STM JOURNALS
Scientific Technical Medical

Journal of Advances in Shell Programming

Volume 1, Issue 1
www.stmjournals.com

An Approach to Design a Compiler in Context of Lexical Analyzer (Scanner) and Parser Generation (Parser) Followed by Non-optimized Intermediate Code Generation using Compiler Construction Tools

Partha Ghosh*
Department of Computer Science & Engineering, Govt. College of Engineering & Ceramic Technology, Kolkata, India

Abstract

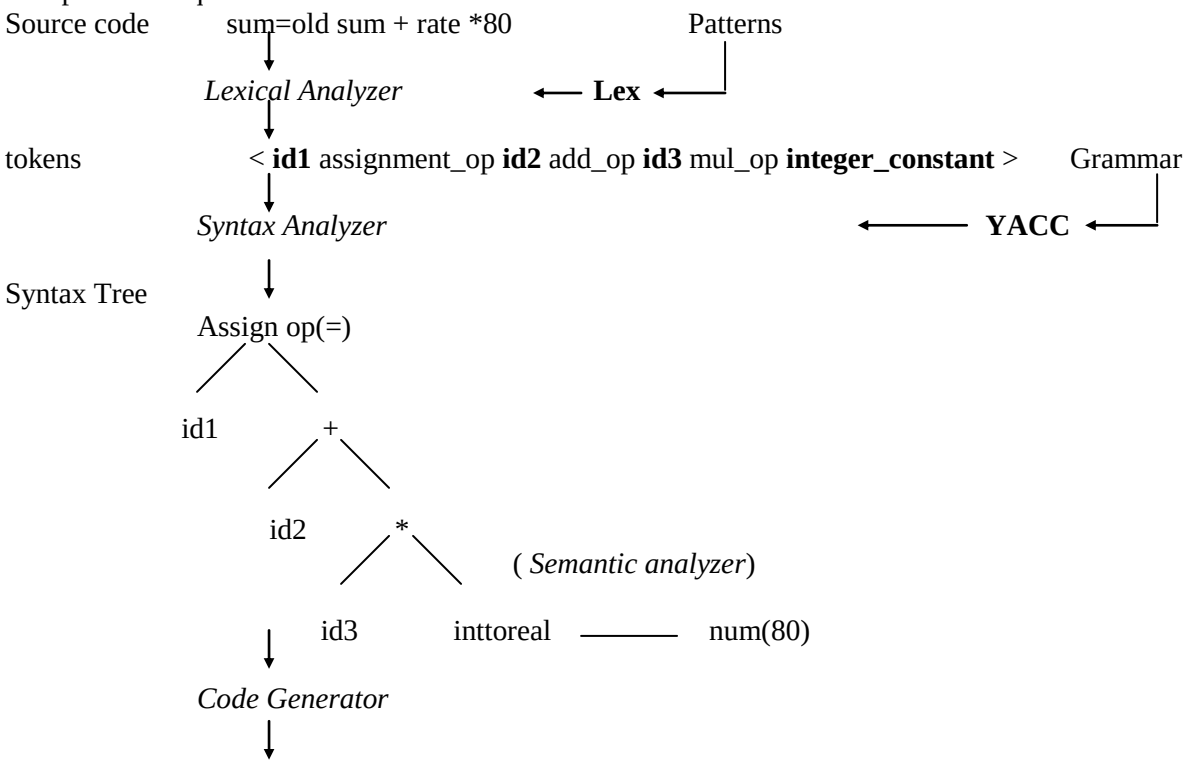
It is very tedious and lengthy task to write a compiler. To realize the various phases of compilers we can use some specific tools. These tools are called compiler construction tools. Furthermore, we call it as compiler-compiler, compiler-generators, or translator writing system. A mixture of compiler building tools are Scanner generator —it generates lexical analyzers. The patterns specified to these generators are in the type of regular expressions. The LINUX has utility for a scanner generator called LEX. The specification given to the LEX consists of regular expressions for representing various tokens. Parser generators — these produce the syntax analyzer. The specification given to these generators is given in the form of CFG. Typically LINUX has a tool called YACC, which is a parser generator. Syntax-directed translation engines — in this tool the parse tree is scanned fully to generate an intermediate code. The translation is done for each node of the tree.

Keywords: *Tokens, Lexeme, Lex, AST, Yacc*

**Author for Correspondence* E-mail: parth_ghos@rediffmail.com

INTRODUCTION

Compilation Sequence:



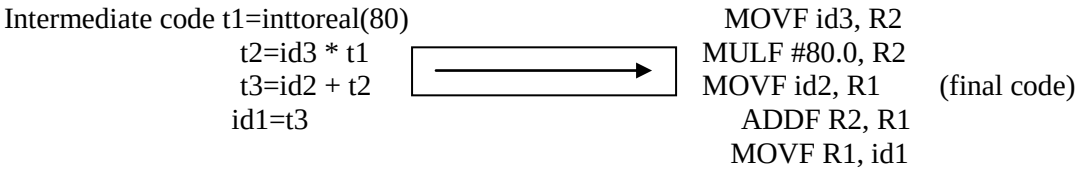
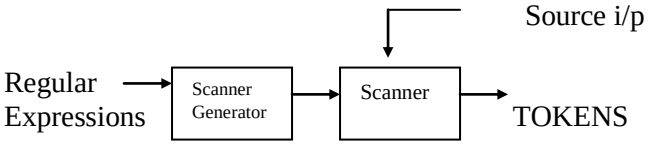
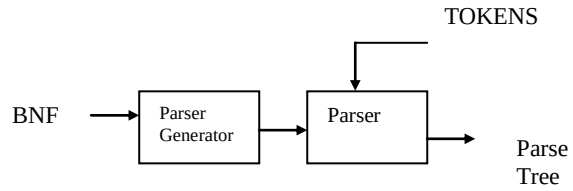


Fig. 1: *Order of Compilation.*

•The main job of a *lexical analyzer (scanner)* is to break up an input stream into more usable elements called *tokens [1]*

a	=	b	+	c	*	d	;
ID	ASSIGN	ID	PLUS	ID	MULT	ID	SEMI





•Lex

•Lex [2–4] is an utility to help you rapidly generate your scanners

•Lex source is a table of

–regular expressions and

–corresponding program fragments

e.g.,

```
digit                [0-9]
letter               [a-zA-Z]
%%
{letter}({letter}|{digit})*   printf("id: %s\n", yytext);
\n                            printf("new line\n");
%%
main() {
                     yylex();
}
```

The tokens are listed below. The bolded words are the token *class* and the words between quotes, " " are the **lexemes** [1].

Type	"void", "int", "real"
Logical Operators	"||", "&&", "!=", "==", "<", ">", "<=", ">="
Numerical Operators	"+", "-", "*", "/", "="
Punctuation	"{", "}", "(", ")", ",", ";"
Keywords	"if", "else", "while", "do", "for"

Real

Plus or minus followed by a number of digits followed by a dot ".", followed by a number of digits. Either the sequence before the dot can be null or the sequence after the dot can be null, but not both.

For example **(type, "void")** and **(Punctuation, "(")** are tokens.

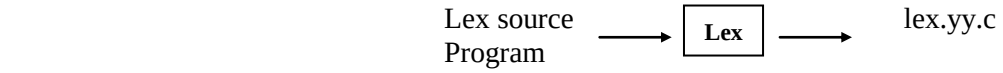
•Yacc [2],[3],[4]

–Yacc generates C code for syntax analyzer, or parser.

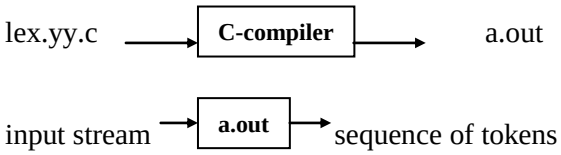
–Yacc uses grammar rules that allow it to analyze tokens from Lex and create a syntax tree.

WORKING PRINCIPLE OF LEX: LEXER GENERATION

The **patterns** in the above Figure 1 are a file we create with a text editor. Lex will interpret this patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on our patterns, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing.



Lexical analyzer enters identifiers in a symbol table from input stream. supplementary information such as data type (integer or real) and location of each variable in memory are also maintained in the symbol. All ensuing references to identifiers refer to the proper symbol table index.



Lex Source

•Lex source is separated into three sections by %% delimiters

•The general format of Lex source is

- A lex specification consists of three parts: regular definitions, C declarations in
  %{ %}
  %%
  translation rules
  %%
  user-defined auxiliary procedures

- The translation rules are of the form:

regexp <one or more blanks> { actions (C code) }

p_1 {action$_1$}
p_2 {action$_2$}
…
p_n {action$_n$}

•The absolute minimum Lex program is thus
%%

Extended Regular Expressions of LEX

•A regular expression matches a set of strings

***Character Classes* [ ]**

•[abc] matches a single character, which may be a, b, or c

•Every operator meaning is ignored except \ - and ^

•e.g.
[ab] => a or b
[a-z] => a or b or c or…or z
[-+0-9] => all the digits and the two signs
[^a-zA-Z] => any character which is not a letter.

Arbitrary Character.

• The operator character (**.**) is used to match almost character of all characters excluding newline

• Octal 40 (blank) to octal 176 (tilde~) are used to matches all printable characters in the ASCII character set

Optional & Repeated Expressions

a? => zero or one instance of a
a* => zero or more instances of a
a+ => one or more instances of a

•E.g.
ab?c => ac or abc
[a-z]+ => all strings of lower case letters
[a-zA-Z][a-zA-Z0-9]* => all alphanumeric strings with a leading alphabetic character

Precedence of Operators

•Level of precedence
–Kleene closure (*), ?, +
–concatenation
–alternation (|)

•All operators are left associative.

•Ex: a*b|cd* = ((a*)b)|(c(d*))

Lex Predefined Variables

yytext → a string containing the lexeme .
yyleng → the length of the lexeme
yyin → the input stream pointer
–the default input of default main() is **stdin**
yyout → the output stream pointer
–the default output of default main() is **stdout**.

- e.g.

[a-z]+ printf("%s", yytext);
[a-z]+ ECHO;
[a-zA-Z]+ {words++; chars += yyleng;}

The unmatched token is using default actions that ECHO from the input to the output.

Lex Library Routines

yylex() – The default main() contains a call of yylex()
yymore() – return the next token
yyless(n) – retain the first n characters in yytext
yywarp() – is called whenever Lex reaches an end-of-file

– The default yywarp() always returns 1

WORKING PRINCIPLE OF YACC: PARSER GENERATION

Backus Naur Form , BNF [5–7] are used to express the grammar of yacc. To express *context-free* languages BNF grammar can be used.

For instance, the grammar intended for an expression that multiplies and adds numbers is
E → E + E
E → E * E
E → id

E, expression are nonterminals. Terms such as **id,** identifier are terminals and only become visible on the right-hand side of a production.

YACC File Format

... definitions ... --> definitions segment contains token declarations and C code bracketed by "**%{**" and "**%}**".
%%
... rules...--> The BNF grammar is to be found in the rules segment
%%
... subroutines ... --> user subroutines

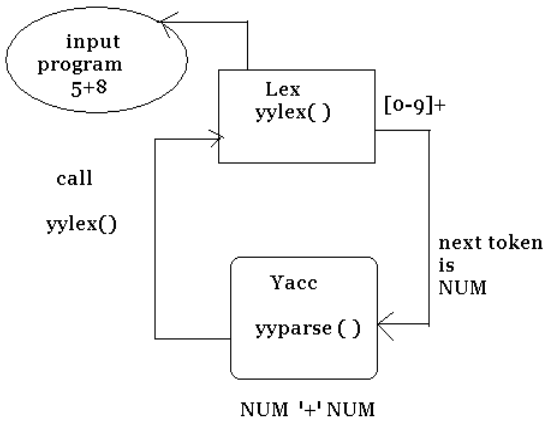


Fig. 2: *Communication between LEX and YACC.*

Definitions Section

```
%{
#include <stdio.h>
#include <stdlib.h>
%}          ⟶   It is a terminal
%token ID NUM
%start expr
```

Start Symbol

The first non-terminal specified in the grammar specification section.
To overwrite it with %start declaraction.
%start non-terminal

Rules Section

This section defines grammar. Normally written like this

- Example:

```
expr   : expr '+' term
       | term
       ;
term   : term '*' factor
       | factor
       ;
factor : '(' expr ')'
       | ID
       | NUM
       ;
```

The Position of Rules

```
                $1    $2    $3
                ↓     ↓     ↓
expr    : expr  '+'   term      { $$ = $1 + $3; }
        | term                  { $$ = $1; }
        ;

term    : term  '*'   factor    { $$ = $1 * $3; }
        | factor                { $$ = $1; }
        ;

factor  : '('   expr  ')'       { $$ = $2; }
        | ID
        | NUM
        ;
                                Default: $$=$1;
```

Shift/Reduce Conflicts

It occurs when a grammar is written in such a way that a decision between shifting and reducing cannot be made.

- ex: IF-ELSE ambiguous.

To resolve this conflict, **yacc** will choose to shift.

YACC Declaration

`%start`	Specify the grammar's start symbol
`%union`	Declare the collection of data types that semantic values may have
`%token`	Declare a terminal symbol

	(token type name) with no precedence or associativity specified
`%type`	Declare the type of semantic values for a non-terminal symbol
`%right`	Declare a terminal symbol (token type name) that is right-associative
`%left'	Declare a terminal symbol (token type name) that is left-associative
`%nonassoc'	Declare a terminal symbol (token type name) that is non-associative

(using it in a way that would be associative is a syntax error, ex: x *op*. y *op*. z is syntax error)

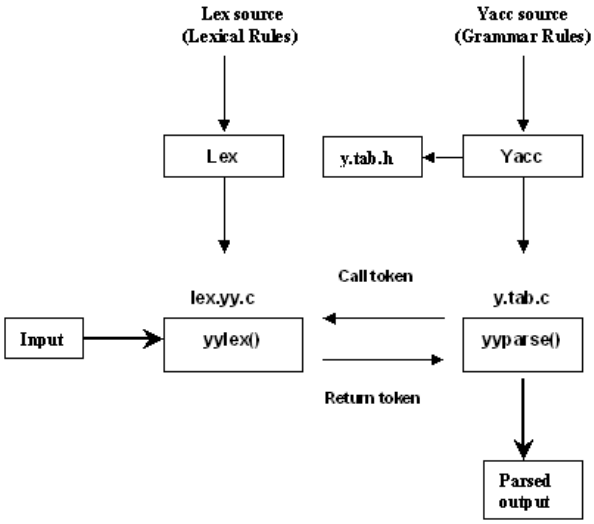
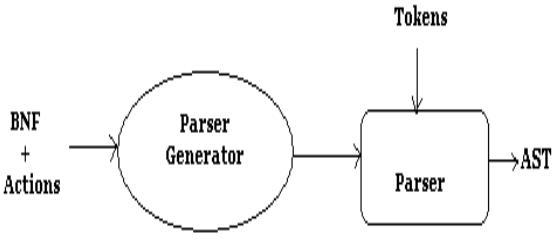


Fig. 3: *Lex with Yacc.*

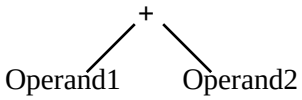
INPLEMENTATION APPROACH OF ABSTRACT SYNTAX TREE: AST

Abstract Syntax Trees (AST)



In programming languages, we choose a grammar that is close to the language constructs. We call this grammar the abstract grammar. The corresponding derivation tree is called the abstract syntax tree. Each node is an component of the language. Operators are represented by non-leaf nodes while *operands* are represented by the leaf nodes.

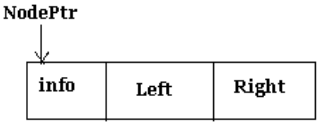
Example: Abstract Syntax Tree for *p* + *q* where p is operand1 and q is operand2.



In parenthesized form, this might be written as:
(+ (p q))

The production that relates to such a node is:
addop_expression → mulop_expression { addop mulop_expression }

If we create a data structure for the nodes of the abstract syntax tree as:



then we can add attributes and functions such as:

addop_expression --> mulop_expression
{addop mulop_expression }
addop_expression.*NodePtr* = **MakeNode**
addop_expression.*Info* = "+"
addop_expression.*Left* =
mulop_expression$_1$.*NodePtr*
addop_expression.*Right* =
mulop_expression$_2$.*NodePtr*

These statements might create an AST node (pointed to by NodePtr) with "+" in the info field and pointers to addop_ and mulop_expression. We have to write the code for **make_node**.

Yacc does have some built-in variables that make this easier: **$n** refers to the value of the nth symbol on the right hand side of the rule; **$$** refers to the value of the non-terminal symbol on the left-hand side. Typically you write **$$ = f($1, $2, ...$m)** next to the production where **f** is a function written by me.

When we write $1, and that value has been assigned via $$ in a previous production, the value is passed up the tree.

For example,

```
addop_expr   :  addop_expr ADDOP
                mulop_expr   { $$ =
                make_node("+",$1,$3); }
```

During parsing a node will be created, pointers to the left and right will be entered into the correct fields and "$$" will contain a pointer to the current AST.

We will have to use, edit or write a "printtree" function that will be called from our main program after yacc has parsed and created the tree, e.g.,

```
int main (void) {return yyparse ( );} { …….. }
```

Intermediate Code generation

It uses the AST created before to generate intermediate code.

EXPERIMENTAL RESULTS

Running lex program to generate the scanner or Lexer:

In Lexical Analysis (Scanning) the following occurs:

Step 1: a version of ***lex*** creating a Scanner that will recognize the tokens described there. It is essentially a C program.

```
[root@localhost]# lex lex.l
```

Now let's see what files are there:

```
[root@localhost]#  ls
lex.l   lex.yy.c
```

We can see that *lex* has created a C program called **lex.yy.c**. **This is our Scanner**, but we have to compile it first:

Step 2: Then the C compiler (*cc*), compiles this to create an executable scanner.

```
[root@localhost]# cc lex.yy.c -ll
[root@localhost]#  ls
a.out    lex.l    lex.yy.c
```

a.out is the **executable scanner**.

Step 3: This scanner runs the input program on the right producing a set of tokens.

```
[root@localhost]# ./a.out
39
integer
r3d2
Identifier
9rdr
positive integer
Identifier
```

Scanner Input:

```
#include <stdio.h>
#include <conio.h>

 void main()
  {
   int a,_b,3c,x,y;
  cslr();
  printf("Enter value of a\n"); /* value of a*/
  scanf("%d",&a);
  printf("Enter value of b\n");
  scanf("%d",&b);
  x=a&b;
  x=(a>b)?a:b;
  y=a|b;

  if((a>b)||(b>5))
  c=a-b;
  else
  c=b-a                //subtractinging of  values
  printf("Result is %d",c); //printing the result
  getch();
  }
```

Output:

```
#include<stdio.h> is a PREPROCESSOR
                        DIRECTIVE
#include<conio.h> is a PREPROCESSOR
                        DIRECTIVE
NEW LINE
NEW LINE
void is a KEYWORD
FUNCTION CALL OR DEFINITION main( )
NEW LINE
BLOCK BEGINS
NEW LINE
Int                  is a KEYWORD
a                    is an IDENTIFIER,
_b                   is a illegal word,
3c                   is a illegal word,
x                    is an IDENTIFIER,
y                    is an IDENTIFIER;
NEW LINE
FUNCTION CALL OR DEFINITION cslr( );
NEW LINE
FUNCTION CALL OR DEFINITION printf(
"Enter value of a\n"       is a STRING);
/* value of a */ is a MULTI LINE COMMENT
NEW LINE
FUNCTION CALL OR DEFINITION    scanf(
"%d"                 is a STRING,
&                    is a BITWISE OPERATOR
a                    is an IDENTIFIER
);
```

```
NEW LINE
FUNCTION CALL OR DEFINITION printt (
"Enter value of b\n"          is a STRING);
NEW LINE
FUNCTION CALL OR DEFINITION scanf(
"%d"          is a STRING,
&             is a BITWISE OPERATOR
b             is an IDENTIFIER
);
NEW LINE
x       is an IDENTIFIER
=       is an ASSIGNMENT OPERATOR
a       is an IDENTIFIER
&       is a BITWISE OPERATOR
b       is an IDENTIFIER;
NEW LINE
x       is an IDENTIFIER
=       is an ASSIGNMENT OPERATOR(
a       is an IDENTIFIER
>       is a RELATIONAL OPERATOR
b       is an IDENTIFIER
)
?:      is a TERNARY OPERATOR;
NEW LINE
y       is an IDENTIFIER
=       is an ASSIGNMENT OPERATOR
a       is an IDENTIFIER
|       is a BITWISE OPERATOR
b       is an IDENTIFIER;
NEW LINE
NEW LINE
FUNCTION CALL OR DEFINITION    if((
a       is an IDENTIFIER
||      is a LOGICAL OPERATOR (
b       is an IDENTIFIER
>       is a RELATIONAL OPERATOR
5       is a NUMBER CONSTANT
)
)
NEW LINE
c       is an IDENTIFIER
=       is an ASSIGNMENT OPERATOR
a       is an IDENTIFIER
-       is an ARITHMETIC OPERATOR
b       is an IDENTIFIER;
NEW LINE
else    is a KEYWORD
NEW LINE
c       is an IDENTIFIER
=       is an ASSIGNMENT OPERATOR
b       is an IDENTIFIER
-       is an ARITHMETIC OPERATOR
a       is an IDENTIFIER
//subtracting of va1ues   is a SINGLE LINE
COMMENT NEW LINE
FUNCTION CALL OR DEFINITION printf(
"Result is %d"  is a STRING,
c               is an IDENTIFIER
);
//printing the result       is a SINGLE LINE
                            COMMENT
NEW LINE
FUNCTION CALL OR DEFINITION getch(
);
NEW LINE
BLOCK ENDS
NEW LINE
Total, no of lines 22
```

Running lex and yacc program to generate bottom-up parser:

As before, if the **lex** file is in a file named **lex.l**, then we just type in

[root@localhost]# lex lex.l

And, as before **lex** creates the file **lex.yy.c**. Now we want yacc to create the parser from our file which we'll call ***ly.y***. Type:

[root@localhost]# yacc –d ly.y

This creates a file called ***y.tab.c*** and the "-d" creates a file of definitions called ***y.tab.h***. which when compiled produce our parser.

To compile:

[root@localhost]# cc y.tab.c lex.yy.c

which, as usual, creates the file **a.out.**

Running a.out

```
./a.out
2 + 3 * 4
(2 + 3 ) * 4
^C (exits) [correct]
```

```
./a.out
3 2 + +
syntax error
^C (exits)
```

YACC Parsing Input:

1+2-5+(5*22);

Output:

```
found NUMBER 1
found factor
found term
found NUMBER 2
found factor
found exp PLUS term
found NUMBER 5
found factor
found exp MINUS term
found NUMBER 5
found factor
found NUMBER 22
found term TIMES factor
found term
found PARENTHESIS exp
found factor
found exp PLUS term
found exp END
```

It is really hard to see the bottom-up parse from the output; you should be able to create a parse from the output (which is the reverse of a leftmost derivation).

We will add semantic actions to create an abstract syntax tree:
Yacc Parsing Input:
1+1;
4-5+7*8;
6-6*7+(9*16);
22-12*(23+45)*56;

Output:
(+ 1 1)
(+ (- 4 5)(* 7 8))
(+ (- 6 (* 6 7))(* 9 16))
(- 22 (* (* 12 (+ 23 45)) 56))

Now we will generateIntermediate Code:
Yacc Parsing Input:
22-12*(23+45)*56;

Output:
(-22 (* (* 12 (+ 23 45)) 56))

```
PUSH 22
PUSH 12
PUSH 23
PUSH 45
POP A
POP B
ADD A= A+B
PUSH A
POP A
POP B
MULT A= A*B
PUSH A
PUSH 56
POP A
POP B
MULT A= A*B
PUSH A
POP A
POP B
SUB A= A-B
PUSH A
```

CONCLUSION

Processing time of FA proportional to the size of the input file and not on the number of regular expressions used. Although the number of expressions may impact number of internal states and therefore in space. Here we use as much as possible regular expressions and as little as possible action processing in C. We use regular expressions more specific at the beginning of the LEX file specification (for example keyword) and more generic regular expressions at the end (for example identifiers). Further study on extending this model with optimization of code and error recovery in all phases of compiler is in steps forward.

REFERENCES

1. Alfred V. Aho, Ravi Sethi, Jeffery D. Ullman, *Compilers- Principles, Techniques, and Tools*: Addison-Wesley; 2007.
2. William A. Barrett, *Compiler Design*, CmpE 152, FALL Version, San Jose State University; 2005.
3. G.M6nier, G. Lorette, Lexical Analyzer based on a Self-Organizing Feature Map, *IEEE Xplore*. 1997. 0-8186-7898-4.
4. Lex, Yacc, *UNIX Programming/UNIX Utilities*. 2nd Edn; John R. Levine, Tony Mason and Doug Brown–O'Reilly (Ed.): ISBN 1-56592-000-7.
5. Jeffrey E.F. Friedl–O'Reilly, *Mastering Regular Expressions*. 1997, ISBN: 1-56592-257-3.
6. William M. Waite, Assad Jarrahian, Michele H. Jackson, Amer Diwan, 2006. *Design and Implementation of a Modern Compiler Course*, ACM 1595930558/06/0006.
7. K.L.P. Mishra, N.Chandrasekharan, *Theory of Computer Science* 2007, 3rd Edn., PHI.