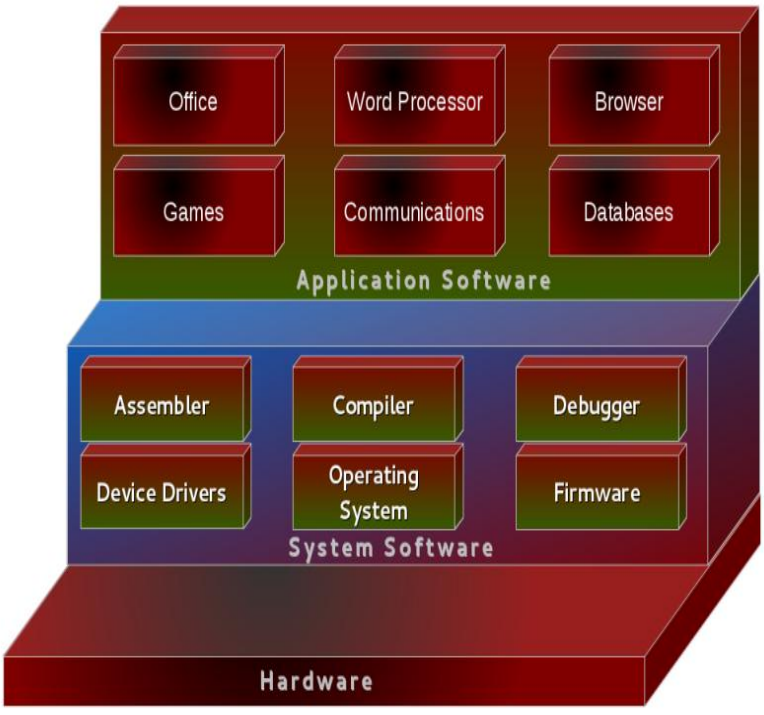
Maharaja Institute of Technology-Mysore

2015

# System Software & Operating Systems Laboratory
# MIT V CSE/ISE




**Department of Information Science & Engineering**

**MIT-Mysore**

**SYSTEM SOFTWARE & OPERATING SYSTEMS LABORATORY**

**Subject Code: 10CSL58** **I.A. Marks: 25**

**Hours/Week: 03** **Exam Hours: 03**

**Total Hours: 42** **Exam Marks: 50**

**PART - A**

**LEX and YACC Programs:**

**Design, develop, and execute the following programs using LEX:**

1. **a)** Program to count the number of **characters**, **words**, **spaces** and **lines** in a given input file.

   **b)** Program to count the **numbers of comment lines** in a given C program. Also eliminate them and copy the resulting program into separate file.
2. **a)** Program to recognize a **valid arithmetic expression** and to recognize the **identifiers and operators present**. Print them separately.

   **b)** Program to recognize whether a given sentence is **simple** or **compound.**
3. Program to recognize and count the **number of identifiers** in a given input file.

**Design, develop, and execute the following programs using YACC:**

4. **a)** Program to recognize a **valid arithmetic expression** that uses operators +, -, * and /.

   **b)** Program to recognize a **valid variable,** which starts with a letter, followed by any number of letters or digits.
5. **a)** Program to **evaluate an arithmetic expression** involving operators +, -, * and /.

   **b)** Program to recognize strings **'aaab', 'abbb', 'ab'** and **'a'** using the grammar **($a^n b^n >= 0$)**.
6. Program to recognize the grammar **($a^n b^n >= 10$)**.

**PART B**

**UNIX Programming:**

Design, develop, and execute the following programs:

7. **a)** Non-recursive shell script that **accepts any number of arguments** and prints them in the **reverse order**, (For example, if the script is named rargs, then executing rargs A B C should produce C B A on the standard output).

   **b)** C program that creates a child process to **read commands** from the standard input and **execute them** (a minimal implementation of a shell – like program). You can assume that no arguments will be passed to the commands to be executed.

8. **a)** Shell script that **accepts two file names** as arguments, checks if the **permissions** for these files are **identical** and if the **permissions are identical, outputs the common permissions**, otherwise **outputs each file name followed by its permissions**.

   **b)** C program to create **a file with 16 bytes of arbitrary data** from the beginning and another **16 bytes of arbitrary data from an offset of 48.** Display the file contents to demonstrate how **the hole in file is handled**.

9. **a)** Shell script that accepts file names specified as arguments and creates a shell script that contains this file as well as the **code to recreate** these files. Thus if the script generated by your script is executed, it would recreate the original files(This is same as the "bundle" script described by Brain W. Kernighan and Rob Pike in " The Unix Programming Environment", Prentice – Hall India).

   **b)** C program to do the following: Using **fork ( )** create a child process. The child process prints its own **process-id** and **id of its parent** and then exits. The parent process waits for its child to finish (by executing the wait ()) and prints its own **process-id** and the **id of its child process** and then exits.

**Operating Systems:**

**10.** Design, develop and execute a program in C / C++ to simulate the working of **Shortest Remaining Time and Round-Robin Scheduling** Algorithms. Experiment with different quantum sizes for the Round-Robin algorithm. In all cases, determine the average turn-around time. The input can be read from key board or from a file.

**11.** Using OpenMP, Design, develop and run a multi-threaded program to generate and print **Fibonacci** Series. One thread has to generate the numbers up to the specified limit and another thread has to print them. Ensure proper **synchronization**.

**12.** Design, develop and run a program to implement the **Banker's** Algorithm. Demonstrate its working with different data values.

**Instructions:**

In the examination, a combination of one LEX and one YACC problem has to be asked from Part A for a total of 30 marks and one programming exercise from Part B has to be asked for a total of 20 marks.

**CHAPTER 1**

# INTRODUCTION TO LEX AND YACC

## 1.1 Introduction to Compiler

The use of higher level languages for programming has become widespread. In order to execute a high level language program written by a programmer, it is necessary to convert the program into the language understood by the machine. A translator is a program, which performs translation from a program written in one language to program in another language of a computer.

The important tasks of a translator are:

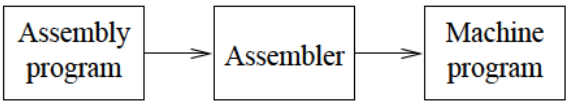
i) Translating the high level language program input into an equivalent machine language program.

ii) Providing diagnostic messages wherever the programmer violates specifications of the high level language.

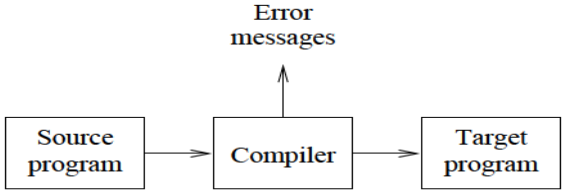
Language translators are generally classified into two groups.

a) Assemblers. b) Compilers.

Assemblers are translators for the lower level assembly language of Computers. It translates program written in assembly language to machine language.

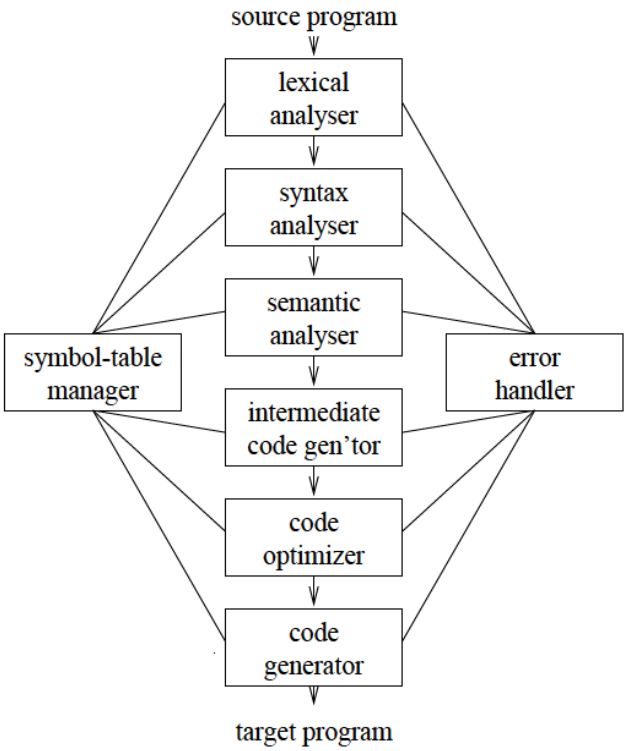



A Compiler is a translator program that translates a program written in high level language into an equivalent program in low level language.



### 1.1.1 Structure of Compiler

A compiler takes as input a source program and produces as output an equivalent sequence of machine instructions. The different phases of the compiler are as given in the following diagram.




The first phase of the compiler is called **Lexical Analyzer** or **Scanner.** It reads the input source and divides the input into meaningful units. These units are called **tokens**. A token is a sub-string of the source program that is to be treated as a single unit.

Example:

if ( x < 5.0 ) THEN x=x+2 ELSE x=x-3;

TOKENS:

Keywords: IF, THEN, ELSE

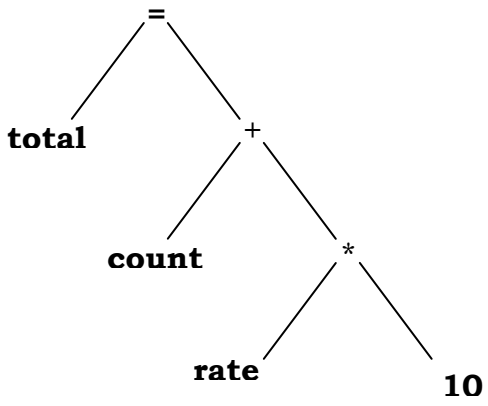
Identifier(s): x

Constants: 2, 3, 5.0

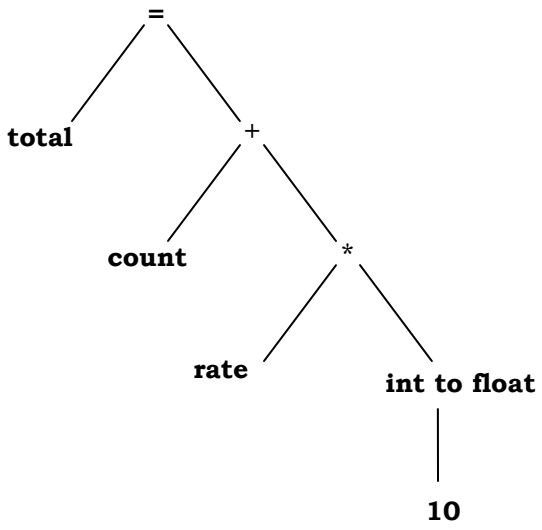
Operators: < , +, -

The Syntax Analyzer groups' tokens together into syntactic structure called as **expression**. The syntactic structure can be regarded as a tree called as **parse trees**.

For example, for the expression total = count + rate * 10 the parse tree can be generated as follows:



The Semantic Analyzer determines the meaning of the source string. For example, meaning of the source string means matching of the parenthesis in the expression or checking the scope of operation as shown below.




## 1.2 Introduction to LEX

Lex and Yacc helps you to write programs that transforms structured input. Lex generates C code for lexical analyzer where as Yacc generates Code for Syntax analyzer.

Lexical analyzer is build using a tool called LEX. Input is given to LEX and lexical analyzer is generated.

Lex is an UNIX utility. It is a program generator designed for lexical processing of character input streams. Lex generates C code for lexical analyzer. It uses the **patterns** that match **strings in the input** and converts **the strings** to tokens. Lex helps you by taking a set of descriptions of possible tokens and producing a C routine, which we call a lexical analyzer.The token description that lex uses are known as regular expressions.

### 1.2.1 Uses of LEX

- Simple text search program creation
- C compiler creation

### 1.2.2 Steps in writing LEX Program

1st step: Using vi editor create a file with extension l. For example: prg1.l

2nd Step: lex prg1.l

3rd Step: cc lex.yy.c –ll

4th Step: ./a.out

### 1.2.3 Structure of LEX source program

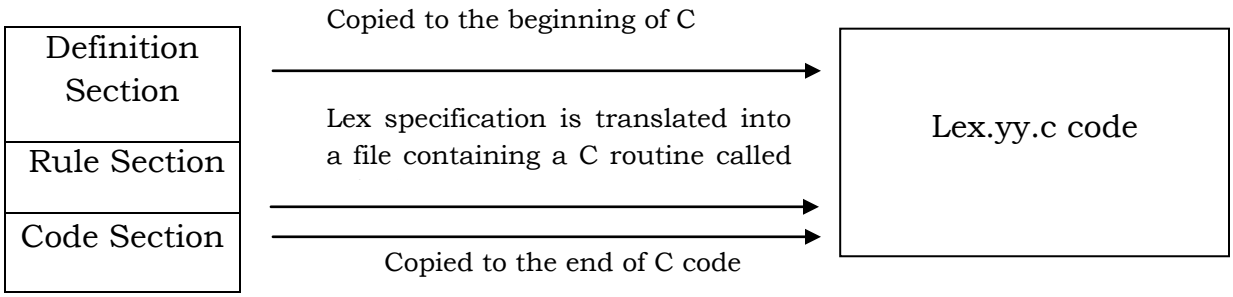
Definition section

%%

Rules section

%%

C code section




%% is a delimiter to mark the beginning of the Rule section. The second %% is optional, but the first is required to mark the beginning of the rules. The definitions and the code /subroutines are often omitted.

### 1.2.3.1 Definition Section

There are three things that can go in the definitions section:

**C code**

It introduces any initial C program code we want copies into the final program. We surround the C code with the special delimiters "%{" and "%}". Lex copies the material between %{ and %} directly to the beginning of generated C file, so you may write any valid C code here. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.

**Substitution:** A **Substitution** is very much like #define cpp directive.

For example

letter [a-zA-Z]

digit [0-9]

punct [,.:;!?]

nonblank [^ \t]

Substitution line does not end with semicolon. This Substitution can be used in the rules section: one could start a rule {letter} + {...

**State definitions** If a rule depends on context, it's possible to introduce states and incorporate those in the rules. A state definition looks like %s STATE, and by default a state INITIAL is already given. If the application of a rule depends on context, there are a couple of ways of dealing with this. We distinguish between 'left state' and 'right state', basically letting a rule depend on what comes before or after the matched token.

### 1.2.3.2 Rule Section

%% marks the beginning of the Rule section. Each rule is made up of two parts. A pattern and an action separated by white space. The lexer i.e Lexical

Analyzer will execute the action when it recognizes the pattern. A number of combinations of pattern and action: if the action is more than a single command it needs to be in braces. References to the substitution in the rules section are surrounded by braces to distinguish them from literals. For example: {letter}

#### 1.2.3.3 Code Section

This section consists of any legal C code and declarations. Lex copies it to the C file after the end of the lex generated code. This can be very elaborate, but the main ingredient is the call to yylex, the lexical analyser. If the code segment is left out, a default main is used which only calls yylex. If the lex program is to be used on its own, this section will contain a main program. If you leave this section empty you will get the default main.

Note that this section has to include the yywrap() function. Lex has a set of functions and variables that are available to the user. One of them is yywrap. Typically, yywrap() is defined as shown in the example below.

```
int main()
{
        yylex();
        return 0;
}
int yywrap()
{
        return 1;
 }
```

where yylex is the parser that is built from the rules.

### 1.2.4 Advanced Lex

Lex has several functions and variables that provide different information and can be used to build programs that can perform complex functions. Some of these variables and functions, along with their uses, are listed in the following tables.

**Lex variables**

| | |
|---|---|
| yyin | Of the type FILE*. This points to the current file being parsed by the lexer. |
| yyout | Of the type FILE*. This points to the location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output. |
| yytext | The text of the matched pattern is stored in this variable (char*). |
| yyleng | Gives the length of the matched pattern. |
| yylineno | Provides current line number information. (May or may not be supported by the lexer.) |

**Lex functions**

| | |
|---|---|
| yylex() | The function that starts the analysis. It is automatically generated by Lex. |
| yywrap() | This function is called when end of file (or input) is encountered. If this function returns 1, the parsing stops. So, this can be used to parse multiple files. Code can be written in the third section, which will allow multiple files to be parsed. The strategy is to make yyin file pointer (see the preceding table) point to a different file until all the files are parsed. At the end, yywrap() can return 1 to indicate end of parsing. |
| yyless(int n) | This function can be used to push back all but first 'n' characters of the read token. |
| yymore() | This function tells the lexer to append the next token to the current token. |

### 1.2.5 Regular Expressions

It is used to describe the pattern. It is widely used to in lex. It uses meta language. The character used in this meta language are part of the standard ASCII character set. An expression is made up of symbols. Normal symbols are characters and numbers, but there are other symbols that have special meaning in Lex. The following two tables define some of the symbols used in Lex and give a few typical examples.

| Character | Meaning |
|---|---|
| A-Z, 0-9, a-z | Characters and numbers that form part of the pattern. |
| . | Matches any character except \n. |
| - | Used to denote range. Example: A-Z implies all characters from A to Z. |
| [ ] | A character class. Matches any character in the brackets. If the first character is ^ then it indicates a negation pattern. Example: [abC] matches either of a, b, and C. |
| * | Match zero or more occurrences of the preceding pattern. |
| + | Matches one or more occurrences of the preceding pattern.(no empty string)<br>Ex: [0-9]+ matches “1”,”111” or “123456” but not an empty string. |
| ? | Matches zero or one occurrences of the preceding pattern.<br>Ex: -?[0-9]+ matches a signed number including an optional leading minus. |
| $ | Matches end of line as the last character of the pattern. |
| { } | 1) Indicates how many times a pattern can be present. Example: A{1,3} implies one to three occurrences of A may be present.<br>2) If they contain name, they refer to a substitution by that name.<br>Ex: {digit} |

| Character | Meaning |
|---|---|
| \ | Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.<br><br>Ex: \n is a newline character, while "*" is a literal asterisk. |
| ^ | Negation. |
| \| | Matches either the preceding regular expression or the following regular expression. Ex: cow\|sheep\|pig matches any of the three words. |
| "< symbols>" | Literal meanings of characters. Meta characters hold. |
| / | Look ahead. Matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input. |
| ( ) | Groups a series of regular expressions together into a new regular expression. Ex: (01) represents the character sequence 01. Parentheses are useful when building up complex patterns with *,+ and \| |

**Examples of regular expressions**

| Regular expression | Meaning |
|---|---|
| joke[rs] | Matches either jokes or joker. |
| A{1,2}shis+ | Matches AAshis, Ashis, AAshi, Ashi. |
| (A[b-e])+ | Matches zero or one occurrences of A followed by any character from b to e. |
| [0-9] | 0 or 1 or 2 or……9 |
| [0-9]+ | 1 or 111 or 12345 or …At least one occurrence of preceding exp |
| [0-9]* | Empty string (no digits at all) or one or more occurrence. |

| **Regular expression** | **Meaning** |
|---|---|
| -?[0-9]+ | -1 or +1 or +2 ….. |
| [0.9]*\.[0.9]+ | 0.0,4.5 or .31415 But won't match 0 or 2 |

**Examples of token declarations**

| **Token** | **Associated expression** | **Meaning** |
|---|---|---|
| number | ([0-9])+ | 1 or more occurrences of a digit |
| chars | [A-Za-z] | Any character |
| blank | " " | A blank space |
| word | (chars)+ | 1 or more occurrences of chars |
| variable | (chars)+(number)*(chars) *( number)* | |

**Design, develop, and execute the following programs using LEX:**

**1. a) Program to count the number of characters, words, spaces and lines in a given input file.**

```
%{
        #include<stdio.h>
        int ch=0,li=0,bl=0,wd=0;
%}
%%
       [a-z A-Z 0-9]   {ch++;}
       [\n]          {li++;wd++;ch++;}
       [ ]          {bl++;wd++;ch++;}
%%
main()
{
       yyin=fopen("text1.txt","r");
       yylex();

       printf("\nNumber of Characters : %d\n",ch);
       printf("\nNumber of Blanks     : %d\n",bl);
       printf("\nNumber of Words      : %d\n",wd);
       printf("\nNumber of Lines      : %d\n",li);
}
```

**Output:-**

gedit 1a.l

lex 1a.l

cc lex.yy.c -ll

./a.out

Number of Characters: 23

Number of Blanks : 2

Number of Words : 5

**b) Program to count the number of comment lines in a given C program. Also eliminate them and copy the resulting program into separate file.**

```
%{
        #include<stdio.h>
        int comment=0;
%}
%%
       "//"[^\n]*              {comment++;}
       "/*"([a-z A-Z 0-9])*"*/"  {comment++;}
%%

main()
{
       yyin=fopen("input.c","r");
       yyout=fopen("output.c","w");
       yylex();

       printf("\nNumber of comment lines : %d\n",comment);
}
```

**Output:-**

Number of comment lines: 3

**2. a) Program to recognize a valid arithmetic expression and to recognize the identifiers and operators present. Print them separately.**

```
%{
        #include<stdio.h>
        int operatorc=0,operandc=0,i=0,j,k=0,flag=0,top=-1;
        char op[15],iden[15],symbol;
%}
%%
        "("   {top++;}
        ")"   {top--;}
        [0-9]+|[a-z A-Z]*    {operandc++;
        if(k==0)
                iden[k++]=yytext[0];
        else
        {
                symbol=yytext[0];

        for(j=0;j<k;j++)
                if(symbol==iden[j])
                        flag=1;
                if(flag==0)
                        iden[k++]=yytext[0];
        }
}
"+"|"-"|"*"|"/"    {op[i++]=yytext[0];operatorc++;}
%%
main()
{
          printf("\nEnter an expression : ");
          yylex();

         if((operandc==operatorc+1) && (top==-1))
        {
                printf("\nValid expression!!\n");
                printf("\nNumber of operators = %d\nNumber of operands =
                %d\n\n",operatorc,operandc);

        for(j=0;j<i;j++)
                 printf("Operator < %c >\n",op[j]);

                printf("\nNumber of Identifiers = %d\n\n",k);

        for(j=0;j<k;j++)
                printf("Identifier < %c >\n",iden[j]);
        }
```

```
        else
                printf("\nInvalid Expression!!\n");
}
```

**Output:-**

```
Enter an expression : a+b-c

Valid expression!!

Number of operators = 2
Number of operands = 3

Operator < + >
Operator < - >

Number of Identifiers = 3

Identifier < a >
Identifier < b >
Identifier < c >
```

**b) Program to recognize whether a given sentence is simple or compound.**

```
%{
        #include<stdio.h>
        int flag=0;
%}
%%
       " AND "|" OR "|" BUT "|" SO "|" IF "|" BECAUSE "|" and "|" or "|"
       but "|" so "|" if "|" because "    {flag=1;}
%%
main()
{
       printf("\nEnter a Sentence : ");
       yylex();

       if(flag==1)
              printf("\n\tCOMPOUND SENTENCE!!\n");
       else
       printf("\n\tSIMPLE SENTENCE!!\n");
}
```

**Output:-**

Enter a Sentence

I am going to Mysore

SIMPLE SENTENCE!!

Enter a Sentence

I am going to Mysore and I will meet somebody

COMPOUND SENTENCE!!

**3. Program to recognize and count the number of identifiers in a given input file.**

```
%{
        #include<stdio.h>
        int count=0;
%}
%%
       "int"|"float"|"char"|"double"
       {
              char ch;
              ch=input();

              for(;;)
              {
              if(ch==',')
                     count++;
              else if(ch==';')
                     break;
              ch=input();
              }
              count++;
       }
       "main"|"printf"|"scanf"   {count++;}
%%
main(int argc, char* argv[])
{
       yyin=fopen(argv[1],"r");
       yylex();

       printf("\nNumber of identifiers : %d\n",count);
}
```

## 1.3 Introduction to YACC

Yacc provides a general tool for imposing structure on the input to a computer program. The input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. Yacc prepares a specification of the input process. Yacc generates a function to control the input process. This function is called a parser.

The YACC is an acronym for "Yet Another Compiler Compiler". YACC generates the code for the parser in the C programming language. YACC was developed at AT & T for the UNIX operating system. Yacc has also been rewritten for other languages, including Java, Ada.

The function parser calls the lexical analyzer to pick up the tokens from the input stream. These tokens are organized according to the input structure rules. The input structure rule is called as grammar. When one of the rule is recognized, then user code supplied for this rule (user code is action) is invoked. Actions have the ability to return values and makes use of the values of other actions.

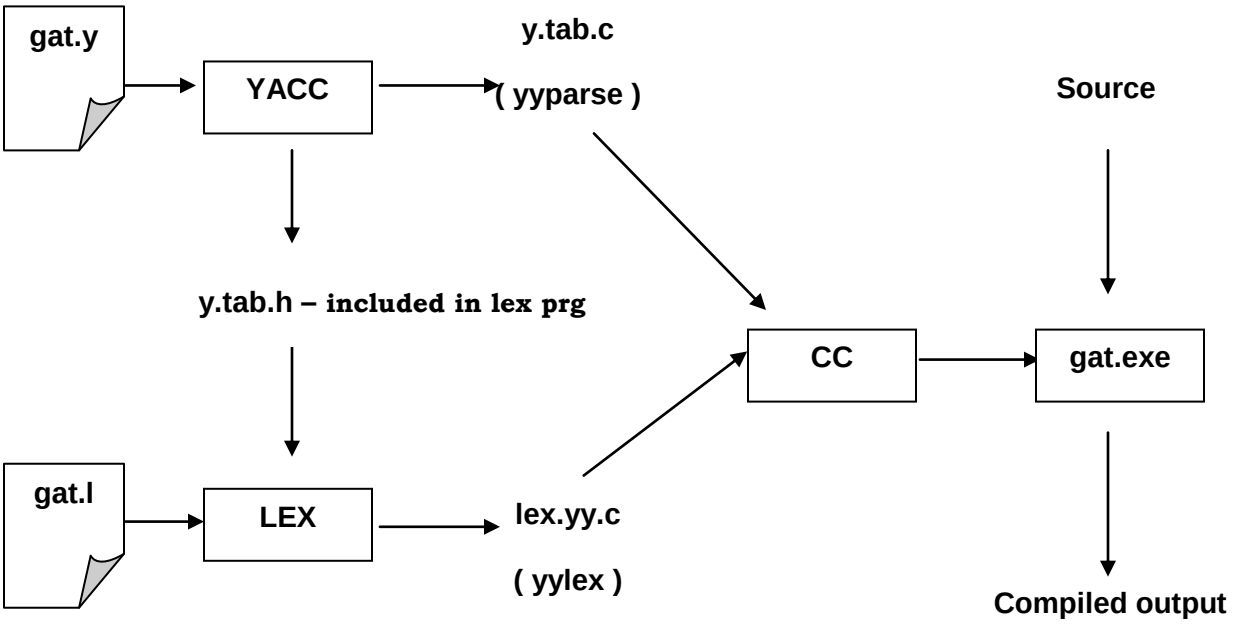
The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

Date: month_name day ',' year

Here date, month_name, day and year represent structure of interest in the input process; 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. When July 4, 1776 is given as input, the lexical analyzer (user routine) reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. The structure recognized by the

lexical analyzer is called terminal symbol. The structure recognized by the parser is called non-terminal symbol. The terminal symbols will usually be referred to as tokens.

Since the parser generated by Yacc requires a lexical analyzer, it is often used in combination with a lexical analyzer generator. Lex divides the input stream into pieces (tokens) and then yacc takes these pieces and groups them together logically.




### 1.3.1 Steps in writing YACC Program

**1st step:** Using vi editor create a file with extension y. For example: prg1.y

**2nd Step:** yacc –d prg1.y

**3rd Step:** lex prg1.l

**4th Step:** cc y.tab.c lex.yy.c –ly -ll

**5th Step:** ./a.out

When we run yacc, it generates a parser in file y.tab.c and also creates an include file y.tab.h. To obtain tokens, yacc calls yylex. Function yylex has a return type of int, and returns the token. Values associated with the token are returned by lex in variable yylval.

### 1.3.2 Structure of YACC source program

**Basic Specification:**

Every yacc specification file consists of three sections.The declarations, Rules (of grammars), programs. The sections are separated by double percent "%%" marks. The % is generally used in yacc specification as an escape character.
The general format for the YACC file is very similar to that of the Lex file.

```
%{
        DEFINITION SECTION
%}
yacc declarations
%%
        RULE SECTION
%%
Additional C code
```

**%%** is a delimiter to the mark the beginning of the Rule section.

### 1.3.2.1 Definition Section

| | |
|---|---|
| %union | It defines the Stack type for the Parser. It is a union of various datas/structures/ objects |
| %token | These are the terminals returned by the yylex function to the yacc. A token can also have type associated with it for good type checking and syntax directed translation. A type of a token can be specified as %token <stack member> tokenName.<br><br>Ex: %token NAME NUMBER<br>%token INTEGER |
| %type | The type of a non-terminal symbol in the Grammar rule can be specified with this.The format is %type <stack member>non-terminal. |
| %noassoc | Specifies that there is no associativity of a terminal symbol. |
| %left | Specifies the left associativity of a Terminal Symbol |
| %right | Specifies the right assocoativity of a Terminal Symbol. |
| %start | Specifies the L.H.S non-terminal symbol of a production rule which should be taken as the starting point of the grammar rules. |
| %prec | Changes the precedence level associated with a particular rule to that of the following token name or literal |

In Declarations C code bracketed by %{ and %} is used. Apart from the legal 'C' declarations there are few Yacc specific declarations which begins with a %sign.

Some declarations used in the definition section is as follow.

You can use single quoted characters as tokens without declaring them, so we don't need to declare "=","+", or "-".

### 1.3.2.2 Rules Section

The rules section simply consists of a list of grammar rules.

A grammar rule has the form:

**A : BODY ;**

A represents a nonterminal name, the colon and the semicolon are yacc punctuation and BODY represents names and literals. The names used in the body of a grammar rule may represent tokens or nonterminal symbols. The literal consists of a character enclosed in single quotes. For a number of technical reasons, the NUL character (\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.

Names representing tokens must be declared as follows in the declaration sections:

**%token name1 name2...**

Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule. Of all the no terminal symbols, one, called the start symbol has a particular importance. The parser is designed to recognize the start symbol. 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 declaration section using the %starts keyword.

The end of the input to the parser is signaled by a special token, called the end marker. If the tokens up to, but not including, the end marker form a structure which matches the start symbol, the parser function returns to its caller after the end-marker is seen. It accepts the input. If the end marker is seen in any

other context, it is an error. It is the job of the user-specified lexical analyzer to return the end marker when appropriate.

With each grammar rule, the user may associate actions to be. These actions may return values, and may obtain the values returned by the previous actions. Lexical analyzer can return values for tokens, if desired. An action is an arbitrary C statement. Actions are enclosed in curly braces.

**Example 01**:

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

**Example 02**:

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

These are all grammar rules with actions. To facilitate the communication between the actions and the parser "$" is used as a signal to yacc. For example if the rule is

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

Then $1 is the value returned by B, $2 is the value returned by C and $3 is the value returned by D.

```
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).

Yacc permits an action to be written in the middle of a rule as well as at the end.

**Example:**

```
A : B { $$ = 1;}

        C { X=$2; Y=$3;}
```

The Yacc parser uses only names beginning in "yy". The user should avoid such names.

### 1.3.3 How the YACC parser works

If you want to use Lex with yacc, note that what Lex writes is a program named yylex(), the name required by Yacc for its analyzer. Normally, the default main program on the lex library calls this yylex() routine, but if yacc is loaded, and its main program is used, yacc will call yylex(). In this case each lex rule should end with Return (token), where the appropriate token value is returned.

The yacc programs can be executed in two ways:

The yacc programs get the tokens from the lex program. Hence a lex program has been written to pass the token to the yacc. That means we have to follow different procdure to get the executable file.

i. The lex progrm <lex file.l> is first compiled using lex compiler to get lex.yy.c
ii. The yacc program <yacc file.y> is compiled using yacc compiler to get y.tab.c
iii. Using c compiler both the lex and yacc intermediate files are compiled with the lex library function. cc y.tab.c lex.yy.c –ll
iv. If necessary out file name can be included during compiling with-o option.

**Design, develop, and execute the following programs using YACC.**

**4. a) Program to recognize a valid arithmetic expression that uses operators +, -, * and /.**

**Lex part**

```
%{
  #include "y.tab.h"
%}
%%
[A-Za-z]   {return ALPHA;}
[0-9]      {return NUM;}
[\t\n]+    ;
.          {return yytext[0];}
%%
```

**Yacc part**

```
%{
        #include<stdio.h>
%}
        %token NUM ALPHA
        %left '+' '-'
        %left '*' '/'
%%
        expression:expression'+'expression
                  |expression'-'expression
                  |expression'*'expression
                  |expression'/'expression
                  |'+'expression
                  |'-'expression
                  |'('expression')'
           |NUM
           |ALPHA;
%%
int main()
{
        printf("\nEnter the expression : ");
        yyparse();

        printf("\n\tVALID EXPRESSION!!\n");
        return 0;
}
```

```
int yyerror()
{
        printf("\n\tINVALID EXPRESSION!!\n");
        exit(0);
}
```

**Output:-**

```
lex 4a.l
yacc 4a.y
cc lex.yy.c y.tab.c -ll
./a.out

Enter the expression : 2+3

        VALID EXPRESSION!!

Enter the expression : 23-

        INVALID EXPRESSION!!

Enter the expression : 2/3

        VALID EXPRESSION!!

Enter the expression : ab+bc-c*v

        INVALID EXPRESSION!!
```

**b) Program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits.**

**Lex part**

```
%{
  #include "y.tab.h"
%}
%%
[0-9]      {return NUMBER;}
[a-zA-Z]   {return LETTER;}
[\t]+
[\n]       {return 0;}
.          {return yytext[0];}
%%
```

**Yacc part**

```
%{
        #include<stdio.h>
%}
        %token NUMBER LETTER
%%
        variable: LETTER alphanumeric
          |LETTER;
        alphanumeric: LETTER alphanumeric
                |NUMBER alphanumeric
                |LETTER
                |NUMBER;
%%
int main()
{
        printf("\nEnter a string : ");
        yyparse();

        printf("String is a VALID variable!!\n");
        return 0;
}

int yyerror()
{
        printf("\nINVALID string variable!!\n");
        exit(0);
}
```

**Output:-**

```
Enter a string : ab
String is a VALID variable!!
Enter a string : 234dfgh
INVALID string variable!!
```

**5. a) Program to evaluate an arithmetic expression involving operators +, -, * and /.**

**Lex Part**

```
%{
        #include <stdlib.h>
        #include "y.tab.h"
        extern int yylval;
%}
%%
        [0-9]+ {yylval=atoi(yytext);
                        return NUM;}
        [\t];
        \n      return 0;
        . return yytext[0];
%%
```

**Yacc Part**

```
%{
        #include < stdio.h >
%}
%token NUM
%left '+''-'
%left '*''/'
%%
expr: e {printf("Result :%d\n",$1); return 0;};
e: e'+'e{$$=$1+$3;}
| e'-'e {$$=$1-$3;}
| e'*'e {$$=$1*$3;}
| e'/'e {$$=$1/$3;}
| '('e')' {$$=$2;}
| NUM {$$=$1;};
%%
main()
{
        printf("type the expression and press enter key\n");
        yyparse();
        printf("valid expression\n");
}
yyerror()
{
        printf("invalid expression\n");
        exit(0);
}
```

**b) Program to recognize strings 'aaab', 'abbb', 'ab' and 'a' using the grammar ($a^n b^n$, n>= 0).**

**Lex Part**

```
%{
        #include "y.tab.h"
%}
%%
a       return A;
b       return B;
.       return yytext[0];
\n      return yytext[0];
%%
```

**Yacc Part**

```
%{
        #include <stdio.h>
%}
%token A B
%%
str: s '\n' { return 0;}
s: A s B;
| ;
%%
main()
{
        printf("Type the strings\n");
        if(!yyparse())
        printf("Valid string\n");
}
int yyerror()
{
        printf("invalid string\n");
        return(1);
}
```

**Output:-**

Type the string

aaab

Valid string

Type the string

abbb

valid string

**6. Program to recognize the grammar ($a^n b$, n>= 10).**

**Lex Part**

```
%{
        #include "y.tab.h"
%}
%%
a return A;
b return B;
.       return yytext[0];
\n      return yytext[0];
%%
```

**Yacc part**

```
%{
        #include <stdio.h>
%}
%token A B
%%
str: s '\n' { return 0;}
s: x B;
x: A A A A A A A A A A T ;
T: T A
| ;
%%
main()
{
        printf("Type the string\n");
        if(!yyparse())
        printf("Valid string\n");
}
int yyerror()
{
        printf("Invalid string\n");
        return(1);
}
```

**Output:-**

Type the string

aaaaaaaaaab

Valid string

Type the string

Aaaab

invalid string

**or**

```
/* program to recognize the grammar(a^n b>=10)*/
%{
        #include <stdio.h>
%}
%token A B
%%
str: s '\n' { return 0;}
s: x B;
x: A{10,25 ;}
%%
main()
{
        printf("Type the string\n");
        if(!yyparse())
        printf("Valid string\n");
}
int yyerror()
{
        printf("Invalid string\n");
        return(1);
}
```

**Output:-**

Type the string

aaaaaaaaaab

Valid string

Type the string

Aaaab

invalid string

# CHAPTER 2

## INTRODUCTION TO UNIX

### 2.1 Basic UNIX commands

**Folder/Directory Commands and Options**

| Action | UNIX options & | DOS filespec & |
|---|---|---|
| Check current Print Working Directory | **pwd** | **cd** |
| Return to user's home folder | **cd**<br>**cd ~** | **cd /** |
| Up one folder | **cd ..** | |
| Make directory | **mkdir** proj1 | |
| Remove empty directory | **rmdir /usr/sam** | **rmdir**<br>**orrd** |
| Remove directory – recursively | **rm -r** | **rmdir** |

**File Listing Commands and Options**

| Action | UNIX options & | DOS filespec & |
|---|---|---|
| List directory tree -recursively | **ls -r** | tree |
| List last access dates of files, with hidden files | **ls -l -a** | |
| List files by reverse date | **ls -t -r *.*** | **dir *.exe** |

| | | |
|---|---|---|
| List files verbosely by size of file | **ls -l -s *.*** | **dir *.* /v** |
| List files recursively including contents of other directories | **ls -R *.*** | **dir *.* /s** |
| List number of lines in folder | **wc -l *.xtuml**<br>**sed -n '$='** | **sed -n "$="** |
| List files with x anywhere in the name | **ls \| grep x** | |
| Create new (blank) file | **touch** | |
| Copy old.file to new.file<br><br>-p preserve file attributes (e.g. ownership and edit dates)<br><br>-r copy recursively through | **cp** old.file new.file | **copy** old.file new.* |
| Move old.file (**-i** interactive flag prompts before overwriting files) | **mv** -i old.file<br><br>/tmp | **Copy** old.file |
| Remove file (-intention) | **rm -i** sam.txt | **del** sam.txt |
| Compare two files and show differences | **diff** | **comp fc** |

**File Utilities**

| Action | UNIX options & filespec | DOS filespec & |
|---|---|---|
| View a file | **vi file.txt** | **edit file.txt** |
| Edit file | **pico** myfile | **edit** myfile |

| | | |
|---|---|---|
| Concatenate files | **cat file1 file2 to standard output.** | **copy file2 >>file1** |
| Counts -lines, -words, and - characters in a file | **wc** -l | |
| Displays line-by-line differences between pairs of text files. | **diff** | |
| calculator | **bc** | |
| calendar for September, 1752 (when leap years began) | **cal** 9 1752 | |

**Pattern Matching**

| | **Pattern** | **Example** |
|---|---|---|
| Position | **?** stands for any single character | ls ?1 |
| Position | ***** stands for any number of characters | ls 2* |
| Specific characters | **[AB]** stands for any number of characters | **ls [AB]1** would yield A1 and B1 |
| Range of characters | **[A-Z]** stands for letters from A | |

### 2.1.1 Files

- **ls** command

  ls command is most widely used command and it displays the contents of directory.

  **options**

  - ls will list all the files in your home directory, this command has many options.
  - ls -l will list all the file names, permissions, group, etc in long format.
  - ls -a will list all the files including hidden files.
  - ls -lt will list all files names based on the time of creation, newer files bring first.
  - ls -Fxwill list files and directory names will be followed by slash.
  - ls -Rwill lists all the files and files in the all the directories, recursively.
  - ls -R | more will list all the files and files in all the directories, one page at a time.

- **more filename :-** shows the first part of a file, just as much as will fit on one screen. Just hit the space bar to see more or **q** to quit. You can use **/pattern** to search for a pattern.
- **mv fname1 fname2 :-**moves a file (i.e. gives it a different name, or moves it into a different directory (see below)
- **cp fname1 fname2 :-**copies a file
- **rm filename :-**removes a file. It is wise to use the option rm - i, which will ask you for confirmation before actually deleting anything. You can make this your default by making an alias in your .cshrc file.
- **diff fname1 fname2 :-**compares files, and shows where they differ
- **wc fname :-**tells you how many lines, words, and characters there are in a file.

- **chmod opt fname :-** lets you change the read, write, and execute permissions on your files. The default is that only you can look at them and change them, but you may sometimes want to change these permissions. For example, **chmod o+r filename** will make the file readable for everyone, and **chmod o-r filename** will make it unreadable for others again. Note that for someone to be able to actually look at the file the directories it is in need to be at least executable.

### 2.1.2 Directories

Directories, like folders on a Macintosh, are used to group files together in a hierarchical structure.

- **mkdir dirname:-** make a new directory
  Ex: mkdir chandru will create new directory, i.e. here chandru directory is created.
- **cd dirname:-**change directory. You basically 'go' to another directory, and you will see the files in that directory when you do 'ls'. You always start out in your 'home directory', and you can get back there by typing 'cd' without arguments. 'cd ..' will get you one level up from your current position. You don't have to walk along step by step – you can make big leaps or avoid walking around by specifying **pathnames.**

**Finding things**

- **ff :-** find files anywhere on the system. This can be extremely useful if you've forgotten in which directory you put a file, but do remember the name. In fact, if you use **ff -p** you don't even need the full name, just the beginning. This can also be useful for finding other things on the system, e.g. documentation.
- **grep string fname(s) :-**looks for the string in the files. This can be useful a lot of purposes, e.g. finding the right file among many, figuring out which is the right version of something, and even doing serious corpus work. grep comes in several varieties (**grep**, **egrep**, and **fgrep**) and has a

lot of very flexible options. Check out the man pages if this sounds good to you.

### 2.1.3 Others

- **kill PID :-**kills (ends) the processes with the ID you gave. This works only for your own processes, of course. Get the ID by using **ps**. If the process doesn't 'die' properly, use the option -9. But attempt without that option first, because it doesn't give the process a chance to finish possibly important business before dying. You may need to kill processes for example if your modem connection was interrupted and you didn't get logged out properly, which sometimes happens.
- **Date :-**Date displays today's date, to use it type date at prompt.
    - Sun Dec 7 14:23:08 EST 1997

  is similar to what you should see on screen.

**Controlling program execution for C-shell**

| | |
|---|---|
| **&** | run job in background |
| **^c** | kill job in foreground |
| **^z** | suspend job in foreground |
| **Fg** | restart suspended job in foreground |
| **Bg** | run suspended job in background |
| **;** | delimit commands on same line |
| **()** | group commands on same line |
| **!** | re-run earlier commands from history |
| **jobs** | list current jobs |

**Controlling program input/output for C-shell**

| | |
|---|---|
| **\|** | pipe output to input |
| **>** | redirect output to a storage file |
| **<** | redirect input from a storage file |
| **>>** | append redirected output to a storage file |
| **tee** | copy input to both file and next program in pipe |
| **script** | make file record of all terminal activity |

**Editors and formatting utilities**

| | |
|---|---|
| **sed** | programmable text editor for data streams |
| **vi** | full-featured editor for character terminals |
| **pico** | very simple text editor |

## 2.2 Introduction to SHELL programming

Shell is the command interpreter of any UNIX system. It interprets the commands that the use gives at the prompt and sends them for execution to the kernel. The shell is essential for interactive computing where the user desires instant output of his/her commands. The shell has the features of re-directing the standard input, output and error files to devices other than the standard devices. The shell creates temporary files to hold the intermediate results and erases them once the command execution is over. The shell has the capability of file name expansion, using Meta characters or wildcards. Using pipe feature different commands can be combined to solve a particular problem. More than one command can be given at the same line using the command terminator ";".

Multitasking feature of UNIX is supported by the shell using background processing method, where more than one process can be started in the background.

Different types of shells available in the UNIX system are

- Bourne shell
- C shell
- Korn shell

Shell's capabilities do not end with it being a command interpreter. It is also a programming language that offers standard programming structures like loops, conditional branching of control, defining and manipulating of variables, file creation and parameter passing.

This is possible by writing a shell script, which is essentially a program file containing UNIX commands that are executed one after the other. The shell script is similar to the batch files in DOS but it is more powerful and complete.

The main features of the shell programming languages are:

- Structured language constructs.
- I/O interaction.
- Subroutine constructs.
- Variables.
- arguments
- Interrupt handling.

Creating and executing a shell scripts:

- Open a file in vi editor.
- Write any unix command
- Save the file under a given name.

- At the shell prompt give the command sh followed by the file name.
- The command written in that file will be executed.

### 2.2.1 Shell variables

**User defined variable**: created by the user. Ex: $age=56

**Environmental variable**: created by shell.

PATH : contains the search path string

TERM: Holds the terminal specification information

HOME: specifies the full path names for the user login directory

LOGNAME: Holds the user login name

PS1: stores the primary prompt string

PS2: specifies the secondary prompt string

SHELL: stores the name of the shell

**Using variables:**

Displaying the contents of the variables: $ echo $variable

**Method of setting Values of the variable:**

- Assigning the values
- Reading values into variable
- Command Substitution:

Assigning: <var> = <value> Ex: name=david

Reading from the standard input: $read <var> <enter> Ex: $read name

Command Substitution: $var=`command` (back quote – You will find this, above the tab key).

**Evaluation shell variables:**

| | Variables | Meaning | Examples |
|---|---|---|---|
| 1) | $var | value of the variable 'var' | |
| 2) | ${ var – value } | Value of the variable, if defined, otherwise "value" | $name=mano<br>$echo ${name-ajay}<br>Output: mano Because name has been defined. |
| 3) | ${ var – value } | | $echo $(year-1995}<br>Output: 1995. Because year has not been defined. |
| 4) | ${ var = value } | | $echo ${class=zoom}<br>$Echo $class<br>Zoom. Because class has been assigned a value zoom. |
| 5) | ${var?message} | value of the variable, if defined,otherwise the shell exits after printing the "messag" | echo${drinks?"not available"}<br>Echo done |

### 2.2.2 Computation on Shell variables

$ expr val1 op val2 Ex: $ expr 5 + 7 output: 12

$ expr $var1 op $var2

Ex: $a=10

$b=10

Echo sum is $a + $b output: echo sum is 20

$var3=`expr $var1 op $var2` (back quote – You will find this, above the tab key).

**Conditional Execution operators:**

01. & &

    Syntax: $command1 && command2

02. ||

    Syntax: $command || command2

03. if <condition>
    then

    command1

    elseif<condition>

    then

    <command2>

    elseif

    <command3>

**Test:**

**Operators on numeric variables:**

-eq equals to

-ne not equals to

-gt greater than

-lt less than

-le less than or equal to

-ge grater than or equal to

**Operators on string variables:**

= equality of strings

!= not equal

-z zero length

-n string length is nonzero

**Operators on files:**

-s file exists and the file size is non zero

-f the file exists

-d directory exists.

-w file exists and has write permission

-r file exists and has read permission

-x file exists and execute permission.

**Logical comparison operators:**

-a logical AND

-o LOGICAL OR

! logical NOT

The [ …. ] alias:

In place of test command ina shell script, we can also use the alias [ .. ]

Ex:

If test "$HOME" = "/usr/mano" we can write this as

If [ "$HOME" = "/usr/mano" ]

**Iteration constructs:**

01. for <variable> in <list>

do

commands

done

02. while command

do

commands

done

03. until command

do

commands

done

**continue and break in loops:**

continue: To return control to the beginning of a loop

break: to exit a loop

The case…. esac construct:

Case word in

Pattern) command;;

Pattern) command;;

*) default;;

Esac

**Example:**

```
# display the options to the user

echo “1.date and time     2.directory listing”

echo

echo “3.User information          4.Current directory”

echo

echo “Enter choice (1,2,3,4):\c”

read choice

case $choice in

        1)  date;;
        2)  ls –l ;;
        3)  who;;
        4)  pwd;;

        *) echo wrong choice;;

esac
```

### 2.2.3 Parameters to Shell Scripts

**Parameter count** : $#

**All parameters** : $*

**The command name**: $0

**Positional parameters**: $1,$2,$3….$n

**Shifting parameters:** shift

**PID of current shell**: $$

**Listing of shell variables**: set

**Shifting parameters** : shift

If more than 9 parameters are passed to a script, it is not possible to refer to the parameters beyond the 9th one. This is because shell accepts a single digit following the dollar sign as a **positional parameter** definition.

The shift command is used to shift the parameters one position to the left. On execution of the shift command, the parameter is overwritten by the second, the second by the third and so on. This implies, that the content of the first parameter are lost once the shift command is executed. This command can be used to shift parameters effectively where the number of parameters to a script varies in number.

**Listing the shell variables**: set

The command set is used to display all the shell variables and the values associated with them. This command has another important use. It executes a command that provided as a parameter, breaking the output of the command into words and store them in different variables, stating from $1 to $n, depending upon the number of words in the output.

Ex: We can store the output of the command date into different variables, by giving the following command:

$set `date`

Here, the variable:

$1 will have “wed’

$2 will have ‘Aug’

…..

$6 will have ‘1995’

Positioning the cursor: tput

| | | |
|---|---|---|
| Tput smso | : | sets reverse video on |
| Tput rmso | : | sets reverse video off |
| Tput cup <row><col> | : | places the cursor at <row> and <col> |
| Tput invis | : | turn screen display off |
| Tput blink | : | sets blink on |
| Tput reset | : | reset the terminal settings to the original state. |

### 2.2.4 Interrupt Handling

Several programs such as editors and compilers create temporary files as they execute. These temporary files are created in a special directory, called /tmp. These temporary files are required to be removed to make space available for other programs or commands. The programs automatically clean up the directory once their execution is over. But, if the command or program terminates abruptly in between, the temporary files are not deleted automatically. In some cases, if a sensitive program is in a running state, terminating it before the execution is complete may create problems. In either case, we want that the program should run till the end of execution. Any interrupt in between should be ignored.

Any process under UNIX is able to receive certain signals while running. One such signal is generated by the interrupt key (either ctrl + c or the del key). It interrupts the process and causes it to end. Other signals include the one produced by hanging up a phone connection and the one sent by the kill command.

The normal effect of most signals is to halt the concerned process, but UNIX allows the user to alter the effect of signals. In shell scripts, this is done by using the trap command.

Trap is the shell's built-in command that sets up a sequence of commands to be executed when a signal occurs.

The format of the trap command is:

$ trap <command-list> <signal-list>

Signals:

0 : shell exit

1 : hang-up

2 : interrupt key pressed

3 : quit

9 : kill

15 : terminate

Ex:

Trap "echo interrupted exit" 1 2 15

Will print the message and exit the script containing the line. However, if the interrupt is caused before the program reaches this line, the interrupt will act in the usual way i.e terminate the program.

**PART B**

**UNIX Programming:**

**Design, develop, and execute the following programs:**

**7. a) Non-recursive shell script that accepts any number of arguments and prints them in the Reverse order, ( For example, if the script is named rargs, then executing rargs A B C should produce C B A on the standard output).**

```
echo "number of arguments are: $#"
len=$#
while [ $len -ne 0 ]
        do
                eval echo \$$len
                len=`expr $len - 1`
done
```

**OUTPUT:-**

```
chmod 777 1a.sh
./1a.sh a b c
Number of arguments are: 3
c
b
a
```

**b) C program that creates a child process to read commands from the standard input and execute them (a minimal implementation of a shell – like program). You can assume that no arguments will be passed to the commands to be executed.**

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>

int main()
{
        pid_t pid;
        char com[10];

        pid=fork();

        if(pid!=0)
        {
                system("clear");
                printf("Child process created!!\n");

                printf("\n\t\tEnter the command to be executed : ");
                scanf("%s",com);

                system(com);
        }

        else
        {
                printf("\nChild process cannot be created!!\n");
                return 0;
        }
        printf("Parent process terminated!!\n\n");
        return 0;
}
```

**Output:-**

```
Child process created!!

        Enter the command to be executed: date
Tue Jun 25 13:14:16 IST 2013
Parent process terminated!!
```

**8. a) Shell script that accepts two file names as arguments, checks if the permissions for these files are identical and if the permissions are identical, outputs the common permissions, otherwise outputs each file name followed by its permissions.**

```
ls -l $1 | cut -d " " -f1 > file1
ls -l $2 | cut -d " " -f1 > file2
if cmp file1 file2
then
        echo "Both the files have same permission"
        cat file1
else
        echo "Both the files have different permission"
        echo "The permission of the first file $1 is "
                cat file1
        echo "The permission of the second file $2 is "
        cat file2
fi
```

**OUTPUT:-**

```
$chmod 777 2a.sh
$cat > file1
This is the first file
$cat > file2
This is the second file
$./2a.sh file1 file2
Both the files have same permission
-rw-r--r--
$chmod 777 file2
```

$./2a.sh file1 file2

Both the files have different permission

The permission of the first file file1 is

-rw-r--r--

The permission of the second file file2 is

-rwxrwxrwx

**b) C program to create a file with 16 bytes of arbitrary data from the beginning and another 16 bytes of arbitrary data from an offset of 48. Display the file contents to demonstrate how the hole in file is handled.**

```
#include<sys/types.h>
#include<sys/stat.h>
#include<stdlib.h>
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
int main()
{
        int fd;
        char buf1[]="Department of CS";
        char buf2[]="Department of IS";
        fd=creat("cse", 0622);
        if(fd<0)
        {
                printf("\nError in creating file");
                exit(0);
        }
        write(fd, buf1, 16);
        lseek(fd, 48, SEEK_SET);
        write(fd, buf2, 16);
        exit(0);
}
```

**OUTPUT:-**

```
cc 2b.c
./a.out
od –c cse
0000000 D e p a r t m e n t O f C S
0000020 \0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
*
0000060 D e p a r t m e n t O f I S
0000100
```

**9. a) Shell script that accepts file names specified as arguments and creates a shell script that contains this file as well as the code to recreate these files. Thus if the script generated by your script is executed, it would recreate the original files(This is same as the "bundle" script described by Brain W. Kernighan and Rob Pike in " The Unix Programming Environment", Prentice – Hall India).**

```
echo '#to bundle, sh this file'
for i in $*
do
        echo "echo $i 1> &2"
        echo "cat>$i << 'End of $i'"
        cat $i
        echo "End of $i"
done
```

**OUTPUT:-**

```
$chmod 777 4a.sh
$ls
10b.c  1b.c  4a.sh  5a.sh  5b.c  6a.sh  6b.c  7a.sh  8a.sh  9a.sh  a
$cat > file1
This is the first file
$cat > file2
This is the second file
$ls
10b.c  4a.sh  5b.c    6b.c    8a.sh     a         file2
1b.c   5a.sh  6a.sh   7a.sh   9a.sh     file1
$./4a.sh file1 file2 > new.sh
$ls
10b.c  4a.sh  5b.c    6b.c    8a.sh   a         file2
```

```
1b.c    5a.sh  6a.sh  7a.sh   9a.sh   file1   new.sh
$rm file1
rm: remove regular file ‘file1’? y
$rm file2
rm: remove regular file ‘file2’? y
$ls
10b.c  4a.sh  5b.c    6b.c    8a.sh   a
1b.c    5a.sh  6a.sh  7a.sh   9a.sh   new.sh
$chmod 777 new.sh
$./new.sh
file1
file2
$ls
10b.c  4a.sh  5b.c    6b.c    8a.sh   a       file2
1b.c    5a.sh  6a.sh  7a.sh   9a.sh   file1   new.sh
$cat file1
This is the first file
$cat file2
This is the second file
```

**b) C program to do the following: Using fork ( ) create a child process. The child process prints its own process-id and id of its parent and then exits. The parent process waits for its child to finish (by executing the wait ( )) and prints its own process-id and the id of its child process and then exits.**

```
#include<sys/types.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
        int pid;
        pid=fork();
        if(pid<0)
        printf("fork error");
        if(pid==0)
        {
                printf("\nThis is child process");
                printf("\nChild PID: %d", getpid());

                printf("\nParent PID: %d", getppid());
                printf("\nThis is child process");
                execlp("date",NULL);
                exit(0);
        }
        else
        {
                wait(NULL);
                printf("\nThis is parent process");
                printf("\nParent PID: %d", getpid());
                printf("\nChild PID: %d\n", pid);
                execlp("date",NULL);
                exit(0);
        }
}
```

**OUTPUT:-**

```
cc 4b.c

./a.out

This is child process

Child PID: 4691
```

Parent PID: 4690

Tue Jun 25 14:57:29 IST 2013

This is parent process

Parent PID: 4690

Child PID: 4691

Tue Jun 25 14:57:29 IST 2013

# CHAPTER 3

## INTRODUCTION TO OPERATING SYSTEMS

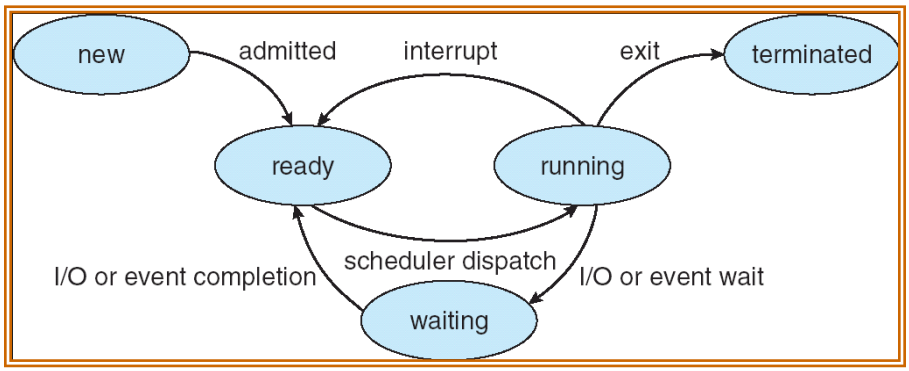
### 3.1 Introduction

An Operating System is a program that manages the Computer hardware. It controls and coordinates the use of the hardware among the various application programs for the various users.

A Process is a program in execution. As a process executes, it changes state

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a process
- **terminated**: The process has finished execution

Apart from the program code, it includes the current activity represented by




- Program Counter,
- Contents of Processor registers,
- Process Stack which contains temporary data like function parameters, return addresses and local variables
- Data section which contains global variables
- Heap for dynamic memory allocation

A Multi-programmed system can have many processes running simultaneously with the CPU multiplexed among them. By switching the CPU between the processes, the OS can make the computer more productive. There is Process Scheduler which selects the process among many processes that are ready, for program execution on the CPU. Switching the CPU to another process requires performing a state save of the current process and a state restore of new process, this is Context Switch.

## 3.2 Scheduling Algorithms

- CPU Scheduler can select processes from ready queue based on various scheduling algorithms. Different scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.

The scheduling criteria include:

- CPU utilization:
- Throughput: The number of processes that are completed per unit time.
- Waiting time: The sum of periods spent waiting in ready queue.
- Turnaround time: The interval between the time of submission of process to the time of completion.
- Response time: The time from submission of a request until the first response is produced.

The different scheduling algorithms are

- **FCFS:** First Come First Served Scheduling
- **SJF:** Shortest Job First Scheduling
- **SRTF:** Shortest Remaining Time First Scheduling
- Priority Scheduling
- Round Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

### 3.3 Deadlocks

A process requests resources; and if the resource is not available at that time, the process enters a waiting state. Sometimes, a waiting process is never able to change state, because the resource is has requested is held by another process which is also waiting. This situation is called Deadlock.

Deadlock is characterized by four necessary conditions

- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait

Deadlock can be handled in one of these ways,

- Deadlock Avoidance
- Deadlock Detection and Recover

**10. Design, develop and execute a program in C / C++ to simulate the working of Shortest Remaining Time and Round-Robin Scheduling Algorithms. Experiment with different quantum sizes for the Round-Robin algorithm. In all cases, determine the average turn-around time. The input can be read from key board or from a file.**

**Aim:**

The aim of this problem is to schedule some given processes with the help of shortest remaining time (SRT) scheduling and round robin (RR) scheduling algorithm and to find out the average turnaround time. Test round robin with different quantum and compare the average turnaround time.

**Algorithm:-**

**Shortest Remaining Time First:-** It is a pre-emptive SJF algorithm. The algorithm associates with each process; the length of process's remaining CPU burst time In this algorithm the CPU scheduler will select the process with the least remaining burst time as the next process to use the CPU.

Round Robin scheduling algorithm: It is similar to FCFS algorithm with pre-emption added to it. A small unit of time, called a time quantum or time slice, is defined. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating CPU to each process for a time interval of up to 1 time quantum.

**Program Code:-**

```
#include<stdio.h>
int i_proc_count, i_time_quantum, i_proc_id[10], i_ariv_time[10], i_burst_time[10], i_turn_arnd_time[10], i_wait_time[10];
int flag[10],flag1[10], swt=0,stat=0, rbt[10], i_total_burst_time=0;
float f_avg_turn_arnd_time=0.0,f_avg_wait_time=0.0;
```

```
int srtf()
{
        int i,j,k=0;
        for(i=0; i<i_proc_count; i++)
        {
                flag[i]=0;
                flag1[i]=0;
                rbt[i]=i_burst_time[i];
        }
        for(i=0; i<i_proc_count; i++)
        {
                i_total_burst_time = i_total_burst_time + i_burst_time[i];
        }
        printf("\ntotal burst time is:%d\n", i_total_burst_time);
        printf("0-");
        for(i=0; i<i_total_burst_time; i++)
        {
                for(j=0; j<i_proc_count; j++)
                {
                        if(i_ariv_time[j]<=i)
                        {
                                flag[j]=1;
                        }
                 }
                k=findmin(i_proc_count);
                printf("%d-", i_proc_id[k]);
                printf("%d-",i+1);
                if(rbt[k]>0)
                {
                        rbt[k]--;
                }
```

```
                if(rbt[k]<=0)
                {
                        i_turn_arnd_time[k] = i-i_ariv_time[k]+1;
                        flag1[k]=1;
                }
        }
        return 0;
}

int findmin(int i_proc_count)
{
        int min=99,index=0;
        int i;

        for(i=0;i<i_proc_count;i++)
        {
                if(rbt[i]<min && flag[i]==1 && flag1[i]!=1 && rbt[i]>0)
                {
                        min=rbt[i];
                        index=i;
                }
        }

        return index;
}

rr()
{
        int st[10];
        int i,count=0,temp,sq=0;
```

```
for(i=0; i<i_proc_count; i++)
{
        st[i]=i_burst_time[i];
}

while(1)
{
        for(i=0,count=0; i<i_proc_count; i++)
        {
                temp = i_time_quantum;
                if(st[i]==0)
                {
                        count++;
                        continue;
                }
                if(st[i] > i_time_quantum)
                        st[i]= st[i]-i_time_quantum;
                else
                        if(st[i]>=0)
                        {
                                temp=st[i];
                                st[i]=0;
                        }
                sq = sq + temp;
                i_turn_arnd_time[i] = sq;
        }
        if(i_proc_count==count)
                break;
}

return 1;
```

```
}

int main()
{
        int i, ch;
        printf("1: SRTF\n");
        printf("2: RR\n");
        printf("3: EXIT\n");
        printf("Enter the choice\n");
        scanf("%d", &ch);
        printf("\nEnter the no of processes:");
        scanf("%d",&i_proc_count);
        if(ch == 1)
        {
                printf("Enter arrival time for sequences:");
                for(i=0; i<i_proc_count; i++)
                {
                        scanf("%d",&i_ariv_time[i]);
                 }
        }
        printf("Enter burst time for sequences:");
        for(i=0; i<i_proc_count; i++)
        {
                i_proc_id[i] = i+1;
                scanf("%d",&i_burst_time[i]);
        }
        if(ch == 2)
        {
                printf("Enter time quantum:");
                scanf("%d", &i_time_quantum);
        }
```

```
        switch(ch)
        {
                case 1: srtf();
                        break;
                case 2: rr();
                        break;
                case 3: return 1;
         }

        printf("\n Process_ID Burst time Wait time Turn around time\n");
        for(i=0; i<i_proc_count; i++)
        {
                i_wait_time[i] = i_turn_arnd_time[i] - i_burst_time[i];
                printf("%d\t\t %d\t %d\t %d", i+1, i_burst_time[i], i_wait_time[i],
i_turn_arnd_time[i]);
        printf("\n");
        }

         for(i=0; i<i_proc_count; i++)
         {
                swt = swt + i_wait_time[i];
                stat = stat + i_turn_arnd_time[i];
        }
         f_avg_wait_time = (float)swt/i_proc_count;
         f_avg_turn_arnd_time = (float)stat/i_proc_count;
         printf("\n\n Average waiting time is %f \n Average turnaround time is
%f",f_avg_wait_time, f_avg_turn_arnd_time);
        return 1;
}
```

**OUTPUT:-**

```
1: SRTF
2: RR
3: EXIT
Enter the choice
2
Enter the no. of processes:3
Enter burst time for sequences: 6 6 6
Enter time quantum: 2
Process_ID     Burst time    Wait time   Turnaround time
1              6             8           14
2              6             10          16
3              6             12          18
Average waiting time is 10.0
Average turnaround time is 16.0
Enter the choice
1
Enter the no. of processes:4
Enter the arrival time for sequences: 0 1 2 3
Enter burst time for sequences: 8 4 9 5
Total burst time is 26
0-1-1-2-2-2-3-2-4-2-5-4-6-4-7-4-8-4-9-4-10-1-11-1-12-1-13-1-14-1-15-1-16-
1-17-3-18-3-19-3-20-3-21-3-22-3-24-3-25-3-26
Process_ID     Burst time    Wait time   Turnaround time
1              8             9           17
2              4             0           4
3              9             15          24
4              5             2           7

Average waiting time is 6.5
Average turnaround time is 13
```

**11. Using OpenMP, Design, develop and run a multi-threaded program to generate and print Fibonacci Series. One thread has to generate the numbers up to the specified limit and another thread has to print them. Ensure proper synchronization.**

**Aim:-**

The aim of this problem is to generate print Fibonacci Series using a multi-threaded program where one thread has to generate the numbers up to the specified limit and another thread has to print them while ensuring proper synchronization.

**Algorithm:-**

1. [input the value of n]

Read n

2. [indicate the required no. of threads]

Call omp_set_num_threads(2)

3. [initialize]

a[0]=0

a[1]=1

3. [generate the fibonacci numbers]

Use #pragma omp parallel directive to run the following structured block

{

Use #pragma omp single directive to run the following block using one thread

For i=2 to n-1

a[i]=a[i-2]+a[i-1];

using omp_get_thread_num( ) display the id of thread involved in the computation of ith fib number

end for

Use #pragma omp single directive to run the following block using one thread For i=2 to n-1

Display the fibonacci elements along with the thread id

End for

}

4. [finished]

**Program code:-**

```
#include <stdio.h>
#include<omp.h>
int main(void)
{
	int i, j, n;
	int a[100];
	a[0]=0;
	a[1]=1;
	omp_set_num_threads(2);
	printf("Enter the number of fibonacci numbers to be generated\n");
	scanf_s("%d", &n);
	printf("Fibonacci series of %d numbers\n",n);
	if(n>=1)
	{
		printf("%d\n", a[0]);
	}
	if(n>=2)
	{
		printf("%d\n", a[1]);
	}
	#pragma omp parallel sections num_threads(2)
	{
		#pragma omp section
```

```
                {
                        for(i=2;i<(n-1);i++)
                        {
                                a[i]=a[i-1]+a[i-2];
                        }
                        printf("Computing thread = %d\n ", omp_get_thread_num());
                }
                #pragma omp section
                {
                        for(j=2;j<(n-1);j++)
                        {
                                printf(" %d \n",a[j]);
                        }
                        printf(" Printing thread = %d\n ", omp_get_thread_num());
                }
        }
}//end of main
```

**OUTPUT:-**

Enter the number of fibonacci numbers to be generated

10

Fibonacci series of 10 numbers

0

1

Computing thread = 0

1

2

3

5

8

13

21

Printing thread = 1

**12. Design, develop and run a program to implement the Banker's Algorithm. Demonstrate its working with different data values.**

**Aim:-**

The aim of this problem is to design and develop a program to avoid deadlock using Banker's Algorithm. This algorithm is used to avoid deadlock in a system where every resources have multiple instances. It will generate a safe sequence, if possible, using which resources should be allocated to each process and execute them otherwise restart the system.

**Algorithm:-**

The Banker's algorithm can be described as follows:

01. Let work[r] and finish[p] be two vectors. With initialization of work = available and finish = false
02. Find i such that,

    Finish[i] == false

    Need[i] <= work

    If no such i exists, go to step 4.

03. Work = work + allocation

    Finish = true

    Go to step2

04. if finish == true for all p, then the system is in safe state

Several data structures must be maintained to implement Banker's algorithm. Assume there are 'p' processes and 'r' resource types in the system,

- avail[r] – no. of available resources of each type
- work[r] – basket to collect resources when it becomes free
- sseq[p] – sequence of processes to be executed for safe termination;

for every process

- all[r] - indicates the allocated resources for that process
- max[r] - indicates the maximum resources allowed for that process
- need[r] - indicates the needed resources for that process during further execution
- request[r] - indicates the request made for the resources by the process
- finish – indicates whether the process can complete

**Program Code:-**

```
#include<stdio.h>

struct process

{

        int i_all[6],i_max[6],i_need[6],i_finished, i_request[6];

} p[10];

int i_avail[6], i_sseq[10], i_ss=0, i_check1=0, i_check2=0, n, i_pid, i_work[6];

int i_nor, i_nori;

void main()

{

        int safeseq(void);

        int tj,ch,i=0,j=0,k,i_pid,ch1;

        printf("\n Enter number of processes : ");

        scanf("%d",&n);

        printf("\n Enter the Number of Resources : ");
```

```
scanf("%d", &i_nor);
printf("\n Enter the Available Resources : ");
for(j=0;j<n;j++)
{
        for(k=0; k<i_nor; k++)
        {
                if(j==0)
                {
                        printf("\n For Resource type %d : ",k);
                        scanf("%d", &i_avail[k]);
                }
                p[j].i_max[k]=0;
                p[j].i_all[k]=0;
                p[j].i_need[k]=0;
                p[j].i_finished=0;
                p[j].i_request[k]=0;
        }
}
printf("\n Enter Max resources for all processes\n");
for(i=0;i<n;i++)
{
```

```
            for(j=0; j< i_nor; j++)
            {
                  scanf("%d",&p[i].i_max[j]);
            }
      }
      printf("\n Enter Allocated resources for all processes\n");
      for(i=0;i<n;i++)
      {
            for(j=0;j<i_nor;j++)
            {
                  scanf("%d",&p[i].i_all[j]);
                  if(p[i].i_all[j]>p[i].i_max[j])
                  {
                        printf("\n Allocation should be less < or == max");
                        j--;
                  }
                  else
                        p[i].i_need[j]=p[i].i_max[j]-p[i].i_all[j];
            }
      }
      if(safeseq()==1)
```

```
        {
                printf("\n The System is in Safe state\n ");
        }
        else
                printf("\n The System is Not in safe state\n ");
        printf("\n Need\n");
        for(i=0;i<n;i++)
        {
                for(j=0;j<i_nor;j++)
                printf(" %d ",p[i].i_need[j]);
                printf("\n");
        }
}
int safeseq()
{
        int tk,tj,i,j,k;
        i_ss=0;
        for(j=0; j<i_nor; j++)
        i_work[j] = i_avail[j];
        for(j=0;j<n;j++)
        p[j].i_finished=0;
```

```
for(tk=0; tk<i_nor; tk++)
{
    for(j=0;j<n;j++)
    {
        if(p[j].i_finished==0)
        {
            i_check1=0;
            for(k=0; k<i_nor; k++)
            if(p[j].i_need[k]<=i_work[k])
            i_check1++;
            if(i_check1== i_nor)
            {
                for(k=0;k< i_nor;k++)
                {
                    i_work[k]= i_work[k]+p[j]. i_all[k];
                    p[j]. i_finished=1;
                }
                i_sseq[i_ss]=j;
                i_ss++;
            }
        }
```

```
            }
        }
        i_check2=0;
        for(i=0;i<n;i++)
            if(p[i].i_finished==1)
        i_check2++;
        if(i_check2>=n)
        {
            printf("The Safe Sequence is\t :");
            for(tj=0;tj<n;tj++)
                printf("%d, ", i_sseq[tj]);
            return 1;
        }
        return 0;
}
```

**Output:-**

Enter the number of processes: 5

Enter the number of resources: 3

Enter the available Resources:

For Resources type 0: 3

For Resources type 1: 3

For Resources type 2: 2

Enter the Max Resources For All Processes:

7 5 3

3 2 2

9 0 2

2 2 2

4 3 3

Enter Allocated Resources For All Processes:

0 1 0

2 0 0

3 0 2

2 1 1

0 0 2

The Safe Squence is : P1, P3, P4, P0, P2

The System is in Safe state

Need

7 4 3

1 2 2

6 0 0

0 0 1

4 3 1

**BIBLIOGRAPHY:-**


| Book Title | Author | Publisher |
|---|---|---|
| Lex and YACC | John R. Levine, Tony Mason & Doug Brown | O'REILLY |
| UNIX Shell Programming | Sumitabha Das | |


**Web sites:-**


http://www.nixcraft.com/

http://www.nixcraft.com/uniqlinuxfeatures/tools/

# APPENDIX

## Viva Questions

- **What is a Parser?**

A Parser for a Grammar is a program which takes in the Language string as its input and produces either a corresponding Parse tree or an Error.

- **What is the Syntax of a Language?**

The Rules which tells whether a string is a valid Program or not are called the Syntax.

- **What is the Semantics of a Language?**

The Rules which gives meaning to programs are called the Semantics of a Language.

- **What are tokens?**

When a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator, keyword etc of the language, these substrings are called the tokens of the Language.

- **What is the Lexical Analysis?**

The Function of a lexical Analyzer is to read the input stream representing the Source program, one character at a time and to translate it into valid tokens.

- **How can we represent a token in a language?**

The Tokens in a Language are represented by a set of Regular Expressions. A regular expression specifies a set of strings to be matched. It contains text characters and operator characters. The Advantage of using regular expression is that a recognizer can be automatically generated.

- **How are the tokens recognized?**

The tokens which are represented by an Regular Expressions are recognized in an input string by means of a state transition Diagram and Finite Automata.

- **Are Lexical Analysis and parsing two different Passes?**

These two can form two different passes of a Parser. The Lexical analysis can store all the recognized tokens in an intermediate file and give it to the Parser as an input. However it is more convenient to have the lexical Analyzer as a co routine or a subroutine which the Parser calls whenever it requires a token.

- **How do we write the Regular Expressions?**

The following are the most general notations used for expressing a R.E.

| Symbol | Description |
|---|---|
| \| | OR (alternation) |
| () | Group of Subexpression |
| * | 0 or more Occurrences |
| ? | 0 or 1 Occurrence |
| + | 1 or more Occurrences |
| {n,m} | n-m Occurrences |

Suppose we want to express the 'C' identifiers as a regular Expression:-

identifier=letter(letter|digit)*

Where letter denotes the character set a-z & A-Z
In LEX we can write it as [a-zA-Z][a-zA-Z0-9]*

The Rules for writing R.E are:-

* An operator character may be turned into a text character by enclosing it in quotes, or by preceding it with a \ (backslash).
* a/b matches "a" but only if followed by b (the b is not matched)
* a$ matches "a" only if "a" occurs at the end of a line
* ^a matches "a" only if "a" occurs at the beginning of a line
* [abc] matches any charcter that is an "a", "b" or "c"
* [^abc] matches any charcter but "a", "b" and "c".
* ab?c matches abc and ac
* Within the square brackets most operators lose their special meanings except "\" and "-". the "^" which takes there special meaning.
* "\n" always matches newline, with or without the quotes. If you want to match the character "\" followed by "n", use \\n.

- **What are the Advantages of using Context-Free grammars?**
  - ⇒ It is precise and easy to understand.
  - ⇒ It is easier to determine syntatic ambiguities and conflicts in the grammar.

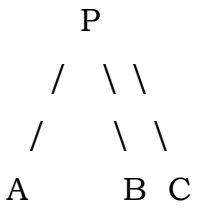
- **If Context-free grammars can represent every regular expression, why do one needs R.E at all?**
  - ⇒ Regular Expression are Simpler than Context-free grammars.
  - ⇒ It is easier to construct a recognizer for R.E than Context-Free grammar.
  - ⇒ Breaking the Syntactic structure into Lexical & non-Lexical parts provide better front end for the Parser.
  - ⇒ R.E are most powerful in describing the lexical constructs like identifiers, keywords etc while Context-free grammars in representing the nested or block structures of the Language.

- **What are the Parse Trees?**

Parse trees are the Graphical representation of the grammar which filters out the choice for replacement order of the Production rules.

e.g

for a production P→ABC the parse tree would be




- **What are Terminals and non-Terminals in a grammar?**

**Terminals:** - All the basic symbols or tokens of which the language is composed of are called Terminals. In a Parse Tree, the Leafs represents the terminal symbol. **Non-Terminals**:- These are syntactic variables in the grammar which represents a set of strings the grammar is composed of. In a Parse tree all the inner nodes represents the Non-Terminal symbols.

- **What are Ambiguous Grammars?**

A Grammar that produces more than one Parse Tree for the same sentences or the Production rules in a grammar is said to be ambiguous. E.g consider a simple mathematical expression E→E*E this can have two Parse tree according to assocciativity of the operator '*'

- **What is bottom up Parsing?**

The Parsing method is which the Parse tree is constructed from the input language string beginning from the leaves and going up to the root node. Bottom-Up parsing is also called shift-reduce parsing due to its implementation. The YACC supports shift-reduce parsing.

e.g Suppose there is a grammar G having a production E

E→E*E

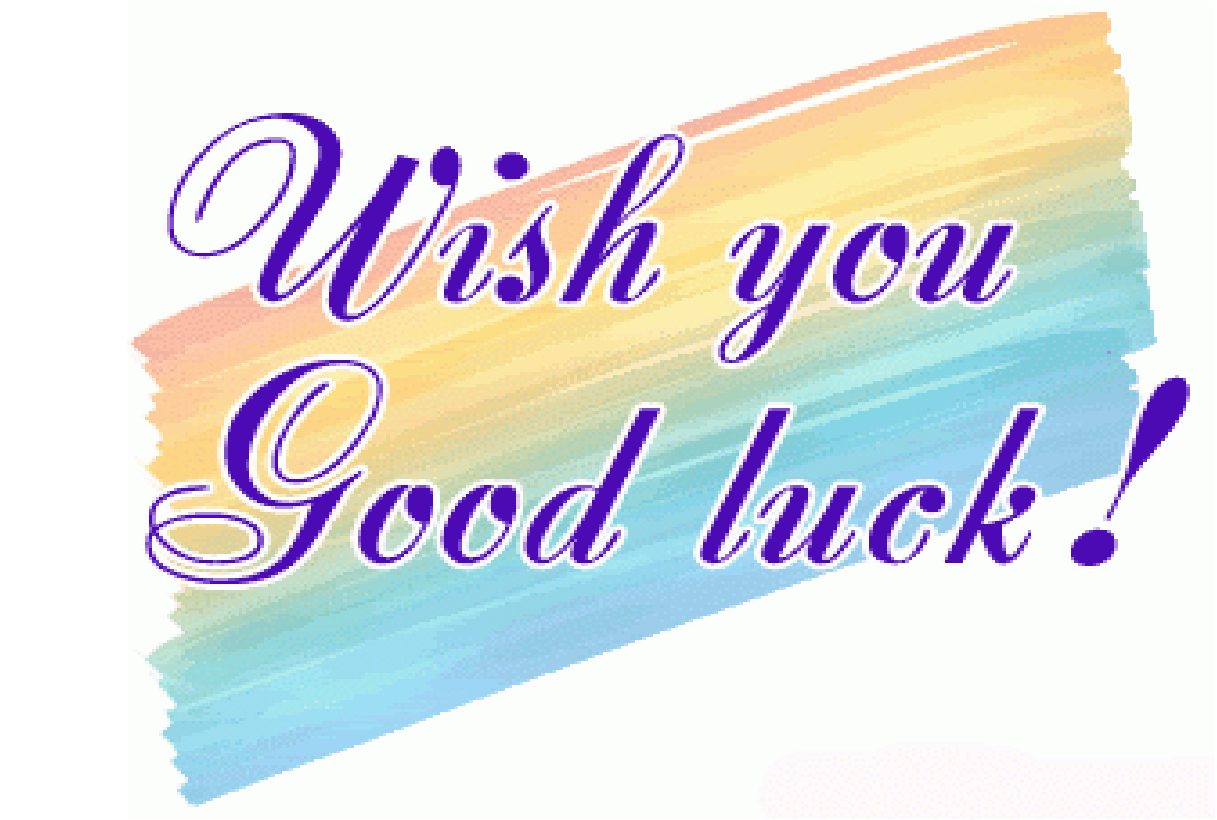
and an input string x*y.

The left hand sides of any production are called Handles. thus the handle for this example is E. The shift action is simply pushing an input symbol on a stack. When the R.H.S of a production is matched the stack elements are popped and replaced by the corresponding Handle. This is the reduce action. Thus in the above example, the parser shifts the input token 'x' onto the stack. Then again it shifts the token '*' on the top of the stack. Still the production is not satisfied so it shifts the next token 'y' too. Now the production E is matched so it pops all the three tokens from the stack and replaces it with the handle 'E'. Any action that is specified with the rule is carried out.

If the input string reaches the end of file /line and no error has occurred then the parser executes the 'Accept' action signifying successful completion of parsing. Otherwise it executes an 'Error' action.

- **What is the need of Operator precedence?**

The shift reduce Parsing has a basic limitation. Grammars which can represent a left-sentential parse tree as well as right-sentential parse tree cannot be handled by shift reduce parsing. Such a grammar ought to have two non-terminals in the production rule. So the Terminal sandwiched between these two non-terminals must have some associability and precedence. This will help the parser to understand which non-terminal would be expanded first.


Wish you
Good luck!