LAB MANUAL OF COMPILER DESIGN

Department of Electronics & Computer Engg.
Dronacharya College Of Engineering
Khentawas, Gurgaon – 123506

LIST OF EXPERIMENTS

S. No.	AIM OF EXPERIMENT
1.	STUDY OF LEX AND YACC TOOLS.
2	TO CONVERT REGULAR EXPRESSION INTO NFA.
3	WAP TO FIND FIRST IN CFG.
4.	WAP TO FIND STRING IS KEYWORD OR NOT.
5.	WAP TO FIND STRING IS IDENTIFIER OR NOT.
6.	WAP TO FIND STRING IS CONSTANT OR NOT.
7.	WAP TO COUNT NO. OF WHITESPACES AND NEWLINE.
8.	WAP TO GENERATE TOKENS FOR THE GIVEN GRAMMER.
9.	AN ALGO TO CONVERT NFA TO DFA.
10.	AN ALGO FOR MINIMIZING OF DFA.
11.	WAP TO CHECK STRING IS IN GRAMMER OR NOT.
12.	WAP TO CALCULATE LEADING FOR ALL NON TERMINALS .
13.	WAP TO CALCULATE TRAILING FOR ALL NON TERMINALS .

PROGRAM NO:-1

PRACTICE OF LEX/YACC OF COMPILER WRITING

A compiler or interpreter for a programming language is often decomposed into two parts:

1. Read the source program and discover its structure.
2. Process this structure, e.g. to generate the target program.

Lex and *Yacc* can generate program fragments that solve the first task.

The task of discovering the source structure again is decomposed into subtasks:

1. Split the source file into tokens (*Lex*).
2. Find the hierarchical structure of the program (*Yacc*).

Lex - A Lexical Analyzer Generator

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a

deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

Lex generates programs to be used in simple lexical analysis of text. The input *files* (standard input default) contain regular expressions to be searched for and actions written in C to be executed when expressions are found.

A C source program, lex.yy.c is generated. This program, when run, copies unrecognized portions of the input to the output, and executes the associated C action for each regular expression that is recognized.

The options have the following meanings.
–t Place the result on the standard output instead of in file lex.yy.c.
–v Print a one–line summary of statistics of the generated analyzer.
–n Opposite of –v; –n is default.
–9 Adds code to be able to compile through the native C compilers.

EXAMPLE

This program converts upper case to lower, removes blanks at the end of lines, and replaces multiple blanks by single blanks.

```
%%
[A–Z]       putchar(yytext[0]+'a'–'A');
[ ]+$
[ ]+ putchar(' ')
```

1. Introduction.

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called ``host languages.'' Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran (in the form of Ratfor [2] has been available in the past. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

```
        +-------+
Source -> |  Lex  |  -> yylex
        +-------+

        +-------+
Input ->  | yylex | -> Output
        +-------+
```

An overview of Lex
Figure 1

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$   ;
```

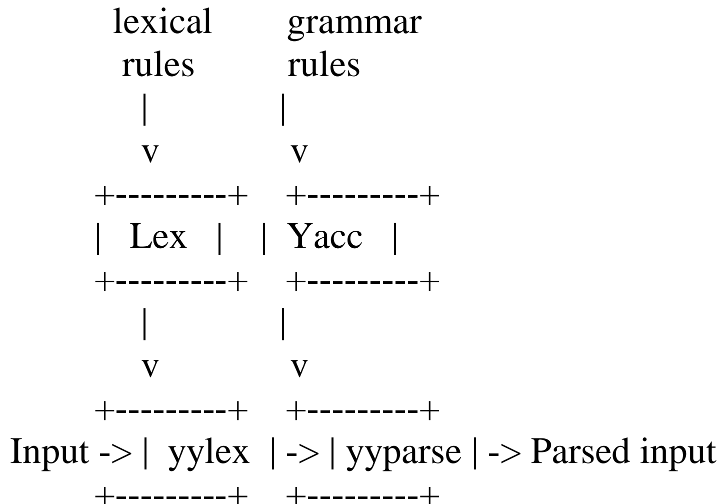
is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates ``one or more ...''; and the $ indicates ``end of line,'' as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$   ;
[ \t]+    printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc [3]. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces.

The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex.



Lex with Yacc
Figure 2

Yacc users will realize that the name yylex is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for ab and another for abcdefg, and the input stream is abcdefh, Lex will recognize ab and leave the input pointer just before cd. . . Such backup is more costly than the processing of simpler languages.

2. Lex Source.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions (see section 3) and the right column contains actions, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer   printf("found keyword INT");
```

to look for the string integer in the input stream and print the message ``found keyword INT'' whenever it appears. In this example the host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour     printf("color");
mechanise  printf("mechanize");
petrol     printf("gas");
```

would be a start. These rules are not quite enough, since the word petroleum would become gaseum; a way of dealing with this will be described later.

3. Lex Regular Expressions.

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string integer wherever it appears and the expression

```
a57D
```

looks for the string a57D.

Operators. The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string xyz++ when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C escapes with \ are

recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes. Classes of characters can be specified using the operator pair []. The construction [abc] matches a single character, which may be a, b, or c. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ - and ^. The - character indicates ranges. For example,

[a-z0-9<>_]

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using - between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0-z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character - in a character class, it should be first or last; thus

[-+0-9]

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

[^abc]

matches all characters except a, b, or c, including all special or control characters; or

[^a-zA-Z]

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character . is the class of all characters except newline. Escaping into octal is possible although non-portable:

[\40-\176]

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator ? indicates an optional element of an expression. Thus

```
ab?c
```

matches either ac or abc.

Repeated expressions. Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive a characters, including zero; while

```
a+
```

is one or more instances of a. For example,

```
[a-z]+
```

is all strings of lower case letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

```
(ab|cd)
```

matches either ab or cd. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
(ab|cd+)?(ef)*
```

matches such strings as abefef, efefef, cdef, or cddd; but not abc, abcd, or abcdef.

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [] operators. If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

```
ab/cd
```

matches the string ab, but only if followed by cd. Thus

ab$

is the same as

ab/\n

Left context is handled in Lex by start conditions as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition x, the rule should be prefixed by

<x>

using the angle bracket operator characters. If we considered ``being at the beginning of a line'' to be start condition ONE, then the ^ operator would be equivalent to

<ONE>

Start conditions are explained more fully later.

Repetitions and Definitions. The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

{digit}

looks for a predefined string named digit and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

a{1,5}

looks for 1 to 5 occurrences of a.

Finally, initial % is special, being the separator for Lex source segments.

4. Ambiguous Source Rules.

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1) The longest match is preferred.

2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer   keyword action ...;
[a-z]+    identifier action ...;
```

to be given in that order. If the input is integers, it is taken as an identifier, because [a-z]+ matches 8 characters while integer matches only 7. If the input is integer,

both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. int) will not match the expression integer and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example, '.*' might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, will stop after 'first'. The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like .* stop on the current line. Don't try to defeat this with expressions like (.|\n)+ or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both she and he in an input text. Some Lex rules to do this might be

```
she   s++;
he    h++;
\n    |
.     ;
```

where the last two rules ignore everything besides he and she. Remember that . does not include newline. Since she includes he, Lex will normally not recognize the instances of he included in she, since once it has passed a she those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means ``go do the next alternative.'' It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of he:

```
she   {s++; REJECT;}
```

```
he    {h++; REJECT;}
\n    |
.     ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that she includes he but not vice versa, and omit the REJECT action on he; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+   { ... ; REJECT;}
a[cd]+   { ... ; REJECT;}
```

If the input is ab, only the first rule matches, and on ad only the second matches. The input string accb matches the first rule for four characters and then the second rule for three characters. In contrast, the input accd agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word the is considered to contain both th and he. Assuming a two-dimensional array named digram to be incremented, the appropriate source is

```
%%
[a-z][a-z]   {
          digram[yytext[0]][yytext[1]]++;
          REJECT;
          }
.         ;
\n        ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

5. Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
```

```
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule. As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define Lex substitution strings. The format of such lines is name translation and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D          [0-9]
```

```
E               [DEde][-+]?{D}+
%%
{D}+              printf("integer");
{D}+"."{D}*({E})?  |
{D}*"."{D}+({E})?  |
{D}+{E}
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as 35.EQ.I, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ   printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs.

Yacc: Yet Another Compiler-Compiler

yacc is a computer program that serves as the standard parser generator on Unix systems. The name is an acronym for "Yet Another Compiler Compiler." It generates a parser (the part of a compiler that tries to make sense of the input) based on an analytic grammar written in BNF notation. Yacc generates the code for the parser in the C programming language.
Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

Yacc converts a context–free grammar and translation code into a set of tables for an LR(1) parser and translator. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, y.tab.c, must be compiled by the C compiler to produce a program yyparse. This program must be loaded with a lexical analyzer function, yylex(void) (often generated by *lex*(1)), with a main(int argc, char *argv[]) program, and with an error handling routine, yyerror(char*).

The options are
–o *output* Direct output to the specified file instead of y.tab.c.
–D*n* Create file y.debug, containing diagnostic messages.

v Create file y.output, containing a description of the parsing tables and of conflicts arising from ambiguities in the grammar.
–d Create file y.tab.h, containing #define statements that associate *yacc*–assigned `token codes' with user–declared `token names'. Include it in source files other than y.tab.c to give access to the token codes.
–s *stem* Change the prefix y of the file names y.tab.c, y.tab.h, y.debug, and y.output to *stem*.
–S Write a parser that uses Stdio instead of the print routines in libc.

1: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C[1] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
        date  :  month_name  day  ','  year   ;
```

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma ``,'' is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
        July  4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
        month_name  :  'J' 'a' 'n'   ;
        month_name  :  'F' 'e' 'b'   ;

                . . .

        month_name  :  'D' 'e' 'c'   ;
```
might be used in the above example. The lexical analyzer would only need to recognize individual letters, and month_name would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a month_name was seen; in this case, month_name would be a token.

Literal characters such as ``,'' must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
        date  :  month '/' day '/' year   ;
```
allowing
```
        7 / 4 / 1776
```
as a synonym for
```
        July 4, 1776
```
In most cases, this new rule could be ``slipped in'' to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While

Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to

handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere.[2, 3, 4] Yacc has been extensively used in numerous practical applications, including lint,[5] the Portable C Compiler,[6] and a system for typesetting mathematics.[7]

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

2: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%" marks. (The percent ``%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also;

thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A  :  BODY  ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ``.'', underscore ``_'', and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes ``'''. As in C, the backslash ``\'' is an escape character within literals, and all the C escapes are recognized. Thus

```
'\n'    newline
'\r'    return
'\''    single quote ``'''
'\\'    backslash ``\''
'\t'    tab
'\b'    backspace
```

```
        '\f'    form feed
        '\xxx'  ``xxx'' in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar ``|'' can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
        A       :       B  C  D   ;
        A       :       E  F   ;
        A       :       G   ;
```

can be given to Yacc as

```
        A       :       B  C  D
                |       E  F
                |       G
                ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
        empty :   ;
```

Names representing tokens must be declared; this is most simply done by writing

```
        %token   name1  name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start   symbol
```

The end of the input to the parser is signaled by a special token, called the endmarker. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as ``end-of-file" or ``end-of-record".

3: Actions

With each grammar rule, the user may associate actions to be

performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces ``{" and ``}". For example,

```
A      :      '('  B  ')'
                {      hello( 1, "abc" );  }
```

and

```
XXX    :      YYY  ZZZ
                {      printf("a message\n");
                       flag = 25;    }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol ``dollar sign" ``$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudovariable ``$$" to some value. For example, an action that does nothing but return the value 1 is

```
{  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A   :   B C D ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr  :   '(' expr ')' ;
```

The value returned by this rule is usually the value of the expr in parentheses. This can be indicated by

```
expr  :   '(' expr ')'     { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A   :   B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A   :   B
            { $$ = 1; }
        C
            { x = $2;  y = $3; }
    ;
```

the effect is to set x to 1, and y to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT  :   /* empty */
            { $$ = 1; }
```

```
        ;

A       :       B  $ACT  C
                        {   x = $2;   y = $3;  }
        ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function node, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr   :       expr  '+'  expr
                       {  $$ = node( '+', $1, $3 );  }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks ``%{'' and ``%}''. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{   int variable = 0;   %}
```

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in ``yy''; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

4: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The function returns an integer, the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable yylval.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the ``# define'' mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                . . .
        case '0':
        case '1':
          . . .
        case '9':
                yylval = c-'0';
                return( DIGIT );
                . . .
                }
        . . .
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names if or while will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name error is reserved for error handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number

for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the Lex program developed by Mike Lesk.[8] These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

5: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
        IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ``.'') is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
        .       reduce 18
```

refers to grammar rule 18, while the action

```
        IF      shift 34
```

refers to state 34.

Suppose the rule being reduced is

A : x y z ;

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states

were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a goto action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
        A       goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action ``turns back the clock'' in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable yylval is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable yyval is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input

tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token  DING  DONG  DELL
%%
rhyme   :       sound  place
        ;
sound   :       DING  DONG
        ;
place   :       DELL
        ;
```

When Yacc is invoked with the -v option, a file called y.output is produced, with a human-readable description of the parser. The y.output file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept  :  _rhyme  $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept  :   rhyme_$end

        $end  accept
        .  error

state 2
        rhyme  :   sound_place

        DELL  shift 5
        .  error

        place   goto 4

state 3
        sound   :   DING_DONG
```

```
        DONG  shift 6
        .  error

    state 4
        rhyme  :   sound  place_    (1)

        .   reduce  1

    state 5
        place  :   DELL_    (3)

        .   reduce  3

    state 6
        sound   :   DING  DONG_    (2)

        .   reduce  2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
    DING  DONG  DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, DING, is read, becoming the lookahead token. The action in state 0 on DING is is ``shift 3'', so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, DONG, is read, becoming the lookahead token. The action in state 3 on the token DONG is ``shift 6'', so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
        sound  :   DING  DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on sound,

```
        sound   goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, DELL, must be read. The action is ``shift 5'', so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on place, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on rhyme causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by ``$end'' in the y.output file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as DING DONG DONG, DING DONG, DING DONG DELL DELL, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

PROGRAM:-2

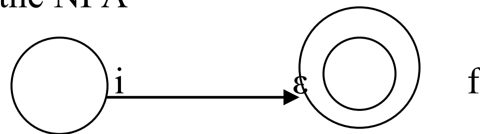
ALGORITHM TO CONVERT REGULAR EXPRESSION TO NFA

Input: A regular expression R over alphabet Σ.

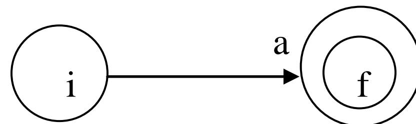
Output: A NFA n accepting the language denoted by R

Method: We first decompose R into the primitive components. For each component we construct a finite automaton as follows.

1. For ε we construct the NFA

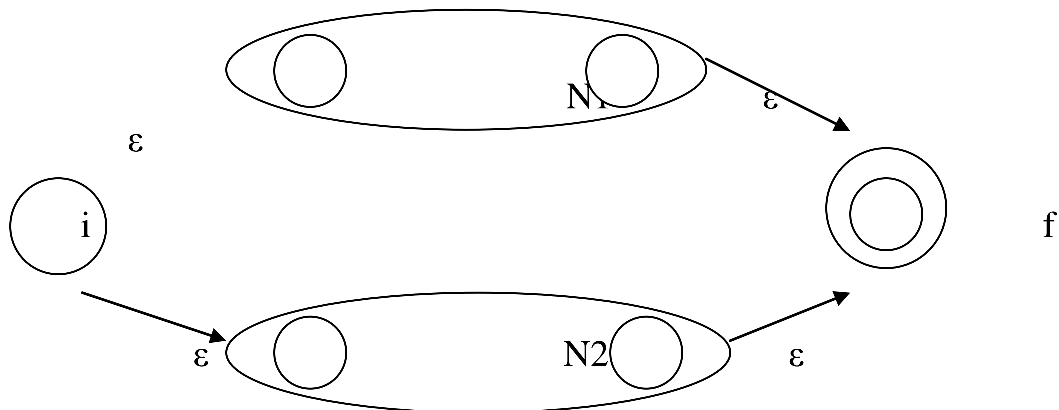


2. For a in Σ we construct the NFA

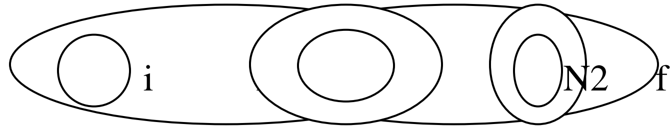


3. Having constructed components for the basic regular expressions, we proceed to combine them in ways that correspond to the way compounded regular expressions are formed from small regular expressions.

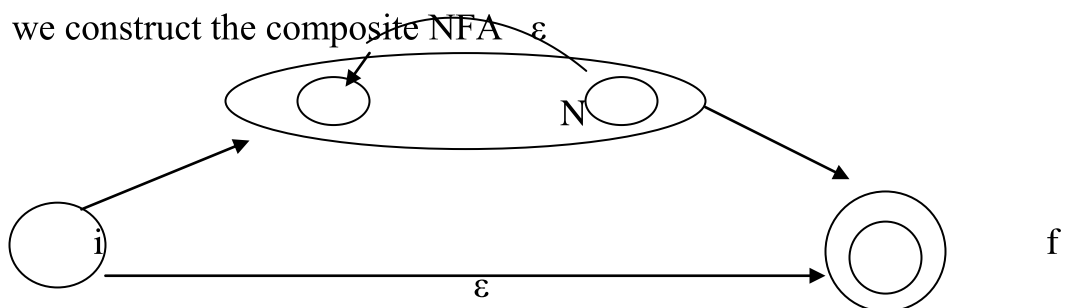
For regular expression R1/R2 we construct the composite NFA



For R1, R2 we construct the composite NFA



For R1* we construct the composite NFA



where i is initial state and
f is final state

ALGORITHM:-3

WAP TO FIND FIRST IN CONTEXT FREE GRAMMER

1. If X is a terminal, then FIRST(X) is X
FIRST(X)={X}

2. If X is a non-terminal where X->aα/ Є then add a to FIRST(X)
X->+T
FIRST(X)={+, Є }

3. If X is non-terminal and X->Y1,Y2,Y3,----,Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and Є is in all of FIRST(Y1)------,FIRST(Yi); that is, Y1---Yi-1=> Є. If Є is in FIRST(Yj) for all j=1,2,---,k, then add Є to FIRST(X).

WAP TO FIND FIRST IN CONTEXT FREE GRAMMER

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

void main()
{

	char t[5],int[10],p[5][5],first[5][5],temp;
	int i,j,not,nont,k=0,f=0;
	clrscr();

	printf("\nEnter the no. of Non-terminals in the grammer:");
	scanf("%d",&nont);
	printf("\nEnter the Non-terminals in the grammer:\n");
	for(i=0;i<nont;i++)
	{
		scanf("\n%c",&nt[i]);
	}

	printf("\nEnter the no. of Terminals in the grammer: ( Enter { for absiline )
");
	scanf("%d",&not);
	printf("\nEnter the Terminals in the grammer:\n");
	for(i=0;i<not||t[i]=='$';i++)
	{
		scanf("\n%c",&t[i]);
	}

	for(i=0;i<nont;i++)
	{
		p[i][0]=nt[i];
		first[i][0]=nt[i];
	}

	printf("\nEnter the productions :\n");
	for(i=0;i<nont;i++)
```

```
	{
		scanf("%c",&temp);
		printf("\nEnter the production for %c ( End the production with '$'
sign ) :",p[i][0]);
		for(j=0;p[i][j]!='$';)
		{
			j+=1;
			scanf("%c",&p[i][j]);
		}
	}

	for(i=0;i<nont;i++)
	{
		printf("\nThe production for %c -> ",p[i][0]);
		for(j=1;p[i][j]!='$';j++)
		{
			printf("%c",p[i][j]);
		}
	}

	for(i=0;i<nont;i++)
	{
		f=0;
		for(j=1;p[i][j]!='$';j++)
		{
			for(k=0;k<not;k++)
			{
				if(f==1)
				break;

				if(p[i][j]==t[k])
				{
`					first[i][j]=t[k];
					first[i][j+1]='$';
					f=1;
					break;
				}

				else if(p[i][j]==nt[k])
```

```
				{
					first[i][j]=first[k][j];
					if(first[i][j]=='{')
					continue;
					first[i][j+1]='$';
					f=1;
					break;
				}
			}
		}
	}

	for(i=0;i<nont;i++)
	{
		printf("\n\nThe first of %c -> ",first[i][0]);
		for(j=1;first[i][j]!='$';j++)
		{
			printf("%c\t",first[i][j]);
		}
	}

getch();
}
```

Output

Enter the no. of Non-terminals in the grammer:3

Enter the Non-terminals in the grammer:
E
T
V

Enter the no. of Terminals in the grammer: (Enter { for absiline) 5

Enter the Terminals in the grammer:
+
*
(
)
i

Enter the productions :

Enter the production for E (End the production with '$' sign) :(i)$

Enter the production for T (End the production with '$' sign) :i*E$

Enter the production for V (End the production with '$' sign) :E+i$

The production for E -> (i)
The production for T -> i*E
The production for V -> E+i

The first of E -> (

The first of T -> i

The first of V -> (

ALGORITHM:-4

ALGORITHM TO FIND WHETHER GIVEN STRING IS KEYWORD OR NOT

1. Start
2. Declare i, flag as int and str[10] as char
3. Declare array a[5][10]={“printf”,”scanf”,”if”,”else”,”break”}
4. Allow user to enter the string
5. Initialize the loop i=0 to i<strlen(str)
   a. Compare str and a[i]
   b. If same, assign flag=1 else flag=0
6. If flag=1, print Keyword else print String
7. Stop

PROGRAM NO:-4

PROGRAM TO FIND WHETHER GIVEN STRING IS KEYWORD OR NOT

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

void main()
{
	char a[5][10]={"printf","scanf","if","else","break"};
	char str[10];
	int i,flag;

	clrscr();

	puts("Enter the string :: ");
	gets(str);

	for(i=0;i<strlen(str);i++)
	{
		if(strcmp(str,a[i])==0)
		{
			flag=1;
			break;
		}
		else
			flag=0;
	}

	if(flag==1)
		puts("Keyword");
	else
		puts("String");

	getch();
}
```

Output

Enter the string ::
printf
Keyword

Enter the string ::
vikas
String

ALGORITHM:-5

ALGORITHM TO FIND WHETHER GIVEN STRING IS IDENTIFIER OR NOT

1. Start
2. Declare i,j,flag=1,len as int and str[10] as char
3. Allow the user to enter the string
4. Initialize the loop i=0,j=1 to i<len
    b. if str[j] is digit, assign flag=0
    c. else if str[i] is alphabet, increment flag
        1) if str[i] is alphanum, increment flag
        2) else if str[i] is not digit, assign flag=0
        3) else increment flag
    d. else if str[i] is not alphanum, assign flag=0
5. if flag==0, print Identifier else print Not Identifier
6. Stop

PROGRAM NO:-5

PROGRAM TO FIND WHETHER GIVEN STRING IS IDENTIFIER OR NOT

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
#include<stdlib.h>

void main()
{
	int i,j,flag=1,len;
	char str[10];

	clrscr();

	puts("Enter the string :: ");
	gets(str);

	len=strlen(str);

	for(i=0,j=1;i<len;i++)
	{
		if(isdigit(str[j]))
		{
			flag=0;
			break;
		}


		else if(isalpha(str[i]))
		{
			{
				flag++;
				continue;
			}
```

```
			if(isalnum(str[i]))
			{
				flag++;
				continue;
			}
			else if(!isdigit(str[i]))
			{
				flag=0;
				break;
			}
			else
				flag++;
		}
else if(!isalnum(str[i]))
		{
			flag=0;
			break;
		}


	}


	if(flag==0)
		puts("Not Identifier");
	else
		puts("Identifier");


	getch();
}
```

Output

Enter the string ::
printf
Identifier

Enter the string ::
123scanf
Not Identifier

Enter the string ::
printf_scanf
Not Identifier

ALGORITHM:-6

ALGORITHM TO FIND WHETHER GIVEN STRING IS CONSTANT OR NOT

1. Start
2. Declare i, flag as int and a[5] as char
3. Allow user to enter the value
4. Initialize the loop from i=0 to i<strlen(a)
    a. if a[i] is digit, assign flag=1
    b. else assign flag=0
5. if flag==1, print Value is Constant else print Value is Variable
6. Stop

PROGRAM:-6

PROGRAM TO FIND WHETHER GIVEN STRING IS CONSTANT OR NOT

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

void main()
{
        int i,flag;
        char a[5];

        clrscr();

        puts("Enter the value :: ");
        gets(a);

        for(i=0;i<strlen(a);i++)
        {
                if(isdigit(a[i]))
                        flag=1;

                else
                {
                        flag=0;
                        break;
                }
        }


        if(flag==1)
                puts("Value is constant");
        else
                puts("Value is a variable");
```

```
        getch();
}
```

Output

Enter the value ::
123
Value is constant

Enter the value ::
vikas
Value is a variable

ALGORITHM:-7

PROGRAM TO COUNT BLANK SPACE AND COUNT THE NO. OF LINES

1. Start
2. Declare flag=1 as int and I,j=0,temp[100] as char.
3. Allow the user to enter the sentence
4. Read the sentence until it is not empty

b. if i==' ', replace i with ;

c. else if i=='\t', replace i with ",

d. else if i=='\n', increment the value of flag

5. Assign temp[j++]=i and temp[j]=NULL
6. Print the value of temp for removing blanks
7. Print the value of flag for counting the no. of lines
8. Stop

PROGRAM:-7

PROGRAM TO COUNT BLANK SPACE AND COUNT THE NO. OF LINES

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

void main()
{
        int flag=1;
        char i,j=0,temp[100];

        clrscr();

        printf("Enter the Sentence (add '$' at the end) :: \n\n");

        while((i=getchar())!='$')
        {
                if(i==' ')
                        i=';';
                else if(i=='\t')
                        i='"';
                else if(i=='\n')
                        flag++;

                temp[j++]=i;
        }

        temp[j]=NULL;
        printf("\n\n\nAltered Sentence :: \n\n");
        puts(temp);

        printf("\n\nNo. of lines = %d",flag);


        getch();
}
```

Output

Enter the Sentence (add '$' at the end) ::

vikas kapoor
hello   world
welcome$

Altered Sentence ::

vikas;kapoor
hello"world
welcome

No. of lines = 3

ALGORITHM:-8

ALGORITHM TO GENERATE TOKENS FOR THE GIVEN GRAMMER

1. Start
2. Declare i as int, str[20] as char
3. Allow user to enter the string
4. Initialize the loop until str[i] is not equal to NULL

   a. if str[i]==’(‘ || str[i]==’{‘, print 4

   b. if str[i]==’)’ || str[i]==’}‘, print 5

   c. if str[i] is digit, increment i and print 1

   d. if str[i]==’+’, print 2

   e. if str[i]==’*’, print 3
5. Stop

PROGRAM:-8

PROGRAM TO GENERATE TOKENS FOR THE GIVEN GRAMMER

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

void main()
{
	int i=0;
	char str[20];
	clrscr();
	printf(" \n Input the string ::");
	gets(str);
	printf("Corresponding Tokens are :: ");
	while(str[i]!='\0')
	{
		if((str[i]=='(')||(str[i]=='{'))
		{
			printf("4");
		}

		if((str[i]==')')||(str[i]=='}'))
		{
			printf("5");
		}

		if(isdigit(str[i]))
		{
			while(isdigit(str[i]))
			{
				i++;
			}
			i--;
			printf("1");
```

```
		}

		if(str[i]=='+')
		{
			printf("2");
		}

		if(str[i]=='*')
		{
			printf("3");
		}

		i++;
	}

	getch();
}
```

Output

Input the string :: (12+23*34)
Corresponding Tokens are :: 4121315

PROGRAM:-9

AN ALGO TO CONVERT NFA TO DFA

Input: An NFA N.

Output: A DFA D accepting the same language

Method: Our algorithm constructs a transition table Dtran for D. Each DFA state is a set of NFA states and we construct Dtran so that D will simulate "in parallel" all possible moves N can make on a given input string

Before it sees the first input symbol, N can be in any of the states in the set €-closure(s0), where s0 is the start state of N. Suppose that exactly the states in set T are reachable from s0 on a given sequence of input symbols, and let a be the next input symbol. On seeing a, N can move to any of the states in the set move(T,a). When we allow for €-transitions, N can be in any of the states in €-closure(move(T,a)), after seeing the a.

```
initially, €-closure(s0) is the only state in Dstates and it is unmarked.
while there is a n unmarked state T in Dstates do begin
        mark T;
        for each input symbol a do begin
                U:= €-closure(move(T,a));
                if U is not in Dstates then
                        add U as unmarked state in Dstates
                Dtran[T,a]:=U
        end
end
```

The Subset Construction

We construct Dstates, the set of states of D, and Dtran, the transition table for D, in the following manner. Each state of D corresponds to a set of NFA states that N could be in after reading some sequence of input symbols including all possible €-transitions before or after symbols are read. The start state of D is €-closure(s0). States and transitions are added to D using algorithm of Subset Constructions. A

state of D is an accepting state if it is a set of NFA states containing at least one accepting state of N.

```
push all states in T onto stack;
initialize €-closure(T) to T;
while stack is not empty do begin
        pop t, the top element, off of stack;
        for each state u with an edge from t to u labeled € do
                if u is not in €-closure(T) do begin
                        add u to €-closure(T);
                        push u onto stack;
                end
end
```

Computation of €-closure

The computation of €-closure(T) is a typical process of searching a graph for nodes reachable from a given set of nodes. In this case the states of T are the given set of nodes, and the graph consists of just the €-labeled edges of the NFA. A simple algorithm to compute €-closure(T) uses a stack to hold states whose edges have not been checked for €-labeled transitions. Such a procedure is shown in Computation of €-closure

PROGRAM:-10

MINIMIZATION THE NUMBER OF STATES OF A DFA

Input: A DFA M with set of states S, set of inputs ∑, transitions defined for all states and inputs, start state s0, and set of accepting states F.

Output: A DFA M' accepting the same language as M and having as few states as possible.

Method:

1. Construct as initial partition ∏ of set of states with two groups: the accepting states F and the non accepting states S-F
2. Apply the procedure to ∏ to construct a new partition ∏new.
3. If ∏new = ∏, let ∏final = ∏ and continue with step (4). Otherwise, repeat step (2) with ∏ = ∏new.
4. Choose one state in each group of the partition ∏final as the representative for that group. The representatives will be the states of the reduced DFA M'. Let s be a representative state, and suppose on input a there is a transition of M from s to t. let r be the representative of t's group. Then M' has a transition from s to r on a. Let the start state of M' be the representative of the group containing the start state s0 of M, and let the accepting states f M' be the representatives that are in F. Note that each group of ∏final either consists only of states in F or has no states in F.
5. If M' has a dead state, that is, a state d that is not accepting and that has transitions to itself on all input symbols, then remove d from M'. Also remove any states not reachable from the start state. Any transitions to d from other states become undefined.

```
for each group G of ∏ do begin
        partition G into subgroups such that two states s and t
                of G are in the same subgroup if and only if  for all
                input symbols a, states s and t have transitions on a
                to states in the same group of ∏;
        replace G in ∏new by the set of all subgroups formed
end
```

Construction of ∏new

ALGORITHM=11

WAP To CHECK IF STRING IS IN GRAMMER OR NOT.

1. Start.
2. Declare the character array str[], token, and initialize integer variables a=0, b=0,c,d.
3. Input the string from user.
4. Repeat step 5 to 12 till str[a]='10'.
5. If str[a]=='(' orstr[a]=='{' then token[b]='4',b++.
6. If str[a]==')'or str[a]=='}' then token[b]='5',b++
7. Check if isdigit(str[a]) then repeat step 8 till isdigit(str[a])
8. a++.
9. a--, token[b]='1', b++.
10. If (str[a]=='+') then token[b]='2',b++.
11. If (str[a]=='*') then token[b]='3',b++.
12. a==.
13. token[b]='\0'.
14. Then print the token genarated for string.
15. b=0.
16. Repeat step 17 to 18 till token[b='\0'.
17. If token[b]=='1' then token[b]=='6'.
18. b++
19. Print the token.
20. b=0
21. Repeat step 22 to 31 till token[b]='\0'.
22. c=0.
23. Repeat step 24 to 36 til token[b]=='6'.
24. token[c]='6'.
25. c++
26. Repeat step 27 to 28 till token[c]='0'.
27. token[c]==token[c+2].
28. c++
29. Token[c-2]='\0'.
30. put token.
31. b++
32. Compare token with 6 and result store in d.
33. If d=0, then put the string in the grammer.
34. Else print string is not in the string .
35. Stop.

PROGRAM=11

WAP To CHECK IF STRING IS IN GRAMMER OR NOT.

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
void main()
{
int a=0,b=0,c;
char str[20],tok[11];
clrscr();
printf("input the expression=");
gets(str);
while(str[a]!='\0')
{
if((str[a]==')')||str[a]=='{'))
{
tok[b]='4';
b++;
}
if((str[a]==')')|str[a]=='}'))
{
tok[b]='5';
b++;
}
if(isdigit(str[a]))
{
while(isdigit(str[a]))
{
a++;
}
a--;
tok[b]='0';
b++;
}
if(str[a]=='+')
{
```

```
tok[b]='2';
b++;
}
if(str[a]=='*')
{
tok[b]='3';
b++;
}
a++;
}
tok[b]='\0';
puts()tok);
b=0;
while(tok[b]!='\0')
{
if(((tok[b]=='6')&&(tok[b+1]=='2')&&(tok[b+2]=='6'))||((tok[b]=='6')&&(tok[b
+1]=='3')&&(tok[b+2]=='6'))||
((tok[b]=='4')&&(tok[b+1]=='6')&&(tok[b+2]=='5')))
{
tok[b]='6';
c=b+1;
while(tok[c]!='\0')
{
tok[c]=tok[c+2];
c++;
}
tok[c]='\0';
puts(tok);
b=0;
}
else
{
b++;
puts(tok);
}
}
int d;
d=strcmp(tok,"6");
if(d==0)
```

```
{
printf("it is in grammer"):
}
else
{
printf("it is not in grammer");
}
getch();
}
```

OUTPUT

Input the expression=(23+)
4625
4625
4625
4625
4625
It is not in grammer.

Input the expression=(2+(3+4)+5)
46246265265
46246265265
46246265265
46246265265
46246265265
462462652
462462652
462462652
462462652
462462652
462462
462462
462462
462462
462462
462
462
462
462
462
6
6
6
6
6

ALGORITHM:-12

WAP TO CALCULATE LEADING OF ALL THE NON TERMINALS IN GIVEN GRAMMER.

```
procedure INSTALL(A,a);
if not L(A,a) then
        begin
                L[A,a]:= true;
                push (A,a) onto STACK
        end
```

The main procedure is given below:

```
begin
        for each nonterminal A and terminal a do L(A,a):= false;
        for each production of the form A->a or A->B  do
        INSTALL(A,a);
        while STACK not empty do
                begin
                        pop top pair (B,a) from STACK;
                        for each production of the form A->B  do
                        INSTALL(A,a)
                end
end
```

PROGRAM:-12

WAP TO CALCULATE LEADING OF ALL THE NON TERMINALS IN GIVEN GRAMMER.

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

char arr[18][3] =
	{
		{'E','+','F'},{'E','*','F'},{'E','(','F'},{'E',')','F'},{'E','i','F'},{'E','$','F'},
		{'F','+','F'},{'F','*','F'},{'F','(','F'},{'F',')','F'},{'F','i','F'},{'F','$','F'},
		{'T','+','F'},{'T','*','F'},{'T','(','F'},{'T',')','F'},{'T','i','F'},{'T','$','F'},
	};
char prod[6] = "EETTFF";
char res[6][3]=
	{
		{'E','+','T'},{'T','\0'},
		{'T','*','F'},{'F','\0'},
		{'(','E',')'},{'i','\0'},
	};
char stack [5][2];
int top = -1;

void install(char pro,char re)
{
	int i;
	for(i=0;i<18;++i)
	{
		if(arr[i][0]==pro && arr[i][1]==re)
		{
			arr[i][2] = 'T';
			break;
		}
	}
	++top;
	stack[top][0]=pro;
	stack[top][1]=re;
```

```
}

void main()
{
	int i=0,j;
	char pro,re,pri=' ';

	clrscr();
	for(i=0;i<6;++i)
	{
		for(j=0;j<3 && res[i][j]!='\0';++j)
		{

	if(res[i][j]=='+'||res[i][j]=='*'||res[i][j]=='('||res[i][j]==')'||res[i][j]=='i'||res[i][j]
=='$')
			{
				install(prod[i],res[i][j]);
				break;
			}
		}
	}

	while(top>=0)
	{
		pro = stack[top][0];
		re = stack[top][1];
		--top;
		for(i=0;i<6;++i)
		{
			if(res[i][0]==pro && res[i][0]!=prod[i])
			{
				install(prod[i],re);
			}
		}
	}
	for(i=0;i<18;++i)
	{
		printf("\n\t");
		for(j=0;j<3;++j)
```

```
			printf("%c\t",arr[i][j]);
	}
	getch();
	clrscr();
	printf("\n\n");
	for(i=0;i<18;++i)
	{
		if(pri!=arr[i][0])
		{
			pri=arr[i][0];
			printf("\n\t%c -> ",pri);
		}
		if(arr[i][2] =='T')
			printf("%c ",arr[i][1]);
	}
	getch();
}
```

Output

```
E   +   T
E   *   T
E   (   T
E   )   F
E   i   T
E   $   F
F   +   F
F   *   F
F   (   T
F   )   F
F   i   T
F   $   F
T   +   F
T   *   T
T   (   T
T   )   F
T   i   T
T   $   F
```

```
E -> + * ( i
F -> ( i
T -> * ( i
```

ALGORITHM:-13

WAP TO CALCULATE TRAILING OF ALL THE NON TERMINALS IN GIVEN GRAMMER

```
begin
        for each non terminal A and terminal a do L(A,a):= false;
        for each production of the form A->αa or A->αaB  do
        INSTALL(A,a);
        while STACK not empty do
                begin
                        pop top pair (B,a) from STACK;
                        for each production of the form A->αB  do
                        INSTALL(A,a)
                end
end
```

INSTALL

```
procedure INSTALL(A,a);
if not L(A,a) then
        begin
                L[A,a]:= true;
                push (A,a) onto STACK
        end
```

PROGRAM:-13

WAP TO CALCULATE TRAILING OF ALL THE NON TERMINALS IN GIVEN GRAMMER

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

char arr[18][3] =
	{
		{'E','+','F'},{'E','*','F'},{'E','(','F'},{'E',')','F'},{'E','i','F'},{'E','$','F'},
		{'F','+','F'},{'F','*','F'},{'F','(','F'},{'F',')','F'},{'F','i','F'},{'F','$','F'},
		{'T','+','F'},{'T','*','F'},{'T','(','F'},{'T',')','F'},{'T','i','F'},{'T','$','F'},
	};
char prod[6] = "EETTFF";
char res[6][3]=
	{
		{'E','+','T'},{'T','\0','\0'},
		{'T','*','F'},{'F','\0','\0'},
		{'(','E',')'},{'i','\0','\0'},
	};
char stack [5][2];
int top = -1;

void install(char pro,char re)
{
	int i;
	for(i=0;i<18;++i)
	{
		if(arr[i][0]==pro && arr[i][1]==re)
		{
			arr[i][2] = 'T';
			break;
		}
	}
	++top;
	stack[top][0]=pro;
	stack[top][1]=re;
}
```

```
void main()
{
	int i=0,j;
	char pro,re,pri=' ';

	clrscr();
	for(i=0;i<6;++i)
	{
		for(j=2;j>=0;--j)
		{

	if(res[i][j]=='+'||res[i][j]=='*'||res[i][j]=='('||res[i][j]==')'||res[i][j]=='i'||res[i][j]
=='$')
			{
				install(prod[i],res[i][j]);
				break;
			}
			else if(res[i][j]=='E' || res[i][j]=='F' || res[i][j]=='T')
			{
				if(res[i][j-1]=='+'||res[i][j-1]=='*'||res[i][j-1]=='('||res[i][j-
1]==')'||res[i][j-1]=='i'||res[i][j-1]=='$')
				{
					install(prod[i],res[i][j-1]);
					break;
				}
			}
		}
	}

	while(top>=0)
	{
		pro = stack[top][0];
		re = stack[top][1];
		--top;
		for(i=0;i<6;++i)
		{
			for(j=2;j>=0;--j)
			{
				if(res[i][0]==pro && res[i][0]!=prod[i])
```

```
                        {
                                install(prod[i],re);
                                break;
                        }
                        else if(res[i][0]!='\0')
                                break;
                }
        }
    }
    for(i=0;i<18;++i)
    {
        printf("\n\t");
        for(j=0;j<3;++j)
            printf("%c\t",arr[i][j]);
    }
    getch();
    clrscr();
    printf("\n\n");
    for(i=0;i<18;++i)
    {
        if(pri!=arr[i][0])
        {
            pri=arr[i][0];
            printf("\n\t%c -> ",pri);
        }
        if(arr[i][2] =='T')
            printf("%c ",arr[i][1]);
    }
    getch();

}
```

Output

```
E  +  T
E  *  T
E  (  F
E  )  T
E  i  T
E  $  F
F  +  F
F  *  F
F  (  F
F  )  T
F  i  T
F  $  F
T  +  F
T  *  T
T  (  F
T  )  T
T  i  T
T  $  F
```

```
E -> + * ) i
F -> ) i
T -> * ) i
```