a/b$
$A \rightarrow \bullet b, a/b$

$I_4 = \text{GOTO}(I_0, b)$
$I_4 : A \rightarrow b\bullet, a/b$

$I_5 = \text{GOTO}(I_2, A)$
$I_5 : S \rightarrow AA\bullet, \$$

$I_6 = \text{GOTO}(I_2, a)$
$I_6 : A \rightarrow a\bullet A, \$$
$A \rightarrow \bullet aA, \$$
$A \rightarrow \bullet b, \$$

$I_7 = \text{GOTO}(I_2, b)$
$I_7 : A \rightarrow b\bullet, \$$

$I_8 = \text{GOTO}(I_3, A)$
$I_8 : A \rightarrow aA\bullet, a/b$

$I_9 = \text{GOTO}(I_6, A)$
$I_9 : A \rightarrow aA\bullet, \$$

The goto table will be

Table 2.31.1.

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		

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

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

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

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

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

Table 2.31.2.

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		

DFA :

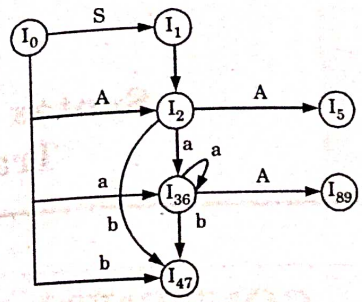


Fig. 2.31.1.

☺☺☺

3

UNIT

Syntax Directed Translation

CONTENTS

PART-1

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

CONCEPT OUTLINE

- Syntax directed definition/translation is a generalization of context free grammar.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

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. **AKTU 2013-14, Marks 10**

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 $a := f(b_1, b_2, \ldots b_k)$, 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 scheme is useful because it enables the compiler designer to express the generation of intermediate code directly in terms of the syntactic structure of the source language.
6. The syntax directed translation is partitioned into two subsets called the synthesized and inherited attributes of grammar.

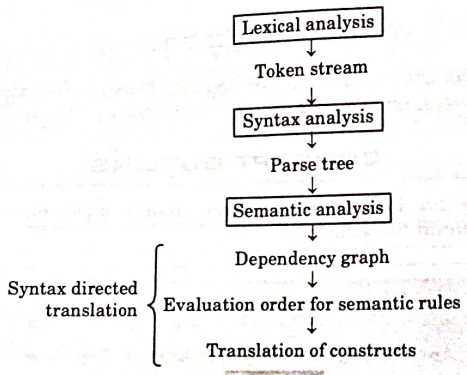


Fig. 3.1.1.

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

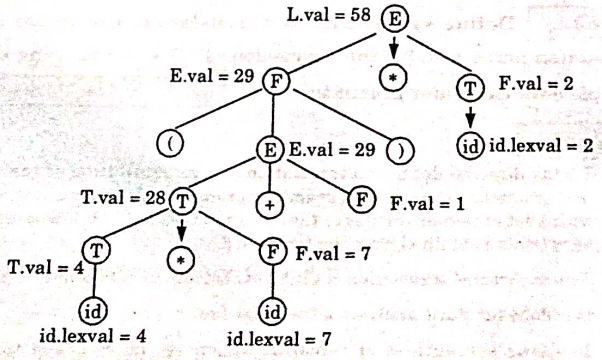


Fig. 3.2.1.

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

AKTU 2014-15, Marks 10

Answer

Syntax directed translation : Refer Q. 3.1, Page 3–2B, 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.

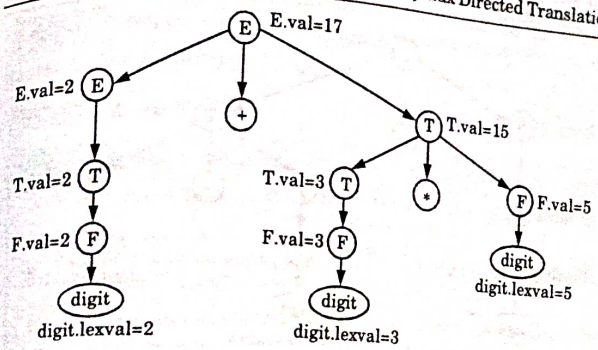


Fig. 3.2.1.

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.

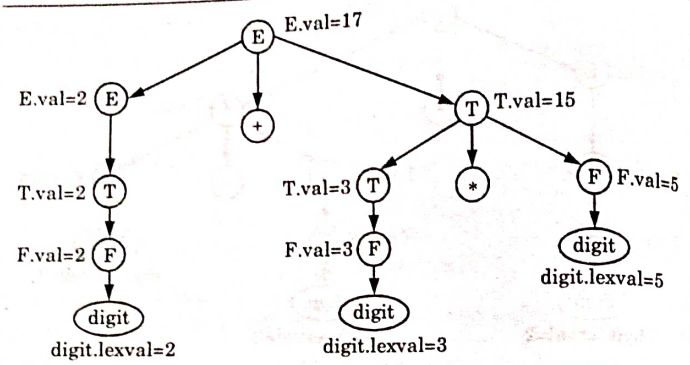


Fig. 3.3.1. An annotated parse tree for expression 2 + 3*5.

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.
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 :

$D \rightarrow TL$	L.type : = T.Type
$T \rightarrow$ int	T.type : = integer
$T \rightarrow$ real	T.type : = real
$L \rightarrow L_1, id$	L_1.type : = L.type
	enter (id.prt, L.type)
$L \rightarrow id$	enter (id.prt, L.type)

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

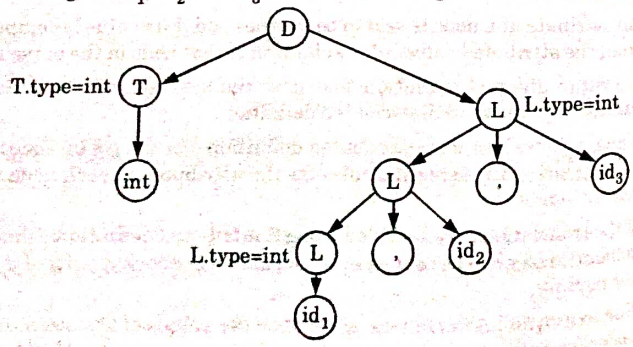


Fig. 3.3.2. Parse tree with inherited attributes for the string int id_1, id_2, id_3.

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.

PART-2

Intermediate Code.

CONCEPT OUTLINE

- Three address code is sequence of statements of the form $X = Y$ op Z where X and Y are operands and op is operator.
- A syntax tree is condensed form of parse tree.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.5. **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.

PART-3

Postfix Notation, Parse Trees and Syntax Trees.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.6. **What is postfix translation ? Explain it with suitable example.** **AKTU 2014-15, Marks 10**

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 calculate as :

$(20 + t_1 * 6 + 12)$	$t_1 = 5 -$
$20 + t_2 + 12$	$t_2 = t_1 6^*$
$t_3 + 12$	$t_3 = 20\ t_2 +$
t_4	$t_4 = t_3\ 12 +$

Now putting values of t_4, t_3, t_2, t_1

$$t_4 = t_3\ 12 +$$
$$20\ t_2 + 12 +$$
$$20\ t_1\ 6 * + 12 +$$
$$(20)\ 5 - 6 * + 12 +$$

Que 3.7. **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 :

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 $A \rightarrow X_1X_2 \ldots X_K$. Then for a vertex, label A, the children node, will be $X_1X_2 \ldots X_K$.
v. 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 3.8. **What is syntax tree ? What are the rules to construct syntax tree for an expression ?**

Answer

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.

Rules for constructing a syntax tree for an expression :

1. Each node in a syntax tree can be implemented as a record with several fields.
2. In the node for an operator, one field identifies the operator and the remaining field contains pointer to the nodes for the operands.
3. The operator often is called the label of the node.
4. 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.
   a. **Mknode(op, left, right) :** It creates an operator node with label op and two field containing pointers to left and right.
   b. **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.
   c. **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, $p_1, p_2, \ldots\ldots\ldots, p_5$ 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 := mkleaf (*id*, entry *a*);
p_2 := mkleaf (num, 4);
p_3 := mknode ('–', p_1, p_2);
p_4 := mkleaf (*id*, entry *c*);

p_5 := mknode ('+', p_3, p_4);

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. The call mknode ('–', p_1, p_2) then constructs the interior node with the leaves for *a* and 4 as children. The syntax tree will be :

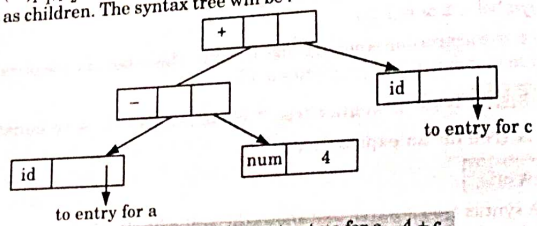


Fig. 3.8.1. The syntax tree for a – 4 + c.

Que 3.9. 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

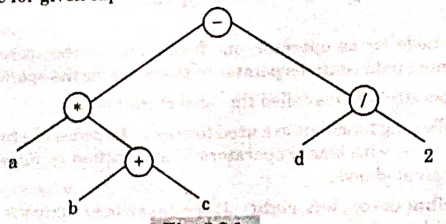


Fig. 3.9.1.

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

$t_1 = bc +$	$(a * t_1 - d/2)$
$t_2 = a\ t_1\ *$	$(t_2 - d/2)$
$t_3 = d\ 2\ /$	$(t_2 - t_3)$
$t_4 = t_2\ t_3\ -$	(t_4)

Put value of t_1, t_2, t_3

$$t_4 = t_2\ t_3\ -$$
$$= t_2\ d\ 2\ /\ -$$
$$= a t_1\ *\ d\ 2\ /\ -$$
$$= abc + *\ d\ 2\ /\ -$$

PART-4

Three Address Code, Quadruples and Triples.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.10. 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 atmost 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 3.11. What are different ways to write three address code ? Write the three address code for the following code segment :
while *A* < *C* and *B* < *D* do
if *A* = 1 then *C* = *C* + 1
else while *A* < = *D* do *A* = *A* + 2. **AKTU 2013-14, Marks 10**

Answer

Different ways to write three address code are :

1. **Quadruple representation :**
   a. The quadruple is a structure with atmost 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 $A <= D$ goto 6
4. $t_1 = A + 2$
5. $A = t_1$

6. $t_2 = C + 1$
7. $C = t_2$

Que 3.12. **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 2015-16, 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 3.13. **Generate three address code for the following code segment :**
while ($a < b$) do
If ($c < d$) then $x = y + z$

AKTU 2014-15, Marks 10

Answer

L_1 : if $a < b$ goto L_2
goto L_4
L_2 : if $c < d$ goto L_3
L_3 : $t_1 := y + z$
$x := t_1$

goto L_1
L_4 : Next

Que 3.14. **Describe various representation of three address codes. Translates the expression : – (a + b) * (c + d) + (a + b + c).**

AKTU 2014-15, Marks 10

Answer

Three address code representation : Refer Q. 3.10, Page 3–10B, Unit-3.

Numerical : The three address code for given expression :

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

i. The quadruple representation :

Location	Operator	Operand 1	Operand 2	Result
(1)	+	a	b	t_1
(2)	+	c	d	t_2
(3)	*	t_1	t_2	t_3
(4)	+	t_1	c	t_4
(5)	+	t_3	t_4	t_5

ii. The triple representation :

Location	Operator	Operand 1	Operand 2
(1)	+	a	b
(2)	+	c	d
(3)	*	(1)	(2)
(4)	+	(1)	c
(5)	+	(3)	(4)

iii. The indirect triple representation :

	Operator	Operand 1	Operand 2
(1)	+	a	b
(2)	+	c	d
(3)	*	(11)	(12)
(4)	+	(11)	c
(5)	+	(13)	(14)

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

Que 3.15. **Generate three address code for**

$C[A[i, j]] = B[i, j] + C[A[i, j]] + D[i, j]$ (You can assume any data for solving question, if needed) Assuming that all array elements are integer. Let A and B a 10×20 array with $low_1 = low = 1$.

AKTU 2017-18, Marks 10

Answer

Given : $low_1 = 1$ and $low = 1$, $n_1 = 10$, $n_2 = 20$.

$$B[i, j] = ((i \times n_2) + j) \times w + (base - ((low_1 \times n_2) + low) \times w)$$

$$B[i, j] = ((i \times 20) + j) \times 4 + (base - ((1 \times 20) + 1) \times 4)$$

$$B[i, j] = 4 \times (20\, i + j) + (base - 84)$$

Similarly, $A[i, j] = 4 \times (20\, i + j) + (base - 84)$

and, $D[i, j] = 4 \times (20\, i + j) + (base - 84)$

Hence, $C[A[i, j]] = 4 \times (20\, i + j) + (base - 84) + 4 \times (20\, i + j) + (base - 84) + 4 \times (20\, i + j) + (base - 84)$

$= 4 \times (20\, i + j) + (base - 84)\ [1 + 1 + 1]$

$= 4 \times 3 \times (20\, i + j) + (base - 84) \times 3$

$= 12 \times (20\, i + j) + (base - 84) \times 3$

Therefore, three address code will be

$t_1 = 20 \times i$
$t_2 = t_1 + j$
$t_3 = base - 84$
$t_4 = 12 \times t_2$
$t_5 = t_4 + 3 \times t_3$

PART-5

Translation of Assignment Statements.

CONCEPT OUTLINE

- Assignment statements can be converted into semantic action using syntax directed translation scheme.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.16. **How would you convert the following into intermediate code ? Give a suitable example.**

i. **Assignment statements**

ii. **Case statements**

AKTU 2016-17, Marks 15

Answer

i. **Assignment statements :**

Production rule	Semantic actions

$S \rightarrow id := E$	{ *id*_entry := look_up(*id*.name);
	if *id*_entry ≠ 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*. place:= E_1.place }

$E \rightarrow id$	{ *id*_entry: = look_up(*id*.name);
	if *id*_entry ≠ nil then
	append (*id*_entry ':=' *E*.place)
	else error; /* *id* not declared*/
	}

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 :

PART-6

Boolean Expressions, Statements that alter the Flow of Control.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.17. **Define backpatching and semantic rules for boolean expression. Derive the three address code for the following expression : $P < Q$ or $R < S$ and $T < U$.**

AKTU 2013-14, 2015-16; Marks 10

OR

Write short notes on backpatching. **AKTU 2014-15, Marks 05**

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 :

$B \rightarrow B_1 \| MB_2 \mid B_1$ AND $MB_2 \mid !B_1 \mid (B_1) \mid E_1$ rel $E_2 \mid$ True $\mid$ False

$M \rightarrow \varepsilon$

The translation scheme is as follows :

i. $B \rightarrow B_1 \| MB_2$ {backpatch (B_1.falselist, M.instr);

B.truelist = merge (B_1.truelist, B_2.truelist);
B.falselist = B_2.falselist;}

ii. $B \rightarrow B_1$AND MB_2 {backpatch (B_1.truelist, M.instr);
B.truelist = B_2.truelist;
B.falselist = merge (B_1.falselist, B_2.falselist);}

iii. $B \rightarrow !B_1$ {B.truelist = B_1.falselist;
B.falselist = B_1.truelist;}

iv. $B \rightarrow (B_1)$ {B.truelist = B_1.truelist;
B.falselist = B_1.falselist;}

v. $B \rightarrow E_1$ rel E_2 {B.truelist = makelist (nextinstr);
B.falselist = makelist (nextinstr + 1);
append ('if' E_1.addr relop E_2.addr 'goto_');
append ('goto_');}

vi. $B \rightarrow$ true {B.truelist = makelist (nextinstr);
append ('goto_');}

vii. $B \rightarrow$ false {B.falselist = makelist (nextinstr);
append ('goto_');}

viii. $M \rightarrow \varepsilon$ {M.instr = 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 3.18. Explain translation scheme for boolean expression.

Answer

Translation scheme for boolean expression can be understand by following example.

Consider the boolean expression generated by the following grammar :

$E \rightarrow E$ OR E
$E \rightarrow E$ AND E
$E \rightarrow$ NOT E
$E \rightarrow (E)$
$E \rightarrow id$ relop id

$E \rightarrow$ TRUE
$E \rightarrow$ FALSE

Here the relop is denoted by ≤, ≥, ≠, <, >. 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
$E \rightarrow E_1$ OR E_2	{ E.place := newtemp() append(E.place ':=' E_1.place 'OR' E_2.place) }
$E \rightarrow E_1$ AND E_2	{ E.place := newtemp() append(E.place ':=' E_1.place 'AND' E_2.place) }
$E \rightarrow$ NOT E_1	{ E.place := newtemp() append(E.place ':=' 'NOT' E_1.place) }
$E \rightarrow (E_1)$	{ E.place := E_1.place }
$E \rightarrow id_1$ relop id_2	{ E.place := newtemp() append('if' id_1.place relop.op id_2.place 'goto' next_state + 3); append(E.place ':=' '0'); append('goto' next_state +2); append(E.place := '1') }
$E \rightarrow$ TRUE	{ E.place := newtemp(); append(E.place ':=' '1') }
$E \rightarrow$ FALSE	{ E.place := newtemp() append(E.place ':=' 0) }

PART-7

Postfix Translation : Array References in Arithmetic Expressions.

CONCEPT OUTLINE

- Production can be factored to achieve postfix form.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.19. 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 : $S \rightarrow$ while $M1\ E$ do $M2\ S1$

Can be factored as :

1. $S \rightarrow C\ S1$
2. $C \rightarrow W\ E$ do
3. $W \rightarrow$ while

A suitable transition scheme is given as :

Production rule	Semantic action
$W \rightarrow$ while	W.QUAD = NEXTQUAD
$C \rightarrow W\ E$ do	C W E do
$S \rightarrow C\ S1$	BACKPATCH (S1.NEXT, C.QUAD) S.NEXT = C.FALSE GEN (goto C.QUAD)

Que 3.20. What is postfix notations ? Translate $(C + D)^*(E + Y)$ into postfix using Syntax Directed Translation Scheme (SDTS).

AKTU 2017-18, Marks 10

Answer

Postfix notation : Refer Q. 3.6, Page 3–7B, 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 :

$$E.\text{code} = E_1.\text{code} \parallel T_1\text{code} \parallel \text{'+'}$$
$$E_1.\text{code} = T.\text{code}$$
$$T_1.\text{code} = T_1.\text{code} \parallel F.\text{code} \parallel \text{'x'}$$
$$T_1.\text{code} = F.\text{code}$$
$$F.\text{code} = E.\text{code}$$
$$F.\text{code} = id.\text{code}$$

where '||' sign is used for concatenation.

PART-8

Procedures Call.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.21. 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. $S \rightarrow$ call *id*(Elist)
2. Elist $\rightarrow$ Elist, E
3. Elist $\rightarrow E$

A suitable transition scheme for procedure call would be :

Production rule	Semantic action
$S \rightarrow$ call *id*(Elist)	for each item p on QUEUE do GEN (param p) GEN (call *id*.PLACE)
Elist $\rightarrow$ Elist, E	append E.PLACE to the end of QUEUE
Elist $\rightarrow E$	initialize QUEUE to contain only *E*.PLACE

Que 3.22. 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. 3.22.1 for row-major and column-major are given as :

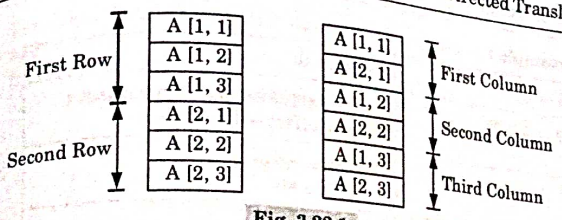


Fig. 3.22.1.

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,

$$((...((i_1 n_2 + i_2) n_3 + i_3)...) n_k + I_i) * w + \text{base} - ((...((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3)...) n_k + \text{low}_k) * w$$

PART-9

Declarations Statements.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.23. 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(*id*.name, T.type,offset); offset:= offset + T.width)}
$T \rightarrow$ integer	{T.type:= integer; T.width:= 8}
$T \rightarrow$ real	{T.type:= real; T.width:= 8}
$T \rightarrow$ array[num] of T_1	{T.type:= array(num.val,T_1.type) T.width:= num.val × T_1.width}
$T \rightarrow {}^*T_1$	{T.type:= pointer(T.type) T.width:= 4}

1. Initially, the value of offset is set to zero. The computation of offset can be done by using the formula offset = offset + width.
2. 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.
3. 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.
4. The width of array is obtained by multiplying the width of each element by number of elements in the array.
5. The width of pointer types of supposed to be 4.

Que 3.24. Give the syntax directed definition for if-else statement.

Answer

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 :

$S \rightarrow$ if E then S_1

E.true := new_label()
E.false := S.next
S_1.next := S.next
S.code := E.code || gen_code(E.true ':') || S_1.code

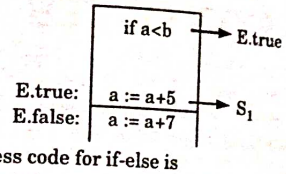
3. In the given translation scheme || is used to concatenate the strings.
4. The function gen_code is used to evaluate the non-quoted arguments passed to it and to concatenate complete string.

5. The S.code is the important rule which ultimately generates the three address code.

$S \rightarrow$ if E then S_1 else S_2

E.true := new_label()
E.false := new_label()
S_1.next := S.next
S_2.next := S.next
S.code := E.code || gen_code(E.true ':') ||
S_1.code := gen_code('goto', S.next) ||
gen_code(E.false ':') || S_2.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.true*/
103 : $L2$ a := a+7

Hence, E.code is "if $a < b$" $L1$ denotes E.true and $L2$ denotes E.false is shown by jumping to line 103 (*i.e.*, S.next).

4
UNIT

Symbol Tables

CONTENTS

PART-1

Symbol Tables : Data Structure for Symbol Tables, Representing Scope Information.

CONCEPT OUTLINE

- Symbol table is a compile time data structure that is used by compiler to collect information about source program.
- Scope information characterizes the declaration of identifiers.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

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 which are used in the source program to identify the various program elements, like variables, constants, procedures, and the labels of statements.
2. 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 ? Discuss.**

AKTU 2016-17, Marks 10

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
c	float
c	int
b	int
a	int

vi. After entering into Level 4.

m	int
d	float
c	float
c	int
b	int
a	int

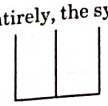
vii. On leaving the control from Level 4.

d	float
c	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.** **AKTU 2013-14, Marks 10**

OR

Discuss the various data structures used for symbol table with suitable example. **AKTU 2014-15, 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 structure used in implementing symbol table are as :

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.

Example : Consider a part of C program

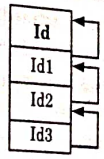
```
int x, y;
msg ( );
```

1. Unordered list :

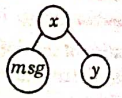
S. No.	Name	Type
1	x	int
2	*msg*	function
3	y	int

2. Ordered list :

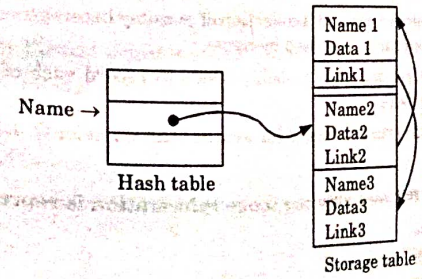
Id	Name	Type
Id1	x	int
Id2	y	int
Id3	*msg*	function



3. Search tree :



4. Hash table :



Que 4.5. **Describe symbol table and its entries. Also, discuss various data structure used for symbol table.** **AKTU 2015-16, Marks 10**

Answer

Symbol table : Refer Q. 4.1, Page 4–2B, Unit-4.

Entries in the symbol table are as follows :

1. **Variables :**
   a. Variables are identifiers whose value may change between executions and during a single execution of a program.
   b. They represent the contents of some memory location.
   c. The symbol table needs to record both the variable name as well as its allocated storage space at runtime.
2. **Constants :**
   a. Constants are identifiers that represent a fixed value that can never be changed.
   b. Unlike variables or procedures, no runtime location needs to be stored for constants.
   c. These are typically placed right into the code stream by the compiler at compilation time.
3. **Types (user defined) :**
   a. A user defined type is combination of one or more existing types.
   b. Types are accessed by name and reference a type definition structure.
4. **Classes :**
   a. Classes are abstract data types which restrict access to its members and provide convenient language level polymorphism.
   b. This includes the location of the default constructor and destructor, and the address of the virtual function table.
5. **Records :**
   a. Records represent a collection of possibly heterogeneous members which can be accessed by name.
   b. 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 4–5B, Unit-4.

Que 4.6. Discuss how the scope information is represented in a symbol table.

Answer

1. Scope information characterizes the declaration of identifiers and the portions of the program where it is allowed to use each identifier.
2. 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.

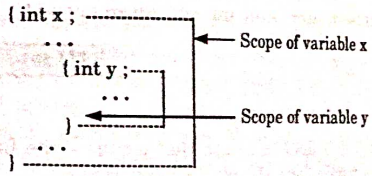
3. Thus, the same identifier may be declared several times as distinct names, with different attributes, and with different intended storage locations.
4. The symbol table is thus responsible for keeping different declaration of the same identifier distinct.
5. To make distinction among the declarations, 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.

   There are mainly two semantic rules regarding the scope of an identifier :

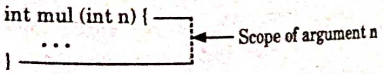
   a. Each identifier can only be used within its scope.
   b. Two or more identifiers with same name and are of same kind cannot be declared within the same lexical scope.

   The scope declaration of variables, functions, labels and objects within a program is shown below :

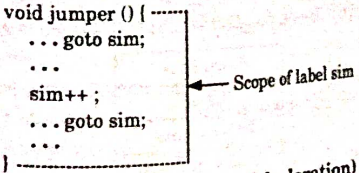
Scope of variables in statement blocks :



Scope of formal arguments of functions :



Scope of labels :



7. **Scope in class declaration (scope of declaration) :** The portion of the program in which a declaration can be applied is called the scope of that declaration. In a procedure, a name is said to be local to the procedure if it is in the scope of declaration within the procedure, otherwise the name is said to be non-local.

Scope of object fields and methods :

```
class X {
  public :
    void A ()
      {
      m = 1;
      }
    private :
      int m;
      ...
}
```

← Scope of variable m and method A

PART-2

Run-Time Administration : Implementation of Simple Stack Allocation Scheme.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 4.7. **Write a short note on activation record.**

AKTU 2014-15, Marks 05

Answer

Activation record is a data structure that contains the important state information for a particular instance of a function call.

Activation record contains the following :

1. The values of actual parameters.
2. The count of number of arguments.
3. The return address.
4. The return value.
5. The value of SP (Stack Pointer) for the activation record.

Local data
Old SP
Return value
Return address
Arg count
Actual parameters

Fig. 4.7.1. Pictorial representation of activation record.

Que 4.8. **How to sub-divide a run-time memory into code and data areas. Explain.**

AKTU 2016-17, Marks 10

Answer

Sub-division of run-time memory into codes and data areas is shown in Fig. 4.8.1.

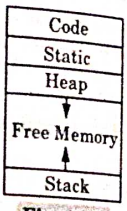


Fig. 4.8.1.

1. **Code :** It stores the executable target code which is of fixed size and do not change during compilation.
2. **Static allocation :**
   a. The static allocation is for all the data objects at compile time.
   b. The size of the data objects is known at compile time.
   c. 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.
   d. 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.
   e. 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 :**
      i. When all access path to a object are destroyed but data object continue to exist, such type of objects are said to be garbaged.
      ii. The garbage collection is a technique which is used to reuse that object space.
      iii. In garbage collection, all the elements whose garbage collection bit is 'on' are garbaged and returned to the free space list.
   b. **Reference counter :**
      i. Reference counter attempt to reclaim each element of heap storage immediately after it can no longer be accessed.
      ii. 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.

iii. 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 :**
   a. Stack allocation is used to store data structure called activation record.
   b. The activation records are pushed and popped as activations begins and ends respectively.
   c. 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.
   d. Furthermore, these values of locals are deleted when the activation ends.
   e. Suppose that the register marks the top of stack. At run-time, an activation record can be allocated and deallocated by incrementing and decrementing top, respectively, by the size of record.

Que 4.9. **Why run-time storage management is required ? How simple stack implementation is implemented ?**

AKTU 2013-14, Marks 10

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 4.10. **Discuss the following parameter passing techniques with suitable example.**

i. Call by name

ii. Call by reference

Answer

i. Call by name :

1. In call by name, the actual parameters are substituted for formals in all the places whereformals 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

PART-3

Storage Allocation in Block Structured Language.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 4.11. **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(name) will result any integer between 0 to $k-1$. We can search any name by position = h(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. 4.11.1.

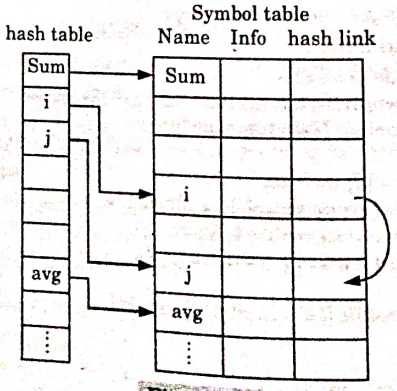


Fig. 4.11.1.

7. The hash function should result in uniform distribution of names in symbol table.
8. The hash function should have minimum number of collision. Collision is such a situation where hash function results in same location for storing the names.
9. Various collision resolution techniques are open addressing, chaining, rehashing.

Que 4.12. **Write a short note on scoping.**

AKTU 2014-15, 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 4.13. **Differentiate between lexical 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.

PART-4

Error Detection and Recovery : Lexical Phase Errors, Syntactic Phase Errors, Semantic Errors.

CONCEPT OUTLINE

- Errors are detected during different phases of compiler.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 4.14. **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 4.15. **What are lexical phase errors, syntactic phase errors and semantic phase errors ? Explain with suitable example.**

AKTU 2013-14, 2015-16; 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. Reason 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 4.16. What do you understand by lexical error and syntactic error ? Also, suggest methods for recovery of errors.

AKTU 2014-15, Marks 10

OR

Explain logical phase error and syntactic phase error. Also suggest methods for recovery of error.

AKTU 2017-18, Marks 10

Answer

Lexical and syntactic error : Refer Q. 4.15, Page 4–15B, 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

while ($x > 0$) $y = a + b$;

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

$E \rightarrow +E|-E|*A|/A$

$A \rightarrow E$

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.

☺☺☺

5 UNIT

Code Generation

CONTENTS

PART-1

Code Generation : Design Issues.

CONCEPT OUTLINE

- Code generation take intermediate code as input and produce target program as output.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

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.

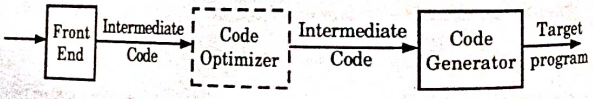


Fig. 5.1.1. Position of code generator.

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 that is used to determine the run-time addresses of the data objects denoted by the names in the IR.
   b. The many choices for the IR include three address representations such as quadruples, triples, indirect triples and graphical representations such as syntax trees and DAGs.
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. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them 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.

PART-2

The Target Language, Address in Target Code.

CONCEPT OUTLINE

- Target language is a language which is generated after the compilation of source program.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.2. **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.

CONCEPT OUTLINE

- Basic block is a sequence of consecutive statements in which flow of controls enters at start and leaves at the end without half.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.3. **Write an algorithm to partition a sequence of three address statements into basic blocks.** **AKTU 2016-17, Marks 10**

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 5.4. **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. $t_6 := \text{prod} + t_5$
9. prod := t_6
10. $t_7 := i + 1$
11. $i := t_7$
12. if $i <= 10$ goto (3)

The flow graph for the given code can be drawn as follows :

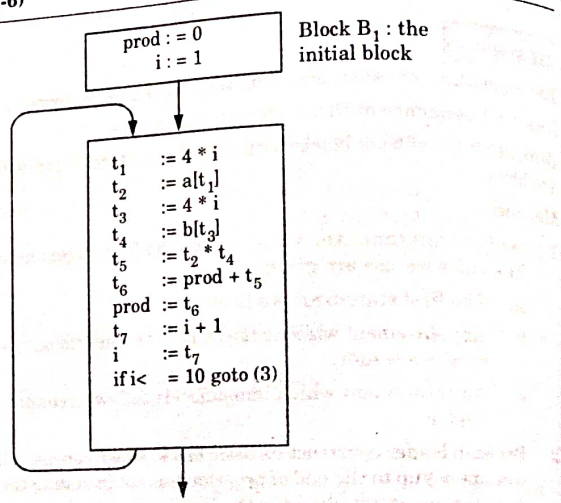


Fig. 5.4.1. Flow graph.

Que 5.5. **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. That means 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 :**
   a. 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 dom n.
   b. By definition, every node dominates itself.
   c. There are a number of related concepts :
      i. A node d strictly dominates a node n if d dominates n and d is not equal to n.
      ii. 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.

      iii. The dominance frontier of a node d is the set of all nodes n such that d dominates an immediate predecessor of n, but d does not strictly dominate n. It is the set of nodes where d's dominance stops.
      iv. 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 :**
   a. Loop in a flow graph can be denoted by $n \rightarrow d$ such that d dom n.
   b. These edges are called back edges and for a loop there can be more than one back edge.
   c. If there is $p \rightarrow q$ then q is a head and p is a tail and head dominates tail.
   d. 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.
3. **Pre-header :**
   a. The pre-header is a new block created such that successor of this block is the header block.
   b. All the computations that can be made before the header block can be made before the pre-header block.

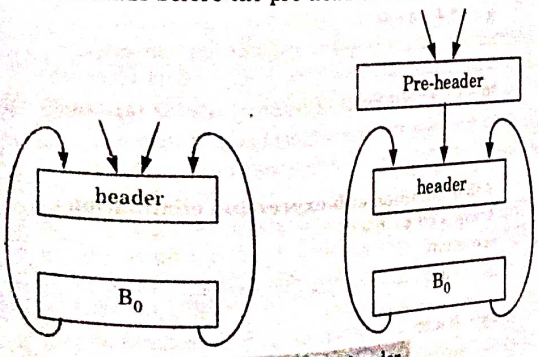


Fig. 5.5.1. Pre-header.

4. **Reducible flow graph :**
   a. The reducible graph is a flow graph in which there are two types of edges, 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 head dominates their tail.

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 5.6. **Discuss in detail the process of optimization of basic blocks. Give an example.** **AKTU 2016-17, Marks 10**

OR

What are different issues in code optimization ? Explain it with proper example. **AKTU 2013-14, Marks 10**

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. During the compilation, execution efficiency can be improved by using such kind of certain actions *i.e.*, replacement of one constant by an equivalent constant can improve efficiency of the compiler.
     - iii. 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 5.7. **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. 5.6, Page 5–8B, Unit-5.
      ii. **Dead code elimination :** Refer Q. 5.6, Page 5–8B, Unit-5.
      iii. **Interchange of statement :** Suppose we have a block with the two adjacent statements,
      temp1 = $a + b$
      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. 5.6, Page 5–8B, Unit-5.

PART-4

Machine Independent Optimizations, Loop Optimization.

CONCEPT OUTLINE

- Code optimization is used by compiler to improve the efficiency of generated object code.
- Loop optimization is done to improve the efficiency of program.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.8. **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 :

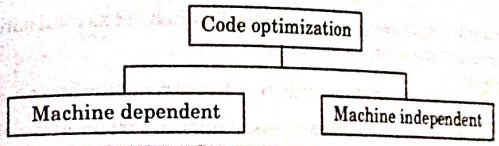


Fig. 5.8.1. Classification of code optimization.

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 5.9. **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 5.10. **Explain what constitute a loop in a flow graph and how will you do loop optimizations in code optimization of a compiler.**

AKTU 2014-15, Marks 10

OR

Write a short note on loop optimization.

AKTU 2017-18, Marks 05

Answer

Following term constitute a loop in flow graph : Refer Q. 5.5, Page 5–6B, 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. A variable x is called an induction variable of loop L if the value of variable gets changed every time.
   b. It is either decremented or incremented by some constant.
3. **Reduction in strength :**
   a. In strength reduction technique the higher strength operators can be replaced by lower strength operators.

   b. The strength of certain operator is higher than other.
   c. 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 :

```
int i = 1;
while(i<=100)
{
    a[i]=b[i];
    i++;
}
```

Can be written as →

```
int i = 1;
while(i<=100)
{
    a[i]=b[i];
    i++;
    a[i]=b[i];
    i++ ;
}
```

6. **Loop fusion or loop jamming :** In loop fusion method, several loops are merged to one loop.

For example :

```
for i:=1 to n do
for j:=1 to m do
a[i,j]:=10
```

Can be written as →

```
for i:=1 to n*m do
a[i]:=10
```

Que 5.11. **Consider the following sequence of three address codes :**

1. **Prod : = 0**
2. **I : = 1**
3. **T_1 : = 4*I**
4. **T_2 : = addr (A) – 4**
5. **T_3 : = T_2 [T_1]**
6. **T_4 : = 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 <= 20 goto (3)**

Perform loop optimization.

AKTU 2015-16, Marks 10

OR

Consider the following three address code segments :

***PROD* : = 0**

***I* : = 1**

$T1 := 4*I$
$T2 := addr(A) - 4$
$T3 := T2[T1]$
$T4 := addr(B) - 4$
$T5 := T4[T1]$
$T6 := T3*T5$
$PROD := PROD + T6$
$I := I + 1$
If $i <= 20$ 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.

AKTU 2017-18, Marks 10

Answer

1. As first statement of program is leader statement.
   ∴ PROD = 0 is a leader.
2. Fragmented code represented by two blocks is shown below :

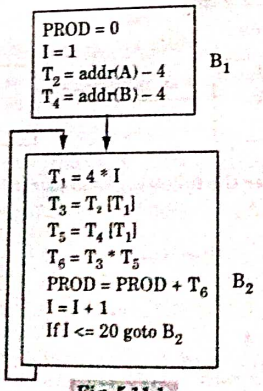


Fig. 5.11.1.

3. A variable I is called an induction variable of loop L if every time the variable I changes values.
4. Generally induction variables are

$$a := I*b \text{ or } a := I \pm b$$

where b is constant. In Fig. 5.11.2, I and T_1 are induction variables.
5. The values of I varies from 1 to 20 and value T_1 varies from (4, 8, ... , 80).

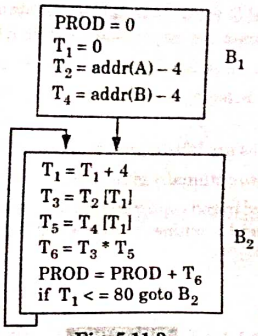


Fig. 5.11.2.

PART-5

DAG Representation of Basic Blocks.

CONCEPT OUTLINE

- DAG stands for Directed Acyclic Graphs.
- DAG is used for implementation of basic blocks.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.12. **What is DAG ? How DAG is created from three address code ? Write algorithm for it and explain it with a relevant example.**

AKTU 2013-14, Marks 10

OR

Write a short note on direct acyclic graph.

AKTU 2017-18, Marks 05

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.

Algorithm :

Input : A basic block.

Output : A DAG with label for each node (identifier).

Method :

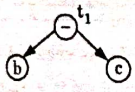
1. Create nodes with one or two left and right children.
2. Create linked list of attached identifiers for each node.
3. Maintain all identifiers for which a node is associated.
4. Node (identifier) represents value that identifier has the current point in DAG construction process. Symbol table store the value of node (identifier).
5. If there is an expression of the form $x = y$ 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 : $a * (b - c) + (b - c) * d$

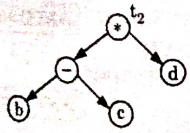
The construction of DAG with three address code will be as follows :

Step 1 :



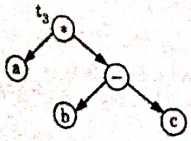
$t_1 = b - c$

Step 2 :



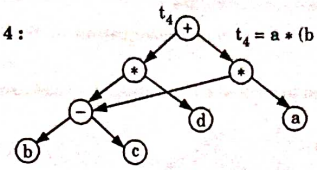
$t_2 = (b - c) * d$

Step 3 :



$t_3 = a * (b - c)$

Step 4 :



$t_4 = a * (b - c) + (b - c) * d$

Que 5.13. What is DAG ? What are its advantages in context of optimization ? **AKTU 2014-15, Marks 10**

Answer

DAG : Refer Q. 5.12, Page 5–15B, Unit-5.

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 5.14. 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.

AKTU 2015-16, Marks 15

Answer

DAG v/s Syntax tree :

1. Directed Acyclic Graph is used for transformations on the basic block. To apply the transformations on a basic block a DAG is constructed from three address statement.
2. A DAG can be constructed for the following types of label on nodes :
   a. Leaf nodes labeled as identifiers operator.

b. Interior nodes store operator values.

3. While syntax tree is an abstract representation of the language constructs.

4. The syntax trees are used to write the translation routines using syntax directed definitions.

AG for the given code is :

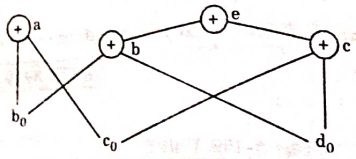


Fig. 5.14.1.

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. In this representation, data enters a processing element through its incoming edges and leaves the element through its outgoing edges.
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 5.15. **Define a directed acyclic graph. Construct a DAG and write the sequence of instructions for the expression :**

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

AKTU 2016-17, Marks 15

Answer

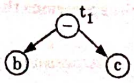
Directed acyclic graph : Refer Q. 5.12, Page 5–15B, Unit-5.

Numerical :

Given expression : $a + a * (b - c) + (b - c) * d$

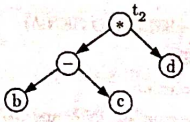
The construction of DAG with three address code will be as follows :

Step 1 :



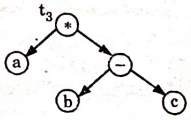
$t_1 = b - c$

Step 2 :



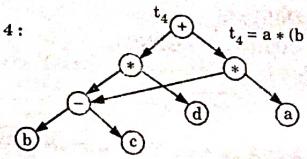
$t_2 = (b - c) * d$

Step 3 :



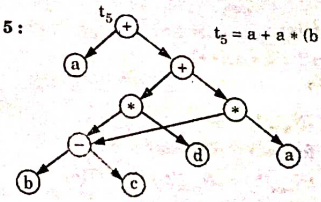
$t_3 = a * (b - c)$

Step 4 :



$t_4 = a * (b - c) + (b - c) * d$

Step 5 :



$t_5 = a + a * (b - c) + (b - c) * d$

Que 5.16. **Give the algorithm for the elimination of local and global common sub-expressions algorithm with the help of example.**

AKTU 2017-18, Marks 10

Answer

Algorithm for elimination of local common sub-expression : DAG algorithm is used to eliminate local common sub-expression.

DAG : Refer Q. 5.12, Page 5–15B, Unit-5.

Algorithm for elimination of global common sub-expression :

a. Global common sub-expressions are expressions that compute the same value but in different basic blocks.

b. To eliminate the global common sub-expression, we need to compute the following equations :

$$OUT(b) = IN(b) - KILL(b) \text{ O } GEN(b)$$

$$IN(b) = \text{O } OUT(p)$$

Here, we obtain the smallest solution.

c. The algorithm for computing the smallest IN(b) and OUT(b) is as follows :

1. $IN(b_1) = \phi$
   $OUT(b_1) = GEN(b_1)$;
2. For ($i = 2$; $i <= n$; i++)
   {
   $IN(b_i) = U$
   $OUT(b_i) = U - GEN(b)$
   }
3. flag = true
4. While (flag) do
   {
   flag = false
   for ($i = 2$; $i <= n$; i++)
   {
   $IN_{new}(b_i) = \phi$
   for each predecessor p of b_i
   $IN_{new}(b_i) = IN_{new}(b_i)$ O $OUT(p)$
   if $IN_{new}(b_i) \neq IN(b_i)$ then
   {
   flag = true
   $IN(b_i) = In_{new}(b_i)$
   $OUT(b_i) = IN(b_i) - KILL(b_i)$ O $GEN(b_i)$
   }
   }
   }

d. After computing IN(b) and OUT(b), eliminating the global common sub-expressions is done as follows :

i. For every statement s of the form $x = y$ *op* z such that y *op* z is available at the beginning of the block containing s, and neither y nor z is defined prior to the statement $x = y$ *op* z in that block, do :

1. Find all definitions reaching up to the s statement block that have y *op* z on the right.
2. Create a new temp.
3. Replace each statement $U = y$ *op* z found in step 1 by :
   temp = y *op* z
   U = temp
4. Replace the statement $x = y$ *op* z in block by x = temp.

PART-6

Value Numbers and Algebraic Laws, Global Data Flow Analysis.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.17. Write a short note on data flow analysis.

OR

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

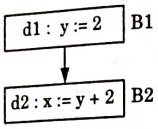
AKTU 2014-15, Marks 10

Answer

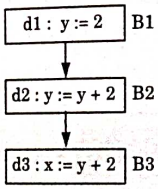
1. Data flow analysis is a process in which the values are computed using data flow properties.
2. In this analysis, the analysis is made on data flow.
3. A program's Control Flow Graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate.
4. The information gathered is often used by compilers when optimizing a program.
5. 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.
6. 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.



2. A definition D of variable x is killed when there is a redefinition of x.



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 5.18. **Write short notes (any two) :**

i. Global data flow analysis
ii. Loop unrolling
iii. Loop jamming

AKTU 2013-14, Marks 10

AKTU 2015-16, Marks 15

OR

Write short note on global data analysis.

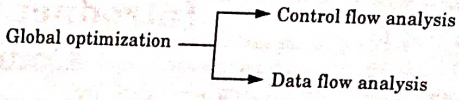
AKTU 2017-18, Marks 05

Answer

i. Global data flow analysis :

1. The local optimization has a very restricted scope on the other hand the global optimization over a broad scope such as procedure or function body.
2. For a global optimization a program is represented in the form of program flow graph. The program flow graph is a graphical representation in which each node represents the basic block and edges represent the flow of control from one block to another.

3. There are two types of analysis performed for global optimizations : control flow analysis and data flow analysis.



ii. Loop unrolling : Refer Q. 5.10, Page 5–12B, Unit-5.
iii. Loop fusion or loop jamming : Refer Q. 5.10, Page 5–12B, Unit-5.

Que 5.19. **Discuss the role of macros in programming language.**

AKTU 2014-15, Marks 05

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.

☺☺☺