A

LABORATORY MANAUL

On

“Computer Laboratory-I”

SEMESTER–(I)

Prepared by: Prof. P.P.Boraste
Checked by: Prof.B.S.Tarle,
Prof.V.S.Tidke

NAME OF LABORATORY: Computer Laboratory-I.

DEPARTMENT OF COMPUTER ENGINEERING

Nashik District Maratha Vidya Prasarak Samaj’s
Karmaveer Adv. Baburao Ganpatrao Thakare College of Engineering
Nashik

A.Y. 2017-18

NashikDistrict Maratha VidyaPrasarakSamaj's
KARMAVEER ADV. BABURAO GANPATRAO THAKARE
COLLEGE OF ENGINEERING, NASHIK
Udoji Maratha Boarding Campus, Near Pumping station, Gangapur Road

DEPARTMENT OF COMPUTER ENGINEERING

SUBJECT:Computer Laboratory-I (CODE: 410446)

INDEX

	<table><tr><td>Transaction ID</td><td>Item1</td><td>Item2</td><td>Item3</td><td>Item4</td></tr><tr><td>T1</td><td>Mnago</td><td>Onion</td><td>Jar</td><td>Key-chain</td></tr><tr><td>T2</td><td>Nuts</td><td>Onion</td><td>Jar</td><td>Key-chain</td></tr><tr><td>T3</td><td>Mnago</td><td>Apple</td><td>Key-chain</td><td>Eggs</td></tr><tr><td>T4</td><td>Mnago</td><td>Toothbrush</td><td>Corn</td><td>Key-chain</td></tr><tr><td>T5</td><td>Corn</td><td>Onion</td><td>Chocolate</td><td>Key-chain</td></tr></table>			
12	Implement Naive Bayes for categorical and continuous data.			
13	Implementation of k-NN approach takes suitable example.			
14	Implementing recursive descent parser for sample language.			
15	**Group C:** Code generation using “iburg” tool.			

This is to certify that Mr/Ms._____Roll No.:_____, students of year Computer Engineering, has completed the above said experiments/Term work for semester-I of the academic year 2017-18.

PRN NO:	*EXAMINATION NO:*

Name:	**Dr. V. S. Pawar**	**DR. K. S. HOLKAR**
SUBJECT INCHARGE	**HEAD OF DEPT.**	**PRINCIPAL**

EXPERIMENT NO: -01 (Group A) **DATE:**

1.1 Title: Using Divide and Conquer Strategies design a function for Binary Search using C.

1.2 Aim: Design a function for Binary Search using divide and conquer strategies.

1.3 Objectives:

- Understand the importance Divide and Conquer Strategies

-To learn binary search

1.4 Hardware used: Experimental Setup/Instruments/ Devices

1.5 Software used (if applicable) / Programming Languages Used:

1.6 Theory:-

The name "divide and conquer" is sometimes applied also to algorithms that reduce each problem to only one sub-problem, such as the binary search algorithm for finding a record in a sorted list. Divide and conquer is a powerful tool for solving conceptually difficult problems:

1.) Breaking the problem into sub-problems
2.) Solving the trivial cases
3.) Combining sub-problems to the original problem.

Divide and conquer algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors. Divide-and-conquer algorithms naturally tend to make efficient use of memory caches. The reason is that once a sub-problem is small enough, it and all its sub-problems can, in principle, be solved within the cache, without accessing the slower main memory. An algorithm designed to exploit the cache in this way is called cache-oblivious, because it does not contain the cache size(s) as an explicit parameter.

Algorithm:

Control Abstraction:

```
Algorithm DC(P)
{
if P is too small then
return solution of P
else
        {
          Divide (P) aqnd obtain P1, P2,……Pn
          Where n>=1
          Apply DC to each subproblem
          Return combine (DC(P1), DC(P2), DC(P3)…… DC(Pn));
        }
```

Binary Search:

The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array. If the target value is equal to the middle element's value, then the position is returned and the search is finished. If the target value is less than the middle element's value, then the search continues on the lower half of the array; or if the target value is greater than the middle element's value, then the search continues on the upper half of the array. This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (and its associated element position is returned), or until the entire array has been searched (and "not found" is returned).

Recursive and non _recursive binary search:

The non-recursive binary search on the left is a function you've seen before. It maintains a range between two variables **low < high**. This range is cut roughly in half at each step of the algorithm. Termination of this algorithm for an unsuccessful search is quite tricky, with **low** managing to meander over to the right of **high**, so that **low > high** and the while loop terminates.

While in recursive search, it maintains the same range as parameters to the function. The very first step is what is always needed in a recursive function: a test of some final condition, to stop the recursion. If instead there were a recursive call at the start, we would go into an "infinite recursive regression," where the program tries to call itself (or call another program) forever. This process terminates when memory runs out. So the condition **if (low > high) return -1;** terminates the recursion with the same condition that terminates the while loop in the non-recursive version. If the termination condition is not met, then the recursive version, instead of giving new values to **low** and **high**, and then going around the loop, puts these new values into parameters of a recursive call.

Divide-and-conquer (or divide-and-combine) approach to solve problems:

```
methodDivideAndConquer(Arguments)
        if(SmallEnough(Arguments))          //Termination
        return Answer
        else                                // “Divide”
        DivideAndConquer(SmalleArguments))
        Combine(SomeFunc(Arguments))        // “Combine”
```

Time complexity:

The time complexity of the binary search algorithm:
Best case: O(1)
Average case: O(log n)
Worst case: O(log n)

1.7 Algorithm:

It can be done either recursively or iteratively:

1. get the middle element;
2. if the middle element equals to the searched value, the algorithm stops;
3. otherwise, two cases are possible:
    - Searched value is less, than the middle element. In this case, go to the step 1 for the part of the array, before middle element.
    - Searched value is greater, than the middle element. In this case, go to the step 1 for the part of the array, after middle element.

Now we should define when iterations should stop. First case is when searched element is found. Second one is when sub array has no elements. In this case, we can conclude, that searched value doesn't present in the array.

1.8 Example:

Example1. Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 (middle element is 19 > 6): -1 5 6 18 **19** 25 46 78 102 114

Step 2 (middle element is 5 < 6): -1 **5** 6 18 19 25 46 78 102 114

Step 3 (middle element is 6 == 6): -1 5 **6** 18 19 25 46 78 102 114

Example 2: Find 103 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}

Step 1 (middle element is 19 < 103): -1 5 6 18 **19** 25 46 78 102 114

Step 2 (middle element is 78 < 103): -1 5 6 18 19 25 46 **78** 102 114

Step 3 (middle element is 102 < 103): -1 5 6 18 19 25 46 78 **102** 114

Step 4 (middle element is 114 > 103): -1 5 6 18 19 25 46 78 102 **114**

Step 5 (searched value is absent): -1 5 6 18 19 25 46 78 102 114

Disadvantages:

- The disadvantage of binary search is that you must provide an ordered list, usually an array, for the algorithm to search.
- Arrays are either static in size or, if dynamic, require special processing when they grow in size.
- A linked list does not work, because you need to make random access to the elements.
- The more generalized solution of a dynamic binary search is a binary tree, instead of a linked list, but trees also have special processing requirements and, in order to maintain O(Log N) processing time, must have provision for being maintained in a balanced state.

1.9 Conclusion:

In this way, we have successfully completed the study of binary search algorithm using Divide and Conquer strategy.

Signature of Staff with date

1.10 Questions

Q.1 Write recurrence relation for Quick Sort.
Q.2 what is time complexity of Quick Sort?
Q.3 what is multi-threading?
Q.4 What are different parameters of pthread_create()?

EXPERIMENT NO: -02 (Group A) **DATE:-**

1.1 Title: Using Divide and Conquer strategies design a class for concurrent quick sort using C++.

1.2 Aim: Implement a Concurrent Quick Sort using divide and conquer strategy

1.3: Objectives:

- Understand the importance Divide and Conquer Strategies
- To learn Quick sort
-To learn binary search

1.4 Hardware used: Experimental Setup/Instruments/ Devices:

1.5 Software used (if applicable) / Programming Languages Used:

1.6 Theory:-

Introduction to quick sort:

Quick sort is a sorting algorithm that uses the divide and conquers strategy. The three steps of quick sort are as follows.

Divide: Split the array into two sub arrays that each element in the left sub array is greater than the middle element. The splitting of the array into two sub arrays is based on pivot element. All the elements sub array and all the elements that are more than pivot should be in right sub array.

Conquer: Recursively sort the two sub arrays

Combine: Combine all the sorted elements in a group to form a list of sorted elements.

In merge sort the division of array elements, but in quick sort this division is based on actual value of the elements. Consider an array A[i] where i is ranging from 0 to n-1 then we can formulize the division of array elements as follows.

Let us understand this algorithm with the help of some example.

A[0]----------A[m-1],A[m],A[m-1]----------A[n-1]

B. Quick sort algorithm steps:

Step 1: Pick on elements, called a pivot, from the array.

Step 2: Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it. After this partitioning, the pivot is in its final position. This is called the partition operation.

Step 3: Recursively apply the above steps to the sub array of elements with smaller values and separately to the sub array of elements with greater values.

Step 4: Repeat steps 1 to 3 for partitions of having more than one element.

1.4 Example

Input: 65 70 75 80 85 60 55 50 45

P:65 i

Pass 1 **<u>65</u>** 70 75 80 85 60 55 50 45

1) i j →swap (A[i], A[j])

<u>65</u> 45 75 80 85 60 55 50 70

2) i j →swap (A[i], A[j])

<u>65</u> 45 50 80 85 60 55 75 70

3) i j →swap (A[i], A[j])

<u>65</u> 45 50 55 85 60 80 75 70

4) i j →swap (A[i], A[j])

<u>65</u> 45 50 55 60 85 80 75 70

5) j i if (i>=j) break

60 45 50 55 <u>65</u> 85 80 75 70 swap (A[left], A[j])

Items smaller than or equal to 65

Items greater than or equal to 65

Result of Pass 1: 3 sub-blocks:

60 45 50 55 <u>65</u> 85 80 75 70

Pass 2a (left sub-block): **<u>60</u>** 45 50 55 (P = 60)

i j

<u>60</u> 45 50 55

j i if (i>=j) break

55 45 50 <u>60</u> swap (A[left], A[j])

Pass 2b (right sub-block): **<u>85</u>** 80 75 70 (P = 85)

i j

<u>85</u> 80 75 70

j i if (i>=j) break

70 80 75 **<u>85</u>** swap (A[left], A[j])

Combining Pass A and Pass B:

55 45 50 60 70 80 75 85

C. Complexity of Quick sort:

1. **Worst-case Performance : $O(n^2)$**
2. **Best-case Performance: O(n log n)**
3. **Average-case Performance - O(n log n)**

D. How Concurrency is applied to quick sort?

1) Quick sort can be implemented in a concurrent form by representing the partition as subtasks of the sorting task.

2) Concurrent Quick sort uses a collection of worker threads and a coordinator thread.

3) The coordinator sends a message to an idle worker telling it to sort the array and waits to receive messages from workers about the progress of the workers about the progress of the algorithm.

4) A worker thread function partitions a sub-array, and every time that worker gets ready to call the partition routine on a smaller array, it checks to see if there is an idle worker to assign the work to if so, it sends a message to the worker to start working on the sub-problem if not the current worker makes calls the partition routine itself.

5) After each partitioning, two recursive calls are (usually) made, so there are plenty of chances to start other worker threads.

6) Since the worker proceed working concurrently, it is no longer guaranteed that he smaller elements in the array will be ordered before the larger, what is certain is that the two worker will never try to manipulate the same elements.

7) A worker can complete working either because it has directly completed all the work sorting the sub array it was initially called on, or because it has ordered a subset of that array but has passed some or all of the remaining work to other worker threads in either case, it reports the number of elements it has ordered back to the coordinator thread.

The number of elements a worker thread has ordered in the number of partitions of sub arrays has been ordered, it tells the worker threads that there is nothing left to do, and the worker threads exit.

1.7 Conclusion : In this way, we have studied Quick Sort using divide and conquer strategy.

Signature of Staff with date

1.8 Questions:

1. What is concurrent programming?

2. What is time complexity of Quick sort algorithm?

3. What is the worst-case behavior (number of comparisons) for quick sort?

4. Quick sort uses Divide and Conquer Technique Justify.

5. What is the output of quick sort after the 2nd iteration given the following sequence of numbers: 65 70 75 80 85 60 55 50 45(Ans: 55 45 50 60 65 70 80 75 85)

EXPERIMENT NO: -03 (Group A) **DATE:-**

1.1Title: - Lexical analyzer for sample language using LEX

1.2Aim: To implement Lexical analyzer for sample language using LEX.

1.3 Objectives: - To study LEX
- To understand Lexical analyzer

1.4 Hardware used: Experimental Setup/Instruments/ Devices:

1.5 Software used (if applicable) / Programming Languages Used:

1.6 Theory:-

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

1) **LEX Specifications:-**

Structure of the LEX Program is as follows

Declaration Part

%%

Translation Rule

%%

Auxiliary Procedures

Declaration part:-
Contains the declaration for variables required for LEX program and C program.
Translation rules:-
Contains the rules like
Reg. Expression { action1 }
Reg. Expression { action2 }
Reg. Expression { action3 }

Reg. Expression { action-n }

Auxiliary Procedures:-
Contains all the procedures used in your C – Code.

2) Built-in Functions i.e. yylex() , yyerror() , yywrap() etc.

1) yylex() :-
This function is used for calling lexical analyzer for given translation rules.
2) yyerror() :-
This function is used for displaying any error message.
3) yywrap() :-
This function is used for taking i/p from more than one file.

3) Built-in Variables i.e. yylval, yytext, yyin, yyout etc.

1) yylval :-
This is a global variable used to store the value of any token.
2) yytext :-
This is global variable which stores current token.
3) yyin :-
This is input file pointer used to change value of input file pointer. Default file pointer is pointing to **stdin** i.e. keyboard.
4) yyout :-
This is output file pointer used to change value of output file pointer. Default output file pointer is pointing to **stdout** i.e. Monitor.

3) How to execute LEX program:-

For executing LEX program follow the following steps
1) Compile *.l file with lex command

lex *.l
It will generate lex.yy.c file for your lexical analyzer.
2) Compile lex.yy.c file with cc command

cc –o out_file lex.yy.c -ll
Here –o option create executable file named out_file.out and –ll will link LEX program with lexical library.
3) Execute the *.out file to see the output

./out_file sample.c
Which will separate the tokens from sample.c file and display that tokens in token table format.

1.7 Conclusion: Thus, we successfully performed lexical analyzer for sample language using LEX.

Signature of Staff with date

1.8 Questions:

Q.1 What are different phases of compiler?
Q.2 How lexical analysis works?
Q.3 Describe various in-built variables and functions available in LEX.
Q.4 Write sequence of commands used to compile LEX program

EXPERIMENT NO: -04 (Group A) **DATE:-**

1.1 Title: Parser for sample language using YACC.
1.2 Aim: To implement Parser for sample language using YACC.

1.3 Objectives:

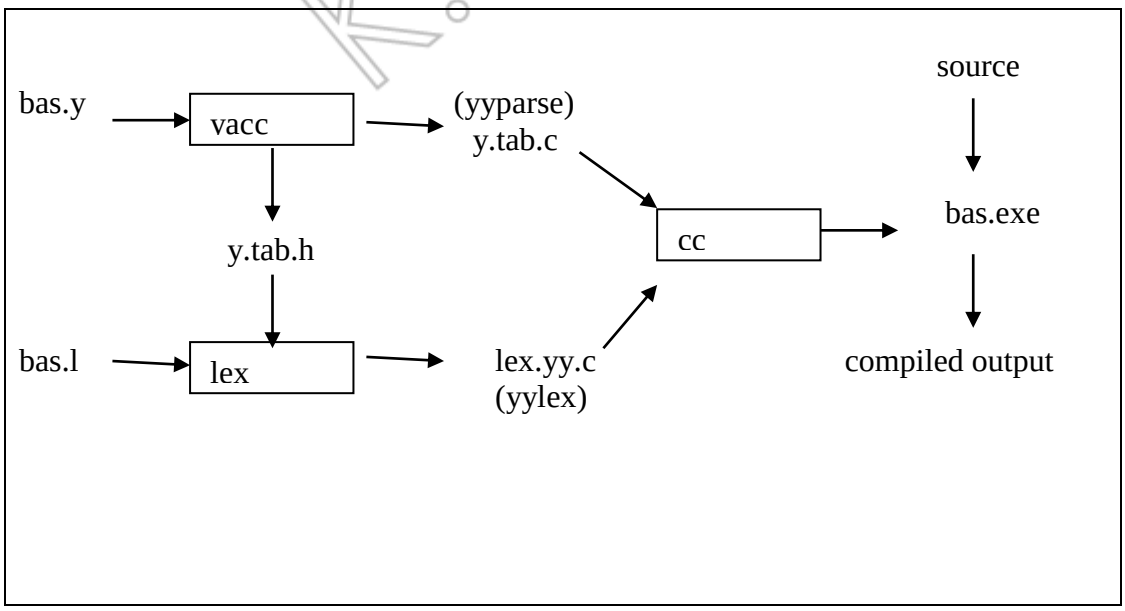
-To study YACC
-To understand parser of compiler: Syntax Analysis.

1.4 Hardware used: Experimental Setup/Instruments/ Devices:

1.5 Software used (if applicable) / Programming Languages Used:

1.6 Theory:-

Lex and Yacc help us to write programs that transform structured input. This includes an enormous range of applications-anything from a simple text search program that looks for patterns in its input file to a C compiler that transforms a source program into optimized object code. In programs with structured input, two tasks that occur over and over are dividing the input into meaningful units, and then discovering the relationship among the units. For a text search program, the units would probably be lines of text, with a distinction between lines that contain a match of the target string and lines that don't. For a C program, the units are variable names, constants, strings, operators, punctuation, and so forth. This division into units (which are usually called tokens) is known as lexical analysis, or lexing for short. Lex helps us by taking a set of descriptions of possible tokens and producing a C routine, which we call a lexical analyzer, or a lexer, or a scanner for short that can identify those tokens. The set of descriptions we give to lex is called a lex specification. The token descriptions that lex uses are known as regular expressions, extended versions of the familiar patterns used by the grep and egrep commands. Lex turns these regular expressions into a form that the lexer can use to scan the input text extremely fast, independent of the number of expressions that it is trying to match. Alex lexer is almost always faster than a lexer that we might write in C by hand.



STRUCTURE OF LEX FILE:

The structure of Lex file is intentionally similar to that of YACC file.A Lex program consists of three sections: the definition section, the rules section, and the user subroutines section.
... **definition section** ...
%%
. . . **rules section** . . .
%%
. . . **user routines section** . . .

The sections are separated by lines consisting of two percent signs. The first two sections are required, although a section may be empty. The third section and the preceding "%%" line may be omitted.

Definition Section

The definition section can include a literal block, C code copied verbatim to the beginning of the generated C file; usually containing declaration and #include lines. There may be %union, %start, %token, %type, %left**,** %right, and %nonassoc declarations.
It can also contain comments in the usual C format, surrounded by "/*n and "*/". All of these are optional, so in a very simple parser the definition section may be completely empty.

Rules Section

The rules section contains lex specification through which the lexer forms tokens.

User Subroutines Section

Lex copies the contents of the user subroutines section verbatim to the C file. This section typically includes routines called from the actions. In a large program, it is sometimes more convenient to put the supporting code in a separate source file to minimize the amount of material recompiled when we change the lex file.

Actions

An action is C code executed when lex matches a lexems with a rule in the lex specification. The action must be a C compound statement,
e.g.: [a-z] { printf("letters from a to z is present"); }

Yacc specifications:
Like lex, yacc has its own specification language Ayacc specification is structured along the same lines as Lex specification.

```
% { /*  C declaration  & includes  */
% }
/* yacc token and type declarations */
% %
/*yacc specification in the form of grammar rules like this */
Symbol: symbols tokens {$$ =  my_c_code( $1) ;}
;
%%
/*c language program  ( the rest ) */
```

The yacc specification rules are the place where we glue various tokens together that lex has conveniently provided to us. Each grammar rule defines a symbol in terms of:

i) Other Symbols

ii) Tokens which come from laxer

Each rule can have associated action, which is executed after all the components symbols of the rule have been parsed. Actions are basically C program statements surrounded by curly braces.

Yacc Grammar Rules:

1. **Simple rule :**

Yacc rules define what is a legal sequence of tokens in our specification language .

```
e.g. menu_item : LABEL EXEC
                ;
```

This rule defines a non terminal symbol, menu_item in terms of 2 tokens LABEL& EXEC. Tokens are known as "terminal symbols" because parser does not need to expand them any further.

2. **Alternate Rule:**

Any given menu_item may also have keyword DEFAULT appear between the label & the executable command. Yacc allows multiple alternative definitions.

```
e.g. menu_item : LABEL EXEC
               | LABEL DEFAULT EXEC
               ;
```

The colon [:], semicolon [;] & 'or' symbol [|] are part of yacc syntax. They are not part of our menu_item definition. All yacc rules follow the basic syntax shown above and end with semicolon.

DEFAULT appears literally, we have defined %token called DEFAULT, and the lexer returns this token when it finds a certain piece of text.

3. **Literal Characters In Rule:**

There's a way to include literal text within rule but it requires that the lexer passes the character to the parser 1 by 1 as tokens.Whenlexer encounters < OR> it returns the character as a token . We can include literal character in a grammar rule , like this :

```
e.g. menu_item : LABEL EXEC '\n'
 | '< 'LABEL' >' exec '\n'
                ;
```

4. **Recursive Rule:**

We have defined a single menu_item, whereas our file can contain any number of such menu_item. Yacc handles this allowing recursive rules.

```
e.g. menu_items :menu_item
               | menu_itemsmenu_items
                           ;
```

By defining menu_items in terms of itself, we now have rule which means “one or more menu items”.

5. Empty Rule:

The rule for single menu_item, there is another way we could accommodate the optional keyword; by defining empty rule.

```
e.g. menu_item : LABEL default EXEC '\n'
default : /* empty */
        | DEFAULT
```

Token Types:

Token types are declared in yacc using yacc declaration %union like this,

```
%union {
        char *str;
        intnum;
}
```

This defines yylval as being a union types (char *) & (int). this is classical C program union, so any number of types may be defined and union may even contain struct types. we also need to tell yacc which tyoe is associated with which token. This is done by modifying our %token declaration to include a type, like this:

```
% token <str> LABEL
% token <str> EXEC
% token <num> INT
```

Now, we need to modify lexer .

```
yylval .str=strdup(yytext);
```

just before the lexer returns the LABEL & EXEC tokens, we need t include-

```
yylval .num=atoi(yytext);
```

just before the lexer returns the INT token.

yyparse( ):

Yacc generates a single function called yyparse(). This function requires no parameters and returns either 0 on success or 1 on failure. “Failure” in the case of the parser means “if it encounters a syntax error”.’

yyerror( ) function:

The yyerror() function is called when yacc encounters an invalid syntax. The yyerror() is passed a single string (char *) as argument. Unfortunately, this string usually says “parse error”, so on its own, it’s pretty useless.

```
e.g. :yyerror(char *err)
        {       fprintf(stderr,"%s\n",err);     }
```

Shift / Reduce conflicts:

A shift/reduce conflict occurs when there would be enough tokens shifted (saved) to make up a complete rule , but the next token may allow a longer rule to be applied. In the event of shift/reduce conflict, the parser will opt for shift operations and hence try to build the longer rule. If our grammar has “optional structure” such as an optional “else” following an “if” statement , then it may not be possible to eliminate all shift/reduce conflict from grammar rule.

Yacc calls its output file y.temp.c for parser and y.tab.h for token definition. The y.tab.h contains the token definitions.

Execution:

1. lexfilename.l

2. yacc –d filename.y
3. cc lex.yy.cy.tab.c –ll –ly
4. ./a. out

Aim: Lex&Yacc parser for calculator.

Directions for program:

1. We first create an ".l" file which contains header file "y.tab.h" i.e. for parser.
2. We write rules to identify lexemes and return tokens to parser.
3. Then we create one ".y" file, in which the rule section consists of declaration of %union which contains attribute name and its type for various tokens and non terminals.
4. %left and %right is used to define precedence and associatively of operators.
5. CFG contains the expression for addition, subtraction, multiplication and division.
6. For division, we'll check whether divisor is zero or not. If it is, the displays "divide by zero" error message.
7. Finally, print the result of evaluation.

Test case:

Output:

3 * 2 =

6.00

1.7 Conclusion: Thus, we successfully performed parser for sample language using YACC.

Signature of Staff with date

1.11 Questions

Q.1 what is parser? What are different types of parser?
Q.2 how parser works?
Q.3 Describe various in-built variables and functions available in YACC.
Q.4 Write sequence of commands used to compile LEX and YACC program
Q.5 what is the difference between lex and yacc?

EXPERIMENT NO: -05 (Group A) **DATE:-**

1.1Title: Int. code generation for sample language using LEX and YACC
1.2 Aim: To implement Intermediate code generation for sample language using LEX and YACC.

1.3 Objectives: - To understand need of Intermediate code generation
- To study different forms of Intermediate code generation

1.4 Hardware used: Experimental Setup/Instruments/ Devices:

1.5 Software used (if applicable) / Programming Languages Used:

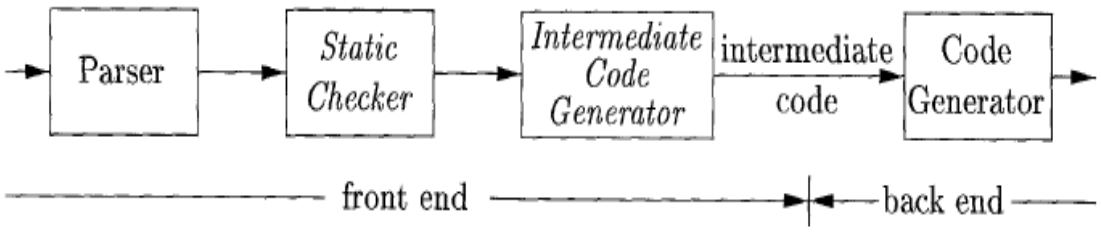
1.6 Theory:-

Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis. After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine. We consider an intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register. The output of the intermediate code generator in figure (b) consists of the three-address code sequence.

tl = inttofloat (60)
t2 = id3 * tl
t3 = id2 + t2
id1 = t3

There are several points worth noting about three-address instructions. First, each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done; the multiplication precedes the addition in the source program. Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction. Third, some "three-address instructions" like the first and last in the sequence, above, have fewer than three operands.



Fig(a) : Position of intermediate code generation module in front end of compiler

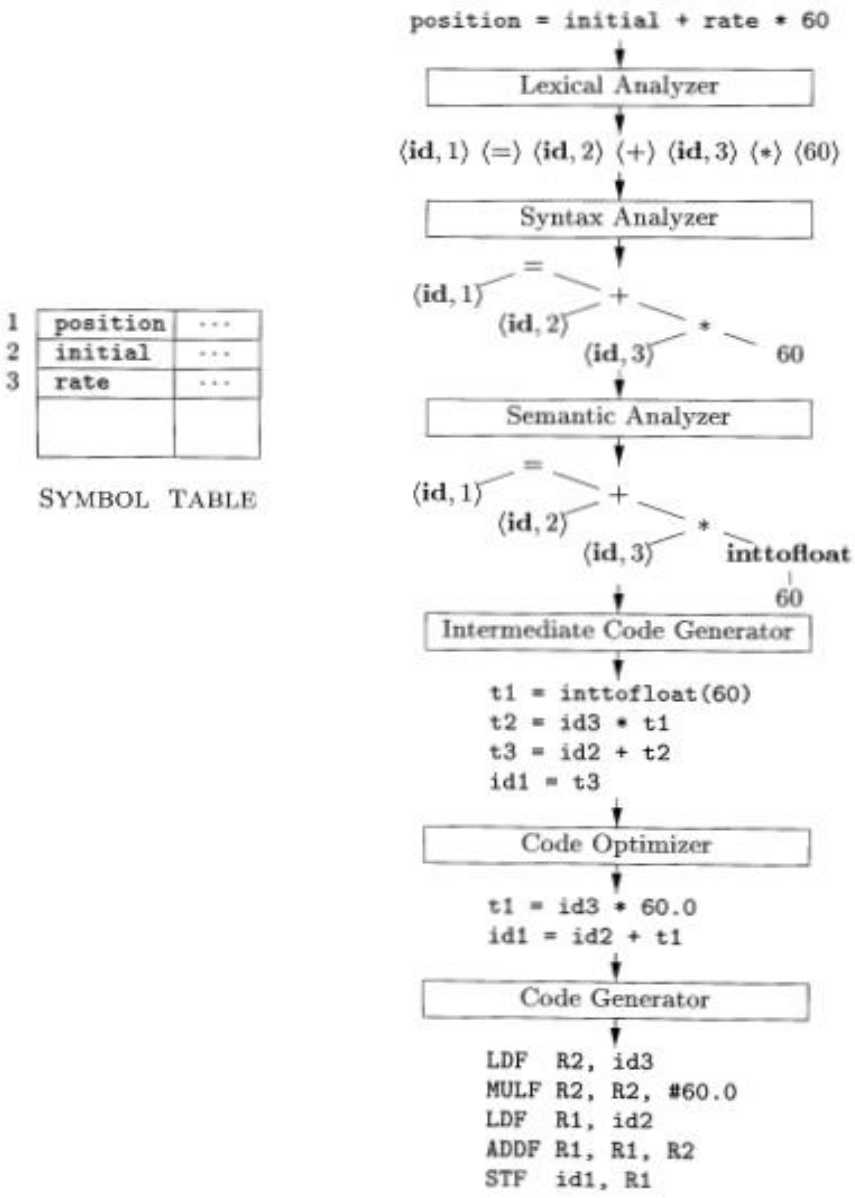


Figure (b): Generation of intermediate code

Direction for program:

1) The three address code are represented in the form of a = b + c in which there are atmost three operand & two operator.

2) For implementation using parser,we define rules related to alphabets [a-z] in lex file & "$" symbol we return 0.

4) The token union defines one character array of place.

5) And then we define token for left associative +,-,*,/.

6) Then we generate three address code according to the priority & associativity of operators present in expression.

Right hand side values of expression will be taken from input where for LHS the temporary string will be created for every new expression

Test case Output:

f=a+b-c/d*e
t1=a+b
t2=c/d
t3=t2*e
t4=t1-t3
f=t4

1.7 Conclusion: Thus, we successfully implemented a parser for three address code.

Signature of Staff with date

1.8 Questions:

1. What is intermediate code generation?
2. What are the difference between Syntax tree and DAG?
3. What are advantages of three address code?
4. What is role of intermediate code in compiler?

EXPERIMENT NO: -06 (Group A) **DATE:-**

1.1 Title: Implement a K-Means Clustering using C++
1.2 Aim: Implement a simple approach for k-means clustering using C++.

1.3 Objectives: - To understand need of clustering
- To study clustering using k-means

1.4 Hardware used: Experimental Setup/Instruments/ Devices:

1.5 Software used (if applicable) / Programming Languages Used:

1.6 Theory:-

A] K-means Algorithm:
K-means clustering is the method of classifying items into k groups (where k is the number of pre-chosen groups).
The grouping is done by minimizing the sum of squared distances (Euclidean distances) between items and the corresponding centric.
A centroid is the "center of mass of a geometric object of uniform density", though here, we will consider mean vectors as centroids.
K-means algorithm is used to create partitions means groups and those partitions are defined by the centroids.
And the initial partitioning can be done in a verity of ways:-
Dynamically chosen: This method is good when the amount of data is expected to grow. The initial cluster means can simply be the first few items of data from the set. For instance, if the data is grouped into 3 clusters, then the initial cluster means will be the first 3 items of data.

a. Randomly chosen: Almost self-explanatory, the initial cluster means are randomly chosen values within the same range as the highest and lowest of the data values.
b. Choosing from upper and lower bounds: Depending on the types of data in the set, the highest and lowest (or at least the extremities) of the data range are chosen as the initial cluster means.

K-means Algorithm Steps:

An algorithm outline assumes two clusters, and each individual scores include two variables. Adding another variable for each individual is as easy as adding calculations within the type of step like step 4 or step 5.
Step1: Choose the number of clusters k.
Step2: Set the initial partition, and the initial mean vectors for each cluster.
Step3: For each remaining individual.
Step4: Get averages/means for comparison to the cluster1:

Add individuals A value to the sum of A values of the individuals in cluster1, then divide by the total number of scores that were assumed.

Add individuals B value to the sum of B values of the individuals in cluster1, then divide by the total number of scores that were assumed.
Step5: Get averages/means for comparison to the cluster2:
Add individuals A value to the sum of A values of the individuals in cluster2, then divide by the total number of scores that were assumed.

Step6: If the averages found in step 4 are closer to the mean values of cluster1, then this individual belongs to cluster1, and the averages found now become the new mean vectors for cluster1.

Step7: If there are more individuals to process, continue again with step 4.Otherwise go to step8.

Step8: Now compare each individuals distance to its own clusters mean vector, and to that of the opposite cluster. The distance to its clusters mean vector should be smaller than its distance to the other vector. If not, relocate the individual to the opposite cluster.

Step9: If any relocation occurred in step 8, the algorithm must continue again with step 3, using all individuals and the new mean vectors. If no relocations are occurred, stop. Clustering is complete.

B] K-means Clustering Examples:

1. As a simple illustration of a k-means algorithm consider the following data set consisting of the stores of two variables on each of seven individuals:

Subject	A	B
1.	1.0	1.0
2.	1.5	2.0
3.	3.0	4.0
4.	5.0	7.0
5.	3.5	5.0
6.	4.5	5.0
7.	3.5	4.5

2. This data set is to be grouped into two clusters. As a first step in finding a sensible initial partition, let the A & B values of the two individuals furthest apart (using the Euclidean distance measure) , define the initial cluster means, giving:

	Individual	Mean Vector(Centroid)
Group1	1	(1.0, 1.0)
Group2	4	(5.0,7.0)

3. The remaining individuals are now examined in sequence and allocated to the cluster to which they are closest, in terms of Euclidean distance to the cluster mean. The mean vector is reallocated each time a new member is added. This leads to the following series of steps:

	Cluster1		Cluster2	
Step	Individual	Mean Vector(centroid)	Individual	Mean Vector(centroid)
1.	1	(1.0,1.0)	4	(5.0,7.0)
2.	1,2	(1.2,1.5)	4	(5.0,7.0)
3.	1,2,3	(1.8,2.3)	4	(5.0,7.0)
4.	1,2,3	(1.8,2.3)	4,5	(4.2,6.0)
5.	1,2,3	(1.8,2.3)	4,5,6	(4.3,5.7)
6.	1,2,3	(1.8,2.3)	4,5,6,7	(4.1,5.4)

4. Now the initial partition has changed, and the two clusters at this stage having the following characteristics:

	Individual	Mean Vector(centroid)
Cluster1	1,2,3	(1.8,2.3)
Cluster2	4,5,6,7	(4.1,5.4)

5. But we can't yet be sure that each individual has been assigned to the right cluster. So, we compare each individuals distance to its own cluster mean and to that of the opposite cluster. And we find

Individual	Distance to mean (centroid) of cluster1	Distance to mean (centroid) of cluster2
1.	1.5	5.4
2.	0.4	4.3
3.	2.1	1.8
4.	5.7	1.8
5.	3.2	0.7
6.	3.8	0.6
7.	2.8	1.1

6. Only individual 3 is nearer to the mean of the opposite cluster (cluster2) than its own (cluster1). In other words, each individuals distance to its own clusters mean should be smaller that the distance to the other cluster mean (which is not the case with individual3).Thus individual3 is related to cluster2 resulting in the new partition.

	Individual	Mean Vector(centroid)
Cluster1	1,2,	(1.3,1.5)
Cluster2	3,4,5,6,7	(3.9,5.1)

The interactive relocation would now continue from this new partition until no more relocation occurs. However, in this example each individual is now nearer its own cluster mean than that of the other cluster and the iteration stops, choosing the latest partitioning as the final cluster solution.

Also, it is possible that the k-means algorithm won't find a final solution. In this case it would be a good idea to consider stopping the algorithm after a pre-chosen maximum of iterations.

1.7 Conclusion : In this way, we have implemented a simple approach for k-means clustering.

Signature of Staff with date

1.8 Questions

Q. 1 what is clustering?
Q. 2 what is difference between clustering and classification?
Q. 3 how distance between two data points is calculated?
Q. 4 which are different methods used for clustering?
Q. 5 how mean is calculated in k-means clustering?

EXPERIMENT NO: -07 (Group B) **DATE:-**

1.1 Title: Implementation of 0-1 knapsack problem using branch and bound approach using c++.
2Aim: Implementation of 0-1 knapsack problem using branch and bound approach.

1.3 Objectives: -To understand branch and bound approach.
- To know the working of 0-1 knapsack problem.

1.4 Hardware used: Experimental Setup/Instruments/ Devices:

1.5 Software used (if applicable) / Programming Languages Used:

1.6 Theory:-

Branch & Bound:

1. Branch and Bound is a general algorithmic method for finding optimal solutions of various optimization problems.

2. Branch and bounding method is a general optimization technique that applies where greedy method and dynamic programming fail, however it is much slower. Indeed, it often leads to exponential time complexities in worst case.

3. On the other hand, if applied carefully, it can lead to algorithms that run risibly fast on average.

4. There are two problems that can be solved using branch and bound technique. They are Knapsack problem and Travelling Sales Person problem.

0/1 Knapsack Problem:-

The 0/1 knapsack problem states that there are 'n' objects given & capacity of knapsack is 'm'. Then select some objects to fill the knapsack in such a way that it should not exceed the capacity of knapsack and maximum profit can be earned. The knapsack problem is maximization problem that means we will always seek for maximum PiXi(wherePi represents profit of object Xi).

Minimize profit-∑PiXi

subject to ∑WiXi

such that ∑WiXi<=m and Xi=0 or 1 where 1<=i<=n

Algorithm:

The algorithm for computing C^(x) is given as below.

```
AlgoC_Bound (total_profit,total_wt,k)
{
        pt= total_profit;
        wt= total_wt;
        for(i=k+1 to n )do
        {
                wt = wt + w[i]
                if (wt<= m )
                        pt = pt + p[i]
                else
                        return (pt + (1-(wt-m))/w[i] * p[i]);
        }
        returnpt;
}
```

The algo for computing U(x) is given as below.

```
AlgoU_Bound (total_profit,total_wt,k,m)
{
        pt= total_profit;
        wt = total_wt;
        for(i= k+1 to n )do
        {
                if(wt + w[i] < =m) then
                {
                        pt<=pt - p[i];
                        wt<= wt + w[i];
                }
        }
        return pt;
}
```

Mathematical Model:

Let S be the System

S={I,F,O}

where,I=set of input

F=set of functions

O=set of outputs

I={I1,I2,I3,I4}

I1=total no. of items

I2=Profit of items

I3=wt. of items

I4 = wt. constraints means maximum capacity

F={F1,F2,F3,F4}

F1=Get_iInput()

F2=Sort_iInput()

F3=Calculate_LB()

F4=Knapsack_BB()

O={O1,O2}

O1=solution

O2=maximum profit obtained.

1.7 Conclusion:

In this way we have implemented 0/1 Knapsack problem using branch and bound approach.

Signature of Staff with date

1.8 Questions

Q. 1 Which different algorithm design strategies are used to solve 0-1 knapsack problem?
Q. 2 Compare backtracking and branch and bound approach.
Q. 3 What is difference between LCBB, FIFOBB and LIFOBB?

EXPERIMENT NO: -08 (Group B) **DATE:-**

1.1 Title: Code optimization using DAG/Labeled tree.
1.2 Aim: Code optimization using DAG

1.2 Objectives: -To know the need of code optimization
- To learn different techniques of code optimization

1.4 Hardware used: Experimental Setup/Instruments/ Devices

1.5 Software used (if applicable) / Programming Languages Used:

Programming Languages Used: C / C++

1.6 Theory:-

The Principal Sources of Optimization

A compiler optimization must preserve the semantics of the original program .Except in very special circumstances, once a programmer chooses and implements a particular algorithm, the compiler cannot understand enough about the program to replace it with a substantially different and more efficient algorithm.
A compiler knows only how to apply relatively low-level semantic transformations, using general facts such as algebraic identities like i + 0 = i or program semantics such as the fact that performing the same operation on the same values yields the same result.

Machine-Independent Optimizations

High-level language constructs can introduce substantial run-time overhead if we naively translate each construct independently into machine code. Elimination of unnecessary instructions in object code, or the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing is usually called "code improvement" or "code optimization.".

Most global optimizations are based on data-flow analyses, which are algorithms to gather information about a program. The results of data-flow analyses all have the same form: for each instruction in the program, they specify some property that must hold every time that instruction is executed. The analyses differ in the properties they compute. For example, a constant-propagation analysis computes, for each point in the program, and for each variable used by the program, whether that variable has a unique constant value at that point.

This information may be used to replace variable references by constant values, for instance. As another example, a livens analysis determines, for each point in the program, whether the value held by a particular variable at that point is sure to be overwritten before it is read. If so, we do not need to preserve that value, either in a register or in a memory location.

Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. For example, a straightforward algorithm generates the intermediate code (1), using an instruction for each operator in the tree representation that comes from the semantic analyzer. A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the **inttofloat** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, **t3** is used only once to transmit its value to id1 so the optimizer can transform **(1)** into the shorter sequence.

t1=id3*60.0
id1=id2+t1 …….(2)

There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called "optimizing compilers, a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.

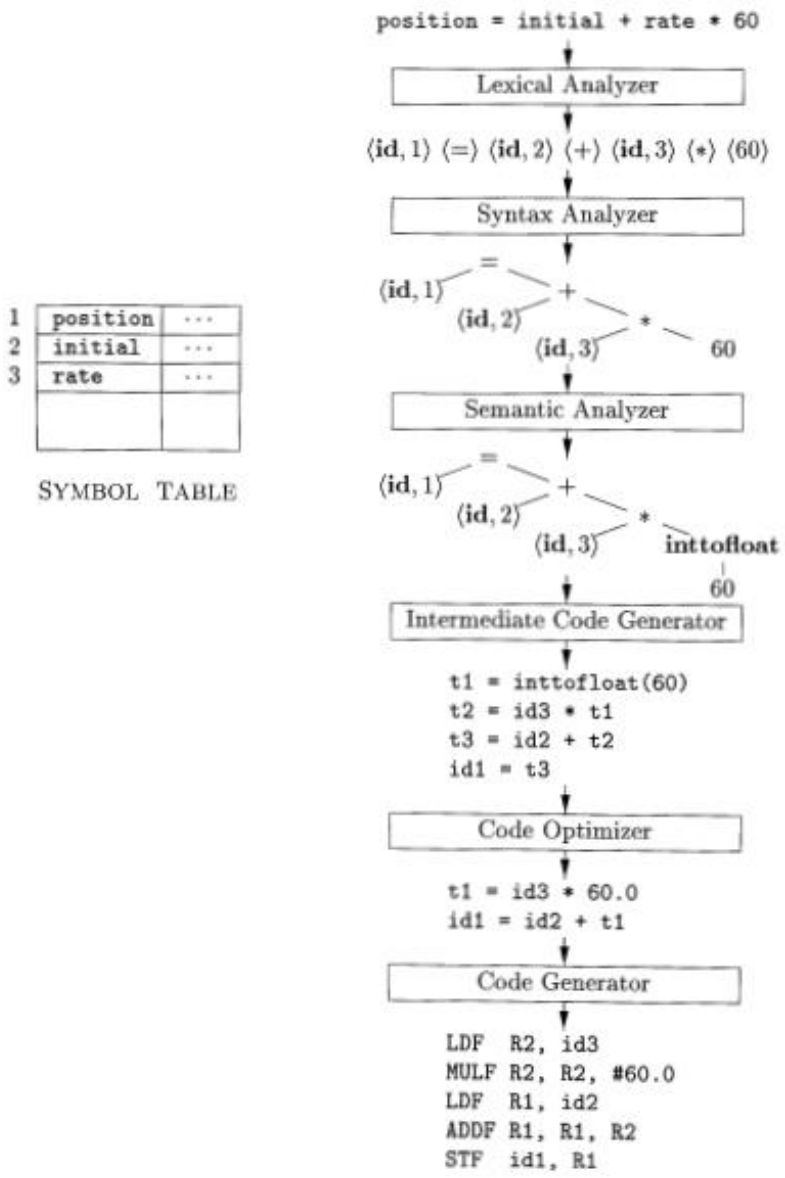


Figure b: Generation of optimized code

Directions for program:

1) As we know, the address code is represented as: a=b+ c*d
2) This 3 address code is converted into target code using lex & yacc
3) This target code of 3 address code is optimized, that means, the unnecessary terms in the target code are reduced and eliminate from target code ,so it will be converted into optimized 3 address code
4) Suppose , our target code will be ,

F=a + b – c / d *e
t1= a + b
t2=c /d
t3=t2*e
t4=t1 - t3
f=t4

In above target code, the t4=t1-t3 is unnecessary term as t4 is assigned to f, we can optimized this form, as F=t1-t3 so the t4=t1-t3 is eliminated

5) For such implementation we will create a Yacc file, in which we scan the complete 3 address code search whether there are some unwanted terms are present or not.

6) In main function, we read a text file which contains 3 address code and generate the optimization for same.

Test case Input:

f=a + b – c / d *e
t1=a + b
t2=c / d
t3=t2 * e
t4=t1 - t3
f=t4

Test case Output:

f=a + b – c / d * e
t1=a + b
t2=c / d
t3=t2 * e
f=t1 - t3

1.7 Conclusion : In this way, we have successfully studied DAG and implemented a code optimization.

Signature of Staff with date

1.8 Questions

Q. 1 What is Data flow equation?
Q. 2 What is live variable analysis?
Q. 3 What is DAG?
Q. 4 What is loop optimization?

EXPERIMENT NO: -09 (Group B) **DATE:-**

1.1 Title: Code generation using DAG / labeled tree.

1.2 Aim: Code generation using labeled tree.

1.3 Objectives:

-To understand the need of code generation
-To study how target code is generated from intermediate code

1.4 Hardware used: Experimental Setup/Instruments/ Devices

1.5 Software used (if applicable) / Programming Languages Used:

Programming Languages Used: C
Compiler: LEX, YACC

1.6 Theory:-

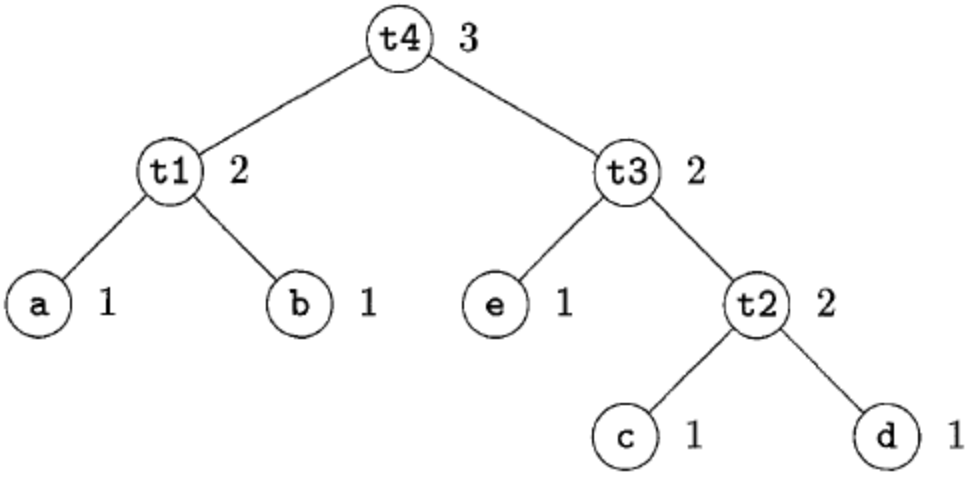
It can be proved that, in our machine model, where all operands must be in registers, and registers can be used by both an operand and the result of an operation, the label of a node is the fewest registers with which the expression can be evaluated using no stores of temporary results. Since in this model, we are forced to load each operand, and we are forced to compute the result corresponding to each interior node, the only thing that can make the generated code inferior to the optimal code is if there are unnecessary stores of temporaries. The argument for this claim is embedded in the following algorithm for generating code with no stores of temporaries, using a number of registers equal to the label of the root.

Algorithm:

Generating code from a labeled expression tree
Input: A labeled tree with each operand appearing once (i.e., no common sub expressions) and a number of registers $r >$ **2.**
Output: An optimal sequence of machine instructions to evaluate the root into a register, using no more than *r* registers, which we assume are R1, Rz , . . . , *R*, .
Method: Apply the following recursive algorithm, starting at the root of the tree, with base b = 1. For a node N with label r or less and we shall not repeat those steps here. However, for interior nodes with a label $k >$ r, we need to work on each side of the tree separately and store the result of the larger sub tree. That result is brought back into memory just before node N is evaluated, and the final step will take place in registers R,-l and R,. The modifications to the basic algorithm are as follows:
1. Node N has at least one child with label r or greater. Pick the larger child (or either if their labels are the same) to be the "big" child and let the other child be the "little" child.
2. Recursively generate code for the big child, using base b = 1. The result of this evaluation will appear in register R.
3. Generate the machine instruction ST tk, R,, where tk is a temporary variable used for temporary results used to help evaluate nodes with label k.

4. Generate code for the little child as follows. If the little child has label r or greater, pick base b = 1. If the label of the little child is *j* < r, then pick b = r - *j*. Then recursively apply this algorithm to the little child; the result appears in R,.
5. Generate the instruction **LD** R,-l, tk.
6. If the big child is the right child of N, then generate the instruction
OP R,, R,, R,-l. If the big child is the left child, generate OP R,, R,-l, R,.

Example:



Let us revisit the expression represented by Fig, but now assume that r = 2; that is, only registers **R1** and **R2** are available to hold temporaries used in the evaluation of expressions. When we apply Algorithm to Fig, we see that the root, with label 3, has a label that is larger than r = 2. Thus, we need to identify one of the children as the "big" child. Since they have equal labels, either would do. Suppose we pick the right child as the big child. Since the label of the big child of the root is 2, there are enough registers. We thus apply Algorithm to this subtree, with b = 1 and two registers. The result looks very much like the code we generated in Fig, but with registers **Rl** and **R2** in place of **R2** and **R3.** This code is

LD R2, d
LD Rl, c
ADD R2, R1, R2
LD R1, e
MUL R2, R1, R2

Now, since we need both registers for the left child of the root, we need to generate the instruction Next, the left child of the root is handled. Again, the number of registers is sufficient for this child, and the code is

LD R2, b
LD Rl, a
SUB R2, Rl, R2

Finally, we reload the temporary that holds the right child of the root with the instruction and execute the operation at the root of the tree with the instruction

ADD R2, R2, R1

The complete sequence of instructions

LD R2, d

LD R1, c
ADD R2, R1, R2
LD R1, e
MUL R2, R1, R2
ST t3, R2
LD R2, b
LD R1, a
SUB R2, R1, R2
LD Rl, t3
ADD R2, R2, R1

Mathematical Model:

Let S be the System

S={I,F,O}

where,I=set of input

F=set of functions

O=set of outputs

I={I1}

I1=labelled tree in form of expression.

F={F1,F2,F3,F4}

F1=Search()

F2=Check()

F3=cog()

F4=add()

O={O1}

O1=Code generation

1.7 Conclusion:

In this way, we have successfully completed the code generation using labeled tree.

Signature of Staff with Date

1.8 Questions:

Q. 1 Explain different storage allocation strategies.
Q. 2 what are advantages of register oriented architecture?
Q. 3 which tools are available for code generation?

EXPERIMENT NO: -10 (Group B) **DATE:-**

1.1 Title: Using LEX and YACC generate the abstract syntax tree.
1.2 Aim: Generating abstract syntax tree using LEX and YACC.

1.3 Objectives:

-To learn parsing phase of compiler

1.4 Hardware used: Experimental Setup/Instruments/ Devices

1.5 Software used (if applicable) / Programming Languages Used:

Programming Languages Used: C
Compiler: LEX, YACC

1.6 Theory:-

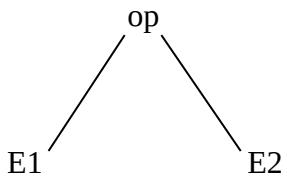
A. Introduction to Intermediate Code Generation:
The front end of a compiler constructs an intermediate representation of the source program from which the back end generates the target program.
There are 2 kinds of Intermediate Representations:

- Trees, including parse trees and (abstract) syntax trees.
- Linear representations, especially "three-address code."

B. Introduction to Abstract Syntax Tree:
A useful starting point for designing a translator is a data structure called an abstract syntax tree. In an *abstract syntax tree* for an expression, each interior node represents an operator; the children of the node represent the operands of the operator. More generally, any programming construct can be handled by making up an operator for the construct and treating as operands the semantically meaningful components of that construct.

Abstract syntax trees, or simply *syntax trees*, resemble parse trees to an extent. However, in the syntax tree, interior nodes represent programming constructs while in the parse tree, the interior nodes represent non terminals. Many non terminals of a grammar represent programming constructs, but others are "helpers" of one sort of another, such as those representing terms, factors, or other variations of expressions. In the syntax tree, these helpers typically are not needed and are hence dropped. To emphasize the contrast, a parse tree is sometimes called a concrete syntax tree, and the underlying grammar is called a concrete syntax for the language.

C. Construction of Syntax Trees:
The syntax tree represents an expression formed by applying the operator op to the sub expressions represented by *El* and *E2.*



Syntax trees can be created for any construct, not just expressions. Each construct is represented by a node, with children for the semantically meaningful components of the construct.

For example, the semantically meaningful components of a C while-statement:

while (expr) *stmt*

These are the expression expr and the statement *stmt.* The syntax-tree node for such a while-statement has an operator, which we call while, and two children-the syntax trees for the expr and the *stmt.*

The translation scheme constructs syntax trees for a representative, but very limited language of expressions and statements. All the non terminals in the translation scheme have an attribute *n,* which is a node of the syntax tree. Nodes are implemented as objects of class Node. Class Node has two immediate subclasses: Expr for all kinds of expressions, and *Stmt* for all kinds of statements. Each type of statement has a corresponding subclass of *Stmt;* for example, operator while corresponds to subclass While. A syntax-tree node for operator while with children x and y is created by the Pseudocode

new While (x, y)

Which creates an object of class While by calling constructor function While, with the same name as the class. Just as constructors correspond to operators, constructor parameters correspond to operands in the abstract syntax.

D. Syntax Trees for Statements:

For each statement construct, we define an operator in the abstract syntax.

add → addl + term { add.n = **new** Op('+',add1.n, term.n); }
| term { add.n = term.n; }

sub → subl - term { sub.n = **new** Op('-',sub1.n, term.n); }
| term { sub.n = term.n; }

mul → mull * term { mul.n = **new** Op('*',mul1.n, term.n); }
| term { mul.n = term.n; }

div → divl / term { div.n = **new** Op('*',div1.n, term.n); }
| term { div.n = term.n; }

block → '{' stmts '}' { b1ock.n = stmts.n; }

factor → (expr) { factor.n = expr.n; }

|**num** { factor.n = **new** Num (**num**.value); }

Fig1: Construction of syntax trees for expressions and statements

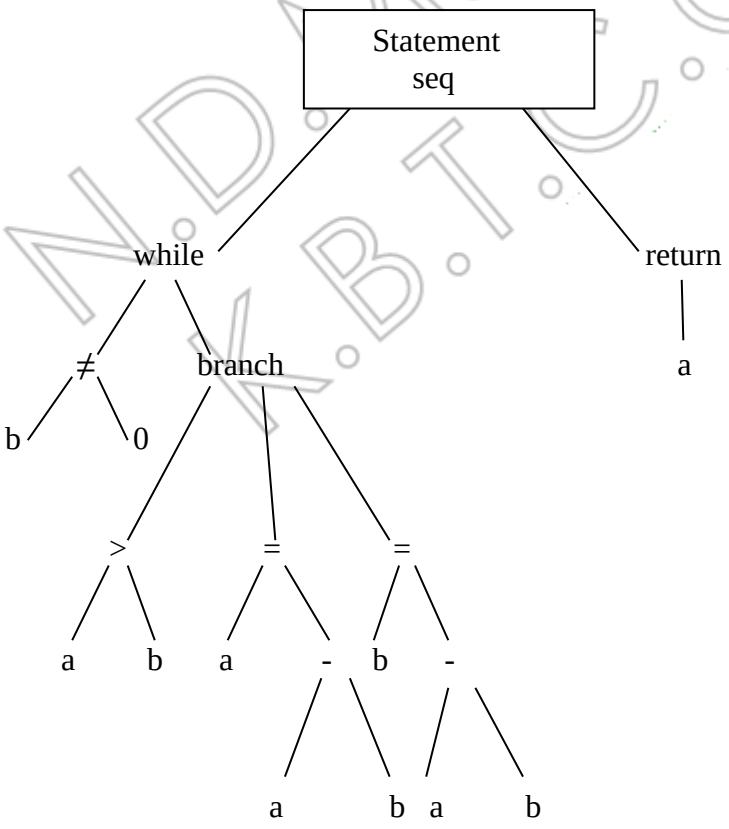
E. Application in Compiler:-

Abstract syntax tree are data structure widely used in compilers due to their property of representing the structure of program code.
As AST is usually the result of syntax analysis phase of compiler. It often serves an intermediate representation of the program through several stages that compilers returns and has a strong impact on the final output of compiler.

Example
Input: Abstract syntax tree for the following code.

```
While b ≠ 0
  if    a > b
        a := a – b
else
       b := b – a
return a.
```

solution: The Abstract syntax tree for the following code is :



1.7 Conclusion : Abstract syntax allows us to group "similar" operators to reduce the number of cases arid subclasses of nodes in an implementation of expressions. Hence in this way, we studied and implemented Abstract Syntax Tree using Lex and Yacc

Signature of Staff with Date

1.8 Questions:

1 What is difference between parse tree and syntax tree? Give example.
2 Define lexical, syntax and semantic errors with their examples?

EXPERIMENT NO: -11 (Group B) **DATE:-**

1.1 Title: Implement Apriori approach to organize the data items on a shelf.

1.2 Aim: Implement Apriori approach for data mining to organize the data items on a shelf using following table of items purchased in a Mall

Transaction ID	Item1	Item2	Item3	Item4	Item 5	Item6
T1	Mango	Onion	Jar	Key-chain	Eggs	Chocolates
T2	Nuts	Onion	Jar	Key-chain	Eggs	Chocolates
T3	Mango	Apple	Key-chain	Eggs	-	-
T4	Mango	Toothbrush	Corn	Key-chain	Chocolates	-
T5	Corn	Onion	Onion	Key-chain	Knife	Eggs

1.3 Objectives:

-To find frequent item sets
-To find support
-To learn to organize data

1.4 Hardware used: Experimental Setup/Instruments/ Devices

1.5 Software used (if applicable) / Programming Languages Used:

1.6 Theory:-

The Apriori Algorithm:

Apriori is a seminal algorithm for mining frequent item sets for Boolean association rules. The name of the algorithm is based on the fact that the algorithm uses *prior knowledge* of frequent item set properties. Apriori employs an iterative approach known as a *level-wise* search, where *k*-item sets are used to explore (*k*+1)-item sets. First, the set of frequent 1-itemsets is found by scanning the database to accumulate the count for each item, and collecting those items that satisfy minimum support. The resulting set is denoted *L*1.Next, *L*1 is used to find *L*2, the set of frequent 2-itemsets, which is used to find *L*3, and so on, until no more frequent *k*-item sets can be found. The finding of each *Lk* requires one full scan of the database.

1. The join step: To find *Lk*, a set of candidate *k*-item sets is generated by joining *Lk*-1 with itself. This set of candidates is denoted *Ck*. Let *l*1 and *l*2 be item sets in *Lk*-1. The notation *li*[*j*] refers to the *j*th item in *li* (e.g., *l*1[*k*-2] refers to the second to the last item in *l*1). By convention, Apriori assumes that items within a transaction or item set are sorted in lexicographic order. For the (*k*-1)-item set, *li*, this means that the items are sorted such that *li*[1] <*li*[2] < ... <*li*[*k*-1]. The join, *Lk*-1 on *Lk*-1,is performed, where members of *Lk*-1 are joinable if their first (*k*-2) items are in common. That is, members *l*1 and *l*2 of *Lk*-1 are joined if (*l*1[1] = *l*2[1]) ^ (*l*1[2] = *l*2[2]) ^...^(*l*1[*k*-2] = *l*2[*k*-2]) ^(*l*1[*k*-1] <*l*2[*k*-1]). The condition *l*1[*k*-1] <*l*2[*k*-1] simply ensures that no duplicates are generated. The resulting item set formed by joining *l*1 and *l*2 is *l*1[1], *l*1[2], ... , *l*1[*k*-2], *l*1[*k*-1], *l*2[*k*-1].

2. The prune step: *Ck* is a superset of *Lk*, that is, its members may or may not be frequent, but all of the frequent *k*-item sets are included in *Ck*. A scan of the database to determine the

count of each candidate in *Ck* would result in the determination of *Lk* (i.e., all candidates having a count no less than the minimum support count are frequent by definition, and therefore belong to *Lk*).

TABLE:1 Transactional data for an *All Electronics* branch.

TID	*List of item_IDs*
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

EXAMPLE:

Let's look at a concrete example, based on the *All Electronics* transaction database, *D*, of Table 1.1.We use Figure 1.2 to illustrate the Apriori algorithm for finding frequent item sets in *D*.

1. In the first iteration of the algorithm, each item is a member of the set of candidate 1-itemsets, *C*1. The algorithm simply scans all of the transactions in order to count the number of occurrences of each item.

2. Suppose that the minimum support count required is 2, that is,*min sup* = 2. (Here,we are referring to *absolute* support because we are using a support count. The corresponding relative support is 2/9 = 22%). The set of frequent 1-itemsets, *L*1, can then be determined. It consists of the candidate 1-itemsets satisfying minimum support.

3. To discover the set of frequent 2-itemsets, *L*2, the algorithm uses the join $L1 \bowtie L1$ to generate a candidate set of 2-itemsets, *C*2.8 *C*2 consists of $\binom{|L_1|}{2}$ 2-itemsets. Note that no candidates are removed from*C*2 during the prune step because each subset of the candidates is also frequent.

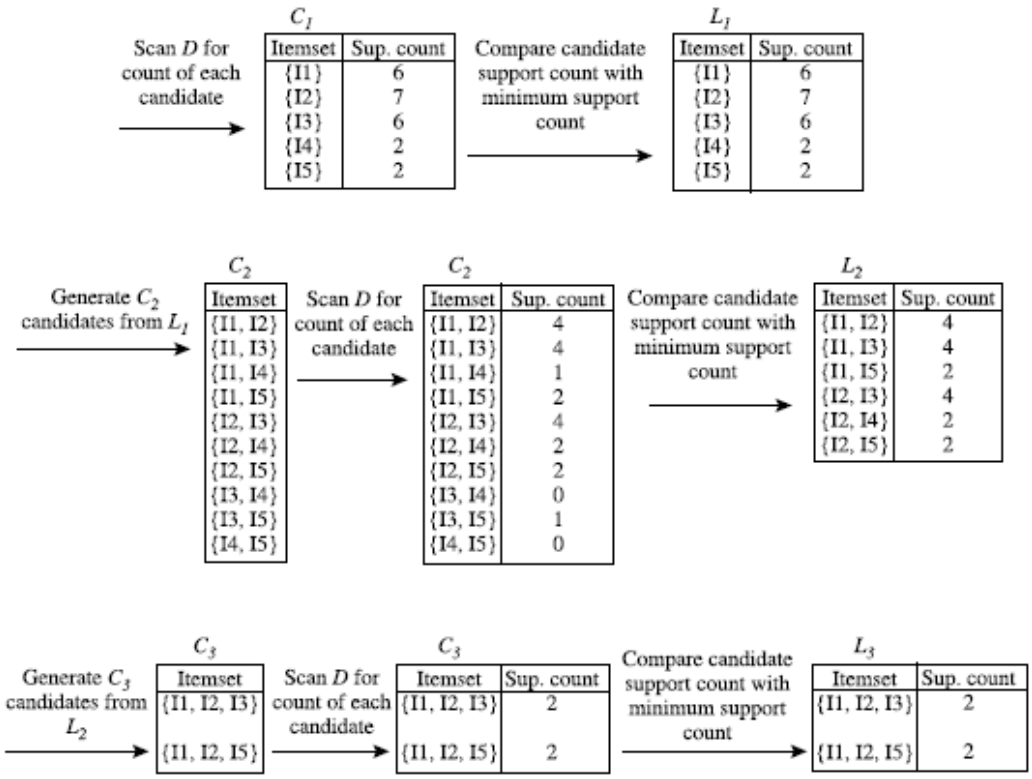


Fig:1.2 Generation of candidate itemsets and frequent itemsets, where the minimum support count is 2.

4. Next, the transactions in*D*are scanned and the support count of each candidate itemset in *C*2 is accumulated, as shown in the middle table of the second row in Figure 1.2.
5. The set of frequent 2-itemsets, *L*2, is then determined, consisting of those candidate 2-itemsets in *C*2 having minimum support.
6. From the join step, we first get
$C_3 = L_2 \bowtie L_2 = \{\{I1, I2, I3\}, \{I1, I2, I5\}, \{I1, I3, I5\}, \{I2, I3, I4\}, \{I2, I3, I5\}, \{I2, I4, I5\}\}$. Based on the Apriori property that all subsets of a frequent itemsetmust also be frequent, we can determine that the four latter candidates cannot possibly be frequent.We therefore remove them from*C*3, thereby saving the effort of unnecessarily obtaining their counts during the subsequent +(Figure 1.2).
8. The algorithm uses *L*3 on *L*3 to generate a candidate set of 4-itemsets, *C*4. Thus, *C*4 = f, and the algorithm terminates, having found all of the frequent itemsets.

PESUDO CODE:
Input:*D*, a database of transactions;
min sup, the minimum support count threshold.
Output: *L*, frequent itemsets in *D*.
Method:
(1) *L*1 = find frequent 1-itemsets(D);
(2) for (*k* = 2;*Lk*-1 6= f;*k*++) f
(3) *Ck*= apriorigen(*Lk*-1);
(4) for each transaction *t* 2 *D* f // scan *D* for counts
(5) *Ct* = subset(*Ck*, *t*); // get the subsets of *t* that are candidates
(6) for each candidate *c* 2 *Ct*
(7) c.count++;
(8) g

(9) *Lk*= f*c* 2 *Ck*j*c*:*countmin sup*g
(10) g
(11) return*L* = [*kLk*;
procedureapriori gen(*Lk*-1:frequent (*k*-1)-itemsets)
(1) for each itemset*l*1 2 *Lk*-1
(2) for each itemset*l*2 2 *Lk*-1
(3) if (*l*1[1] = *l*2[1])^(*l*1[2] = *l*2[2])^:::^(*l*1[*k*-2] = *l*2[*k*-2])^(*l*1[*k*-1] <*l*2[*k*-1]) then f
(4) *c*= *l*1 on *l*2; // join step: generate candidates
(5) if has infrequent subset(*c*, *Lk*-1) then
(6) delete*c*; // prune step: remove unfruitful candidate
(7) else add *c* to *Ck*;
(8) g
(9) return *Ck*;
procedure has infrequent subset(*c*: candidate *k*-itemset;
Lk-1: frequent (*k*-1)-itemsets); // use prior knowledge
(1) for each (*k*-1)-subset *s* of *c*
(2) if*s* 62 *Lk*-1 then
(3) return TRUE;
(4) return FALSE;

Frequent Itemsets:
The sets of item which has minimum support (denoted by L(i) for ith-Itemset).

1.7 Conclusion:
In this way we can implement apriori algorithm for organizing the data items.

Signature of Staff with Date

1.8 Questions (Page no.)
Q. 1 How frequent item sets are calculated?
Q. 2 How support is calculated?
Q. 3 How confidence is used in apriori algorithm?

EXPERIMENT NO: -12 (Group B) **DATE:-**

1.1 Title:- Implement Naive Bayes for Classification.

1.2 Aim: Implement Naive Bayes for categorical data.

1.3 Objectives: -To be able to find prior probability
-To be able to find conditional probability
-To be able to find posterior probability

1.4 Hardware used: Experimental Setup/Instruments/ Devices

1.5 Software used (if applicable) / Programming Languages Used:

1.6 Theory:-

Naive classifiers are a family of simple probabilistic classifiers based on applying Bayes theorem with strong (naive) independence assumptions between the features.

Naive Bayes is a simple technique for constructing classifiers: models that assign class labels to problem instances, represented as vectors of feature values, where the class labels are drawn from some finite set. It is not a single algorithm for training such classifiers, but a family of algorithms based on a common principle: all naive Bayes classifiers assume that the value of a particular feature is independent of the value of any other feature, given the class variable. For example, a fruit may be considered to be an apple if it is red, round, and about 10 cm in diameter. A naive Bayes classifier considers each of these features to contribute independently to the probability that this fruit is an apple, regardless of any possible correlations between the color, roundness and diameter features.

Naive bays implementation Steps:

- Calculate prior Probabilities of class to be predicted
- Calculate conditional probabilities
- Calculate posterior probability
- Highest probability among above is predicted class for query tuple.

Problastic model:

- D is a dataset consisting tuples X
- $D=\{X_1,X_2,\ldots,X_n\}$
- Tuple X has attributes {*Work Type, Age, Qualification, Experience*}

Suppose that there are m classes, $C_1,C_2,\ldots,C_m$

Given a tuple X, the classifier will predict that X belongs to class having the highest posterior probability conditioned on X.

$P(Ci|X)>P(Cj|X)$ for $1 \leq j \leq m, j \neq i$

- Thus the *maximum prediction hypothesis* by bays theorem:

$$P(Ci|X) = \frac{P(X|Ci) * P(Ci)}{P(X)}$$

- To predict class label of X:

$$P(X|Ci) * P(Ci) > P(X|Cj) * P(Cj) \text{ for } 1 \leq j \leq m, j \neq i$$

1.8 Conclusion:

In this way, we have implemented Naive Bays for categorical data only.

Signature of Staff with Date

1.9 Questions:

1. What is Bayes theorem?
2. Formula for bayes theorem?
3. Write uses of naive bayes classification
4. What are different types of learning?
5. How posterior probability is calculate

EXPERIMENT NO: -13 (Group B) **DATE:-**

1.1Title:- Implementation of K-NN approach with suitable example**.**
1.2 Aim: Implementation of k-NN approach with suitable example

1.3 Objectives: To perform classification using k-NN

1.4 Hardware used: Experimental Setup/Instruments/ Devices

1.5 Software used (if applicable) / Programming Languages Used:

Programming Languages Used: C / C++

1.6 Theory:

The k Nearest Neighbour (kNN) is a very intuitive method that classifies unlabeled examples based on their similarity with examples in the training set.kNN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until classification. The kNN algorithm is among the simplest of all machine learning algorithms. In pattern recognition, the k-Nearest Neighbours algorithm (or kNN for short) is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. The output depends on whether kNN is used for classification or regression:
1.In kNN classification, the output is a class membership. An object is classified by a majority vote of its neighbours, with the object being assigned to the class most common among its k nearest neighbours (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbour.
2.In kNN regression, the output is the property value for the object. This value is the average of the values of its k nearest neighbours.

kNN advantages:
1. The cost of the learning process is zero.
2. No assumptions about the characteristics of the concepts to learn have to be done.
3. Complex concepts can be learned by local approximation using simple procedures.
4. Robust to noisy training data.

kNN disadvantages:
1. Need to determine value of parameter k.
2. In distance based learning, it is not clear which type of distance to use and which attribute to use to produce the best results.
3. Computation cost is quite high because we need to compute distance of each query instance to all the training samples.

kNN Example:
1. Let us suppose there is a data from the questionnaires survey (to ask people opinion) and objective testing with two attributes (acid durability and strength) to classify whether a special paper tissue is good or not. Here are four training samples Now the factory produces a new paper tissue that pass laboratory test with X1 = 3 and X2 = 7. Without another expensive survey, we need to guess what the classification of this new tissue is?

X1 = Acid Durability (seconds)	X2 = Strength (kg/square meter)	Y = Classification
7	7	Bad
7	4	Bad
3	4	Good
1	4	Good

2.Solving the above problem using kNN.

1. Determine parameter K = number of nearest neighbors,Say K = 3.

2. Calculate the distance between the query instance and all the training samples Coordinate of query instance is (3, 7), instead of calculating the distance compute square distance which is faster to calculate (without square root).

X1 = Acid Durability (seconds)	X2 = Strength (kg/square meter)	**Square Distance to query instance (3, 7)**
7	7	$(7-3)^2+(7-7)^2=16$
7	4	$(7-3)^2+(4-7)^2=25$
3	4	$(3-3)^2+(4-7)^2=9$
1	4	$(1-3)^2+(4-7)^2=13$

3. Sort the distance and determine nearest neighbors based on the K-th minimum distance.

X1 = Acid Durability (seconds)	X2 = Strength (kg/square meter)	Square Distance to query instance (3, 7)	**Rank minimum distance**	**Is it included in 3-Nearest neighbors?**
7	7	$(7-3)^2+(7-7)^2=16$	**3**	**Yes**
7	4	$(7-3)^2+(4-7)^2=25$	**4**	**No**
3	4	$(3-3)^2+(4-7)^2=9$	**1**	**Yes**
1	4	$(1-3)^2+(4-7)^2=13$	**2**	**Yes**

4. Gather the category Y of the nearest neighbors. Notice in the second row last column that the category of nearest neighbor (Y) is not included because the rank of this data is more than 3 (=K).

X1 = Acid Durability (seconds)	X2 = Strength (kg/square meter)	Square Distance to query instance (3, 7)	Rank minimum distance	Is it included in 3-Nearest neighbors?	**Y = Category of nearest Neighbor**
7	7	$(7-3)^2+(7-7)^2=16$	3	Yes	**Bad**
7	4	$(7-3)^2+(4-7)^2=25$	4	No	**-**
3	4	$(3-3)^2+(4-7)^2=9$	1	Yes	**Good**
1	4	$(1-3)^2+(4-7)^2=13$	2	Yes	**Good**

5. Using simple majority of the category of nearest neighbors as the prediction value of the query instance.
Thus we have 2 good and 1 bad result. Since 2 is greater than 1 then we conclude that a new paper tissue that pass laboratory test with X1 = 3 and X2 = 7 is included in Good category.

Algorithm kNN (k, training samples, query instance):
1. start.
2. Calculate the distance between the query instance and all the training samples.
3. Sort the distance and determine nearest neighbors based on the K-th minimum distance.
4. Gather the category Y of the nearest neighbors.
5. Use simple majority of the category of nearest neighbors as the prediction value of the query instance.
6. stop.

1.7 Conclusion : In this way, we have implemented kNN approach.

Signature of Staff with Date

1.8Questions Q. 1 What is classification?
Q. 2 What are different methods of classification?
Q. 3 How nearest neighbor is calculated?
Q. 4 How value of k is selected?
Q. 5 Why k-NN is called lazy learner?

EXPERIMENT NO: -14 **DATE:-**

1.1Title: Implementing recursive descent parser for sample language.
1.2 Aim: Implementing recursive descent parser for sample language.

1.3 Objectives: To be able to find recursive descent parsing with backtracking.

1.4 Hardware used: Experimental Setup/Instruments/ Devices

1.5 Software used (if applicable) / Programming Languages Used

1.6 Theory:

Recursive descent parser is a kind of top-down parser built from a set of mutually recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the productions of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

Predictive parser is a recursive descent parser that does not require backtracking. Predictive parsing is possible only for the class of LL(*k*) grammars, which are the context-free grammars for which there exists some positive integer *k* that allows a recursive descent parser to decide which production to use by examining only the next *k* tokens of input. The LL(*k*) grammars therefore exclude all ambiguous grammars, as well as all grammars that contain left recursion. Any context-free grammar can be transformed into an equivalent grammar that has no left recursion, but removal of left recursion does not always yield an LL(*k*) grammar. A predictive parser runs in linear time.

Recursive descent with backtracking is a technique that determines which production to use by trying each production in turn. Recursive descent with backtracking is not limited to LL(k) grammars, but is not guaranteed to terminate unless the grammar is LL(k). Even when they terminate, parsers that use recursive descent with backtracking may require exponential time.

Although predictive parsers are widely used, and are frequently chosen if writing a parser by hand, programmers often prefer to use a table-based parser produced by a parser generator, either for an LL(*k*) language or using an alternative parser, such as LALR or LR. This is particularly the case if a grammar is not in LL(*k*) form, as transforming the grammar to LL to make it suitable for predictive parsing is involved. Predictive parsers can also be automatically generated, using tools like ANTLR.

1.7 Conclusion : In this way, we have implemented recursive descent parser for sample language.

Signature of Staff with Date

1.8Questions Q. 1 What is Recursive descent parser?

Q2 Explain the role of Predictive parser?

Q3. How to implement the parser?

EXPERIMENT NO: -15 (Group C) **DATE:-**

1.1Title: Code generation using IBURG tool

1.2 Aim: Code generation using "iburg" tool.

1.3 Objectives:

-To study "iburg" tool.

1.4 Hardware used: Experimental Setup/Instruments/ Devices

1.5 Software used (if applicable) / Programming Languages Used:

Tool used: iburg

1.6 Theory:-

A] iburg tool-

i] iburg is a code generator tool that uses dynamic programming at compile time,it is based on tree pattern matching and dynamic programming at compile time.It is based on tree pattern .

ii] It accepts tree patterns,associated tools costs and semantic actions [for register allocation and object code emission] and produce tree matches that produces cover of minimum cost.Cost can involve arbitary computations.

B] iburg Tree Matcher

iburg tree produces two functions . Label reduce are used to generate tree matcher.User calls these subroutines. Label[P] makes a bottom up,left to right pass over tree for chain rules to indicate that the patterns that associated with rule 'm'matches the node with cost c.

C] EBNF grammer for iburg specification

grammar ->y.y{dci} %% {rule}
dci/ %term {identifier=integer}
rule->nonterm:tree=integer[cost];
cost->(integer)
tree->term(tree,tree)
|tree(tree)
|term
|nonterm
Iburg specification-
1] %term ADDI =309 ADDRLP=295 ASCCNI=53
2]%term CNSTI=21 CVCI=85 IOI=661 INDIRC=67
3]%%
4]stmt ASGNI(disp,reg)=4(1);
5]stmt:reg=5;
6]reg:ADDI(reg,rc)=6(1);
7]reg:CVCI(INDI RC(disp))=7(1);

Building and Configuration of iburg

Download iburg torball from http:// code/google.com/pliburg/"
Archieve the contains files table/:iburg tool.
copyright : copyright notice support of most recent changes.
makefile :make specification empty
custom.mk : makes file customization.
iburg.c :srccode for most of iburg
gram.c:yacc output
gram.y:yacc input for burg grammar
iburg.h : header files for iburg.c and gram.y
iburg : test files

1.7Conclusion :

In this way, we have studied how to perform code generation using iburg tool.

Signature of Staff with Date

1.8 Questions:

1. Which are different compilers you have used?
2. Which are different tools used to design compiler?
3. Give examples of parallel and distributed compilers.