The Short Hand Programming Language

Amar Banerjee
Course - Compilers
2019
SERC
IIIT - Hyderabad

August 22, 2019

Abstract

This document describes the syntax and user manual to use the "Short Hand"programming language and it's capabilities. The document contains the purpose of this programming language, the keywords of the language, syntax and semantics and finally the context free grammar for better understanding it's essence. We expect this document to help students and language enthusiasts to start exploring and enriching the capabilities of this language.

This manual is an guideline to enable users to identify the syntax and semantic of the "Short Hand" language. This is a work in progress and this document is expected to undergo numerable changes as we progress on developing this language and it's features.

Contents

1 Introduction

1.1 Need of a "Short Hand" Language

Programming languages intend to raise the level of abstraction and capture the programming intentions of the programmer. Many languages like C, C++, JAVA, Python etc. are meant to serve this purpose. However, most of the available programming languages tend to make the programming very tedious in terms of writing a lot of code to explicate the programming intentions. The "Short Hand Language" attempts at mitigating this by enabling programmers write less code to explicate their programming intentions.

1.2 "Short Hand" structure

The "Short Hand" language follows the following structure:-

1. This language is based on file system, where every file contains a set of programming constructs. Multiple file together can create multiple executables. Unlike other programming languages, the "Short Hand" programme can compile with broken dependency, and it is expected that the dependency will be resolved during the runtime.

→ Chapter 2

   You can find detailed information about how to create a programme is "Short Hand" chapter 2.

2. Data variables and type declaration. This language expects the types to be declared before any usage. The language primarily focusses on enabling users create custom types by composing primitive types.

→ Chapter 2

   You can find detailed information about how to declare types and variables. chapter 2.

3. After the variables and types are declared the language is structured to have functions and constructs. Constructs are statements which execute. The constructs can be written even outside functions.

→ Chapter 6

   Chapter 6 contains more information about the new layout.

2 The "Short Hand" programming

2.1 Common Rules

There are almost no common rules because every kind of programming language has different requirements and needs a specific layout. We have tried to keep the layout in this language very simple. The structure is divided into 2 parts:-

Data / Variables and *Logic*

Any one using this language to create programmes should consider *what* the data is, and *how* to use it in logic.

An important criteria is if the reader will read the document from start to end like a detective novel (linear reading) or if he wants to find certain information as in a telephone directory or a reference manual.

In addition to that, the layout has to consider certain conventions, like the habits of the reader or "Corporate Design" rules that distinguish publications from different publishers. [1]

! → The main purpose of this language to reduce the syntactic effort and stress more on the semantic aspect of programming

The following "rules of thumb" will be valid for most applications:

File extension : The file extension to save a "Short Hand" programme in the file system is ".short". Any ".short" file is identifed by the compiler as a "Short Hand" programme.

Programme Length : It is expected that a user should not write more than 100 lines of code in any '.short' file. This constrain is enforced to keep the size of any programme short. This also encourages users to divide the programme into multiple sub-modules hence maintaining modularity.

3 The Grammar for "Short Hand"

The context free grammar for "Short Hand" is specified below. We enchourage users and reviewers to suggest modifications and optimizations to make this more usable. This section is meant for those who are interested to know about the grammar and abstract syntax tree for this language. Others can ignore this section and procees with the next chapter.

The grammar of the language is expresses as a context free grammar (CFG) form below:-

3.1 Terminals

The CFG uses terminal rules which form the basis to compose other rules of the grammar. The terminal rules are described below:-

id : The *id* is a terminal rules which is described below:-

$id \rightarrow [a - zA - Z0 - 9\_.-]^*$
This rule includes all the possible combinations of characters and numbers that are possible with any number of repetitions, but without any spaces.

E.g., a, var1, hello etc.

character : The *character* is a terminal rules which is described below:-
$character \rightarrow (')[a - zA - Z0 - 9\_.-](')$
This rule includes all the possible characters and numbers, with any number of repetition between two "(' ')".

E.g., 'a', '1', 'Hello World !!'

number : The *number* is a terminal rules which is described below:-
$number \rightarrow [-?0 - 9.]^*$
This rule includes all the possible combinations of numbers that are possible with any number of repetitions. Numbers could be positive, negative or with floating integers.

E.g., 123, -99999, 3.14 etc.

[1] Compare the layout of different daily papers or magazines like "page" or "invers".

boolean : The *boolean* is a terminal rule which is described below:-

$boolean \rightarrow (true)|(false)$

Here *true* and *false* both represent the boolean 1 and 0 values.

primitivetype : The *primitivetype* rule includes the different keywords for corresponding data types. The rule is described below.

$primitivetype \rightarrow (int)|(+int)|(-int)|$

$(float)|(+float)|(-float)|(bool)|(char)|void$

list : The *type* rule defines the terminal symbols to defile arrays or multiple dimentions.
$list(\rightarrow ([)number(]) \cdot list|\epsilon$

operator : The terminal rule defining the operators is defined below:
$operator \rightarrow (+)|(-)|$
$(*)|(\backslash)|(\%)|(<)|$
$(>)|(<=)|(>=)|(\&)|$
$(|)|(==)|(!)|(!=)|$
$(!>)|(!<)$

→ Section **??**

Section **??** describes the keywords and syntax to define variables and data types. LaTeX.

3.2 Non-Terminals

The CFG uses non-terminal rules which form the basis to compose other production rules of the grammar. The non-terminal rules are described below:-

ENTRY_RULE : The programme begins with an entry rule which allows the programmer to start writing the programme in the file. The *ENTRY_RULE* is described as :-
$ENTRY\_RULE \rightarrow PATH\_LIST\_RULE \cdot PROGRAMME\_RULES$

PATH_LIST_RULE : This provides a rule to include references to other ".short" files. The *REFERENCE_RULE* is described as follows:-

$PATH\_LIST\_RULE \rightarrow (refer)id|id \cdot$

$(,)PATH\_LIST\_RULE|\epsilon(;)$

This is analogous to the initial *include* statements used in C and C++.

Here (refer) is a terminal (keyword) and denotes the beginning of the referencing rule.

Here (;) is a terminal (keyword) and denotes the termination of a rule.

PATH : The PATH rule is described below:-
$PATH \rightarrow id$

PROGRAMME_RULES : The PROGRAMME_RULES initiates the grammar to trigger writing the programme. It is described as :-
$PROGRAMME\_RULES \rightarrow STATEMENT\_LIST\_RULES|$
$FUNCTION\_LIST\_RULES$
The PROGRAMME_RULES defines how the language could be used to write programmes. As seen the PROGRAMME_RULES is based on 2 rules described below.

STATEMENT_LIST_RULES : The STATEMENT_LIST_RULE describes the grammar for writing multiple statements. The rule is described below:-
$STATEMENT\_LIST\_RULES \rightarrow$
$STATEMENT\_RULE$
$\cdot\ STATEMENT\_LIST\_RULES|\epsilon$

STATEMENT_RULE : The STATEMENTS_RULE describes the grammar for writing programatic statements. The rule is described below:-
$STATEMENT\_RULE \rightarrow DATA\_DEFINITION\_RULE$
$|DATA\_MANIPULATION\_RULE$
$|FUNCTION\_CALL\_RULE|IO\_RULE|COMMENTS$

DATA_DEFINITION_RULE : This rule describes how data variables can be declared and assigned as a statement:-
$DATA\_DEFINITION\_RULE \rightarrow$
$DATA\_DECLARATION\_RULE \cdot$
$DATA\_OPERATION\_RULE|\epsilon;$

DATA_DECLARATION_RULE : This rule is used to define data variables and its types.
$DATA\_DECLARATION\_RULE \rightarrow$
$type \cdot id|DATA\_DEFINITION\_RULE|\epsilon$

DATA_MANIPULATION_RULE : This rules enables the different constructs which provide a way to manipulate data and write the programming logic. The rule is described below:-

$DATA\_MANIPULATION\_RULE \rightarrow$
$DATA\_OPERATION\_RULE|CONDITIONAL\_RULE|$
$CONDITIONAL\_SHORT\_RULE|LOOPING\_RULE$

DATA_OPERATION_RULE : This rule allows to write operations on data.
$DATA\_OPERATION\_RULE \rightarrow$
$(id = idoperation)VALUE|DATA\_OPERATION\_RULE|\epsilon$

VALUE : The rule for creating values in terms of simple values and array lists is :
$VALUE \rightarrow$
$SIMPLE\_VALUE|ARRAY\_VALUE$

SIMPLE_VALUE : The rule for creating values which are not lists arrays.
$SIMPLE\_VALUE \rightarrow$
$id|number|character|boolean|null$
Here the id, number, character, boolean are all terminals and null is a symbol which denotes the null or ϕ value.

ARRAY_VALUE : The rule for creating values for list array types:
$ARRAY\_VALUE \rightarrow$
$id|number|character|boolean|null|ARRAY\_VALUE|\epsilon$

Here the id, numberm character, boolean are all terminals and null is a symbol which denotes the null or ϕ value.

CONDITIONAL_RULE : This rule allows to write conditions as in if-then-else.
$CONDITIONAL\_RULE \rightarrow (if) \cdot (() \cdot CONDITION \cdot$
$()) \cdot (\{) \cdot STATEMENT\_RULE \cdot (\}) |$
$(else) \cdot (\{) \cdot STATEMENT\_RULE \ (\})$

CONDITIONAL_SHORT_RULE : This rule allows to write conditions as in if-then-else.
$CONDITIONAL\_SHORT\_RULE \rightarrow CONDITION$
$(?) id (:) id$

CONDITION : This rules enables specifying conditions.
$CONDITION \rightarrow id|numbers(>)|(<)|$
$(==) id|boolean|id$

LOOPING_RULE : The looping rules allow to define loop conditions. The language supports looping constructs.
$LOOPING\_RULE \rightarrow loop(DATA\_DEFINITION\_RULE;$
$CONDITION; DATA\_OPERATION\_RULE)$
$(\{) STATEMENT\_RULE (\})$

IO_RULE : The rule for printing on console and taking input from keyboard. $IO\_RULE \rightarrow printVALUE|readid;$

FUNCTION_CALL_RULE : The rule is responsible for the calling of functions and passing parameters. $FUNCTION\_CALL\_RULE \rightarrow id(PARAMS);$

PARAMS : This rule describes the rules to pass parameters in a function. $PARAMS \rightarrow id|id, PARAMS|\epsilon$

FUNCTION_RULES : The FUNCTION_RULES describes the grammar for writing programatic statements. The rule is described below:-
$FUNCTION\_RULES \rightarrow FUNCTION\_DEFINE\_RULE|\epsilon$
$|FUNCTION\_RULES$

FUNCTION_DEFINE_RULE : The STATEMENTS_RULE describes the grammar for writing programatic statements. The rule is described below:-
$FUNCTION\_DEFINE\_RULE \rightarrow (function) \cdot type \cdot id$
$\cdot (() \cdot DATA\_DECLARATION\_RULE \cdot ()) (\{) \cdot STATEMENT \cdot$
$(return) VALUE (;) \cdot (\})$

COMMENTS : Comments can be described by he following rule
$COMMENTS \rightarrow SINGLE\_LINE\_COMMENT|MULTI\_LINE\_COMM$

SINGLE_LINE_COMMENT : Demo
$SINGLE\_LINE\_COMMENT \rightarrow \#id|SINGLE\_LINE\_COMMENT|\epsilon$

MULTI_LINE_COMMENT : Demo
$MULTI\_LINE\_COMMENT \rightarrow \#\#id|MULTI\_LINE\_COMMENT|\epsilon\#\#$

4 Keywords and Syntax

The keywords supported in "Short Hand" are listed below. We encourage people to provide their suggestions about the keywords and their usage.

Pre-programming : The "Short Hand" language starts with including references to existing libraries (optional). This acn be done using the *refer* keyword, followed by the path of the referred file.

Usage → *refer* "path to the file"

Comments : The language provides commenting of single and multiple lines. These lines will not be considered to be processed by the compiler.

Usage → # This is a single line comment

Usage → ## multi line comment ##

Data Types : "Short Hand" supports numeric, character and boolean data types and primitive types.

Null value : The *null* value can be denoted by using the *null* keyword. The *null* denotes that there is no value associated to an identifier.

Usage → *int* anynumber = *null*. Here the variable with the name *anynumber* is assigned a ϕ value.

Numeric Types : The numeric like integer and decimal types can be declared by using the keywords:

int : *int* represents the integer type. *int* can be annotated to represent positive or negative integer type. This int type variable can store both values which are positive or negative or zero.

Usage → *int* anynumber. Here the variable with the name *anynumber* has an integer type.

+int : *+int* represents the positive integer type. This means any variable with this type is intended to hold *ONLY* positive values which are greater than 0.

Usage → *+int* noOfSubjects. Here the variable with the name *noofsubjects* has an integer type and it can contain all non zero positive numbers.

-int : *-int* represents the negative integer type. This means any variable with this type is intended to hold *ONLY* negative values which are less than 0.

Usage → *-int* lossOfMoney. Here the variable with the name *lossOfMoney* has an integer type.

float : *int* represents the decimal type. *int* can be annotated to represent positive or negative integer type. This float type variable can store both values which are positive or negative or zero. However, to store zero a floating decimal point has to be specified.

Usage → *float* anydecimalnumber

+float : *+float* represents the positive decimal type. This means any variable with this type is intended to hold *ONLY* positive decimal values which are greater than 0.0.

Usage → *+float* noOfSubjects. Here the variable with the name *noofsubjects* has an integer type and it can contain all non zero decimal positive numbers.

-int : *-float* keyword represents the negative decimal type. This means any variable with this type is intended to hold *ONLY* negative decimal values which are less than 0.0.

Usage → *-float* lossOfMoney. Here the variable with the name *lossOfMoney* has an integer type.

Character Types : The character types are supported by providing 2 seperate types for single character and multi-character string.

char : The *char* keyword represents the single character type. All single characters (Capital and Small) and special symbols can be assigned to a variable of this type.

Usage → *char* someCharacter; Here the variable *someCharacter* has character type and it can store single numeric, alpha numeric and special symbol characters.

string : The *string* keyword represents the string type. All words and collection of characters can be assigned to a variable of this type.

Usage → *string* someCharacter; Here the variable *myString* has string type and it can store strings of characters.

Boolean Type : The *bool* keyword represents the boolean type. The variables of this type can have values either *true* or *false*.

Usage → *bool* trueOrFalse; Here the variable *trueOrFalse* has boolean type and it can store **true** or **false**.

Arrays : The language supports defining arrays of multiple dimentions. The keyswords to define arrays are:-

[] : The *[]* keyword denotes an array. It can be used multiple times after a variable declaration for defining multi-dimentional arrays. The length of the array has to be specified using a numerical symbol (1-9).

Usage → *type variableName* [number]. The *type* can be any keyword corresponding to a primitive data type like *int, float, char* etc. The *number* is the length of the array.

int myArray [3], denotes that *myArray* is an array which can hold 3 integer values.

int my2DArray [3][4], denotes that *my2DArray* is a 2 dimensional array which can hold 3×4 integer values. Similary, multi dimensional arrays can be defined.

Operators : The language supports primary operations like assignment, addition, subtraction, multiplication, division, modulous, and boolean operations like and, or and not.

= : The = symbol denotes assignment operation. It can be used to assign values to one or more than one variables. It is expected that every variable will be assigned the right type of value. In case of any value mismatch the compiler will throw an error.

Usage → *int* var1 = 24. This statement will assign the integer value 24 to *var1.*

= can also be used to assign multiple values to multiple variables. *int* a, *float* b = (24, 31.76) will assign the integer value 24 to *a* and float value 31.76 to *b.* Likewise multiple values can be assigned together.

int a = 2;
float b, *float* c = (12.34,a); This will produce a compilation error for the assignment of the integer value of a to *float* c

+ : The + symbol denotes addition operation. When supplied with 2 or more operands it will produce the addition of the numbers.

Usage → *int* result = +(var1, var2, var3). This statement will produce result equivalent to var1 + var2 + var3

- : The - symbol denotes subtraction operation. When supplied with 2 or more operands it will produce the difference of the numbers starting from left to right.

Usage → *int* result = -(var1, var2, var3). This statement will produce result equivalent to var1 - var2 - var3

* : The - symbol denotes subtraction operation. When supplied with 2 or more operands it will produce the difference of the numbers starting from left to right.

Usage → *int* result = -(var1, var2, var3). This statement will produce result equivalent to var1 - var2 - var3

and : The *and* symbol denotes the logical AND operation. When supplied with 2 or more boolean operands it will produce the logical AND result.

Usage → *true and false.*

The keywords for programming constructs like conditions, loops, function return types are given below:-

if : For denoting the begin of a condition

Usage → $if(i > 0)/*statements*/$.

else : For denoting the else part

Usage → $if(i > 0)/*statements*/else/*statements*/.$

else : For denoting the else part

Usage → $loop(inti = 0; i < 10; i = i+1)/*statements*/else/*statemen$

else : For denoting the else part

Usage → $loop(inti = 0; i < 10; i = i+1)/*statements*/else/*statemen$

return : This keyword denotes the return statement for the value returned by a function

Usage → $e.g.....returntrue; .....returnvar1;$

print : This prints a value to the console

Usage → $e.g.print`HelloWorld', e.g.print23, printvar2$

read : This reads a value from the key input

Usage → *e.g.readid*

5 Semantic Checks

The language supports some semantic checks to enable successful compilation and avoid bad code running in the system. The semantic checks are given below.

Check for identifier declaration : Check that all identifiers are called and are declared before defining or assigning values to them.

Array size check : All declared arrays of any dimension should have a dimension size of greater than zero at compile time.

Type checks for operators : The operators are responsible to execute operations based on algebraic rule. For e.x.addition of 2 integers, division between 2 floats etc. However, we cannot perform a logical 'and' operation between any values expect for boolean 'true' or 'false'. Hence the compiler checks for such semantic checks and identify such errors.

Type checks for assignments : Types and assignments are very important for a programming language. For e.g. we do not want to store a string value in an integer identifier. Such semantic checks are supported by this language.

Type checks for function returns : The language has return types for functions. Which means that the functions are allowed to only produce values of specific types as defined in the function declaration. In any other case the compiler is expected to show error messages.

Dead code identification : A block of code which is not being used at all, is referred as dead code. This language compiler identifies basic dead code blocks and shows them as warning to the user.

Operations on null value : The NULL value cannot be processed for any operation. Hence the compiler while compilation, identifies basic scenarios where a null value is being operated on. This will result into an error during execution, hence the compiler upfront shows errors corresponding to this to the user.

6 How to write a programme in "Short Hand"?

6.1 $g(N,k) = \sum_{i=1}^{N} i^k$

```
int scanned_N, scanned_k, final_sum;
read scanned_N;
read scanned_k;

int power(int base, int num)
{
 if (num == 0)
     {return 1;}
 else
     {return base * power(base, num - 1);}
}

int g(int N, int k)
{
```

```
for(int i=1;i<=N){
final_sum = final_sum + power(i,k)
}
}

g(N,k);
```

6.1.1 Check for prime number

```
int n, i, flag = 0;
print "Enter a positive integer: ";
read n;
for(i = 2; i <= n/2; ++i)
{
    // condition for nonprime number
    if(n%i == 0)
    {
        flag = 1;
        break;
    }
}
if (n == 1)
{
  print "1 is neither a prime nor a composite number");
}
else
{
    if (flag == 0)
      print n "is a prime number";
    else
      print n "is not a prime number.";
}
```

6.2 The sum of all prime numbers strictly less than N where N is provided as an input

```
int sum(int limit){
        int number = 2;
        int count = 0;
        long sum = 0;
        for(;count < limit;){
            if(isPrimeNumber(number)){
                sum += number;
                count++;
            }
            number++;
        }
        return sum;
    }

    private boolean isPrimeNumber(int number){

        for(int i=2; i<=number/2; i++){
            if(number % i == 0){
```

```
                return false;
            }
        }
        return true;
    }
print ‘‘Enter value of n’’;
int n = read n;
sum(n);
```

6.3 Pythogorean triplets (x, y, z) where x, y, z are integers and $x^2 + y^2 = z^2$ and $z \leq 100000000$

```
int gcd(int m, int n)
{
    int t;
    while (n) { t = n; n = m % n; m = t; }
    return m;
}

int computePythagoreadTriplet()
{
    int a, b, c, pytha = 0, prim = 0, max_p = 100, aa, bb, cc;

    for (a = 1; a <= max_p / 3) {
        aa = a * a;

       ##  max_p/2: valid limit, because one side of triangle
           must be less than the sum of the other two ##

        for (b = a + 1; b < max_p/2) {
            bb = b * b;
            for (c = b + 1; c < max_p/2) {
                cc = c * c;
                if (aa + bb < cc) break;
                if (a + b + c > max_p) break;

                if (aa + bb == cc) {
                    pytha++;
                    if (gcd(a, b) == 1) prim++;
                }
 c = c + 1;
 b = b + 1;
            }
b = b + 1;
        }
a = a + 1;
    }
}

    print "Up to " max_p ", there are " pytha " triples";
```

6.4 Write a program to print all combinations of 1, , n where n is given as an input

```
# The main function that prints all combinations of size r
# in arr[] of size n. This function mainly uses combinationUtil()
```

```
void printCombination(int arr[], int n, int r)
{
    # A temporary array to store all combination one by one
    int data[r];

    # Print all combination using temprary array 'data[]'
    combinationUtil(arr, data, 0, n-1, 0, r);
}


void combinationUtil(int arr[], int data[], int start, int end,
                     int index, int r)
{
    # Current combination is ready to be printed, print it
    if (index == r)
    {
        for (int j=0; j<r)
{
print data[j];
j = j + 1;
}

        print "\n";
        return;
    }

    for (int i=start; i<=end && end-i+1 >= r-index)
    {
        data[index] = arr[i];
        combinationUtil(arr, data, i+1, end, index+1, r);
i = i + 1;
    }
}


    int arr[];
    int r;
arr = read;
n = read;
    printCombination(arr, n);
```

6.5 Insertion Sort

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n)
    {
        key = arr[i];
        j = i - 1;
        for (j >= 0 ; arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
```

```
        arr[j + 1] = key;
i = i + 1;
    }
}

void printArray(int arr[], int n)
{
int i;
for (i = 0; i < n; i++)
cout << arr[i] << " ";
cout << endl;
}


    int arr[];
    int n = sizeof(arr) / sizeof(arr[0]);
   arr = read;
    insertionSort(arr, n);
    printArray(arr, n);
```

6.6 Radix Sort

```
int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n)
{
if (arr[i] > mx)
{mx = arr[i];}
i = i + 1;

}

    return mx;
}

void countSort(int arr[], int n, int exp)
{
    int output[n]; # output array
    int i, count[10] = {0};

    # Store count of occurrences in count[]
    for (i = 0; i < n)
{
count[ (arr[i]/exp)%10 ]++;
i = i + 1;
}

    for (i = 1; i < 10)
{
count[i] += count[i - 1];
i = i + 1;
}
```

```
    for (i = n - 1; i >= 0)
    {
int itemVar = count[ (arr[i]/exp)%10 ];
        output[itemVar - 1] = arr[i];
        itemVar = itemVar - 1;
count[ (arr[i]/exp)%10 ] = itemVar;
i = i - 1;
    }

    for (i = 0; i < n)
{
arr[i] = output[i];
i = i + 1;
}

}

void radixsort(int arr[], int n)
{
    int m = getMax(arr, n);

    for (int exp = 1; m/exp > 0)
{
countSort(arr, n, exp);
exp = exp * 10;
}

}

void print(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        print  arr[i] "\n";
}

    int arr[];
arr = read;
    int n = sizeof(arr)/sizeof(arr[0]);
    radixsort(arr, n);
    print arr n;
```

6.7 Merge Sort

```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
{
L[i] = arr[l + i];
}
```

```
    for (j = 0; j < n2; j++)
{
R[j] = arr[m + 1+ j];
}


    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    for (i < n1 ; j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    for (;i < n1;)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    for (;j < n2;)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        # Same as (l+r)/2, but avoids overflow for
        # large l and h
        int m = l+(r-l)/2;

        # Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}
```

```
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size)
{
 print A[i];
}

    print "\n";
}


    int arr[];
arr = read;
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    print "Given array is \n";
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    print "\nSorted array is \n";
    printArray(arr, arr_size);
}
```

6.8 Sum of 2 matrices

```
   int m, n, c, d, first[10][10], second[10][10], sum[10][10];

   print "Enter the number of rows and columns of matrix\n";
   read m n;
   printf("Enter the elements of first matrix\n");

   for (c = 0; c < m)
{
      for (d = 0; d < n)
{
         read first[c][d];
 d = d + 1;
}
  c = c + 1;
}

   read "Enter the elements of second matrix\n";

   for (c = 0; c < m; c++)
{
for (d = 0 ; d < n; d++)
{
read second[c][d];
}
}


   read "Sum of entered matrices:-\n";
```

```
   for (c = 0; c < m) {
      for (d = 0 ; d < n) {
         sum[c][d] = first[c][d] + second[c][d];
         print sum[c][d];
d = d + 1;
      }
c = c + 1;
      printf("\n");
   }
```

6.9 Product of matrices

```
  int m, n, p, q, c, d, k, sum = 0;
  int first[10][10], second[10][10], multiply[10][10];

  print "Enter number of rows and columns of first matrix\n";
  read m n;
  print "Enter elements of first matrix\n";

  for (c = 0; c < m;)
{
for (d = 0; d < n; d++)
{
read first[c][d];
}
c = c + 1;
}


  print "Enter number of rows and columns of second matrix\n";
  read p q;

  if (n != p)
{
 print "The matrices can't be multiplied with each other.\n";
}

  else
  {
    read "Enter elements of second matrix\n";

    for (c = 0; c < p)
{
for (d = 0; d < q)
{
for (d = 0; d < q; d++)
{
read second[c][d];
}
}
        read second[c][d];
c = c + 1;
}
```

```
    for (c = 0; c < m) {
      for (d = 0; d < q) {
        for (k = 0; k < p) {
          sum = sum + first[c][k]*second[k][d];
  k = k + 1;
        }
  d = d + 1;
        multiply[c][d] = sum;
        sum = 0;
      }
c = c + 1;
    }

    print "Product of the matrices:\n";

    for (c = 0; c < m) {
      for (d = 0; d < q)
        print  multiply[c][d];
  d = d + 1;
       print "\n";
    }
c = c+ 1;
  }
```

6.10 A program to compute the alphabet histogram in given English language text file

```
   char string[100];
   int c = 0, count[26] = {0}, x;

   print "Enter a string\n";
   gets(string);

   for (;string[c] != '\0';) {

      if (string[c] >= 'a' && string[c] <= 'z') {
         x = string[c] - 'a';
         count[x]++;
      }

      c++;
   }

   for (c = 0; c < 26; c++)
{
print c + 'a' " %c occurs %d times in the string.\n", count[c];
}
```

6.11 A program to list all the unique words in a file