

1

UNIT

Introduction to Compiler

CONTENTS

- Part-1** : Introduction to Compiler :.....1-2C to 1-6C
Phases and Passes
- Part-2** : Bootstrapping.....1-6C to 1-7C
- Part-3** : Finite State Machines and1-8C to 1-17C
Regular Expressions and
their Application to Lexical
Analysis, Optimization of
DFA Based Pattern Matchers
- Part-4** : Implementation of.....1-17C to 1-22C
Lexical Analyzers,
Lexical Analyzer Generator,
LEX Compiler
- Part-5** : Formal Grammars and1-22C to 1-25C
their Application to Syntax
Analysis, BNF Notation
- Part-6** : Ambiguity, YACC.....1-25C to 1-27C
- Part-7** : The Syntactic Specification.....1-27C to 1-30C
of Programming Languages :
Context Free Grammar (CFG),
Derivation and Parse Trees,
Capabilities of CFG

PART-1*Introduction to Compiler, Phases and Passes.***Questions-Answers****Long Answer Type and Medium Answer Type Questions****Que 1.1.**

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

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

AKTU 2016-17, Marks 10

Answer

A compiler contains 6 phases which are as follows :

i. Phase 1 (Lexical analyzer) :

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

ii. Phase 2 (Syntax analyzer) :

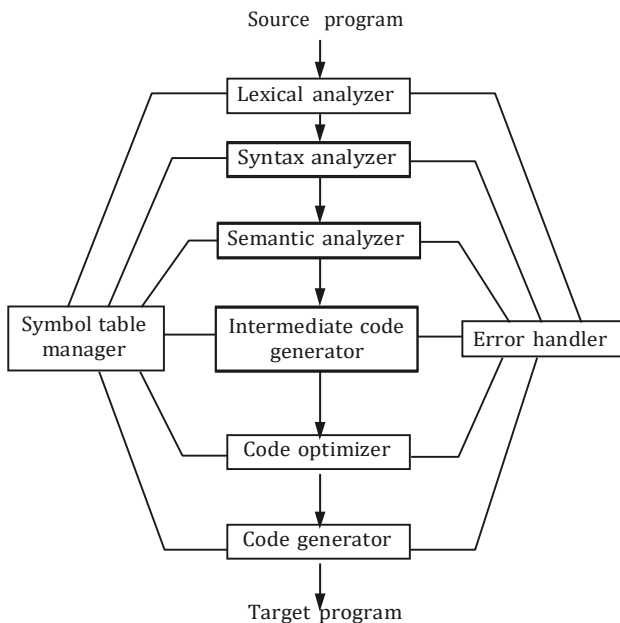
- The syntax analyzer phase is also called parsing phase.
- The syntax analyzer groups tokens together into syntactic structures.
- The output of this phase is parse tree.

iii. Phase 3 (Semantic analyzer) :

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

iv. Phase 4 (Intermediate code generation) :

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

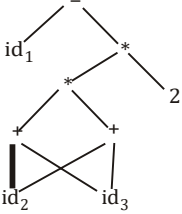
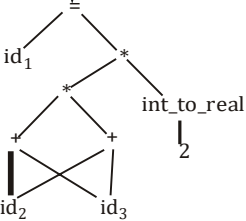
**Fig. 1.1.1.**

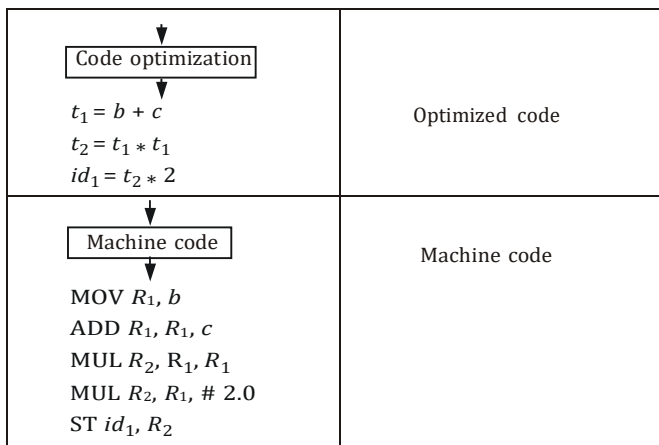
- v. **Phase 5 (Code optimization)** : This phase is designed to improve the intermediate code so that the ultimate object program runs faster and takes less space.
- vi. **Phase 6 (Code generation)** :
- It is the final phase for compiler.
 - It generates the assembly code as target language.
 - In this phase, the address in the binary code is translated from logical address.

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

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

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

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



Que 1.2. What are the types of passes in compiler ?

Answer

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.

Answer

Role of compiler writing tools :

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

Various compiler construction/writing tools are :

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

PART-2*Bootstrapping.***Questions-Answers****Long Answer Type and Medium Answer Type Questions**

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

OR

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

Answer

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

Bootstrapping :

1. Bootstrapping is the process of writing a compiler (or assembler) in the source programming language that it intends to compile.
2. Bootstrapping leads to a self-hosting compiler.
3. An initial minimal core version of the compiler is generated in a different language.

4. A compiler is characterized by three languages :
 - a. Source language (S)
 - b. Target language (T)
 - c. Implementation language (I)
5. ${}^S C^T$ 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 :

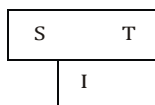


Fig. 1.4.1.

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.)

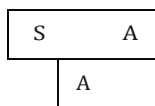


Fig. 1.4.2.

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

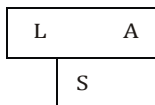


Fig. 1.4.3.

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

$${}^L C^A_S \rightarrow {}^S C^A_A \rightarrow {}^L C^A_A$$

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

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

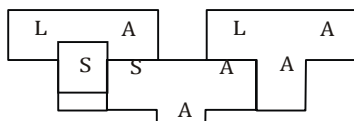


Fig. 1.4.4.

PART-3

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

Questions-Answers**Long Answer Type and Medium Answer Type Questions**

Que 1.5. What do you mean by regular expression ? Write the formal recursive definition of a regular expression.

Answer

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

Formal recursive definition of regular expression :

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

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

Que 1.6. Define and differentiate between DFA and NFA with an example.

Answer**DFA :**

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

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

where,

Q = A set of non-empty finite states

Σ = A set of non-empty finite input symbols

q_0 = Initial state of DFA

F = A non-empty finite set of final state

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

NFA :

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

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

where,

Q = A set of non-empty finite states

Σ = A set of non-empty finite input symbols

q_0 = Initial state of NFA and member of Q

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

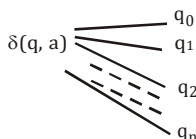


Fig. 1.6.1.

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

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

Difference between DFA and NFA :

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

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

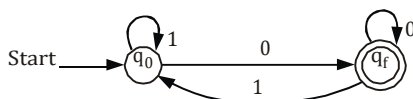


Fig. 1.6.2.

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

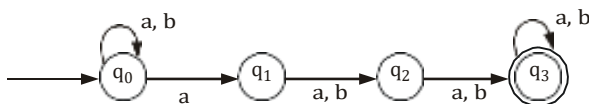


Fig. 1.6.3.

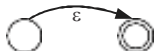
Que 1.7.

Explain Thompson's construction with example.

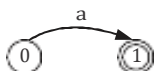
Answer

Thompson's construction :

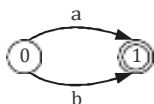
- It is an algorithm for transforming a regular expression to equivalent NFA.
- Following rules are defined for a regular expression as a basis for the construction :
 - The NFA representing the empty string is :



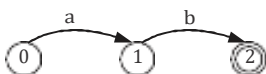
- ii. If the regular expression is just a character, thus a can be represented as :



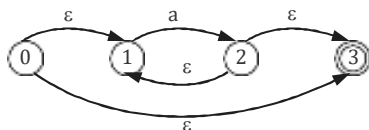
- iii. The union operator is represented by a choice of transitions from a node thus $a|b$ can be represented as :



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



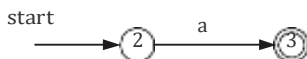
- v. The Kleene closure must allow for taking zero or more instances of the letter from the input; thus a^* looks like :



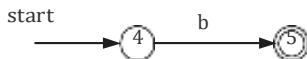
For example :

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

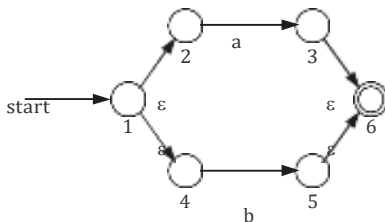
For $r_1 = a$,



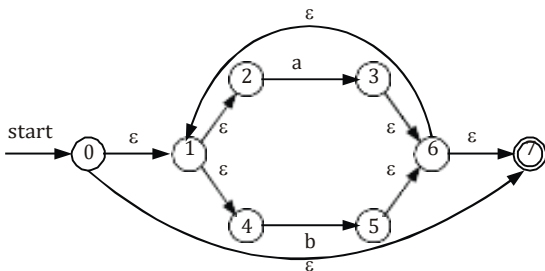
For $r_2 = b$,



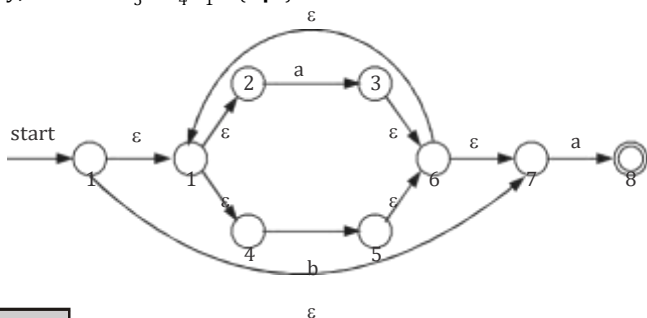
For $r_3 = a|b$



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



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



Que 1.8. Construct the NFA for the regular expression $a|abb|a^*b^+$ 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 :

q_6 ε

q_4

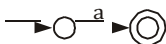
b

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

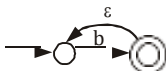
AKTU 2016-17, Marks 10

Answer

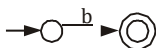
Step 1 : a



Step 2 : b^*



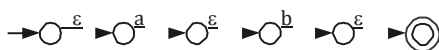
Step 3 : b



Step 4 : ab^*



Step 5 : ab



Step 6 : $ab^*|ab$

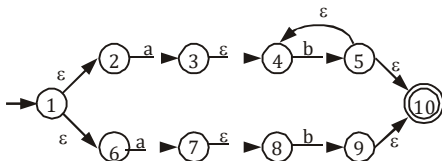


Fig. 1.9.1. NFA of $ab^*|ab$.

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 $N = \langle Q, \Sigma, q_0, \delta, F \rangle$ which recognizes a language L . Then the DFA $D = \langle Q', \Sigma, q_0, \delta', F' \rangle$ can be constructed for language L as :

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

Step 2 : Add q_0 to Q' .

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

it to Q' .

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

Que 1.11. Construct the minimized DFA for the regular expression

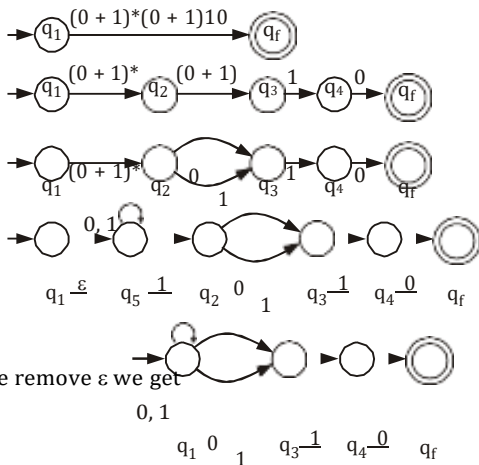
$(0 + 1)^*(0 + 1)10$.

AKTU 2016-17, Marks 10

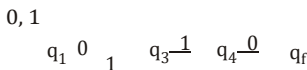
Answer

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

NFA for given regular expression :



If we remove ϵ we get



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

Now, we convert above NFA into DFA :

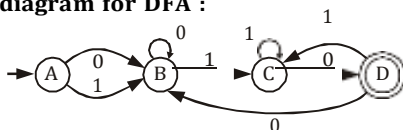
Transition table for NFA :

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
-----------------	---	---	-----

$\rightarrow q_1$ $q_1 q_3$ $q_1 q_3 q_4$ $* q_1 q_3 q_f$	$q_1 q_3$ $q_1 q_3$ $q_1 q_3 q_f$ $q_1 q_3$	$q_1 q_3$ $q_1 q_3 q_4$ $q_1 q_3 q_4$ $q_1 q_3 q_4$	q_1 as A $q_1 q_3$ as B $q_1 q_3 q_4$ as C $q_1 q_3 q_f$ 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 ?

Answer

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

3. This approach is so useful that programs called lexical analyzer generators exist to automate the entire process.

Que 1.13. Write down the regular expression for

- The set of all string over $\{a, b\}$ such that fifth symbol from right is a .
- 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 strings over $\{a, b\}$ such that fifth symbol from right is a :

Regular expression : $(a + b)^* a (a + b) (a + b) (a + b) (a + b)$

2. **Regular expression :**

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

Que 1.14. Convert following NFA to equivalent DFA and hence minimize the number of states in the DFA.

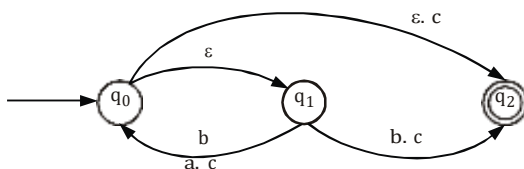


Fig. 1.14.1.

AKTU 2018-19, Marks 07

Answer

Transition table for ϵ -NFA :

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

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

ϵ -closure of $\{q_1\} = \{q_1\}$

ϵ -closure of $\{q_2\} = \{q_2\}$

Transition table for NFA :

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

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

$\{q_1, q_2\} = B$

$\{q_2\} = C$

Transition table for NFA :

δ/Σ	<i>a</i>	<i>b</i>	<i>c</i>
<i>A</i>	<i>A</i>	<i>B</i>	<i>A</i>
<i>B</i>	<i>A</i>	<i>C</i>	<i>A</i>
<i>C</i>	ϕ	ϕ	ϕ

Transition table for DFA :

δ/Σ	<i>a</i>	<i>b</i>	<i>c</i>
<i>A</i>	<i>A</i>	<i>B</i>	<i>A</i>
<i>B</i>	<i>A</i>	<i>C</i>	<i>A</i>
<i>C</i>	ϕ	ϕ	ϕ

So, DFA is given by

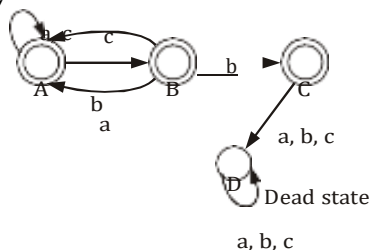


Fig. 1.14.2.

PART-4

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

Questions-Answers**Long Answer Type and Medium Answer Type Questions**

Que 1.15. 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.

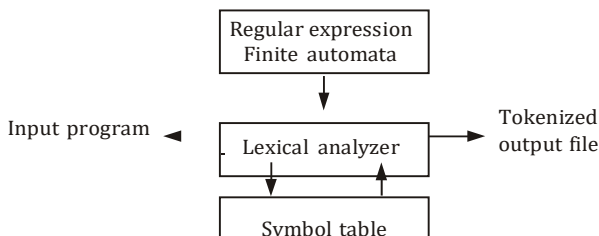


Fig. 1.15.1. Implementation of lexical analyzer.

5. This NFA are then converted into DFA and DFA are minimized by using different method of minimization.
6. The minimized DFA are used to recognize the pattern and broken into lexemes.
7. Each minimized DFA is associated with a phase 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 phases, and a routine which uses them appropriately.

Que 1.16. Write short notes on lexical analyzer generator.

Answer

1. For efficient design of compiler, various tools are used to automate the phases of compiler. The lexical analysis phase can be automated using a tool called LEX.

2. LEX is a Unix utility which generates lexical analyzer.
3. The lexical analyzer is generated with the help of regular expressions.
4. LEX lexer is very fast in finding the tokens as compared to handwritten LEX program in C.
5. LEX scans the source program in order to get the stream of tokens and these tokens can be related together so that various programming structure such as expression, block statement, control structures, procedures can be recognized.

Que 1.17. 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. 1.17.1. This lex.yy.c is a C program which is actually a lexical analyzer program.

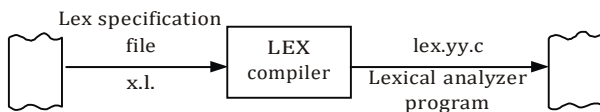


Fig. 1.17.1.

5. The LEX specification file stores the regular expressions for the token and the lex.yy.c file consists of the tabular representation of the transition diagrams constructed for the regular expression.
6. In specification file, LEX actions are associated with every regular expression.
7. These actions are simply the pieces of C code that are directly carried over to the lex.yy.c.
8. Finally, the C compiler compiles this generated lex.yy.c and produces an object program a.out as shown in Fig. 1.17.2.
9. When some input stream is given to a.out then sequence of tokens gets generated. The described scenario is shown in Fig. 1.17.2.

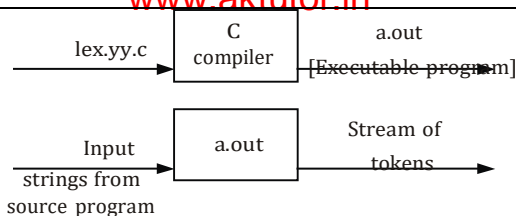


Fig. 1.17.2. Generation of lexical analyzer using LEX.

Que 1.18. Explain different parts of LEX program.

Answer

The LEX program consists of three parts :

```

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

1. Declaration section :

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

2. Rule section :

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

```

R1 {action1}
R2 {action2}
:
.
Rn {actionn}
  
```

Where each R_i is a regular expression and each action _{i} is a program fragment describing what action is to be taken for corresponding regular expression.

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

3. Auxiliary procedure section :

- In this section, all the procedures are defined which are required

by the actions in the rule section.

- b. This section consists of two functions :
- main() function
 - yywrap() function

Que 1.19. 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 1.20. What are the various LEX actions that are used in LEX programming ?

Answer

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

- BEGIN** : It indicates the start state. The lexical analyzer starts at state 0.
- ECHO** : It emits the input as it is.
- yytext()** :
 - yytext is a null terminated string that stores the lexemes when lexer recognizes the token from input token.
 - 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. **yylen** : `yylen` stores the length or number of characters in the input string.

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

Answer

Token :

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

Lexeme :

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

Pattern :

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

PART-5

*Formal Grammars and their Application to Syntax Analysis,
BNF Notation.*

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

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

Que 1.23. What is Context Free Grammar (CFG) ? Explain.

Answer

Context free grammar :

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

where,

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

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

$S \rightarrow$ starting symbol.

$P \rightarrow$ 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.

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

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

3. For example :

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

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

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

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

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

- Optional symbols are written with square brackets.
- For repeating the symbol for 0 or more number of times asterisk can be used.

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

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

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

- The alternative rules are separated by vertical bar.
- The group of items must be enclosed within brackets.

PART-6

Ambiguity, YACC.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.26. What is an ambiguous grammar ? Is the following grammar is ambiguous ? Prove $EE + |E(E)|id$. The grammar should be moved to the next line, centered.

AKTU 2016-17, Marks 10

Answer

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

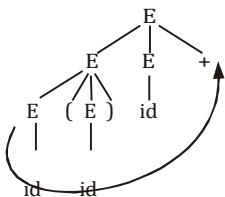
Proof : Let production rule is given as :

$$E \rightarrow EE +$$

$$E \rightarrow E(E)$$

$$E \rightarrow id$$

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



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

Que 1.27. Write short note on :

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

OR

Write a short note on YACC parser generator.

AKTU 2017-18, Marks 05

Answer

- i. **Context free grammar** : Refer Q. 1.23, Page 1-23C, Unit-1.
- ii. **YACC parser generator** :

1. YACC (Yet Another Compiler - Compiler) is the standard parser generator for the Unix operating system.
2. An open source program, YACC generates code for the parser in the C programming language.
3. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code.

Que 1.28. Consider the grammar G given as follows :

$$S \rightarrow AB|aaB$$

$$A \rightarrow a|Aa$$

$$B \rightarrow b$$

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

Answer

Given :

$$S \rightarrow AB|aaB$$

$$A \rightarrow a|Aa$$

$$B \rightarrow b$$

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

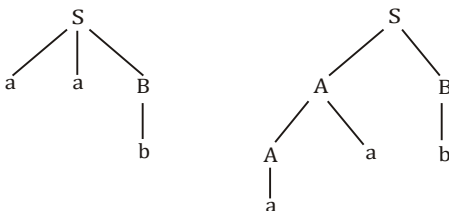


Fig. 1.28.1.

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 .

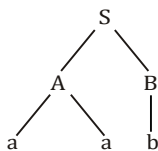


Fig. 1.28.2.

PART-7

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

Answer

Parse tree :

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

Conditions for constructing a parse tree from a CFG :

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

Answer

1. A derivation is a sequence of tokens that is used to find out whether a sequence of string is generating valid statement or not.
2. We can define the notations to represent a derivation.
3. First, we define two notations \Rightarrow_G and $\xRightarrow{*}_G$.
4. If $\alpha \rightarrow \beta$ is a production of P in CFG and a and b are strings in $(V_n \cup V_t)^*$, then

$$a\alpha b \Rightarrow_G a\beta b.$$

5. We say that the production $\alpha \rightarrow \beta$ is applied to the string $a\alpha b$ to obtain $a\beta b$ or we say that $a\alpha b$ directly drives $a\beta b$.
6. Now suppose $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_m$ are string in $(V_n \cup V_t)^*$, $m \geq 1$ and $\alpha_1 \xRightarrow[G]{\quad} \alpha_2, \alpha_2 \xRightarrow[G]{\quad} \alpha_3, \alpha_3 \xRightarrow[G]{\quad} \alpha_4, \dots, \alpha_{m-1} \xRightarrow[G]{\quad} \alpha_m$.
7. Then we say that $\alpha \xRightarrow[G]{\quad}^* \alpha$, i.e., we say α drives α in grammar G . If $\alpha \xRightarrow[G]{\quad}^i \beta$ drives by exactly i steps, we say $\alpha \xRightarrow[G]{\quad}^i \beta$.

Que 1.31. What do you mean by left most derivation and right most derivation with example ?

Answer

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

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

Firstly take the production

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

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

Example : Let us consider a grammar G having production.

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

Start with production

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

Que 1.32. Describe the capabilities of CFG.

Answer

Various capabilities of CFG are :

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

VERY IMPORTANT QUESTIONS

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

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

Ans. Refer Q. 1.1.

Q. 2. Define and differentiate between DFA and NFA with an example.

Ans. Refer Q. 1.6.

Q. 3. Construct the minimized DFA for the regular expression $(0 + 1)^*(0 + 1)10$.

Ans. Refer Q. 1.11.

Q. 4. Explain the implementation of lexical analyzer.

Ans. Refer Q. 1.15.

Q. 5. Convert following NFA to equivalent DFA and hence minimize the number of states in the DFA.

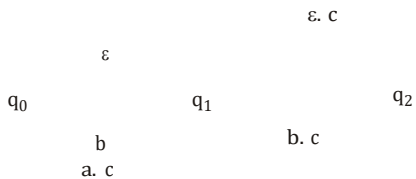
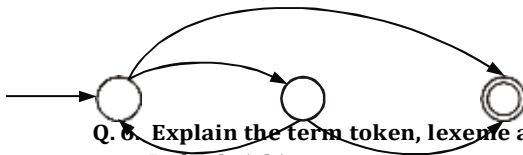


Fig. 1.

Ans. Refer Q. 1.14.



Q. 6. Explain the term token, lexeme and pattern.

Ans. Refer Q. 1.21.

**Q. 7. What is an ambiguous grammar ? Is the following grammar is ambiguous ? Prove $EE + |E(E)|id$.
The grammar should be moved to the next line, centered.**

Ans. Refer Q. 1.26.

Q. 8. Define parse tree. What are the conditions for constructing a parse tree from a CFG ?

Ans. Refer Q. 1.29.

