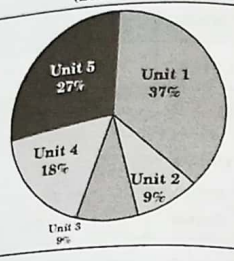
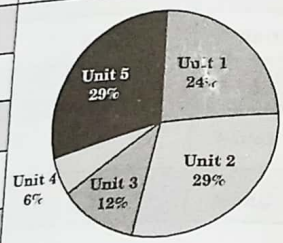
Units	Ques. Asked (2016-17)
Unit-1	4
Unit-2	1
Unit-3	1
Unit-4	2
Unit-5	3

% Weightage of Units (2016-17)



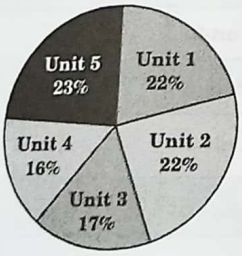
Units	Ques. Asked (2017-18)
Unit-1	4
Unit-2	5
Unit-3	2
Unit-4	1
Unit-5	5

% 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 ..... **1–2B to 1–6B**
Phases and Passes

Part-2 : Bootstrapping ..... **1–6B to 1–8B**

Part-3 : Finite State Machines and ..... **1–8B to 1–16B**
Regular Expressions and
their Application to Lexical Analysis
Optimization of DFA based
Pattern Matchers

Part-4 : Implementation of ..... **1–17B to 1–23B**
Lexical Analyzers
Lexical Analyzer Generator
LEX Compiler

Part-5 : Formal Grammars and ..... **1–23B to 1–24B**
their Application to Syntax Analysis

Part-6 : BNF Notation ..... **1–25B to 1–25B**

Part-7 : Ambiguity ..... **1–26B to 1–28B**
YACC

Part-8 : The Syntactic Specification ..... **1–28B to 1–31B**
of Programming Languages :
Context Free Grammar (CFG)
Derivation and Parse Trees
Capabilities of CFG

Scanned with CamScan

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.

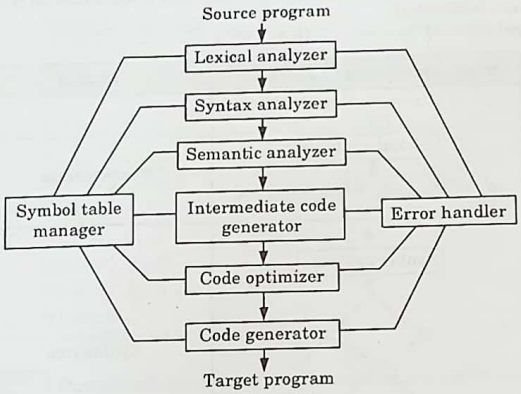


Fig. 1.1.1.

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.

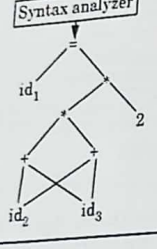
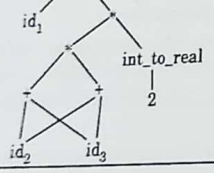
c. In this phase, the address in the binary code is translated from logical address.

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.

Scanned with CamScan

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

Scanned with CamScan

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 :

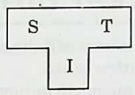


Fig. 1.4.1.

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.)

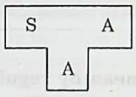


Fig. 1.4.2.

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

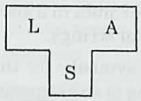


Fig. 1.4.3.

   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.

Scanned with CamScan

$$^{L}C_{S}^{A} \rightarrow {}^{S}C_{A}^{A} \rightarrow {}^{L}C_{A}^{A}$$

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

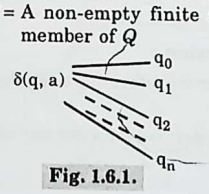


Fig. 1.6.1.

Scanned with CamScanner

$\hat{\delta}$ = It is transition function that takes a state from Q and an input symbol from Σ and returns a subset of Q. The $\hat{\delta}$ is represented as :

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

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

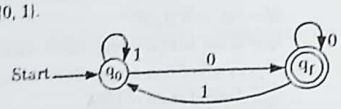


Fig. 1.6.2.

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

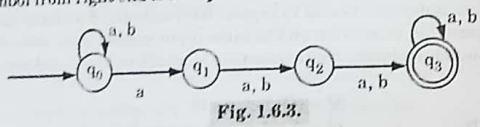


Fig. 1.6.3.

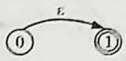
Que 1.7. **Explain Thompson's construction with example.**

Answer

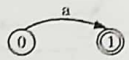
Thompson's construction :

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 :

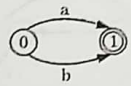
i. The NFA representing the empty string is :



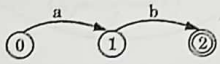
ii. If the regular expression is just a character, thus a can be represented as :



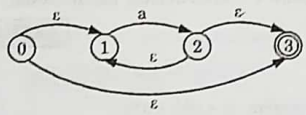
iii. The union operator is represented by a choice of transitions from a node thus $a \mid b$ can be represented as :



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



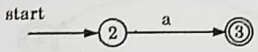
v. The Kleene closure must allow for taking zero or more instances of the letter from the input; thus a^* looks like :



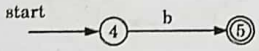
For example :

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

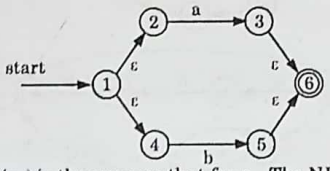
For $r_1 = a$,



For $r_1 = b$,

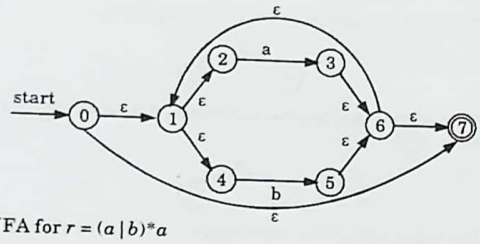


For $r_3 = a \mid b$

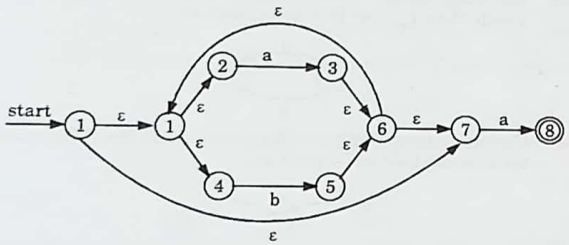


The NFA for $r_4 = (r_3)$ is the same as that for r_3. The NFA for $r_5 = (r_3)^*$

Scanned with CamScan



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



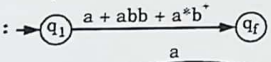
Que 1.8. **Construct the NFA for the regular expression $a \mid abb \mid a^*b^+$ by using Thompson's construction methodology.**

AKTU 2017-18, Marks 10

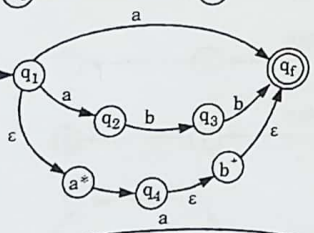
Answer

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

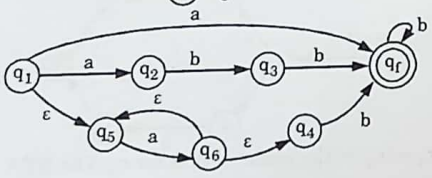
Step 1 :



Step 2 :



Step 3 :



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

AKTU 2016-17, Marks 10

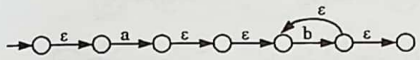
Answer

Step 1 : a

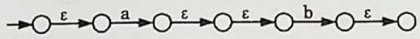
Step 2 : b^*

Step 3 : b

Step 4 : ab^*



Step 5 : ab



Step 6 : $ab^* \mid ab$

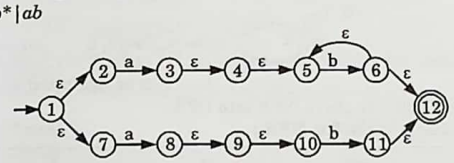


Fig. 1.9.1. NFA of ab* | ab.

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).

Scanned with CamScan

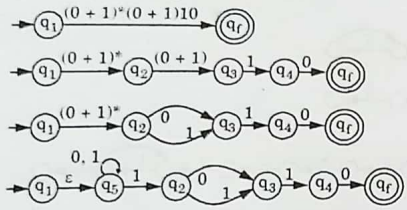
Que 1.11. **Construct the minimized DFA for the regular expression $(0 + 1)^*(0 + 1)\,10$.**

AKTU 2016-17, Marks 10

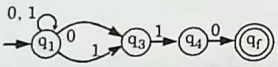
Answer

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 :

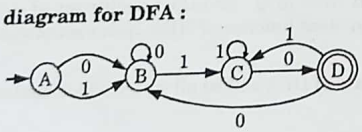
Transition table for NFA :

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.

Scanned with CamScan

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

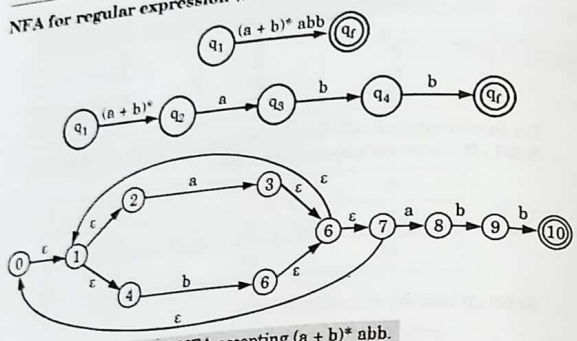


Fig. 1.12.1. NFA accepting (a + b)* abb.

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

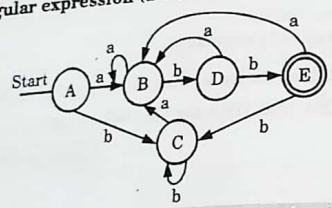


Fig. 1.12.2. DFA accepting (a + b)*abb.

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.

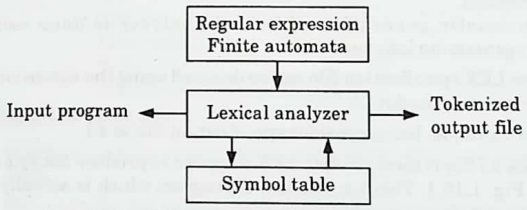


Fig. 1.14.1. Implementation of lexical analyzer.

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.

Scanned with CamScan

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.

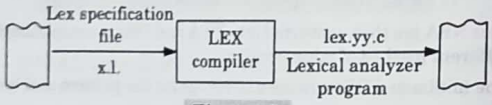


Fig. 1.16.1.

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.

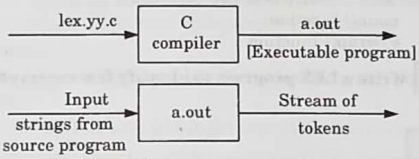


Fig. 1.16.2. Generation of lexical analyzer using LEX.

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 :

Scanned with CamScan

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();
```

Scanned with CamScan

```
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.

Scanned with CamScan

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.

Scanned with CamScan

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 :

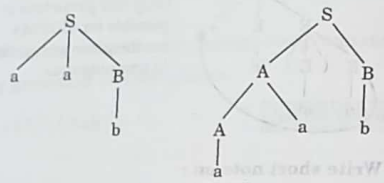


Fig. 1.29.1.

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*.

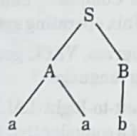


Fig. 1.29.2.

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

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

Scanned with CamScan

5. We say that the production $\alpha \to \beta$ 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 $\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_m$ are string in $(V_n \cup V_t)^*$, $m \geq 1$ and

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

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 $S \to s$ 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 $E \to E + E \mid E * E \mid id$.

Firstly take the production

$$E \to E + E \to \underline{E} * E + E \qquad \text{(Replace } E \to E * E\text{)}$$
$$\to id * \underline{E} + E \qquad \text{(Replace } E \to id\text{)}$$
$$\to id * id + \underline{E} \qquad \text{(Replace } E \to id\text{)}$$
$$\to id * id + id \qquad \text{(Replace } E \to id\text{)}$$

Right most derivation : A derivation $S \to s$ 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.

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

Start with production

$$E \to E * \underline{E}$$
$$\to E * E + \underline{E} \qquad \text{(Replace } E \to E + E\text{)}$$
$$\to E * \underline{E} + id \qquad \text{(Replace } E \to id\text{)}$$
$$\to E * id + id \qquad \text{(Replace } E \to id\text{)}$$
$$\to id * id + id \qquad \text{(Replace } E \to id\text{)}$$

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.

☺☺☺

Scanned with CamScan

2 UNIT

Basic Parsing Techniques

CONTENTS

Part-1 :	Basic Parsing Techniques : Parsers ..... Shift Reduce Parsing	2–2B to 2–5B
Part-2 :	Operator Precedence Parsing	2–5B to 2–9B
Part-3 :	Top-down Parsing ..... Predictive Parsers	2–9B to 2–18B
Part-4 :	Automatic Generation of ..... Efficient Parser : LR Parsers The Canonical Collections of LR(0) Items	2–18B to 2–20B
Part-5 :	Constructing SLR ..... Parsing Tables	2–21B to 2–27B
Part-6 :	Constructing canonical LR..... Parsing Tables	2–27B to 2–28B
Part-7 :	Constructing LALR ..... Parsing Tables Using Ambiguous Grammars An Automatic Parser Generator Implementation of LR Parsing Tables	2–28B to 2–38B

Udit K. Gupta

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.

Udit K. Gupta

Udit Krishna Gupta

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.

Udit

Scanned with CamScanner

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.

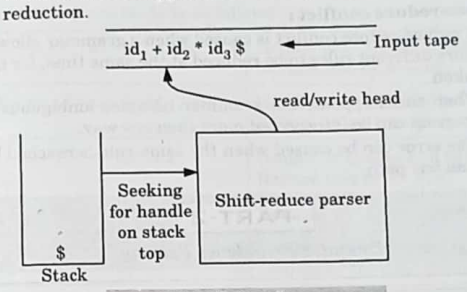


Fig. 2.2.1. Shift-reduce parser.

   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.

Scanned with CamScan

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 $<\cdot\cdot>$. Reduce this by $E \rightarrow id$.
$E + <\cdot id \cdot> * <\cdot id \cdot> \$$	Handle id is obtained between $<\cdot\cdot>$. Reduce this by $E \rightarrow id$.
$E + E * <\cdot id \cdot> \$$	Handle id is obtained between $<\cdot\cdot>$. 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 $<\cdot\cdot>$. This indicates that * becomes handle. That means, we have to reduce $E * E$ operation first.
$\$ <\cdot + \cdot> \$$	Now + becomes handle. Hence, we evaluate $E + E$.
$\$\$$	Parsing is done.

Que 2.5. **Give the algorithm for computing precedence function. Consider the following operator precedence matrix draw precedence graph and compute the precedence function :**

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

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.

Scanned with CamScan

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 $a \lessdot b$, place an edge from the group of g_b to the group of f_a. If $a \gtrdot b$, 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 :

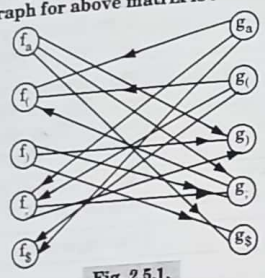


Fig. 2.5.1.

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	\$
+	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	
)	$\gtrdot$	$\gtrdot$		$\gtrdot$		$\gtrdot$
id	$\gtrdot$	$\gtrdot$		$\gtrdot$		$\gtrdot$
\$	$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$	

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	\$
+	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	
)	$\gtrdot$	$\gtrdot$		$\gtrdot$		$\gtrdot$
id	$\gtrdot$	$\gtrdot$		$\gtrdot$		$\gtrdot$
\$	$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$	

Scanned with CamScan

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.

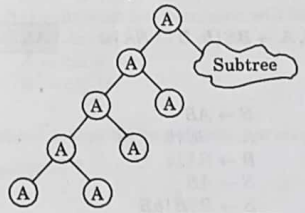


Fig. 2.8.1. Left recursion.

   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 ?**

Scanned with CamScan

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

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$$

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

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_{n-1} \mid \beta_n$$

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

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

AKTU 2013-14, Marks 05

Answer

Eliminating immediate left recursion among all productions, we get :

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

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

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

AKTU 2017-18, Marks 10

Answer

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

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

The production after left recursion is

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

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

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

AKTU 2014-15, Marks 05

Answer

$S \to Aa \mid b$

$A \to \underbrace{A}_{A}\underbrace{c}_{\alpha} \mid \underbrace{Sd}_{\beta_1} \mid \underbrace{e}_{\beta_2}$ ←Left recursion exist

$A \to Sd\,A' \mid eA'$
$A \to c\,A' \mid \varepsilon$

The production after removing left recursion will be :

$S \to Aa \mid b$
$A \to Sd\,A' \mid e\,A'$
$A \to cA' \mid \varepsilon$

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.

Scanned with CamScan

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.

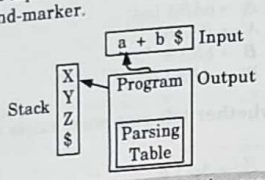


Fig. 2.13.1. Model of a predictive parser.

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.

c. 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 $A \rightarrow X_1 X_2 X_3 \ldots X_k$,

FIRST(A) = (FIRST(X_1) $\cup$ FIRST(X_2) $\cup$ FIRST(X_3).... $\cup$ 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) = {$a \mid S \Rightarrow \alpha A a \beta$ 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).

Scanned with CamScan

c. If there is a production $A \to \alpha B \beta$ or $A \to \alpha B$ 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 $A \to \alpha$ of the grammar, do steps (2) and (3).
2. For each terminal 'a' in FIRST (α), add $A \to \alpha$ to $M[A, a]$.
3. If ε is in FIRST (α), add $A \to \alpha$ to $M[A, b]$, for each terminal 'b' in FOLLOW (A). If ε is in FIRST (α) and \$ is in FOLLOW (A), add $A \to \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M 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 :
$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to -F \mid (E) \mid 1 \mid 0$
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**

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

Scanned with CamScan

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

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

Scanned with CamScan

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.

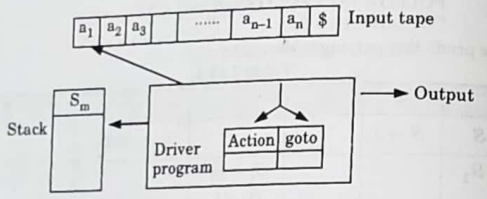


Fig. 2.20.1.

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 $a_i a_{i+1} ..... a_n$:

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	Input
$S_0X_1S_1X_2.....X_{m-r}S_{m-r}X_m$	$S_m a_i S, a_{i+1}, a_{i+2}.....a_n\$$

b. **Reduce :** If action $[S_m, a_i]$ = reduce $A \to \beta$ the parser executes a reduce move using the $A \to \beta$ production of the grammar. If $A \to \beta$ 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	Input
$S_0X_1S_1X_2.....X_{m-r}S_{m-r}AS$	$a_i a_{i+1}.....a_n\$$

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

$S \to \bullet ABC$

$S \to A \bullet BC$

$S \to AB \bullet C$

$S \to ABC \bullet$

The production $S \to \varepsilon$ generates only one item $S \to \bullet$.

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 $S' \to S$. The purpose of this grammar is to indicate the acceptance of input. That is when parser is about to reduce $S' \to S$, it reaches to acceptance state.

3. **Kernel items :** It is collection of items $S' \to \bullet S$ 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.

Scanned with CamScan

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 :

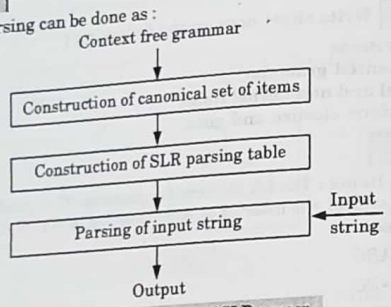


Fig. 2.22.1. Working of SLR parser.

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 $[A \to \alpha \bullet a\beta]$ 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 $[A \to \alpha \bullet]$ is in I_i the set ACTION $[i, a]$ to "reduce $A \to \alpha$" for all 'a' in FOLLOW (A).
3. If $[S' \to S \bullet]$ 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 $[S' \to \bullet S]$.

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 :**

$$S \to A)$$
$$S \to A, P \mid (P, P$$
$$P \to \{\text{num, num}\}$$

AKTU 2015-16, Marks 10

Answer

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

$$S' \to S$$
$$S \to A)$$
$$S \to A, P$$
$$S \to (P, P$$
$$P \to \{\text{num, num}\}$$

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

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

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

I_1 :
$$S' \to S \bullet$$

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

I_2 :
$$S \to A \bullet)$$
$$S \to A \bullet, P$$

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

I_3 :
$$S \to (\bullet P, P$$
$$P \to \bullet\{\text{num, num}\}$$

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

I_4 :
$$P \to \{\bullet \text{ num, num}\}$$

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

I_5 :
$$S \to A) \bullet$$

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

I_6 :
$$S \to A , \bullet P$$
$$P \to \bullet\{\text{num, num}\}$$

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

I_7 :
$$S \to (P\bullet, P$$

Scanned with CamScan

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

	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

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

Let us numbered the production rules in the grammar as :

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

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

Scanned with CamScan

Table : 2.24.1. SLR parsing table.

	Action			Goto	
States	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		

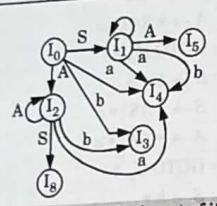


Fig. 2.24.1. DFA for set of items.

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

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

Que 2.25. **Consider the following grammar $E \to E+E \mid E*E \mid (E) \mid id$. Construct the SLR parsing table and suggest your final parsing table.**

AKTU 2017-18, Marks 10

Answer

The augmented grammar is as :

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

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

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

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

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

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

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

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

I_3: $E \to id\bullet$

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

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

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

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

Scanned with CamScan

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.

Scanned with CamScan

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			

Scanned with CamScan

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		

Scanned with CamScan

DFA :

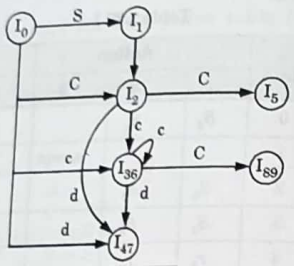


Fig. 2.29.1.

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

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

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

Answer

Augmented grammar G' for the given grammar :

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

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

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

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

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

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

$B \to \bullet d, a$

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

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

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

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

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

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

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

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

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

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			

Scanned with CamScan

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 :**

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

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

Answer

The given grammar is :

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

The augmented grammar will be :

$$S' \to S$$
$$S \to AA$$

$$A \to aA \mid b$$

The LR (1) items will be :

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

I_1 = GOTO (I_0, S)

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

I_2 = GOTO (I_0, A)

$$I_2 : S \to A\bullet A, \$$$
$$A \to \bullet aA, \$$$
$$A \to \bullet b, \$$$

I_3 = GOTO (I_0, a)

$$I_3 : A \to a\bullet A, a/b$$
$$A \to \bullet aA, a/b$$
$$A \to \bullet b, a/b$$

I_4 = GOTO (I_0, b)

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

I_5 = GOTO (I_2, A)

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

I_6 = GOTO (I_2, a)

$$I_6 : A \to a\bullet A, \$$$
$$A \to \bullet aA, \$$$
$$A \to \bullet b, \$$$

I_7 = GOTO (I_2, b)

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

I_8 = GOTO (I_3, A)

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

I_9 = GOTO (I_6, A)

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

The goto table will be

Scanned with CamScan

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

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

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

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

$I_{89}: A \to aA \bullet, a/b/\$$ 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 :

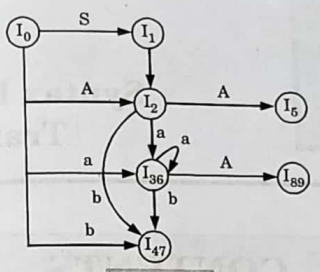


Fig. 2.31.1.

☺☺☺

Scanned with CamScan