# ALGORITHMS AND PROGRAM MODULES FOR PROCESSING INTERRELATED INFORMATION CELLS

BY

ASHRAF HABIB RUMI



A Thesis

Submitted to the Department of Computer Engineering, Bangladesh University of Engineering and Technology, Dhaka, in partial fulfilment of the requirements for

the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

DECEMBER, 1986

## CERTIFICATE OF RESEARCH

Certified that the work presented in this Thesis is the result of the investigation carried out by the candidate under the supervision of Dr. Syed Mahabubur Rahman at the Department of Computer Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh.

3/12/18

Candidate

## DECLARATION

I hereby declare that neither this thesis nor any part thereof has been submitted or is being concurrently submitted in candidature for any degree at any other university.

13/12/88

Candidate

Accepted as satisfactory for partial fulfilment of the requirements for the degree of M. Sc. Engineering in Computer Engineering.

BOARD OF EXAMINERS

| | | |
|---|---|---|
| i. | Dr. Syed Mahabubur Rahman<br>Associate Professor and Head,<br>Department of Computer Engineering,<br>BUET. | Chairman<br>and<br>Supervisor |
| ii. | Prof. Shamsuddin Ahmed<br>Head, Department of Electrical and Electronics, ICTVTR, Dhaka.<br>( Ex. Dean of the Faculty of EEE & Head of the Department of Computer Engineering, BUET, Dhaka.) | Member<br>( External ) |
| iii. | Mr. Dulal Chandra Kar<br>Assistant Professor,<br>Department of Computer Engineering,<br>BUET. | Member |

## ACKNOWLEDGEMENT

It is a matter of great pleasure on the part of the author to acknowledge his profound gratitude to his supervisor, Dr. Syed Mahabubur Rahman, Associate Professor and Head, Department of Computer Engineering, Bangladesh University of Engineering and Technology (BUET), Dhaka for his support, advise, valuable guidance and his constant encouragement throughout the progress of this work.

The author also wishes to express his sincere gratitude to late Dr. A.K.M. Mahfuzur Rahman Khan (Ex. Professor and head of the Computer Engineering department, BUET) for his keen interest and encouragements for this work. He expresses his gratitude to Dr. Shamsuddin Ahmed, (Ex. Dean of the faculty of Electrical and Electronic Engineering and Head of the Department of Computer Engineering, BUET).

Thanks are due to Mr. Javed Sabir Barkatulla and Mr. Dipak Bin Qusem Chowdhury for their friendly discussions on different problems in the progress of the work.

## ABSTRACT

The present research work has undertaken the task of developing algorithms and software programs for processing a large number of information cells containing high level program statements including assignments, arithmetic and boolean expressions, commands etc . Algorithms for the manipulation of informations on a large cell system by using simple but powerful commands have been developed by considering the structuring facilities. In the present work "top-down" approach of structuring has been used for the development of algorithms and program modules. After cosidering all the external specifications, two main algorithmic modules have been designed. They are the expression interpreter and the screen manipulator. Expression interpreter defines algorithms for the evaluation of arithmetic, logical,conditional expressions with built-in functions. Screen manipulator defines algorithm for interaction intelligently with the user through the display screen. Main module defines algorithms for the production of the proposed cell system and for the interaction between the user and the system . The specification for the interaction between the main module with the data structure of the cell system and these two modules has been defined first , then the detailed design has been carried out more or less independently. Finally the algorithms have been implemented in the form of a program module by considering some coding tricks and local optimization. The coding language used here is the CBASIC compiler, a popular high level language.

# CONTENTS

# CONTENTS

# CHAPTER 1

# INTRODUCTION

## 1.1 GENERAL

Today's computers come in a variety of shapes and sizes. Large mainframe to small microcomputers are now used in many factories , businesses , universities , hospitals , and government agencies to carry out simple to sophisticated business or technical calculations . One of the main part of a computer to run is the software programs. Application programs are becoming one of the most important tool in computer world from the user's point of view. Without these programs each computer user must be acquainted elaborately with the system operating programs such as operating system, programming language etc., which needs him to have a desired level of qulification and experience. Thus it is not possible for a man who is novice in this field.But in the case of a particular application, one does not need to know how to work with an operating system, how to develop a program etc. Thus it is possible and easy for him to use a particular program for his application with only some commands and functions . This has made an application program construction an important, practical area of research in computer science.

Research works in Bangladesh related to the development of algorithm and software for widely used application

programs are still at the initial stage . Some works are undergoing in the department of computer engineering, BUET, in the field of data base software compatible for the mainframe computers. Processing and retrieving of information or data in a useful form through the video screen is one of the most customization facilities of today's computer. The proposed research work has undertaken the task of developing algorithms and a suitable implementation in the form of a program module for processing of large number of programmable information cells. The processing of the cells involve analysis of high level program statements including assignments, arithmetic expressions , logical expressions or conditional expressions. The data or information types handled by the cells consist of value, text string, formula etc. The system will be a powerful tool for solving many types of mathematical, business, and financial problems. Since the algorithms or procedures have been developed for determining how the system behaves. The computer program is a medium in which the algorithms can be expressed and applied. Hence a modular software program has been developed in a microcomputer for the implementation of the algorithms.

## 1.2 OBJECTIVES OF THE THESIS WORK

For the development of the desired system the following informations are important to consider as the objectives of the present thesis work:

- Production of a two-dimensional rectangular grid containing a large number of programmable information cells.

. Intelligent interaction with the user through CRT display screen and the hard copy printer.

. Interaction of powerful but easy to use system commands.

. Calculation of powerful logical, conditional and built-in mathematical functions.

One of the main purposes of the developed system is to manipulate data or information on a large rectanguler array system using simple but powerful commands. These commands may be specified for the following purposes:

. editing the information,
. formatting the information,
. displaying the information,
. calculating the formulas,
. storing the informations,
. printing the informations, etc.

## 1.3 APPLICATION AREAS OF THE THESIS WORK

The developed system is an aggregate of concurrently active objects, organized into a rectangular array of cells similar to the paper spreadsheet used by an accountant. Each cell has a rule specifying how its value is to be determined. Every time a value is changed anywhere in the cell system, all values dependent on it are recomputed instantly and the new values are stored and displayed. The system can be defined as a simulated pocket universe that continuously

maintains its fabric; it is a kit for a surprising range of applications. Here the user illusion is simple, direct and powerful. There are few mystifying surprises because the only way a cell can get a value is by having the cell's own value rule put in there. Thus the developed software module acts as a simple means to tap the power of a computer to do time-cosuming, repetative calculations.

The proposed algorithms may be used for the development of sophisticated application program softwares such as

- electronic spread-sheet developing programs,
- text editor or word-processor developing programs,
- compiler writting programs,
- interpreter writting programs etc.

The developed software program module may be useful for the preparation of data sheets in a tabular form, cashflow analysis or forecasting, balance sheets, profit statements, tax estimation, market share analysis & planning, bar charts, patient records, salary records etc.

# CHAPTER 2

# PROGRAM DEVELOPMENT

## 2.1 INTRODUCTION

Many programmers are often preoccupied with what initially seem to be the most difficult aspect of writing programs; translating ideas into a programming language, or coding. However, this activity is only one stage in a computer programming project, which has at least seven definable components:

# Requirements analysis(RA)
# Specification(SF)
# Design(DG)
# Documentation(DO)
# Coding(CD)
# Testing and debugging(TD)
# Maintenance(M)

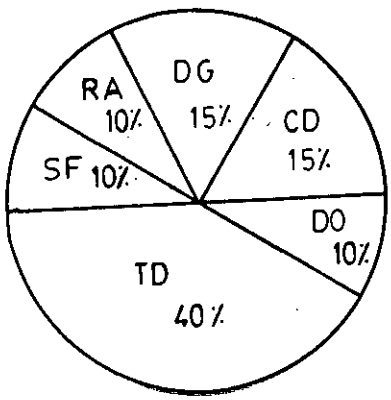


Figure-2.1 Components of program development

The first six stages above take place during the traditional "development cycle" of a programming project. Fig-2.1 shows that coding is just a small part of the development cycle.

## 2.2 STAGES OF PROGRAM DEVELOPMENT [1]

Requirements analysis, present in some form in all problem solving activities, defines the requirements for an acceptable solution to the problem. Most developers do not get involved in this stage, because it usually takes place at a management level and involves issues such as general approach, staffing and other resources, project costs and schedules.

The input and output of a program and their relationships are defined by an external specification. For example, the specification for a text editor defines the format of the text files and it lists all of the editing commands and the effects of each. However, an external specification does not contain a description of how the program achieves these effects; this is part of design.

The structure of a program is defined during the design stage. The design stage often decomposes the problem by outlining a solution in terms of a set of cooperating high-level program modules. This approach requires additional design and internal specification, since each module and its interaction with the others must be specified, and then the internal structure at each module must be designed. Depending on the module complexity, additional decomposition into submodules might also be take place.

Most documentation should be created during the specification and design stage . Concise yet complete documentation is needed to communicate specification and design concepts among the current implementors and future users and maintainers of a program. To a lesser extent, documentation is needed in the coding stage to explain the details of program coding.

In the coding stage the design is translated into a programming language for a specific computer system. When coding begins, design usually stops, which is a good argument for not starting coding prematurely. Several studies have shown that design errors are more common than coding errors. So that a good design is essential to project success. Also one is much more likely to reduce the size of a program or enhance its performance by design improvement in algorithms and data structures than by coding tricks or local optimizations.

The word debugging usually suggests an activity in which the obvious errors in a program are eliminated so that the program runs without "blowing up". Testing refers to a more refined activity that verifies not only that the program runs but also that it meets its external specifications. Testing requires a test plan, basically a set of input patterns and expected responses for varifying the behavior and operating limits of the program. Quite often the test plan is given as part of the original specification to ensure that the tests are not biased in a way that would obscure the errors in a known design.

Large programs require maintenance after they have been put into operation in the field for two reasons. First, there are usually errors that are not detected during the testing stage. Obviously increased effort during the testing stage reduce this maintenance requirement. Second, and almost inevitable, is that users of a system will call for changes and enhancements after the system has been put into operation. The cost of this maintenance is strongly influenced by the specification, design and documentation of a program; therefore maintenance must be considered early in the development cycle.

### 2.2.1 SPECIFICATION AND DESIGN

The external specification of a program comes directly from the results of requirements analysis, while internal specifications come later. During the development cycle, some "looping" between specification and design often occurs as shown in fig-2.2. This looping may be required when attempts to design the program reveal ambiguities , contradiction, or other deficiencies in the external specification. But most of the looping occurs because of the "top-down" approach that is used in problem solving. When the solution to an original external specification is designed as a collection of cooperating modules, each module and its interfaces with its partners must be documented by internal specification. Then the working of each module must be designed. The whole process may be repeated many times as modules at each level are designed as collections of lower level modules.

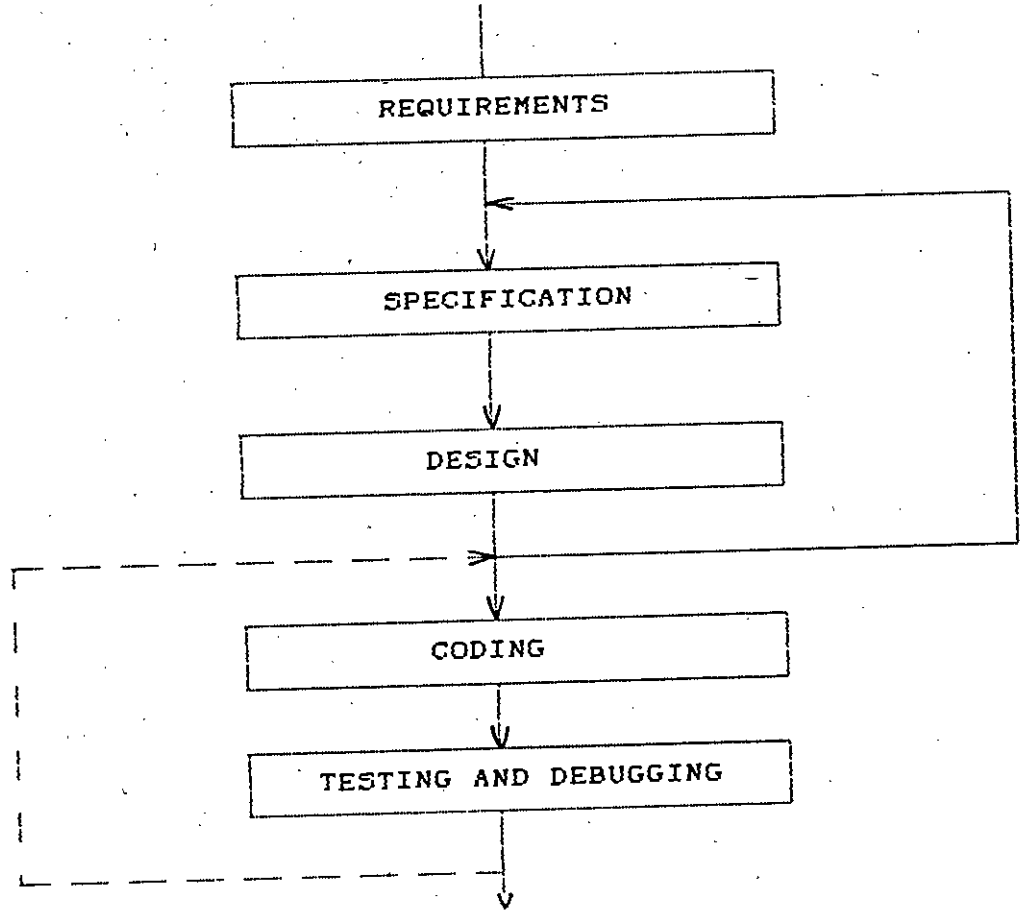


Fig-2.2 The program development cycle

## 2.2.2 DOCUMENTATION

Documentation should not be generated at one fixed time during the program development cycle. Instead, documentation should be generated as appropiate throughout the project. The most important program documentation is generated long before the coding stage, during specification and design.

In general it is a growing practice to make all program self documenting , so that all documentation including specifications and design, contained in the same text as the source code itself. Writing self-documenting code has several advantages over keeping separate handwritten or typed documentation. The advantages are:

* It is easier to relate the documentation to the code.

* Efficient procedures can be instituted for maintaining all code and documentation on a development computer system.

* When a design (or code) is changed, it is convenient to make the appropiate documentation changes ( otherwise there is a tendency for documentation to lag design).

* During revisions and maintenance, if the source code is available then all the documentation is guaranteed to be available too.

The main disadvantage of self-documenting code is that it increases the size of the text files that the development computer system must handle. Including back-up files, a development system may need 200 to 1000 bytes or more of disk stoarge for every byte of object code that is developed. Thus developing a 50k byte application program for a microprocessor may require a microcomputer or minicomputer with a 50 Mbyte system.

### 2.2.3 CODING

Coding is probably the best understood aspect of programming. In fact, coding of a well-designed program is a fairly mechanical operation. The most important to know about coding is that good code must be built upon good design. A programmer cannot expect to make significant imporvements in program performance by clever coding.

The purpose of most coding tricks is to produce faster programs. Since a typical program spends most of its time executing loops, speeding up critical loops can improve a program's performance more than any other coding changes. In short loops, the speed improvement obtained by eliminating just a few instructions can be substantial.

### 2.2.4 TESTING AND DEBUGGING

The purpose of testing and debugging is to make a program meet its speciticationS. Testing is an activity that detects the existance of errors in a program. Debugging finds the

causes of detected errors and then repairs them. As shown in the pie-chart in figure-2.1, testing and debugging form the largest single component in the program development process.

Even after starting with a good design, many programmers have a haphazard approach to the remainder of program development. They code the entire program and then run it the first time with their fingers crossed. A sensible method for developing a large program is code, test, and debug it in small chunks.

One of the two approaches may be used:

* Bottom-up development: The lowest level modules are coded, tested and debugged first. These modules which are now known to be working, may be employed in developing higher level modules.

* Top-down development: The highest-level modules are coded, tested, and debugged first. In order to test them, the lower-level modules which have not yet been coded must be replaced by "stubs" that match their input/ output specification but with much less functionality.

There come advantages and disadvantages to both approaches. In bottom-up development, fundamental errors in design at top-level modules may not be caught until late in the project. In top-down development, problem with program site or performance may not become apparent until critical low-level modules are developed. Both approaches require

additional code to be written for testing. In practice, it is often best to use a combination of the two approaches, developing both high-level and critical low-level modules first and using stubs for less critical modules to be developed later.

In large programming projects the need to partition the testing and debugging problem is well recognized. About half of the total testing and debugging effort is devoted to ensuring that individual modules meet their internal specification, this activity is sometimes called "unit testing". The remaining effort is spent on "system integration and test", in which the modules are linked together and the external specification of the program are checked.

### 2.2.5 MAINTENANCE

In the "real" world, programs are written for and used by customers who expect certain degree of performance. Throughout the lifetime of any nontrivial program, lurking bugs will pop up, and at the same time customers will request new features and improvements. Thus, the program's code and even its specification and design may undergo frequent change. Program maintenance requires a large and sophisticated development system.

## 2.3 ALGORITHMIC REPRENSENTATION (2,3)

The notation for algorithms used here can be described with the aid of examples. Consider the following algorithm for determining the largest algebraic elements of a vector.

Algorithm MAX

This algorithm finds the largest algebraic element of vector A which contains n elements and places the result in MAX. i is used as a subscript to A.

```
1. [ Is the vector empty ? ]
   If n < 1 then print message and exit.

2. [ Initialize ]
   Set MAX <-- A[1];i <-- 2;   ( we assume initially that
   A[i] is the greatest element ).

3. [ All done ? ]
   Repeat steps 4 and 5 while i <= n;

4. [ Exchange MAX if it is smaller than next element ]
   If MAX < A[i] then set MAX <-- A[i];

5. [ Get next subscript ]
   Set i <-- i + 1;

6. [ Finished ]
   Exit.
```

The algorithm is given an identifying name MAX which is followed by a brief description of the tasks the algorithm perform, thus providing an identification for the variables used in the algorithm. This description is followed by the actual algorithm - a sequence of numbered steps.

Every algorithm step begins with a phrase enclosed in square braket which gives an abbreviated description of that step. Following this phrase is an ordered sequence of statements which describe actions to be executed or tests to be performed. In general, the statements in each step must be executed from left to right in order. An algorithm step may terminate with a comment enclosed in parantheses that is intended to define the step clearly. The comments specify no action and are included only for clarity.

Step2 in the example algorithm contains the arrow symbol "<-- " which is used to denote the assignment operator. The statement MAX <-- A[i] is taken to mean that the value of the vector element A[i] is to replace the content of the variable MAX. In this algorithmic notation, the symbol "=" is used as relational operator and never as an assignment operator . One assignment statement , or a group of several assignment statements seperated by commas, is preceded by the word " set ". The action of incrementing i by one in step5 is indicated by i <-- i + 1. Many variables can be set by using multiple assignments. The statement i<-- 0, j <-- 0 and k <-- 0 could be rewritten as the single statement i <-- j <-- k <-- 0. An exchange of the values of two variables can be written as A[i] <--> A[i+1]. And the subscripts for arrays can be written as A[i], where i is the index of the array A.

The execution of an algorithm begins at step1 and cotinues from there in sequential order unless the result of a condition tested or an unconditional transfer ( a "goto") specifies otherwise. In the same sample algorithm begins at

step1 and is first executed. If vector A is empty, the algorithm terminates; otherwise , step2 is performed in which MAX is initialized to the value of A[1] and the subscript variable, i, is set to 2. Step3 leads to the termination of the algorithm if we have already tested the last element A. Otherwise, step4 is taken. In this step, the value of MAX is compared with the value of the next element of the vector. If MAX is less than next element, then MAX is set to this new value. If the test fails, no reassignment takes place. The completion of step4 is followed by step5, where the next subscript value is set; control then returns to the testing step, step3.

## 2.4 IMPLEMENTATION FEATURES

Since in this application we will be mainly concerned with string data processing, it is important to incorporate into the algorithmic notation certain features that facilitate the processing of string information. These features are provided as an addition to the standrad mathematical functions and operations which one would expect to find useful in this application. The algorithms are written in a structured high level programming style . Consequently, it is important to pattern the string data processing features after certain high level language operations or functions. Some of these features are described in the following section.

A close analysis of the basic string handling facilities required of any text creation and editing system should lead to the following list of conventional functions:

1. Create a string of text,
2. Concatenate two or more strings to form another string,
3. Search and replace ( if desired ) a given substring within a string,
4. Test for the identity of a string,
5. Compute the length of a string and
6. Convert to numeric value from a string.

The creation of a string implies not only the ability to construct a representation for a string, but also the ability to retain the value of a string in a variable ( or memory cell location). The ability to create a string must be present in any string handling system. In the algorithmic notation, a string is expressed as any sequence of characters enclosed in single quote marks. To provide a transparent representation for strings, a single quote contained within a string is represented by two single quotes. Variables can be used to retain string values.

The empty or null string is denoted by either two single quotes or the symbol NUL.

Concatenation is the most important operation on a string. For easy and consistent representation , we use + to denote cancatenation in our algorithmic notation. String variables as well as string constants can appear as operands.

When searching for a substring within a given string, there must be some method of returning the position of the substring within the string, if the substring is found. This position is often called the cursor position, and it is given by an integer value indicating the character position of the leftmost character of the substring being sought. The

name of the function used in algorithmic notation to perform this operation is INDEX. INDEX(PATTERN, SUBJECT, CURSOR) returns (as a value) the cursor position of the leftmost occurence of the string, PATTERN, in the string SUBJECT searching after the character given by the argument CURSOR. If PATTERN does not occur in SUBJECT, the value zero is returned. The string associated with PATTERN is applied to the SUBJECT string on a character by character basis from the first character to the last character. This process of applying a PATTERN string to a SUBJECT string is commonly called pattern matching. A wide varity of pattern matching operation exist, and many of these are used in our algorithmic notation.

The ability to extract a substring from a subject string is another important function. In the algorithmic notation, rather than using special marker symbols , we use cursor position plus substring length to isolate a substring. The name given to this function is SUB with arguments SUBJECT, CURSOR and LEN. Thus SUB(SUBJECT, CURSOR, LEN) returns as a value the substring SUBJECT that is specified by the parameters CURSOR and LEN. The parameter CURSOR indicates the starting cursor position of the substring, while LEN specifies the length of the required substring plus the cursor position. If LEN is not provided, then LEN is assumed to be equel to LENGTH, where LENGTH is the length of the argument SUBJECT. To complete a definition of SUB, some additional cases must be handled:

1. If LEN <= 0 (regardless of CURSOR), then null string is returned.
2. If CURSOR <=0 (regardless of LEN), then null string is returned.
3. If CURSOR > LENGTH (regardless of LEN),then null string is returned.
4. If LEN > LENGTH , then LEN is assumed to be LENGTH.

The function SUB can also be used on the left hand side of an assignment (i.e., in a replacement mode of operation). If SUB (SUBJECT,CURSOR,LEN) appears on the left hand side of an assignment statement and CURSOR <= zero or LEN <= zero, then assignment is not executed. If CURSOR <= LENGTH and LEN > LENGTH , then characters are assigned to positions beyond the right hand end of the SUBJECT string. Intermediate character positions which are assigned as a set of blank cahracters.

Testing the identity of a string implies the existance of some form of predicate which returns a true or false value when a comparison is made between a subject string and a known string. In the algorithmic notation we used all the common relational symbols (such as >, <, <>, <= and => ). Comparisons are made on a character by character basis starting from the left most character of each string in the comparison. The presence of any character (even a blank) is always considered to be greater than the omission of a character.

The length of a string is important for checking the character string and in the formatting of the string. In the

algorithmic notation, the computation of the length of a string is achieved by the function LENGHT. If SUBJECT is a character variable then LENGTH(SUBJECT) returns as a value, the number of symbols are in the string represented by SUBJECT. The value zero is returned if the SUBJECT is the empty string.

The value of a string is important for the conversion of a character string to a numeric constant. In the algorithmic notation, the computation of the value of string is performed by the function VALUE with the SUBJECT as an argument. In this function SUBJECT must be a valid character set for a numeric constant. For invalid syntax for a number within the SUBJECT string, a value of zero is returned.

# CHAPTER 3

# SYSTEM ANALYSIS

## 3.1 INTRODUCTION

The purpose of the system to be developed is to manipulate data on a large array system using simple but powerful commands. These commands may be classified to particular works or categories. These categories may be specified according to the following purposes:

- editing the informations,
- formatting the informations,
- displaying the informations,
- calculating the formulas,
- storing the informations,
- printing the informations etc.

## 3.2 EXTERNAL SPECIFICATIONS

In order to develop the desired system the following informations are important to consider for the requirement analysis and specifications:

* Production of a two-dimensional rectangular grid containing a large number of programmable information cells.

* Interaction intelligently with the user through CRT display and hard-copy- printer.

* Interaction of powerful but easy-to-use system commands.

* Calculation of powerful logical, conditional and built-in mathematical functions.

### 3.2.1 NONCOMPUTATIONAL FUNCTIONS

In order to facilitate the production of a two dimentional retangular grid containing a large number of programmable information cells,we can consider the following points.

The location of an information cell may be specified by the vertical and horizontal grid. Each cell can contain several types of information such as:

(a) a cell content,
(b) a cell value,
(b) a display format, and
(c) a formula reference.

We can define the cell content as the basic data that a cell contains. A cell may be empty, contain text, repeating text or a formula. Initially all cell can be defined as empty cell.

A formula is a mathematical expression that calculates to a numerical value. It consists of numerical constants, cell references and function references connected by operators. When a formula is entered into a cell, the value must be calculated and displayed.

The value of a cell is the result obtained by evaluating the content of the cell. All cells can have a value. An empty cell can be defined as zero numeric value. Similarly a text string can have a numeric value of zero. A formula may have a number data, textual, not available or value.

We can propagate cell values and types. This means that the cell value may be referenced by a formula in another cell. It is remarkable that such reference is to the value of the original cell, but not to its content (formula). So the cell content can not be referenced by other cells, but it may be replicated or copied.

Since the cell content and the cell value, both are important for the user, so we must have a option to designate the display format for the cell values or for the cell contents. The format option can be displayed on the status line for cells formatted at the entry level. Altering the display format in no way alters the contents or the value, only the way it is displayed on the console or printed on the printer. By formatting a cell or range, one may tell how the cell value to look on the screen. One can specify the format for an individual cell, a group of cells, rows, columns, or the whole cell system. A text string can be set left justified or right justified. We can set the default setting left justified for text and repeating

text and right justified for numeric value. When the text is longer than the column width and is left justified then it displays the number of characters equal to the width of the cell. A repeating text can display from the active cell on word to the right until it reaches a non-empty cell or end of the column range.

In order to facilitate the ability to interact intelligently with the user through CRT display and hard-copy printer, the following points are important to take into consideration:

The operation of the system can be divided into three distinct modes:

(i) Grid display mode,
(ii) Data entry mode, and
(iii) Command mode.

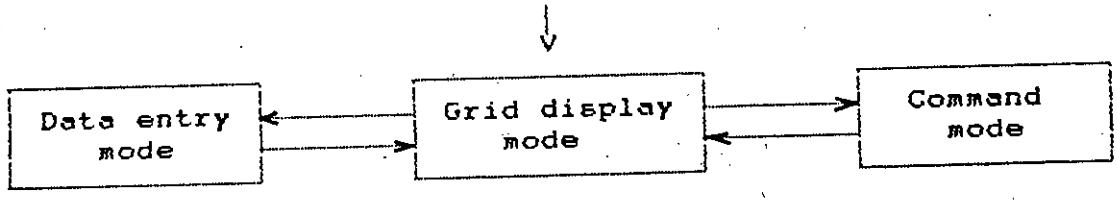


Figure-3.2

Operation modes

Grid display mode is the initial mode of operation. So we must have a option to move from this mode to data entry mode or to command mode. Direct switching from data entry mode to

command mode or vice-versa, is not necessary because grid display mode always shows the present condition. Hence any one can move to data entry mode or command mode from initial grid display mode. But control can not be switched between the data entry mode and command mode without returning to the grid display mode.

In the grid display mode the active cell cursor is active and the edit cursor is inactive. The active cell cursor can be moved over the entire cell system to view the cell contents or values. The status lines are for displaying the followings:

- The active cell status,
- The global status, and
- The edit cursor status.

In data entry mode data can be entered directly into the data entry line. A return key enters the data in the data entry line into the active cell which is indicated by the active cell cursor. In this mode a text, repeating text or a formula can be entered into a particular cell. Any printable character on the keyboard may be used as text. Printable characters usually begins with the ASCII 32 to 127. A repeating text also may contain any printable character. When a cell in formatted to right justification then the text does not repeat.

A formula begin with any of the followings:

- A numeric constant
- A cell reference - cell coordinates
- A mathematical function
- A Boolean function

A formula entry must be always checked whether the entry constitutes a legitimate formula or not. When it is not, an "ERROR" message is to be displayed at the status line.

In order to facilitate the interaction between a powerful but easy-to-use system commands and the user, we can consider the following points :

Command mode must direct the system to perform an action over the system. Before performing any action we must switch to the command mode from the grid display mode. Then perform the particular function according to the type of action. In the command mode we can categorize the action as follows:

# Recalculation of formulas,
# Switch to a particular cell directly,
# Switch display window,
# Helping menu display, and
# Performing all other functions.

From the select mode we can switch to perform a particular action as above by inserting a particular code. These code can be any ASCII character code available . Thus we can use a character code(!) for recalculation of all the formulas. Similarly a character code(=) can be used to

switch to a particular cell directly. A character code of "!" can be used to switch from active to alternate window in the split screen mode. Another character code can be used such as question(?) mark to switch to the helping menu and a slash (/) character to switch to a meta command stage where other command functions are available for further action. Thus in this case we can switch to one of the five command levels by inserting one of the five command characters or by pressing one of the five command keys in the grid display select mode.

The recalculation command must be capable of performing recalculation of the entire information cell system interrelating with formulas which specify mathematical and logical calculations.

Switching to a particular cell directly can be mentioned as GOTO command. This command moves the active cell cursor directly to the cell specified. The active cell cursor moves to the cell specified if it is currently being displayed on the console screen. If not within the display window , the specified cell becomes the upper left cell of the display window. The command without any cell specification shifts the display window to put the active cell in the upper left.

Helping menu display shows all information related to the system operation. Thus this feature help user when he needs help. The system explains on screen his current options, then with a touch of any key, returns him to where he was to continue his work.

The slash command performs all other functions. We can make an option that the system prompt with the first letter of each command, when the user enter the slash character. This is very useful because a long list of commands is difficult to remember. Now when entering the first letter the system can immediately fill the rest of the word on the command line.

By pressing the slash key, the following things may happen immediately :

The bottom status line enters command mode. The position number of the edit cursor display first, the "/" character.

The middle status line changes from global display mode to prompt mode. The slash command prompt may display the following line,

Enter B,C,F,G,L,M,P,Q,S,W,X,Z.

In command mode, the edit cursor becomes active and the active cell cursor inactive. Most commands have several entry levels. When entering a command letter the prompt line changes to the appropiate prompt. The system continues to prompt through the sequence of options until execution of the command.

If one mistakes to press the proper key he can back to the previous level by pressing BACK SPACE key. Thus one can edit commands, like data and formulas, with the in-line editor.

All possible slash commands are shown below:

| | | | |
|---|---|---|---|
| Blank | Global | Print | Window |
| Copy | Load | Quit | Xchange |
| Format | Move | Save | Zap |

### 3.2.2 COMPUTATIONAL FUNCTIONS

In order to evaluate the powerful logical, conditional and built-in-mathematical functions and expressions , an expression interpreter is needed to be developed. For the development of the interpreter we must first specify the functions available for the system and their type of works.

A function returns the value of a calculation. Two types of function can be classified.

(1) Arithmetic
(2) Boolean.

Each function has a name and one or more arguments. The arguments specify the values that one wants to apply to the function. An expression may be used to produce a value. The expression is evaluated first and the value is used as the argument of the function.

## ARITHMETIC FUNCTIONS

The following functions have the arguments which may consist of a value, a range or a list.

Value- An expression evaluating to a numeric value. It may be integer or real.

Range- A group of cells specified by naming the top leftmost cell and bottom rightmost cell, seperated by a delimiter.

List- One or more ranges and values seperated by delimiters.

ABS(value)
Return the absolute value of the given value.

- Equivalent to the value itself if positive
- Equivalent to the value without its sign if negative. This is the additive inverse.
- Equivalent to zero if the expression is zero.

ACS(value)
Returns the radian angle of the cosine value.

ASN(value)
Returns the radian angle of the sine value.

ATN(value)
Returns the radian angle of the tangent value.

AVG(list)
Returns the average(mean) of the range given. The function is equivalent to the SUM of the list divided by the CNT of the list.

COS(value)
Returns the cosine of the radian angle value given.

CNT(list)
Returns the number of the non-blank non-text cells described by the range.

EXP(value)
This function raises the number e exponentially to the value. The value of e is 2.7182818

INT(value)
Returns the integer of the value given, the value is not rounded.

LNE(value)
Returns the natural log, log base e, of the value given.

LOG(value)
Returns the commom log, log base 10, of the value given.

MAX(list)
Retuns the maximum value of the range. Non numeric cells are ignored.

MIN(list)
Returns the minimum value of the range. Non numeric cells are ignored.

RND(value,place-value)
The RND function rounds a value to a specified number of places.

SIN(value)
Returns the sine of the radian angle value given.

SQT(value)
Returns the square root of the value given.

SUM(list)
Returns the sum of the values in the range. Non numeric cells are ignored.

TAN(value)
Returns the tangent of the radian angle value given.

## BOOLEAN FUNCTIONS

A conditional or logical function consists of a relational comparison connected by a logical operator. Complex logical expressions may be formed by using parantheses. Here all the logical expressions used as an argument can be replaced by a cell reference which has a value of logical true(1) or logical false(0).

IF(expression,value1,value2)

If the logical expression is true, then the value1 is entered into the active cell. Otherwise if the logical expression is false, then value2 is entered. If an expression is entered into an IF function, the expression must evaluate properly to a logical value in order for the IF function valid. Otherwise it will cause a formula error.

IF (expressionA,expressionB,expressionC)

Here expressionA is a logical expression and expressionB & expressionC are the arithmetic expression. In this case if expressionA is true then expressionB is evaluated and the result is entered into the active cell. Otherwise if expressionA is false then expressionC is evaluated and the result is stored into the active cell.

AND(expression1,expression2)

A logical AND function has a value of true(numerical value of 1) if both the expressions are true. If either expression

is false, the AND function is false (numerical value of 0). Here both the expressions must be logical expression containing logical or relational operators.

OR(expression1,expression2)

A logical OR function has a value of true if either expression1 or expression2 is true. If both expressions are false, then OR function returns false. Here both the expressions must be logical in nature i.e., containing logical and relational operators only.

NOT(expression)

This function returns the opposite logic values as the expression stated. Here also expression must be logical expression.

EQU(expression1,expression2)

A logical EQU function has a value of logical false if both the expressions are in same logic. Otherwise a value of logical true is returned. Here also both the expression are of logical type.

NEQ(expression1,expression2)

A logical NEQ function has a value of logical true if both the expressions are in same logic. Otherwise a value of logical false is returned. Here also both the expressions are of logical type.

# CHAPTER 4

# SYSTEM DESIGN

## 4.1 INTRODUCTION

The design of the cell system can take place after detail investigation and analysis of the external specification of the system. Thus the detailed design has been carried out after considering all the system requirements and then maintaining enough scope to incorporate the future changes with minimum efforts. All the external specifications of the system design has been considered directly from the result of the requirement analysis and the internal specification has been considered later in the design stage. The linking between the specification and design has been maintained frequently to minimize the ambiguities, contradiction, or other deficiencies in the external specification. But most of the linking occured here because of the "top-down' approach that is used here in the system design.

Initially a solution to an original external specification is designed as a collection of cooperating modules and each module and its interfaces with its partners are documented by internal specifications. Then the working of each module has been designed. The whole process is repeated many times since the modules at each level are designed as collections of lower level modules.

## 4.2 STRUCTURE OF THE SYSTEM

The partitioning of a program into modules can be illustrated and documented in a diagram that shows relationships among

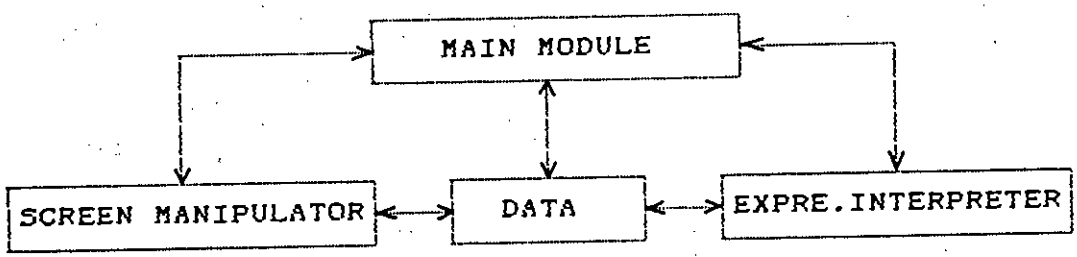


Figure-4.3

Basic structure of the system

the modules. The basic structure of the system, as shown in the figure-4. 3 , is designed after considering all the external specifications. The main module performs interaction with the supporting modules when required. The supporting modules perform specific work and according to their nature of work they have the name as ;

1) Expression interpreter and
2) Screen manipulator.

In the expression interpreter a number of algorithms have been developed for the evaluation of the expressions such as arithmetic expressions with built-in functions, logical expressions, conditional expressions etc.

In the screen manipulator a number of algorithms have been developed for intelligent interaction with the user through the CRT display screen.

In the main module, algorithms for the production of the cell system, and the algorithms for the system command functions have been defined. In the main module, an error routine has also been described for trapping the errors and for displaying the error messages.

The specifications for the interaction between the main module and the other two subordinate modules have been defined first, and then the detailed designed has been carried out more or less independently. At the stage of the detailed design it is convenient to outline algorithms using an informal block-structured language that allows detailed description of action. In this system design data structures are also part of the system and hence it is important to specify exactly when and how the supporting modules access the data from the structure. The detailed of these procedures have been discussed later in different corresponding chapters.

# CHAPTER 5

# EXPRESSION INTERPRETATION

## 5.1 INTRODUCTION

The developed expression interpreter accepts a statement string as an input and then analyses the string and finally evaluates to a value. Expressions are similar to expressions in most programming languages. Expression strings can be represented by a valid arithmetic expression or a logical expression or a conditional expression or a mixed expression. The ingredients are the identifiers, numbers, operators, delimilers and reserved words.

Two basic type of expressions are taken into consideration for the development of the interpreter, they are:

1. Arithmetic expressions perform arithmetic operations on the operands in the expression and

2. Boolean expressions perform logical and comparison operations with boolean results.

### Arithmetic Expressions

The ingredients of the arithmetic expressions are the binary operators, unary operators and the identifiers and numbers

as the operands.

The precedence rule used here may be stated in the way:

When unaltered by parantheses, the order of arithmetic operation performed within one expression is in descending order of precedence.

The arithmetic operations are listed in descending order of precedence as follows:

| Symbol | Operation |
|---|---|
| ^ | Exponentiation (to the power of) |
| *,/ | Multiplication, division |
| +,- | Addition, substruction |

For our system, as in most other high-level languages, exponentiation operator is right associative and other binary operators are left associative.

As employed in the usual algebraic sense, parantheses may be introduced to override the usual rules of precedence for a given expression. In general, when we enclose a portion of an expression in parentheses, we are in effect forming a 'subexpression'. The parantheses rule then may be stated as: " Any subexpression must be evaluated before it can be employed in the rest of the expression ". Within the subexpression the same precedence rule and associative rule has been applied. The use of nested parantheses, suggesting that one subexpression may form part of another

subexpression. Each subexpression is evaluated in sequence, from the inside out, according to the same precedence and associative rule.

Boolean Expressions

The ingredient of the boolean expressions are the boolean operator such as AND, OR, NOT and one or two relational operator and operands. Here relational operator produces a boolean result. A boolean value is either a true or a false. In this system , the logical negation operator returns the complement of a boolean operand. Here we define four binary operations on boolean operands, as shown in the table-5.1

TABLE 5.1 BOOLEAN OPERATORS

| X | Y | NOT(X) | AND(X,Y) | OR(X,Y) | EQU(X,Y) | NEQ(X,Y) |
|---|---|---|---|---|---|---|
| false | false | true | false | false | false | true |
| false | true | true | false | true | true | false |
| true | false | false | false | true | true | false |
| true | true | false | true | true | false | true |

When operands of other types are compared by relational operators, the result of the comparison is a value of type boolean. A relational expression compares two arguments of like type using one of the relational operators: = (equal), < (less than), > (greater than), <= or =< (less than or equal), >= or => (greater than or equal) and <> or >< (not equal).

Functions

An important aspect of this system is the case with which one may write expressions refering to functions of one or more variables. The common mathematical functions such as logarithm, exponential, sine, cosine, sqaure root etc., are specialy easy to incorporate in arithmetic expressions. In the computing sense, a function is typically a separate but subordinate program designed to perform a specific task.It is given certain key information called the argument(s) of the function. Certain commonly used functions are predefined and automatically available in this system.

Implicit in the evaluation of these expressions is an additional rule governing the order of computation which states that function evaluation takes precedence over binary arithmetic operations. As a consequence, the evaluation process proceeds as follows:

. First, argument expression, which are parenthesized subexpressions,
. Second, the function, and
. Third, all the other arithmetic operation.

## 5.2 THE STRUCTURE OF THE INTERPRETER

The developed interpreter takes as input a source statement and produces as output a sequence of meaningful codes. This process is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the interpretation process as occuring in

one single step. For this reason, the interpretation process is subdivided into a series of subproceses, called phases. Each phase is the logically cohesive operations that takes as input one representation of the source and produces as output another representation.

The structural diagram of the designed interpreter is shown below:

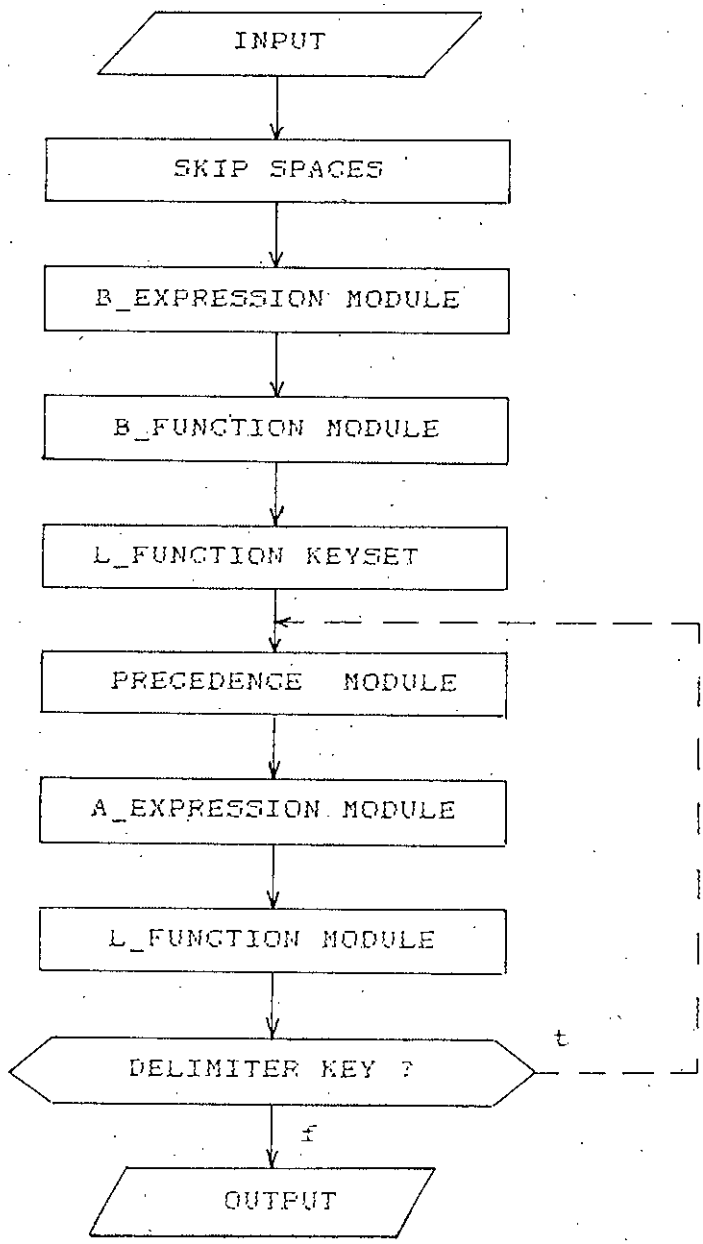


Figure-5.2 STRUCTURE OF THE EXPRESSION INTERPRETER

(2,4)

## 5.3 GRAMMATICAL DESCRIPTION

The grammatical description for the expression interpreter language is given below:

```
< expression > ::= <a_expression >
                  ¦<b_expression >

<a_expression> ::= <term>
                  ¦<a_expression><add/sub.op><term>

<term> ::= <form>
          ¦<term><mult/div.op><form>

<form> ::= <word>
          ¦<form><expont.op><word>

<word> ::= <primary>
          ¦<u_operator><primary>

<primary> ::= <identifier>
             ¦<numeric>
             ¦<b_function>
             ¦<l_function>
             ¦<(><a_expression><)>

<identifier> ::=<name><digit>

<name> ::= <letter>
          ¦<name><letter>
          ¦<name><digit>

<letter> ::= A¦B¦C¦D¦E¦F¦G¦H¦I¦J¦K¦L¦M¦N¦O¦P¦Q¦R¦S¦T¦U
            ¦V¦W¦X¦Y¦Z

<numeric> ::= <digit_string>
             ¦<decimal><digit_string>
             ¦<digit_string><decimal><digit_string>
             ¦<digit_string><e_field><digit_string>
             ¦<digit_string><e_field><digit_string>

<digit_string> ::= <digit>
                  ¦<digit_string><digit>

<digit> ::= 0¦1¦2¦3¦4¦5¦6¦7¦8¦9
```

```
<add/sub.op> ::= -¦+

<mult/div.op> ::= *¦/

<expont.op> ::= ^

<u_operator> ::= +¦-

<e_field> ::= E¦E+¦E-

<decimal> ::= .


<b_expression> ::= <l_identifier><(><logical_term><)>

<l_identifier> ::= IF¦NOT¦OR¦AND¦EQU¦NEQ

<logical_term> ::= <logical_part>
                  ¦<logical_term><,><logical part>
                  ¦<logical_term><,><a_expression>

<logical_part> ::= <a_expression><relation><a_expression>

<relation> ::= =¦<¦>¦<>¦<=¦>=


<b_function> ::= <b_identifier><(><argument><)>

<b_identifier> ::= MAX¦MIN¦SUM¦AVG¦CNT

<argument> ::= <range>
               ¦<argument><;><identifier>
               ¦<argument><;><range>

<range> ::= <identifier><:><identifier>


<l_function> ::= <lib_identifier><(><a_expression><)>

<lib_identifier> ::= SIN¦COS¦TAN¦SQR¦.....

<(> ::= (

<)> ::= )

<;> ::= ;

<:> ::= :
```

Algorithm EXPR_TER

The input parameter of this algorithm is the expression string, the grammatical description of the expression is already defined. The algorithms belong to this interpreter are REBK where blanks are removed from the source string , B_EXPRESSION where boolean expressions are interpreted , B_FUNCTION where built-in functions are interpreted , L_FUNCTION where library functions are recognized , BRAKET where subexpressions are recognized, A_EXPRESSION where arithmetic expressions are interpreted, LIB_EVA where library functions are evaluated etc.

The input parameter of this algorithm is the SOURCE string. This SOURCE expression string is first passed to the algorithm REBK where blank characters (if any) within the string are removed and this is done because the lexical analyzer can not recognize any blank character. Then the string is scanned, looking for a boolean function keyword. If there have any boolean function keyword then the string is treated as a boolean expression string. Hence it passes to the boolean expression interpreter for evaluation to a value. If no boolean function keyword is found then the source string is scanned, looking for a built-in function keyword.If there have any function reference then the source string is passed to the built-in function algorithm where each function is evaluated to a value according to the type of the function referenced and stored in temporary variables. Then the string is scanned, searching for a library function keyword. If there have any library function

reference then the string is passed to the library function algorithm where function keys are set to proper codes according to the type of the functions available. Then the string passes to the subexpression recognition algorithm after identifying any parantheses. Then the subexpression or the expression is passed to the arithmetic expression algorithm. Finally by proper key checking and setting it branches to the various parts of the algorithm or exit from the algorithm.

```
1.[Initialize the logic function & precedence delimiter key]
    Set PKEY <-- LKEY <-- false;

2.[ Remove the blanks from the source string if any]
    Call REBK(SOURCE);

3.[ Call the boolean expression procedure ]
    Call B_EXPRESSION(SOURCE);

4.[ Call the built-in function procedure ]
    Call B_FUNCTION(SOURCE);

5.[ Call the library function procedure ]
    Call L_FUNCTION(SOURCE);

6.[ Check the precedence function & set the key ]
    If INDEX(LPR,SOURCE,1) <> 0 then
    call BRAKET(SOURCE); and goto step-7;
    otherwise set PKEY <-- true;

7.[ Call the arithmetic expression procedure ]
    Call A_EXPRESSION(DSTRING);

8.[ Check the library function key ]
    If FSWITCH = true then call LIB_EVA;
```

```
9.[ Check the precedence delimiter key ]
    If PKEY = false then repeat from step-6;

10.[ Check the logic function key ]
     If LKEY = true then call B_EXPRESSION(RESULT);
     If LKEY = true then repeat from step-4;

11.[ Finished ]
     Exit.
```

## 5.4 ARITHMETIC EXPRESSION ALGORITHMS

This algorithm takes a source statement string as input and produces a numeric value as the output. This process is partitioned into a series of subprocesses called phases as shown in the figure-5.4a. A phase is a logically cohesive operation that takes as input one representation of the source and produces as output another representation.

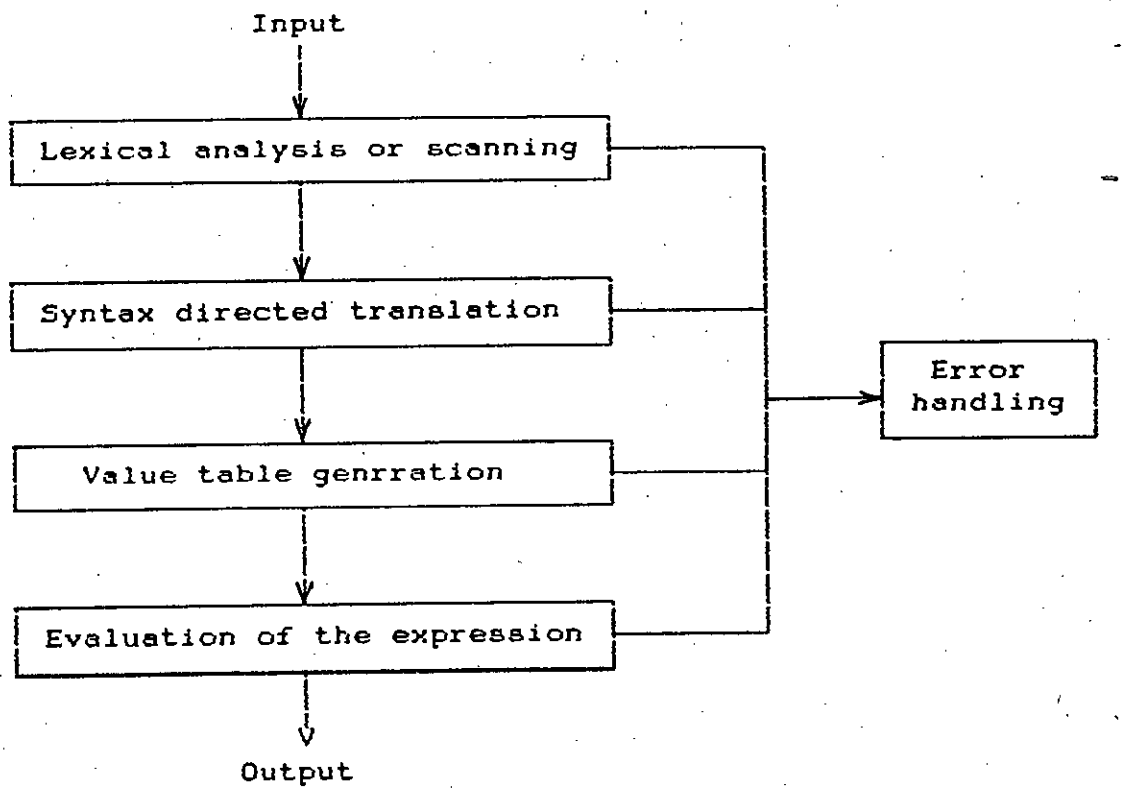


Figure-5.4
Structural block diagram of the interpreter

The first phase, called the lexical analysis, or scanning seperates characters of the source string into groups that logically belong together, these groups are called tokens. The usual tokens are identifiers, operator symbols, numeric constants, reserved words etc. The output of the lexical analyzer is a stream of tokens, which is passed to the next phase, the syntax directed translation or intermediate code generation. This phase decides the hierarchical structure of the incoming token stream by identifying which parts of the token stream should be grouped together. For examlpe, the expression A/B*C has two possible interpretation rule:

a) divide A by B and then multiply by C (as in FORTRAN); or
b) multiply B by C and then use the result to divide A (as in APL)

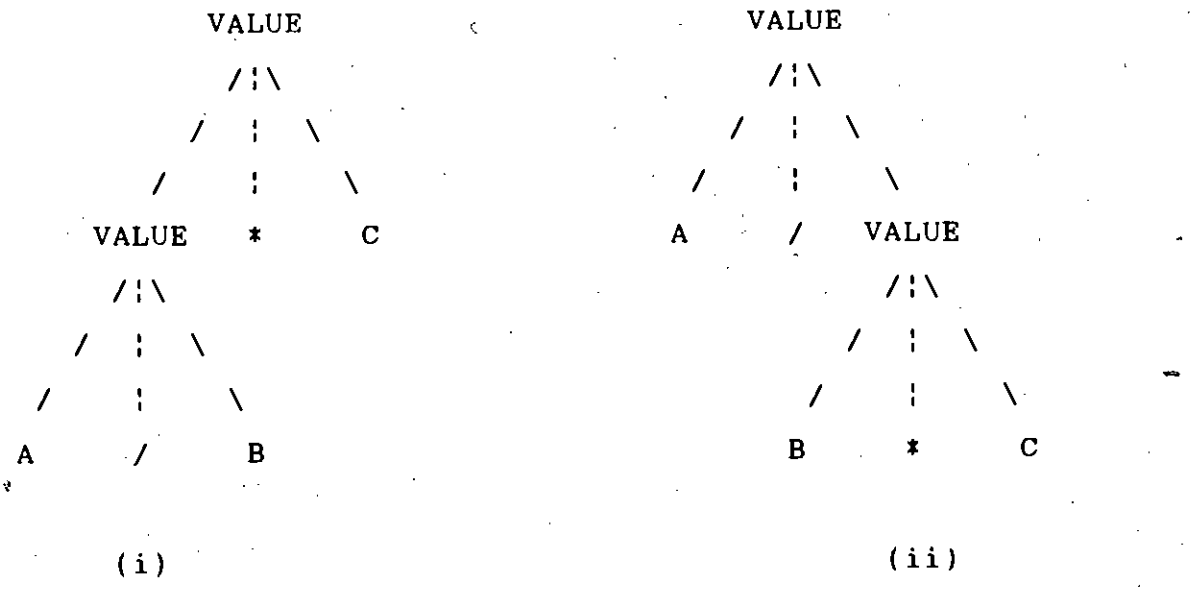


Figure-5.4b Parse trees

These rules form the internal specification of the interpreter. On a logical level the output of the syntax analyzer is some representation of a parse tree. It then transforms this parse tree into an intermediate language representation of the source string. "Three address code" can be used for the intermediate code generation. Consider, A = B op C, as a statement of the three address code, where A, B and C are operands and op is a binary operator. The parse tree in fig-5.4b(i) might be converted into three address code sequence:

```
T1 = A / B
T2 = T1 * C
```

where, T1 and T2 are names of temporary variables.

Thus the usual intermediate text introduces symbols to stand for various temporary quantities such as the value of B*C in the source language expression A+B*C.

Value table generation is the phase where all identifier tokens are replaced by their numerical values. Thus the output of this phase, a stream of constant numbers and operator tokens arranged in a notation meaningful for the interpreter.

Finally, the evaluation of the expression string, phase reduces these stream by considering three address code sequence and return a single value which is the value of the expression.

One of the most important function of an interpreter is the detection and reporting of errors in the source string or execution time. The error message identifies the exact type and position of the error.

The words or lexical components for the input string are identifiers, numeric constants, positive and negative unary operators, and all the binary operators. The syntax of these lexical classes is given by the mathematical description:

```
<artih_expression> ::= <term>
                      ¦<arith_expression><add/sub.op><term>

<term> ::= <form>
          ¦<term><mult/div.op><form>

<form> ::= <word>
          ¦<form><expont.op><word>

<word> ::= <primary>
          ¦<u_operator><primary>

<primary> ::= <identifier>
             ¦<numeric>

<identifier> ::= <name><digit>

<name> ::= <letter>
          ¦<name><letter>
          ¦<name><digit>

<letter> ::= A¦B¦C¦D¦E¦F¦G¦H¦I¦J¦K¦L¦M¦N¦O¦P¦Q¦R¦S¦T¦U¦V¦W¦X
            ¦Y¦Z

<numeric> ::= <digit string>
             ¦<decimal digit string>
             ¦<digit string><decimal><digit string>
             ¦<numeric><e_field><digit string>

<digit string> ::= <digit>
                  ¦<digit string><digit>

<digit> ::= 0¦1¦2¦3¦4¦5¦6¦7¦8¦9

<add/sub.op> ::= -¦+

<mult/div.op> ::= *¦/

<expont.op> ::= ^

<u_operator> ::= +¦-

<e_field> ::= E¦E+¦E-

<decimal> ::= .
```

Algorithm A_EXPRESSION

The input parameter of this algorithm is the source string assigned as SOURCE. This SOURCE string is analyzed by the algorithm named SCAN and produces a token table named TABLE. This TABLE is passed to the next phase defined by an algorithm SYNTAX where these tokens are arranged in a post-fix sequence which are stored in the same table named TABLE. This TABLE is passed to the next phase defined by an algorithm named VTABLE where a table containing value(s) and operator(s) corresponding to the input table is generated. This table is also named as TABLE which finally passed to the next phase defined by an algorithm named as EVALUATE where the table is condensed to a single value and stored in a variable named RESULT.

```
1. [ Call the lexical analyzer ]
     Call SCAN(SOURCE);

2. [ Call the syntax analyzer ]
     Call SYNTAX(TABLE);

3. [ Call the value table generator ]
     Call VTABLE(TABLE);

4. [ Call evaluation procedure ]
     Call EVATUATE(TABLE);

5. [ Finished ]
     Exit.
```

## 5.4.1 LEXICAL ANALYSIS

Source string is the input of the lexical analyzer or scanner and output is the stream of meaningful characters in group which are called tokens. These tokens may contain identifiers called ID-tokens, numeric constants called NUM-tokens or operators called OP-tokens. The following block diagram shows the functional concept of a scanner. In goes a stream of characters and out comes a sequence of character groups as tokens. For example, if the

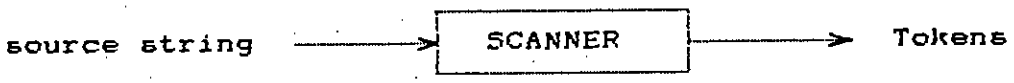


Figure-5.4.1a

source string is A1+B2*20, then the output tokens are A1, +, B2, * and 20, where A1 & B2 are ID-tokens, + & * are OP-tokens and 20 is the NUM-tokens. The functional block diagram of the lexical analyzer or scanner is shown below:

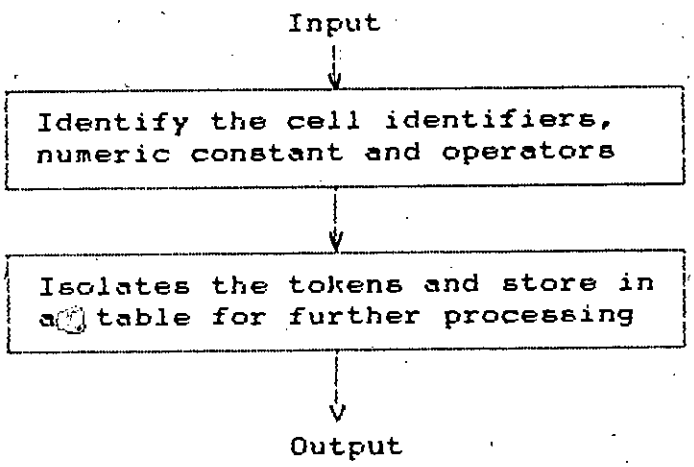


Figure-5.4.1b

Block diagram of the lexical analyzer or scanner

Algorithm SCAN

This algorithm leads a procedure which scans the expression string by operator searching method. The searching pointer scans from left to right of the source language. After identifying an operator it checks the operator whether it is an unary operator or not. If it is not an unary operator then the logic checks the operator whether it belongs to an exponent token or not. If it is not then the operator is treated as a binary operator and the left to this operator is to be considered as one of the two operands of the identified operator token. In the case of an unary operator and exponent token the scanning pointer is advanced to right by one character and the searching process for the next operator is cotinued and the cycle is repeated after the occurence of an operator. In this way the tokens are isolated and stored in a table until the scanning is finished.

The input parameter for this algorithm is the SOURCE. SOURCE is the expression string containing constant number(s) or cell identifier(s) or arithmetic operator(s) tokens.OPERATOR is a string which contains all the arithmetic operators such as +, -, *, / and ^ . TEMPLATE is the temporary buffer where the input string is placed for scanning process. TABLE is an array where tokens are to be stored in a sequence as they are in the expression string. i and j are the character and the table index respectively.

```
1. [ Initialize ]
     Set i  <-- j  <-- 1;

2. [ Assign the input string ]
     Set TEMPLATE <-- SOURCE;

3. [ Assign the length of the buffer string ]
     Set k <-- LENGTH(TEMPLATE);

4. [ Check the input character ]
     If SUB(TEMPLATE,j,1) = OPERATOR then goto step-6;

5. [ Check the character pointer with the length ]
     If j = k then goto step-10;
     otherwise (increment the character pointer)
     Set j <-- j+1  and repeat step-4;

6. [ Check the index for the unary operator ]
     If i = 1 then set j <-- 2; and repeat step-4;

7. [ Check for exponent tokens ]
     If SUB(TEMPLATE,j-1,1) = 'E' then set j <-- j+1; and
     repeat step-4;

8. [ Isolate the tokens ]
     Set TABLE[i] <-- SUB(TEMPLATE,1,j-1);
     Set TABLE[i+1] <-- SUB(TEMPLATE,j,1);
     Set TEMPLATE <-- SUB(TEMPLATE,j+1,k-j);

9. [ Increment & reset the  pointers ]
     Set i <-- i+2;
     Set j <-- 1;  and repeat from step-3;

10. [ Isolate the final token and set the token range ]
      Set TABLE[i] <-- TEMPLATE;
      Set TOKENLEN <-- i;

11. [ Check the validity of tokens ]
      If TABLE[i] = IDTOKEN then goto step-12;
      otherwise if TABLE[i] = NUMTOKEN then goto step-12;
      otherwise if TABLE[i] = OPTOKEN then goto step-12;
      other print error message and goto step-14;

12. [ Check and reset the index ]
      If i = 1 then goto step-13;
      otherwise set i <-- i-1;  and repeat step-11;

13. [ Check the unity of token and set the key ]
      If TOKENLEN = 1 then set SVALUE <-- TRUE;
      otherwise set SVALUE <-- FALSE;

14. [ Finish ]
      Exit.
```

## 5.4.2 SYNTAX DIRECTED TRANSLATION

The input of this phase is the stream of tokens outputted from the lexical analyzer and the output is the stream of tokens arranged according to postfix notation scheme. The ordinary (infix) way of writing the sum of a and b is with the operator at the middle, a+b


Figure-5.4.2a

The postfix or polish postfix notation for the same expression replaces the operator at the right end, as ab+ . In general, if e1 and e2 are any postfix expression, and a is any binary operator, the result of applying a to the values denoted by e1 and e2 is indicated in postfix notation by e1e2a.

Postfix notation can be generalized to k-ary operators for any k >= 1. If k-ary operator a is applied to postfix expression e1,e2,e3,.....ek, then the result is denoted by e1e2e3...eka. If we know the arity of each operator, then we can uniquely decipher any postfix expression by scanning it from either end. For example, consider the postfix string ab+c*. The righthand * says that there are two arguments to its left. Since the next to rightmost symbol is c, a simple operand, we know c must be the second operand of *. Continuing to the left, we encounter the operator +. We now know the subexpression ending in + makes up the first operand of *. Continuing in this way, we deduce that ab+c* is "parsed' as (((a,b)+),c)*.

Algorithm SYNTAX

This algorithm produces a postfix notation token stream by processing the lexical tokens. The method used here to generate a postfix token stream is the swapping between an operator and an operand identified by a pointer scanning the token stream from left to right. The rules followed here can be defined by the following table:

TABLE 5.4.2 PRECEDENCE AND ASSOCIATIVITY OF OPERATORS

| 2nd operator \ 1st operator | - | + | / | * | ^ |
|---|---|---|---|---|---|
| - | > | > | > | > | > |
| + | > | > | > | > | > |
| / | < | < | > | > | > |
| * | < | < | > | > | > |
| ^ | < | < | < | < | < |

> -- swapping between the 1st compared operator and the next available operand.

< -- swapping between the 2nd compared operator and the next available operand.

Given a stream of tokens which is outputted from the algorithm SCAN. These are nothing but the cell identifier tokens, arithmetic operator tokens, constant value tokens

etc. These are stored into a buffer array named SYNA. The length of the array is limited by the number of tokens TOKENLEN. Here OPERATOR is a string containing all the arithmetic operators in sequence.

```
1. [ Initialize ]
   Set i <-- 2 ; j <-- 1; and k <-- 0;

2. [ Exchange the elements ]
   SYNA[i-k] <---> SYNA[i+1];

3. [ Increment the array index and check the range ]
   Set i <-- i + 1;
   If i > RANGE then goto step-7 ;

4. [ Check the operator ]
   Set OP <-- INDEX ( SYNA[i],OPERATOR,1);
   If OP = 0 then repeat step-3;

5. [ Set the operator buffer ]
   Set OPBUFF[j] <-- OP; and j <-- j+1;
   If j=2 then repeat step-4;
   otherwise set j <-- 1;

6. [ Check the precedence rule of operators ]
   If OPBUFF[2] = 1 or 2 then
   set k <-- 0 and repeat step-2;
   otherwise
   if OPBUFF[2] = 5 then
   set k <-- 1; and repeat step-2;
   otherwise
   if OPBUFF[1] = 1 or 2 then
   set k <-- 0 and repeat step-2;
   otherwise set k <-- 0 and repeat step-2;

7. [ Finish ]
   Exit.
```

### 5.4.3 VALUE TABLE GENERATION

The input of this phase is the stream of tokens arranged according to postfix notation scheme outcome from the intermediate code generator and output is a stream of

numeric constants only. In this phase all ID tokens and NUM tokens are replaced by their corresponding numeric constant values. Thus the elements outcame from this phase consisting of only values. And their sequence is maintained according to postfix notation with the help of an index. The following figure illustrates the functional concept of this phase.


Figure-5.4.3a

For example if the input stream has the form B2, 20, *, A1 & + and if ID tokens have the values as B2=5.0 & A1=10.0 then the output table of this phase has the form 5.0, 20 & 10.0. The functional block diagram of the value table generation phase is given below:

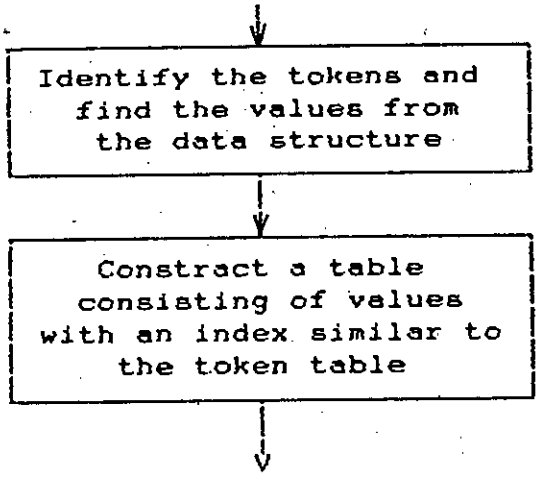


Figure-5.4.3b

Block diagram of the value table generator

Algorithm VTABLE

Given a stream of tokens consisting of identifier tokens, numeric tokens and operator tokens. Sequence of these tokens are identical as these are outcome from the syntax directed translator. These tokens are stored in a table named TABLE with a pointer i. Another table named VTABLE is used here for storing the token values with the same index as they are in TABLE. RANGE is the number represented by the range of the tokens. TEMPLATE is the temporary buffer where tokens are checked for further operation.

```
1. [ Initialize ]
   Set i <-- 1;

2. [ Load the buffer ]
   Set TEMPLATE <-- TABLE [i];

3. [ Check the tokens ]
   If TEMPLATE = NUMTOKEN then
   set VTABLE[i] <-- VALUE(TEMPLATE); and goto step-8;
   otherwise, ( Check for operators)
   if TEMPLATE = OPTOKEN then goto step-8;
   otherwise, ( Check temporary storage values)
   if TEMPLATE = TSTOKEN then
   set VTABLE[i] <-- VALUE(TS); and goto step-8;

4. [ Check for unary operator ]
   If SUB(TEMPLATE,1,1) = OPTOKEN then goto step-5;
   otherwise, (Extract the cell values)
   set [ROW,COLUMN] <-- CELLID(TEMPLATE);
   VTABLE[i] <-- VALUE(CELL[ROW,COLUMN]); and
   goto step-8;

5. [ Isolate the absoluate value ]
   Set TEMPLATE <-- SUB(TEMPLATE,2,LENGTH(TEMPLATE));

6. [ Check the tokens for unary operation ]
   If TEMPLATE = NUMTOKEN then
   set AVALUE <-- VALUE(TEMPLATE);
   otherwise, (Check for temporary storage value)
   if TEMPLATE = TSTOKEN then set AVALUE <-- TS;
   otherwise, (Extract the cell value)
   set [ROW,COLUMN] <-- CELLID(TEMPLATE); and
   AVALUE <-- VALUE(CELL(ROW,COLUMN]);
```

*7. [ Evaluate the values ]*
*If OPTOKEN = PLUS then set VTABLE[i] <-- AVALUE;*
*otherwise set VTABLE[i] <-- -(AVALUE);*

*8. [ Increment and check the token index]*
*Set i <-- i+1; and*
*if i < RANGE then repeat from step-2;*

*9. [ Finish]*
*Exit.*

### 5.4.4 EVALUATION OF THE EXPRESSION

Having generated postfix notations for an expression, we can evaluate it easily using a stack, implemented in software. The general strategy is to scan the postfix code sequence from left to right. We push each operand onto the stack when we see it. If we encounter a k-ary operator, its first (left most) argument will be (k-1) position below the top on the stack, its last argument will be at the top on the stack, and in general, its ith argument is (k-i) position below the top. It is then easy to apply the operator to the top k values on the stack. These values are popped and the result of applying the k-ary operator is pushed onto the stack.

Consider the postfix expression A2B3*C5+. Suppose the identifiers A2, B3 and C5 have the values 5, 3 and 7 respectively. To evaluate 53*7+ we perform the following actions:

1. Stack the value 5
2. Stack the value 3
3. Multiply the two topmost elements, pop them off the stack, and then stack the result, 15

4. Stack the value 7
5. Add the two topmost elements, pop them off the stack, and then stack the result, 22 .

The value on the top of the stack at the end (here 22) is the value of the expression.

Algorithm EVALUATE

Given the values in a table named VTABLE and the tokens in another table named TABLE arranged according to the postfix notation scheme. An index i identifies the elements in the TABLE. Both tables have the same range, RANGE. OPERATOR is the string containing all the binary operators in sequence. k represent another index for the operator string.

```
1. [ Initialize ]
   Set i <-- 1;

2. [ Search for the operator]
   Set k <-- INDEX (TABLE[i],OPERATOR,1);
   If k <> 0 then goto step-4;
   otherwise, (Increment the token index)
   Set i <-- i+1;

3. [ Check the index limit ]
   If i <= RANGE then goto step-2;
   otherwise  print error message; and
   goto step-8 ;

4. [ Perform the operation between the operands ]
   Set VTABLE[i-2] <-- VTABLE[i-2] OP(k) VTABLE[i-1];
```

*5. [ Shift the elements next to the operands ]*
*Set TABLE[i-1] <-- TABLE[i+1]; and*
*VTABLE[i-1] <-- VTABLE[i+1];*

*6. [ Increment and check the index ]*
*Set i <-- i+1;*
*If i < RANGE then repeat from step-5;*

*7. [ Set the new range and check with unity ]*
*Set RANGE <-- (RANGE - 2);*
*If RANGE > 1 then repeat from step-1;*

*8. [ Finish ]*
*Exit.*

## 5.5 BOOLEAN EXPRESSION ALGORITHMS

The interpretation of a boolean expression consists of the following algorithms:

# Boolean expression scanning or lexical analysis
# Value table generation for the tokens
# Evaluation of the value table

The grammatical description of the language for the boolean expression is given below:

```
<b_expression> ::= <lg_identifier><(><logical_term><)>

<lg_identifier> ::= IF¦NOT¦OR¦AND¦EQU¦NEQ

<logical_term> ::= <logical_part>
                  ¦<logical_term><,><logical_part>
                  ¦<logical_term><,><a_expression>

<logical_part> ::= <a_expression><relation><a_expression>

<relation> ::= <¦>¦=¦<>¦>=¦<=
```

Algorithm B EXPRESSION

The input parameter of this algorithm is the expression string assigned as SOURCE. Initially this source string is scanned from left to right, looking for any boolean keyword already defined in a table named LFUN. If no keyword is found in the string then the algorithm exits without any evaluation. If it is a logical expression then the source string is passed through the procedure LCLEX where tokens are isolated and stored in a table named TABLE. Next this table is passed to the next phase called LCVTABLE where a value table is generated . Finally evaluation of the expression occurs in the phase named LCEVALUATE.

```
1. [ Initialize the function keyword table index ]
   Set i <-- 1;

2. [ Check for any boolean keyword ]
   If INDEX(LFUN[i],SOURCE,1) <> 0 then
   set LTYPE <-- i and goto step-3;
   otherwise set i <-- i+1 and repeat step-2 until i>6;
   goto step-7;

3. [ Identify the argument token of the function ]
   Set j <-- INDEX(LPR,SOURCE,1); and
   SOUREC <-- SUB(SOURCE, j+1, LENGTH(SOURCE)-1);

4. [ Call the lexical analyzer ]
   Call LCLEX(SOURCE);

5. [ Call the value table generator ]
   Call LCVTABLE(TABLE);

6. [ Call the evaluation procedure ]
   Call LCEVALUATE(TABLE);

7. [ Finished ]
   Exit.
```

Algorithm LCLEX

This algorithm analyzes the lexical classes of the logical and conditional expression argument string. A class of symbol called delimiters must be handled by this algorithm, and yet their presence is not passed on to the next phase. As described before according to the grammer of the expression language, the argument string handled by this phase consists of at least one relational expression followed by none or another relational expression or arithmetic expressions. Each of these argument classes, called them as tokens, must be delimited by an ASCII character of "," (comma). Each relational expression consists of two arithmetic expressions as operands and a relational operator as described in the grammer of the language. This algorithm identifies the lexical classes and generates a table containing tokens with an index for identifying the tokens.

An argument string is the SOURCE. A delimiter character named COM is used to isolate the lexical classes of the argument string. Another table containing all the possible relational operators is defined as ROP with an index i. Here nine possible combinations of the relational operators are taken into consideration. They are =,<,>,<>,><,<=,=<,>= and =>. Here it is important to mention that double byte operators are at the top of the table and single byte operators are at the bottom. This is done because one byte operators are part of the two byte operators. During searching process they may mislead the operator type. The analyzed tokens are stored in a table named TABLE with an index j. The maximum token range is denoted by LCLIMIT and a temporary variable named BUFFR is used for checking and processing the tokens.

```
1.[ Initialize]
    Set i <-- 0; and j <-- 1;

2.[ Check the source string ]
    If SOURCE = blank then goto step-7;

3.[ Check the argument delimiter and load the buffer]
    Set k <-- INDEX( COM, SOURCE, 1);
    If k = 0 then set BUFFR <-- SOURCE; and
    SOURCE <-- blank and goto step-5 ;

4.[ Load the buffer and update the source ]
    Set BUFFR <-- SUB( SOURCE, 1, k-1);
    SOURCE <-- SUB( SOURCE, k+1, LENGTH(SOURCE));

5.[ Check for the relational operator and isolate tokens ]
    Set i <-- i+1; and k <-- INDEX(ROP[i], BUFFR, 1);
    If k <> 0 then set TABLE[j] <-- SUB(BUFFR, 1,k-1);
    TABLE[j+1]<-- ROP[i];
    TABLE[j+2]<--SUB(BUFFR,k+LENGTH(ROP[i])+1,LENGTH(BUFFR));
    j <-- j+3 and repeat from step-2;
    otherwise repeat step-5 while i <= 8;

6.[ Extraction of token with no relational operator ]
    Set TABLE[j] <-- BUFFER; j <-- j+1; and
    repeat from step-2;

7.[ Set the index limit ]
    Set LCLIMIT <-- j-1;

8.[ finished ]
    Exit.
```

Algorithm LCVTABLE

This algorithm analyzes the tokens and recursively evaluates to values. In the case of relational operator tokens, a code is generated to the corresponding element of the table. Thus the generated table contains values and the code of the operators. During the evaluation process it uses a procedure to evaluate the expression string which is considered as a token in this phase.

Given a table named TABLE generated by the previous lexical phase with an index i. EXPR_TER is the name of the procedure which interprets the valid expression tokens. BUFFER is the temporary variable used for processing tokens. k is the setting for the index displacement depending on the type of the function. RANGE is the valid token number generated by the previous phase.

```
1. [ Initialize]
     Set i <-- 1; k <-- 2; and  IKEY <-- false;

2. [ Load the buffer for evaluation of the expression ]
     Set BUFFER <-- TABLE[i];
     Call EXPR_TER(BUFFER);
     Set VTABLE[i] <-- BUFFER;

3. [ Increment the index and check for processing ]
     Set i <-- i+k;
     If i=3 then repeat step-2;
     otherwise if TABLE[i] = blank then goto step-6;

4. [ Check and set the index]
     If IKEY = false then repeat step-2;
     otherwise set IKEY <-- false; and i <-- 4;

5. [ Check for the range of tokens and set the displacement]
     If RANGE = 6 then set K <-- 2;
     otherwise set k <-- 1;
     repeat step-2;

6. [ Finished ]
     Exit
```

Algorithm LCEVALUATE

This algorithm defines a procedure which analyzes the logical condition given and returns a boolean result or a value. According to the logical operator given it analyzes the relational argument(s) and returns a boolean result. In the case of the conditional operator it analyzes the

relational argument and evaluate the corresponding expression. The logic followed here first checks the argument and with the result checks the operator and proceeds according to the following table:

| 1st condition | IF | NOT | AND | OR | EQU | NEQ |
|---|---|---|---|---|---|---|
| TRUE | > | > | < | > | < | < |
| FALSE | > | > | > | < | < | < |

> -- exit
< -- 2nd condition check

Here OPERAND1 and OPERAND2 are two buffer variables where two operands of a given relational operator are placed and check the logic with the condition given. If the result is true then the true logic table is checked and if not then the false logic table is checked. Finally the evaluated value is stored in the RESULT. VTABLE is the value table generated from the previous algorithm. LTYPE is the function code given. LP and LG are keys set for the recognition of logical keyword and the conditional keyword.

*1.[ Initialize ]*
*Set i <-- 1;*

*2.[ Load the operator and operand token for logic testing ]*
*Set OPERAND1 <-- VTABLE[i]; OPERAND2 <-- VTABLE[i+2];*
*and condition <-- VTABLE[i+1];*

*3.[ Check the condition over the operands ]*
*If (OPERAND1 condition OPERAND2)=true then goto step-5;*

```
4.[ Perform the operation for false logic condition ]
    If LTYPE = 1 then set RESULT <-- VTABLE[i+4]; and exit;
    otherwise if LTYPE = 2 then set RESULT <-- 1; and exit;
    otherwise if LTYPE = 3 or 5 then
      if LP=false then set LP<--true;i<--4; & repeat step-2;
      otherwise if LG=true then set RESULT <-- 1;
                  otherwise set RESULT <-- 0; and exit;
    otherwise if LTYPE = 4 then set RESULT <-- 0; and exit;
    otherwise if LTYPE = 6 then
     if LP=false then set LP<--true;i<-- 4; & repeat step-2;
     otherwise if LG=true then set RESULT <-- 0;
                  otherwise set RESULT <-- 1; and exit;
    otherwise print error message; and exit;

5.[ Perform the operation for true logic condition ]
    If LTYPE=1 then set RESULT<-- VTABLE[i+3]; and exit;
    otherwise if LTYPE=2 then set RESULT<-- 0; and exit;
    otherwise if LTYPE=3 then set RESULT<--1; and exit;
    otherwise if LTYPE=4 or 6 then
    if LP=false then set LP<-LG<-true;i<--4; & repeat step-2;
    otherwise if LG=true then set RESULT<-- 1;
              otherwise set RESULT<--0; and exit;
    otherwise if LTYPE=5 then
    if LP=false then set LP<-LG<-true;i<--4; & repeat step-2;
    otherwise if LG=true then set RESULT<-- 0;
              otherwise set RESULT<-- 1; and exit;

6.[ Print error message ]
    Exit.
```

## 5.6 BUILT-IN FUNCTION ALGORITHMS

The interpretation process of the built-in function argument string consists of the following algorithms:

# argument string scanning or lexical analysis

# Value table generation for the tokens

# Evaluation of the value table

The grammatical description of the language for the built-in function argument string is given below:

```
<b_function> ::= <b_identifier><(><argument><)>

<b_identifier> ::= MAX¦MIN¦SUM¦AVG¦CNT

<argument> ::= <range>
             ¦<argument><;><range>
             ¦<argument><;><identifier>

<range> ::= <identifier><:><identifier>

<:> ::= :

<;> ::= ;

<(> ::= (

<)> ::= )
```

Algorithm B_FUNCTION

The input parameter of this algorithm is the expression string assigned as SOURCE. Initially this source string is scanned from left to right, looking for any function keyword already defined in a table named BFUN. If no keyword is matched then the algorithm exits without any evaluation. If any function matches then the type of function is identified and corresponding code is generated. The argument string is isolated for processing. BTYPE is the function type code. ASTR is the argument string. In the source string the recognized function keyword with the argument is dissolved and a temporary variable name is replaced,the value of which is the evaluated value of the corresponding function. The argument string is passed to the lexical analyzer BLEX where a token table is produced. This table is then passed to the next phase called the value table generator BVTABLE. This recognized function is finally evaluated in the next phase

called BEVALUATE. This process is repeated again for another function (if any) with the reduced source expression.Finally the algorithm exits if no more function is referenced in the reduced expression string.

```
1. [ Initialize the function keyword table index ]
     Set i <-- 1; and l <-- LENGTH(SOURCE);

2. [ Check for any built-in function keyword ]
     Set j <-- INDEX(BFUN[i],SOURCE,1);
     If j <> 0 then set BTYPE <-- i and goto step-3;
     otherwise set i <-- i+1; and repeat step-2 until i>5;
     goto step-8;

3. [ Identify the argument string of the function ]
     Set k1 <-- INDEX(LPR,SOURCE,j);
         k2 <-- INDEX(RPR,SOURCE,k1);
       ASTR <-- SUB(SOURCE,k1+1,k2-1); and
     SOURCE <-- SUB(SOURCE,1,k1-1)+TS[m]+SUB(SOURCE,k2+1,l);

4. [ Call the lexical analyzer ]
     Call BLEX(ASTR);

5. [ Call the value table generator ]
     Call BVTABEL(TABLE);

6. [ Call the evaluation procedure ]
     Call BEVALUATE(TABLE);

7. [ Check for the existance of another keyword ]
     Repeat from step-1;

8. [ Finished ]
     Exit.
```

Algorithm BLEX

This algorithm analyzes the argument string of the built-in function class. The lexical classes of the argument as the input string consist of list of identifiers or list of ranges or list of identifier(s) and range(s). The delimiter character for each indentifier or range is the ASCII

character "; " (semicolon). Each range defines the lower boundary and the upper boundary of a list or block delimited by an ASCII character of ":" (colon) as described in the grammer of this type. The method of identifying the token depends on the delimiter character. In this algorithm two types of token tables are generated. One type defines the individual identifier list while the other type defines the range list. The latter table is sudivided into two table- one for the lower boundary definition while the other for the upper boundary definition of the range. Each table has an index to indicate the token. These three token tables defines all the lexical classes of the argument string.

The argument string is the SOURCE. Identifier or range delimiter within the argument string is denoted by SLON and the boundary delimiter within the range is denoted by CLON. The table for the individual identifier tokens is given as ICT with an index i. The tables for the range tokens are named as LBT and HBT. LBT for the lower boundary token list while HBT for the upper boundary token list. Both the tables have the same index of j because each range must have both the boundary tokens. ILIMIT is the count of individual identifier tokens and RLIMIT is the count of range tokens. BUFFER is the temporary variable used to process the tokens.

*1. [ Initialize ]*
*Set i <-- j <-- 0;*

*2. [ Check the source input]*
*If SOURCE = blank then goto step-7*

*3. [ Check for delimiter ]*
*Set k <-- INDEX( SCLON, SOURCE, 1);*
*If k = 0 then ( Isolate the token )*
*Set BUFFER <-- SOURCE; and*
*SOURCE <-- blank and goto step-5;*

```
4. [ Isolate the token ]
     Set BUFFER <-- SUB( SOURCE, 1, k); and
     SOURCE <-- SUB( SOURCE, k, LENGTH(SOURCE));

5. [ Check for list argument ]
     Set k <-- INDEX( COLON, BUFFER, 1);
     If k = 0 then ( store the individual cell token )
     Set i <-- i+1 , ICT[i] <-- BUFFER; and
     repeat from step-2

6. [ Store the boundary token ]
     Set j <-- j+1 ;
     LBT[j] <-- SUB( BUFFER, 1, k-1);
     HBT[j] <-- SUB( BUFFER, k+1, LENGTH(BUFFER)); and
     repeat fron step-2;

7. [ Set the index limits ]
     Set ILIMIT <-- i; and BLIMIT <-- j;

8. [ Finished ]
     Exit.
```

Algorithm BVTABLE

This algorithm generates a table after analyzing the lexical tokens outcame from the previous phase. In the analysis of the tokens the same three token tables are used. The analysis process involvs identifying the cells and extract values from the data structure of the cell system accoding to the token type. This algorithm uses procedures for the identification of cells and extraction of values. Finally a table consisting of values is generated for further processing.

Given the three tables as indicated in the previous lexical analysis algorithm with two indexes i and j. Here a procedure named CELLID is called for the identification of cells from the identifier token. VALUE is another procedure

to find the corresponding value from the data structure of the cell system. COUNT and RANGE both are the maximum count of the value table. The generated table is named as VTABLE with an index i.

```
1. [ Initialize ]
   Set i <-- j <-- 0;

2. [ Check the cell token table range ]
   If ILIMIT = o then goto step-4;

3. [ Production of cell value for individual cell token ]
   Repeat step-3 while j <= ILIMIT;
   Set [ ROW, COLUMN ] <-- CELLID( ICT[j]);
   VTABLE[J] <-- VALUE( CELL[ROW,COLUMN]); and
   j <-- j+1 ;

4. [ Check the boundary token table range ]
   If BLIMIT = 0 then set RANGE <-- ILIMIT;  and exit.

5. [ Recognition of boundary cells ]
   Set [RL, CL] <-- CELLID( LBT[i] ); and
   [RH, CH] <-- CELLID( HBT[i] );

6. [ Production of cell values for the list token ]
   Set k1 <-- RL; and k2 <-- CL ;
   If RL = RH then goto step-8;
   otherwise if CL = CH then goto step-9;

7. [ Production of table for a list ]
   Set i <-- i+1 ; VTABLE[i] <-- VALUE( CELL[k1,k2] );
   k2 <-- k2+1; and
   repeat step-7 while k2 <= CH;
   otherwise set k2 <-- CL ; k1 <-- k1+1; and
   repeat step-7 while k1 <= RH;
   otherwise goto step-10;

8. [ Production of table for a column list ]
   Set i <-- i+1 ; VTABLE[i] <-- VALUE( CELL[k1,k2] );
   k2 <-- k2+1; and
   repeat step-8 while k2 <= CH;
   otherwise goto step-10;

9. [ Production of table for a row list ]
   Set i <-- i+1; VTABLE[i] <-- VALUE( CELL[k1,k2] );
   k1 <-- k1+1; and
   repeat  step-9 while k1 <= RH;
```

10. [ Set the range of the table ]
    Set COUNT <-- RANGE <-- i;

11. [ Finished ]
    Exit.

Algorithm BEVALUATE

The input parameter of this algorithm is the function code and the value table generated from the previous algorithm. According to the function code it branches to the corresponing routine and evaluation takes place over the value table and finally returns a value. This value is stored in the RESULT and exit from the algorithm.

*1. [ Check the function code and call the routines ]*
*If BTYPE=1 then set RESULT<-- MAXIMUM(TABLE); and exit.*
*otherwise*
*if BTYPE=2 then set RESULT<-- MINIMUM(TABLE); and exit.*
*otherwise*
*if BTYPE=3 then set RESULT<-- SUMMATION(TABLE); and exit.*
*otherwise*
*if BTYPE=4 then set RESULT<-- COUNT(TABLE); and exit.*

*2. [ print error message ]*
*Print error message and exit.*

## 5.7 OTHER ALGORITHMS

ALgorithm REBK

The input parameter of this algorithm is a valid character string. Any space i.e., blank within the string is removed

by this algorithm. Here the input string is searched from left to right for a blank space, if found then the blank character is removed from the string i.e., the length of the string is reduced by a character. If there are no blank space in the input string then the string is returned as it is. Here the input string is assigned as SOURCE and the output string is REBK as the name of the algorithm. i is the searching index used in this algorithm and j is the position of a blank space returned by the function INDEX. And j is used to minimize the searching time.

*1. [ Initialize the searching index ]*
*Set i <-- 1 ;*

*2. [ Set the length and search the string for the blank ]*
*Set l <-- LENGTH(SOURCE); and*
*j <-- INDEX( CHR(32), SOURCE, i);*
*If j = 0 then goto step-6;*

*3. [ Remove the blank space ]*
*Set REBK <-- SUB(SOURCE,1,j-1)+SUB(SOURCE,j+1,l);*

*4. [ Advance the searching index and repeat the loop ]*
*Set i <-- j; and repeat from step-2;*

*5. [ Finished ]*
*Exit.*

Algorithm BRAKET

The input parameter of this algorithm is the SOURCE. In this algorithm any subexpression delimited by parantheses, as convension, is isolated and in the mother expression this subexpression is replaced by a temporary variable. The precedence of parantheses follows the normal arithmetic rule. Inside-right subexpressions are isolated first. Here a

pointer first scans the source string from left to right searching for a left parantheses. When a parantheses is identified then the pointer moves to right and scans for the next parantheses (if any). If no parantheses is identified then the pointer is set after the last parantheses occurance. Now the pointer scans the string to right searching for a right parantheses. When a right parantheses is recognized then the substring is isolated as a subexpression and the substring is replaced by a temporary variable. In this way all subexpressions can be replaced by repeating the process. Here LPR and RPR represent left and right parantheses respectively. PTABLE is the function keyword position table with an index v. FSWITCH is the key representing whether the corresponding subexpression is the argument of a function or a simple expression. DSTRING is the substring representing the subexpression. Here m is the index of the temporary variable.

*1. [ Initialize the character pointer and set the length ]*
*Set j <-- 1; and LEN <-- LENGTH(SOURCE);*

*2. [ Search the delimiter character and advance the pointer]*
*Set i <-- INDEX( LPR, SOURCE, j);*
*If i <> 0 then set j <-- i+1; and repeat step-2;*

*3. [ Check for the library function key ]*
*If (j-1) = PTABLE[v] then set FSWITCH <-- true;*

*4. [ Set the pointer and search the delimiter character ]*
*Set k <-- j and l <-- INDEX(RPR, SOURCE,k);*
*If l = 0 then print error message and exit.*
*otherwise set k <--(l-1);*

*5. [ Set the delimited string and the source string]*
*Set DSTRING <-- SUB( SOURCE,j,k); and*
*SOURCE <-- SUB(SOURCE,1,j-2)+TS[m]+SUB(SOURCE,k+1,LEN);*

*6. [ Finished ]*
*Exit.*

Algorithm LIB_FUN

The input parameter of this algorithm is the SOURCE expression string. Here any library function is identified in the expression and a table is generated with the function code. Another table is generated simultaneously which represents the position of the corresponding function within the source string . During the generation of the tables, function keywords are deleted from the source string. Here FSTR is the string containing all the library function keywords in a sequence. All keywords are of fixed length of three characters. LPR is the string representing the left parantheses. Left parantheses is searched first because all function has an argument delimited by parantheses. FTABLE is the generated function code table and PTABLE is corresponding function position table. Both the table use an index of i. TEST is the function position returned by the INDEX function during the searching of keywords. RANGE is the number of functions i.e., the length of the tables.

*1. [ Initialize the table index and the search pointer ]*
*Set i <--1 and j <-- 3;*

*2. [ Search the delimiter character ]*
*Set k <-- INDEX(LPR, SOURCE, j);*
*If k = 0 then goto step-6;*
*If k < 4 then set j <-- k+1; and repeat step-2;*

*3. [ Search for a function keyword ]*
*Set TEST <-- INDEX(SUB(SOURCE,k-3,k-1),FSTR,1);*
*If TEST = 0 then set j <-- k+1; and repeat step-2;*

*4. [ Set the function code and position ]*
*Set FTABLE[i] <-- ((TEST-1)/3)+1;and PTABLE[i] <-- k-3;*

*5. [ Eliminate the keyword and reset the indexes ]*
*Set SOURCE <-- SUB(SOURCE,1,k-4)+SUB(SOURCE,k+1,1);*
*i <-- i+1; j <-- k-2; and repeat step-2;*

*6. [ Set number of keywords in the table ]*
*Set RANGE <-- i-1;*

*7. [ Finished ]*
*Exit.*

Algorithm CELLID

The input parameter of this algorithm is the cell identifier string. This algorithm resolves the identifier into the corresponding cell address. The address of a cell means its row value and the column value. A cell can be identified uniquely by a row and a column. An identifier is a language containing one or two letter(s) with one or more digits appended.Thus the length of an identifier never exceeds five after removing all the blank(s) , within the identifier. By this representation we can represent a cell of maximum range of row 999 and column (26*26). In this algorithm initially identifier string is packed by removing all the blank(s) within the string. Then the first character is checked whether it is an alphabet or not. If not then an error message is issued. If the first character is a letter then its numeric equivalent is calculated. Then the second character is checked, if it is a letter then its numeric equivalent is calculated otherwise it is treated as a digit and along with the rest of the string is converted to its numeric equvalent. This digit(s) equivalent number represents the row address and the letter(s) equivalent number represents the column address. Now finally these numbers are checked by the valid range given. If they are within the range then no problem but it they are out of the range then an error message is issued. Here LETTER is a string containing all the alphabetic chatacters (upper

case). REBK is the blank removal procedure. RLIMIT and CLIMIT are the maximum rows and the columns defined initially.

```
1.[ Removes all blanks and set the length of the source ]
    Set SOURCE <-- REBK(SOURCE) and l <-- LENGTH(SOURCE);

2.[ Check for the valid length of the idetifier ]
    If l > 5 then print error message and exit.

3.[ Check the first character to match the syntax ]
    Set i <-- INDEX(SUB(SOURCE,1,1),LETTER,1);
    If i = 0 then print error message and exit.

4.[ Check the second character for letter or digit ]
    Set j <-- INDEX(SUB(SOURCE,2,1),LETTER,1);
    If j = 0 then goto step-6;

5.[ Convert the cell into its address for double letter id.]
    Set ROW <-- (26*i+j); and
      COLUMN <-- VALUE( SUB(SOURCE,2,1));
    goto step-7;

6.[ Check the address with the limits given ]
    If ROW (>RLIMIT or <1) then print error message & exit.
    If COLUMN (>CLIMIT or <1) then print error message & exit.

8.[ Finished ]
    Exit.
```

CHAPTER-6

# SCREEN MANIPULATION

## 6.1 INTRODUCTION

The whole system uses computer memory which consists of cells organised into rectanguler grid. Each cell has a name for identification. Thus the location of a cell within the grid defines its cell address. In order to reference a cell, its name can be mentioned with the column letter first and then the row number. The entire grid system is far too large to display on terminal screen at one time. Thus the screen is used as a display window, as shown in the figure-6.1a, that can slide over the entire grid system but showing a portion at a time.

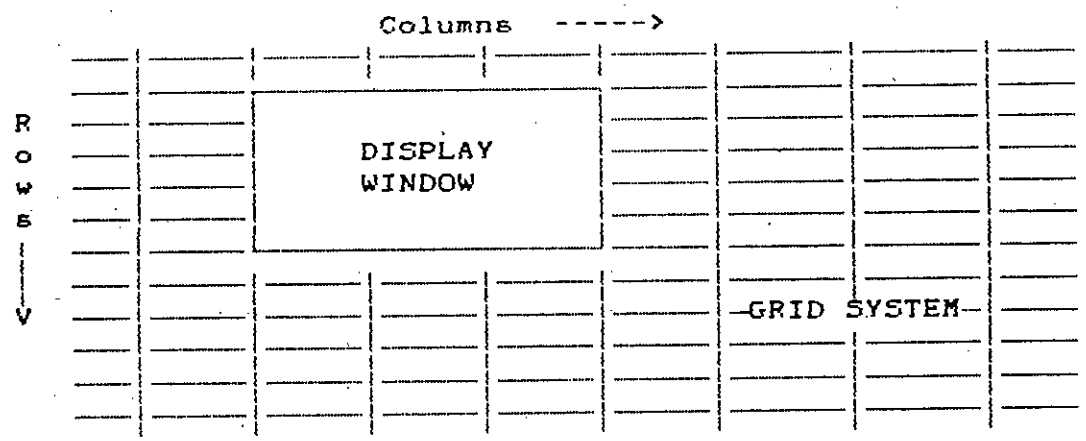


Figure-6.1a The display window and the grid-system

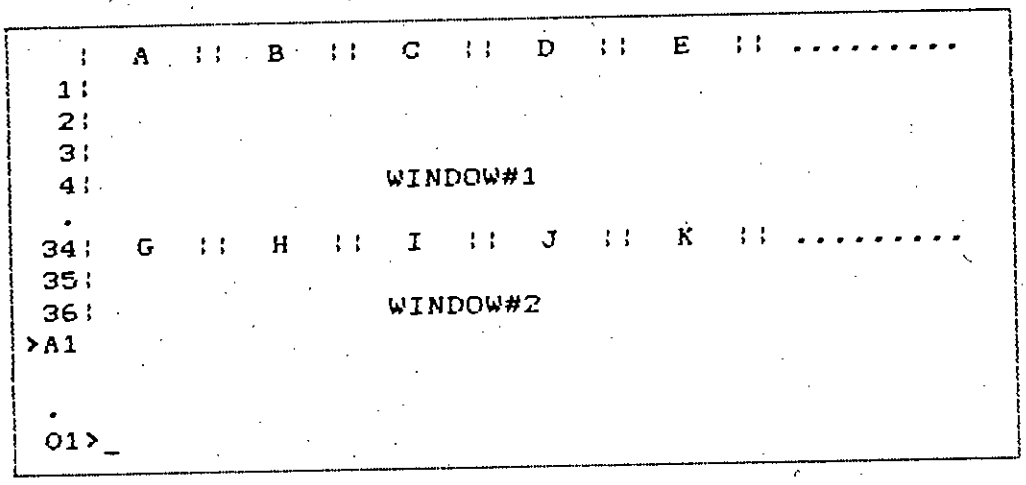


Figure-6.1b Horizontal split screen

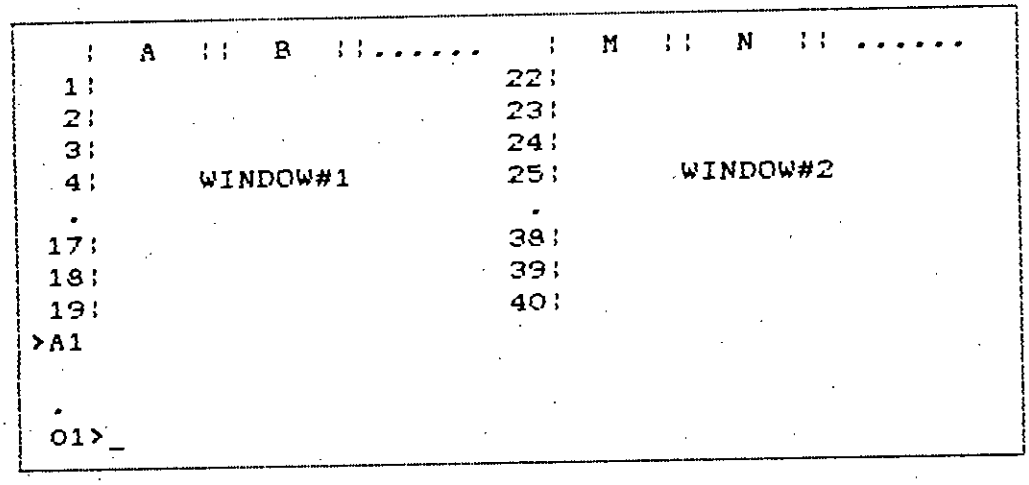


Figure-6.1c Vertical split screen

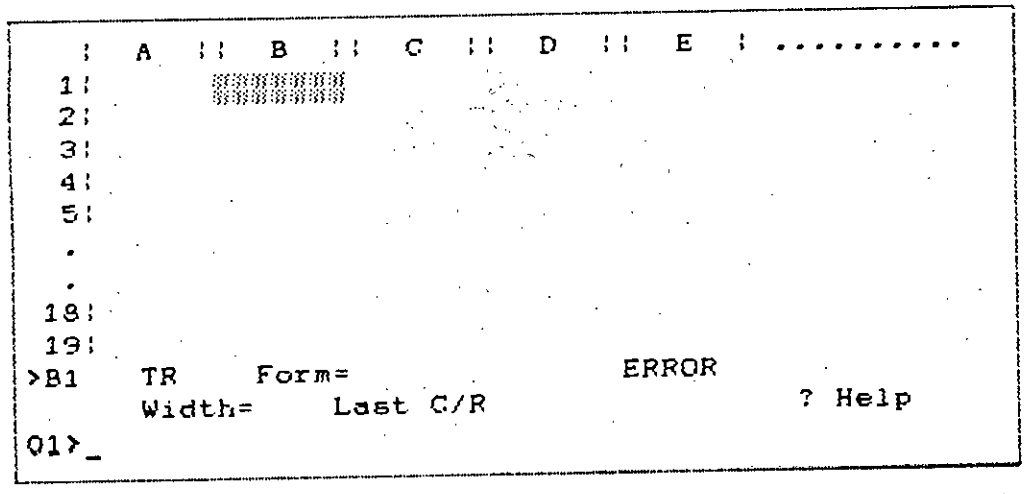


Figure-6.1d

Column & Row border, Active cell cursor and Status lines

The screen can be subdivided to show different portions of the grid system as shown in the figure-6.1b and 6.1c. The screen border identifies the currently displayed columns and rows. The top border, called the column border, contains letters and delimiters. The left border, called the row border, contains numbers and delimiters as shown in the figure-6. 1d. It can be made possible to turn the border off or on as desired. When the border is set to off, then it will not be displayed on the screen and as well as not printed on the printer.

The active cell is the cell affected by the data entry at the present time. In order to identify the active cell, a pointer, called the active cell cursor, identifies the active cell. Only one cell is made active at a time as shown in the figure-6.1d and it always displays because only one cell must active to accept data at one time and it must be identified. The cell cursor can be set to move automatically to an adjacent cell or to remain in the current cell upon data entry. When set to move automatically, it moves in the direction of its previous move to the adjacent cell, which then becomes the active cell. When set to remain stationary, the cursor does not move upon data entry.

The status lines show the active cell status, global status and data entry or command status. As shown in the figure-6.1d, the bottom three lines display the current status.

- Active cell status
- Global status or prompt
- Data entry or command

The active cell status and global status or prompt lines display only the information related to the active cell and the system status. On the data entry or command line , data or command can be entered into the system. For the data entry function, a cursor , called the edit corsor, always displays on this line.

Active cell status line displays information about the active cell. A sample active cell status line looks like this.

```
> A1  TR   Formula= C1*10+D2     ERROR
```

# >, cursor direction. The first character indicates the current direction of motion of the cell cursor. In order to enter data into the active cell, carrage return character must be encountered in the edit line. At the same time the cell cursor moves to the adjacent cell in the direction indicated. This direction is always that of the previous cursor move.

# A1, active cell address. The coordinates of the active cell display. Commands that reference current column or current row use the column/row containing this cell.

# TR, cell format. This shows the active cell display format. TR represents the right justified text.

# Form=, data type of the active cell. The system recognizes three types of data.

VALUE -- Numeric value,
TEXT -- Text string, and
FORMULA -- Formula entry.

# C1*10+D2, cell content . This shows the literal content of the active cell. The content is of the same type as indicated in the cell format.

# ERROR, error message. If an error occurs, an error message displays on the right end of this line.

The global status or prompt is the middle status line. The global status line contains the following informations:

# Width, the column width of the active cell. Initially a default column width is set for each cell. This default setting can be changed by the user as desired.

# Last Column/Row, the intersection of the last column and row that contains data. It is the composite of the last column and last row that have a nonblank cell.

# ? Help, this character shows for pressing in order to switch to the helping menu.

The data entry or command line contains the edit cursor. The number at the left side indicates the current edit cursor position. The data entry or command line serves a number of functions.The character entered into position 1 on this line determines its mode.

i. Data entry mode enters data directly into the active cell. In this mode three types of data as indicated before can be entered into the active cell. The mode can be set to one of the three options plus a repeating text entry option with the help of the four function keys.

ii. Command mode performs specific function according to command character entered in select mode. The command characters are as follows:

=, the goto command moves the cursor directly into the designated cell,

!, the recalculate command forces a recalculation of the entire system,

¦, the switch window command position the cell cursor in the alternate window on a split screen,

/, selects the slash commands, and

?, the help command switch to the help menu.

## 6.2 STRUCTURE OF THE MANIPULATOR

The system operates in three distinct modes:

# Grid display mode
# Data entry mode and
# Command mode

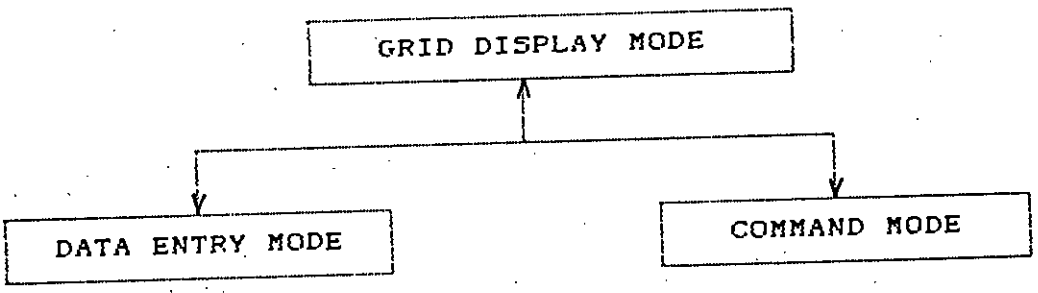


Figure 6.1, Operation modes

Grid display mode is the initial mode of operation. So we must have a option to move from this mode to data entry mode or to command mode. Direct switching from data entry mode to command mode is not necessary because grid display mode always shows the working area of the cell system.

In grid display mode the cell cursor is active and the edit cursor is inactive. At this mode it is possible to move over the system with the help of the cursor control keys. At this mode the window can be slided over the grid system.

In the data entry mode edit cursor is active in the data entry line. A carrage return enters the data from the data entry line into the active cell. During the entry of data in the edit line, data can be edited with the help of the back space key. In the data entry mode data can be edited with the help of the back space key, insert key, delete key etc. and the editing procedure is the same as the in-line editor. In the command mode the system is directed to perform a particular action. With the help of one of the five command keys we can switch to a particular action. In this mode command can also edited with the help of the back space key.

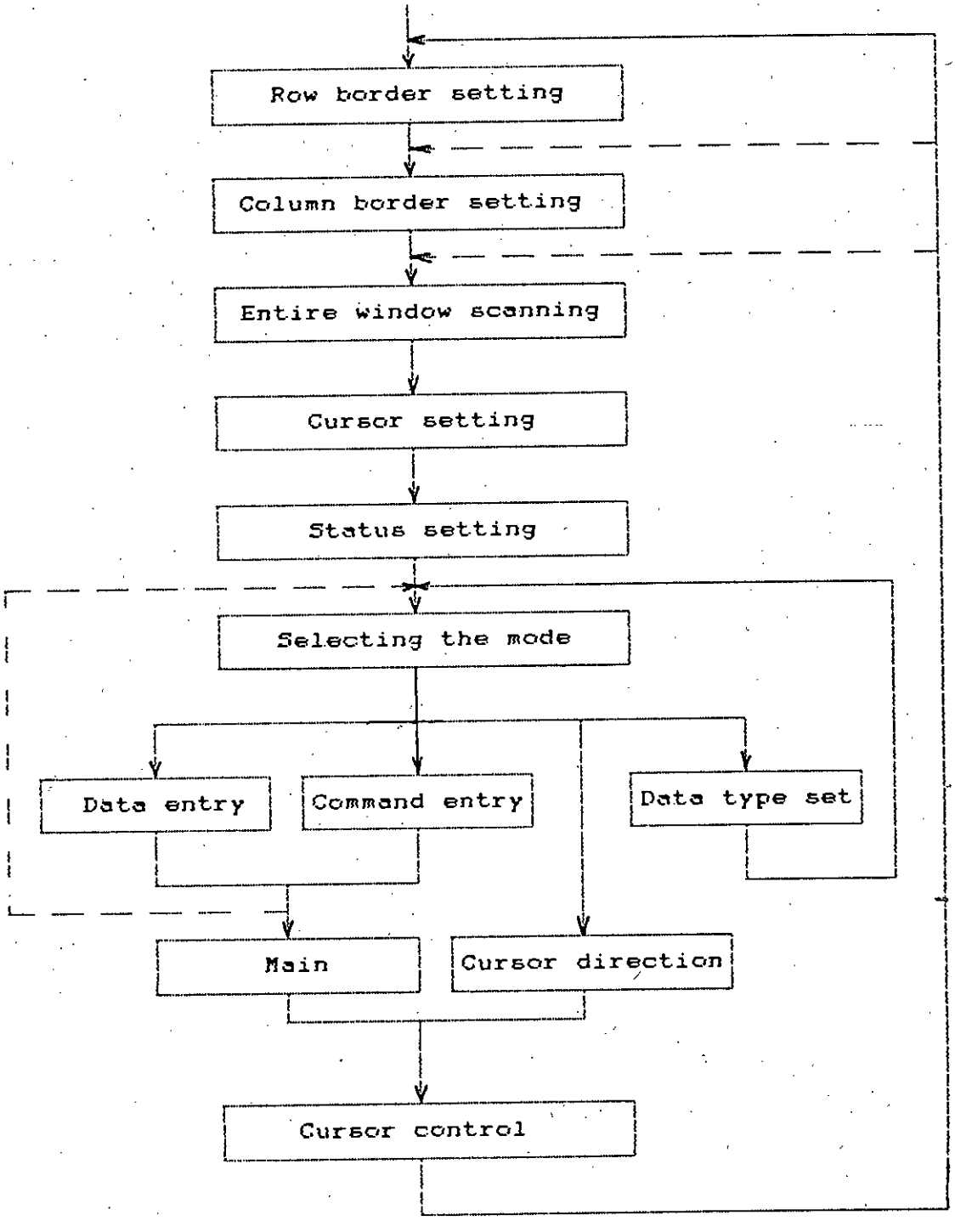


Figure-6.2

## 6.3 SCREEN MANIPULATOR ALGORITHMS

Algorithm DISPLAY

This algorithm defines a procedure which uses a number of algorithms described later in this chapter. These algorithms belong to various functions of the screen manipulator. ROW defines the row border display of the window, COLUMN defines the column border display of the window , WINDOW defines the scanning of all the cells within the window boundary, CURSOR defines the deletion of the previous active cell cursor and position the current active cell cursor, STATUS shows the various status information on the status lines, SELECT identifies the mode to be set and sets the corresponding code, MAIN is the data processing algorithm, INDATA accepts the data , COMMAND defines the command mode, DATA_TYPE_SET defines the data type entry mode, CURSOR_DIR_SET defines the direction of the active cursor and CURSOR_CONTROL sets all the setting keys to perform the active cell cursor movement. In this algorithm SELKEY is the key to set the operation mode and SKEY is the key for setting to switch to various algorithmic steps. BKEY is the border on or off key. CCKEY selects switching to CURSOR_CONTROL algorithm.

*1.[ Initialize the parameters ]*
*Set IR<--IC<--AR<--AC<--PAR<--PAC<--1;*
*CR<--PCR<--2;CC<--PCC<--5; RT<--62;*
*RSTART<--2;REND<--20;CSTART<--5; and CEND<--81;*

*2.[ Print the row and column border for initial setting ]*
*Call ROW(CSTART,RSATAT,REND,IR); and*
*Call COLUMN(RSTART,CSTART,CEND,IC); and goto step-5;*

*3.[ Scan the window with the row border ]*
*If BKEY = true then call ROW(CSTART,RSTART,REND,IR); and*
*Call WINDOW(CSTART,CEND,RSTART,REND,IC,IR);*

```
4.[ Scan the window with the column border ]
    If BKEY = true then call COLUMN(CSTART,RSTART,REND,IR);
    and Call WINDOW(CSTART,CEND,RSTART,REND,IC,IR);

5.[ Erase the previous cursor and print the active cursor]
    Call CURSOR(PCR,PCC,PAC,PAR,CR,CC,AR,AC);

6.[ Print the status lines ]
    Call STATUS;

7.[ Receive the select code and check the code ]
    Call SELECT;
    If SELKEY = 1 then call INDATA and goto step-8;
    otherwise
    if SELKEY = 2 then call COMMAND and goto step-8;
    otherwise
    if SELKEY = 3 then call DATA_TYPE_SET and repeat step-7;
    otherwise
    if SELKEY = 4 then call CURSOR_DIR_SET and goto step-9;

8.[ Branch to the data processing procedure ]
    If (INSTRING=NUL) and (COMKEY=false) then repeat step-7
    otherwise call MAIN

9.[ Set the cursor control key if necessary ]
    If CCKEY = true then call CURSOR_CONTROL;
    If SKEY = 1 then repeat from step-3;
    otherwise if SKEY = 2 then repeat from step-4;
    otherwise if SKEY = 3 then repeat from step-5;
```

Algorithm JUSTIFICATION

The input parameters are INSTRING, SPAN, JUSKEY. Given a character string INSTRING which contains any valid printable characters. This input string is to be justified either right or left depending on the value of a key JUSKEY. If the JUSKEY has a value of logical true then it will be right justified string and if JUSKEY has a value of logical false then it will be a left justified string. SPAN is the length of the justified string. If the source string is smaller in length than the SPAN then blank characters are inserted to maintain the fixed length of the justified

string. If the source is longer than the SPAN then extra characters are truncated to fix the justified string.

```
1. [ Set the length of the source string ]
     Set l <-- LENGTH(INSTRING);

2. [ Check and set for smaller source string case ]
     If l < SPAN then
           if JUSKEY = true then
           set JUSTIFICATION <-- SPACE(SPAN-l)+INSTRING;
           otherwise
           set JUSTIFICATION <-- INSTRING+SPACE(SPAN-l);
           goto step-5;

3. [ Check and set for longer source string case ]
     If l > SPAN then
           if JUSKEY = false then
           set JUSTIFICATION <-- SUB(INSTRING,l-SPAN+1,l);
           otherwise
           set JUSTIFICATION <-- SUB(INSTRING,1,SPAN);
           goto step-5;

4. [ Set for equal source string ]
     Set JUSTIFICATION <-- INSTRING;

5. [ Finished ]
     Exit.
```

Algorithm ROW

This algorithm defines a procedure for displaying the row border of the display window. The format of the border consists of numbers followed by a border delimiter character to define the cells of each row. An index defines the position of the border to be displayed on the screen. The border length also must be defined to display the window border size. A pointer for the row number corresponding to a cell within the cell system must be defined. This pointer represents the lower boundary of the row number. This algorithm produces a border for each row corresponding to the window cell and displays on the screen. This process is

repeated for a number of rows until the row number exceeds the given border length.

Given a column position COLUMN before which the row border is to be printed. ROWSTART and ROWEND are the window boundaries for the row numbers i.e., they represent the boundaries of the index i between where the row numbers are to be printed. ROW is the pointer of the number table which indicates the lower boundary of the row number. Here JUSTIFICATION is a procedure to use to justify the given string to right for a given length. PRINT is another procedure to print a string to a given position of the screen. RSTRING is the string variable used to define the row border for a single row.

*1. [ Initialize ]*
*Set i <-- ROWSTART; j <-- ROW; and k <-- (COLUMN-4);*

*2. [ Set the foreground and background colors ]*

*3. [ Set the row string for each row ]*
*Set RSTRING <-- JUSTIFICATION(STR(J),3,1)+CHR(d);*

*4. [ Print the row string ]*
*PRINT (i,k,RSTRING);*

*5. [ Increment the pointers and check for limits ]*
*Set i <-- i+1; and j <-- j+1;*
*If i <= ROWEND then repeat from step-3;*

*6. [ Reset the foreground and background colors ]*

*7. [ Finished ]*
*Exit.*

Algorithm COLUMN

This algorithm defines a procedure for displaying the column border of the display window. The format of the border consists of letters and delimiter characters to define the

cells and the boundary of the cell widths respectively. An index defines the position of the column border to be displayed on the screen. The span of the border i.e., the upper boundary and the lower boundary of the window is defined by two indexes. A pointer for the column letter corresponding to a cell to be displayed is also defined. This pointer represents the lower boundary of the column letter. This algorithm produces a string containing column letters with delimiter characters setted according to the width of the corresponding cells and finally displayed on the screen.

```
1.[ Initialize ]
    Set j <-- LP; k <-- WIDTH(j); CSPAN <-- (CEND-CSTART);
    and CBORDER <-- SPACE(4);

2.[ Check the column number and set the border ]
    If j<27 then set FIRST <--CHR(32);1<--j and goto step-3;
    otherwise if J<53 then
    set FIRST <-- CHR(65); 1 <-- (j-26); and goto step-3;
    otherwise ifj<79 then
    set FIRST <-- CHR(66); 1 <-- (j-52); and goto step-3;
    otherwise print error message and exit

3.[ Set the width and check for odd or even ]
    Set w <-- WIDTH(j);
    If MOD(w,2) = 0 then
    set w <-- (w-1); and NAME <-- CHR(32)+FIRST;
    otherwise set NAME <-- FIRST;

4.[ Set the blank character number for each cell ]
    Set m <-- INTEGER(w*0.5)-1;

5.[ Set the column border string ]
    Set CBORDER <-- CBORDER+CHR(d)+SPACE(m-1)+NAME+CHR(64+1)
                    +SPACE(m)+CHR(d);

6.[ Increment the letter pointer & set the width & check ]
    Set j <-- j+1; and k <-- k+WIDTH(j);
    If k <= CSPAN then repeat from step-2;
```

*7.[ Set the foreground and the background colors ]*

*8.[ Print the border string ]*
*PRINT (ROW-1,COLUMN-4,JUSTIFICATION(CBORDER,CSPAN+4,0));*

*9.[ Reset the foreground and the background colors ]*

*10.[ Finished ]*
*Exit.*

Algorithm WINDOW

This algorithm defines a procedure for scanning the whole window with a set of cell values or contents depending on the display format of the corresponding cells. This scanning process over the window requires two sets of pointers. One set indicates the position of the window on the screen while the other indicates the set of cells to be displayed on the window. It is important to note that the column length used in this algorithm must coincide with that of the row border. Similarly the row lenght must coincide with the column border. This three procedures must be synchronously active on the display screen. This algorithm directly communicates with the data structure of the cell system for informations such as the cell contents or values or the cell formats etc. In this algorithm actually the window is scanned row by row. All informations corresponding to a single row are arranged in a string and the arrangement depends on the display formats of the corresponding cells. This row scanning procedure is repeated until the compared row pointer exceeds the given row limit.

Given a set of arguments RSTART, REND, CSTART, CEND, C and R. RSTART and REND defines the boundary limits of the window column where CSTART and CEND defines the boundary limits of

the window row. Both are defined in terms of the cell numbers. C and R gives the address of the top-left corner cell of the window. Here WIDTH in a table containing the information related to the width of the cells with an index c. FMT[r,c] and CELL[r,c] represent the format and content information in the data structure corresponding to a cell having a row address of r and column address of c.

*1. [ Initialize ]*
*Set i <-- RSTART; CSPAN <-- (CEND-CSTART); and r <-- R;*

*2. [ Set the foreground and the background colors ]*

*3. [ Set the column pointer, cell width and check with span]*
*Set c <-- C; k <-- WIDTH(j); and RSTR <-- NULL;*
*If k > CSPAN then print error message and exit.*

*4. [ Check the format and set the string ]*
*If SIGN(FMT[r,c])= NEGATIVE then*
*set j <-- true otherwise set j <-- false*
*set RSTR<--RSTR+JUSTIFICATION(CELL[r,c],WIDTH(c),j);*

*5. [ Increment the column pointer & reset the span & check ]*
*Set c <-- c+1 and k <-- k+WIDTH(c)*
*If k <= CSPAN then repeat step-4;*

*6. [ Print the row string on the screen ]*
*PRINT (i,CSTART, JUSTIFICATION(RSTR,CSPAN,0));*

*7. [ Reset the row pointer and check the limit ]*
*Set r <-- r+1; and i <-- i+1;*
*If i <= REND the repeat from step-3;*

*8. [ Reset the foreground and the background colors ]*

*9. [ Finished ]*
*Exit.*

Algorithm CURSOR

The input parameters are RP1,CP1,NPC1,LPC1,RP2,CP2,NPC2 and LPC2. The parameters RP1 and CP1 represent the present

corsor position and NPC1 and LPC1 for the cell identification of the active cell. Similarly RP2 and CP2 represent the previous cursor position and NPC2 and LPC2 represent the previous cell identification. This algorithm defines a procedure for erasing the previous cursor but showing the cell content and printing the present cursor with the active cell content if any. Here FMT represents a table to give the information of the corresponding cell whether it is right justified or left justified. A positive sign gives the information of left justification and a negative sign gives the information of right justification. A procedure named JUSTIFICATION as described before is used to justify the text given.

```
1.[ Check for the justification format of the previous cell ]
    If SIGN(FMT[NPC2,LPC2])=NEGATIVE then set k <-- true;
    otherwise set k <-- false;

2.[ Set the previous cursor string ]
    Set CURSOR<--JUSTIFICATION(CELL[NPC2,LPC2],WIDTH[LPC2],k);

3.[ Erase the previous cursor ]
    PRINT (RP2,CP2,CURSOR);

4.[ Check for the justification format of the active cell ]
    If SIGN(FMT[NPC1,LPC1])=NEGATIVE then set k <-- true;
    otherwise set k <-- false;

5.[ Set the active cursor ]
    Set CURSOR<--JUSTIFICATION(CELL[NPC1,LPC1],WIDTH[LPC1],k);

6.[ Set the background color and print the cursor ]
    PRINT (RP1,CP1,CURSOR);

7.[ Reset the background color and exit ]
    Exit.
```

Algorithm SELECT

This algorithm identifies the input code and enables the corresponding mode by setting keys. Here INKEY is a notation used to indicate a function to receive the input code. INKEY waits for keystroke and when a key responses,it receives the corresponding key code and it executes only for a single time. CODE is the received code. Now here COMSTR is the command code listing, where command codes are stored in a sequence. Here the COMSTR is defined in such a way that each code occupies three character length.Thus if there have five codes then the length of the COMSTR is 15. If any command code returned in the buffer CODE then it must match with any one of the COMSTR codes. If the code is a command code then a key called SELKEY is set to a numeric value (say 2) which is the key code of command mode and hence no further inquery is required. It is important to note that the above code testing is much more efficient than the repetative testing of each code with the returned code. When exit from this algorithm the code position is set to COMCODE in the case of command mode set. Similarly cursor control codes are stored in a string named CURSTR and the code position is set to CURCODE if any cursor control code matches. SELKEY is set to a numeric value of 4 (say). Also for the case of data type code, codes are stored in the string DTYSTR and the code position is stored in DTYCODE. SELKEY is set to 3. Now if the returned code does not match with any of the three types of code then it is assumed to be a data for the cell entry and hence checks the data for valid ASCII code, range between 32 to 127. If the code is valid then SELKEY is set to 1 and exit from the algorithm.

```
1. [ Print the edit line code and set the edit cursor]
     PRINT (23,2,STRING(00)+CHR(62));

2. [ Receive the input command ]
     Set CODE <-- INKEY;

3. [ Check the code with the cursor control code string ]
     Set CURCODE <-- INDEX(STRING(CODE),CURSTR,1);
     If CURCODE <> 0 then set SELKEY <-- 4 and goto step-7;

4. [ Check the code data type code string ]
     Set DTYCODE <-- INDEX(STRING(CODE),DTYSTR,1);
     If DTYCODE <> 0 then set SELKEY <-- 3 and goto step-7;

5. [ Check the code with the command code string ]
     Set COMCODE <-- INDEX(STRING(CODE),COMSTR,1);
     If COMCODE <> 0 then set SELKEY <-- 2 and goto step-7;

6. [ Check the code with the valid code range for data entry]
     If CODE >= 32 or CODE <= 127 then
                    set SELKEY = 1 and goto step-7;
     otherwise repeat from step-1

7. [ Finished ]
     Exit.
```

Algorithm INDATA

This algorithm receives data from the keyboard and/or edits according to the instruction given.When the first input code is not a command or a cursor control or a data type code, received by the algorithm SELECT, then the code is passed as a cell data to this algorithm. Thus the input codes are concatenate with the input string until it encounters a control code of 13 (carrage return), used to return the data to the system i.e., this code indicates that the data entry has been finished here. Another code numbered as 8 (back space) is used to erase the previous inputted character in the input string . After encounting the carrage return code, this algorithm exits and return to the calling

routine. After receiving a back space code this algorithm removes the previous character from the input buffer string as well as from the screen and also the cursor is reset. Now if the back spacing makes the buffer string to null then the algorithm returns to the mother program because the 1st character entry is checked by the algorithm SELECT. Here l represent the length of the string i.e., (l+1) represent the present edit cursor position. In order to display the present cursor position, this number must be converted to string data with justification to right. This string is placed on the screen along with the edit line indicator symbol (>). INSTRING is the buffer string where the inputted character codes are stored. INSTRING is loaded initially by the character code returned from the algorithm SELECT. CHACODE is the code received by the function INKEY.

```
1.[ Initialize the buffer string ]
   Set INSTRING <-- CODE

2.[ Set the cursor position from the length of the buffer ]
   Set l <-- LENGTH(INSTRING); and
   CURPOS <-- JUSTIFICATION(STRING(l+1),2,1);

3.[ Print the buffer string with the cursor position number]
   PRINT (23,2,CURPOS+CHR(62)+INSTRING);

4.[ Wait for the input character and save the code ]
   CHACODE <-- INKEY;

5.[ Check for the returning code ]
   If CHACODE = 13 then
            set COMKEY <-- false; and goto step-8;

6.[ Check for the editing code ]
   If (CHACODE = 8) and (l = 1) then set INSTRING <-- nul;
   and goto step-8;
   otherwise if CHACODE = 8 then
   set INSTRING <-- SUB(INSTRING,1,l-1);
   PRINT (23,1,SPACE(1)); and repeat from step-4;
```

*7.[ Concatenate the new code with the previous codes ]*
*Set INSTRING <-- INSTRING + CHR(CHACODE); and*
*repeat fron step-2;*
*8.[ Finished ]*
*Exit.*

Algorithm COMMAND

This algorithm decodes the command code returned from the algorithm SELECT. CKEY represents the translated code and COMKEY represents the command mode character code.

*1. [ Analyze the command code ]*
*Set CKEY <-- (COMCODE-1)/3+1;*

*2. [ Set the command key ]*
*Set COMKEY <-- true;*

*3. [ Finished ]*
*Exit.*

Algorithm DATA TYPE SET

This algorithm sets the data type entry mode key and corresponding mode title. MKEY is the translated code of the actual mode setting key codes. DTYCODE is the code returned from the algorithm SELECT. Four types of data entry modes are possible to set. These are value entry, text entry, repeating text entry and the formula entry. HEAD is the title string which represents the type of mode on the display screen. ROW and COLUMN represent the position where the title is to be printed.

*1.[ Analyze the data type code ]*
*Set MKEY <-- (DTYCODE-1)/3+1;*

*2.[ Set the mode according to the translated code ]*
*If MKEY = 1 then set HEAD <-- "VALUE" and goto step-3;*
*otherwise*
*if MKEY = 2 then set HEAD <-- "TEXT" and goto step-3;*
*otherwise*
*if MKEY = 3 then set HEAD <-- "R-TEXT" and goto step-3;*
*otherwise*
*if MKEY = 4 then set HEAD <-- "FORMULA" and goto step-3;*
*otherwise print error message and exit.*

*3.[ Set the backgorund and the foreground colors ]*

*4.[ Print the title of the corresponding mode ]*
*PRINT (ROW,COLUMN,HEAD);*

*5.[ Reset the backgorund and the foreground colors ]*

*6.[ Finished ]*
*Exit.*

Algorithm CURSOR DIR SET

This algorithm sets the cursor direction key to one of the four codes. Code lr represents the cursor direction from left to right, code rl represents from rigth to left, code ud repersents from up to down and code du represents from down to up. CURCODE is the translated code of the actual cursor control key code. RT represents the cursor direction.

*1. [ Analyze the cursor direction code ]*
*Set CURCODE <-- (CURCODE-1)/3+1;*

*2. [ Check the code AND set the mode ]*
*If CURCODE = 1 then set RT <-- rl; and goto step-3;*
*otherwise*
*if CURCODE = 2 then set RT <-- lr; and goto step-3;*
*otherwise*
*if CURCODE = 3 then set RT <-- ud; and goto step-3;*
*otherwise*
*if CURCODE = 4 then set RT <-- du; and goto step-3;*
*otherwise pritn error message and exit.*

*3. [ Finished ]*
*Exit.*

Algorithm CURSOR CONTROL

The input parameters of this algorithm are the present active cell address, previous active cell address, top-left cell address of the window, row and column ranges and the cursor direction code. Here first cursor direction code is checked and switches to the corresponding steps to set one of the four possible settings.To set the right direction parameters we use two procedures LIMIT and SPAN, which are defined after this algorithm. Both the procedures check the window width with the summation of the individual column widths. LIMIT returns the address of the last cell whereas SPAN returns the starting address of the accomoded last cell within the window. Here various indexes are checked and reset according to the logic. SKEY is the setting for identification to branch to various levels.

1. *[ Initialize the indexes of the previous active cell ]*
   *Set PAR <-- AR; PAC <-- AC; PCR <-- CR; and PCC <-- CC;*

2. *[ Check the cursor direction code and select settings ]*
   *If RT = lr then goto step-6;*
   *otherwise if RT = rl then goto step-7;*
   *otherwise if RT = ud then goto step-8;*
   *otherwise if RT = du then goto step-9;*
   *otherwise goto step-9;*

3. *[ Check and set the right cursor movement indexes ]*
   *If CC = CLIMIT then set SKEY <-- 3 and exit.*
   *otherwise set CC <-- CSTART;*

4. *[ Check the accomoded width of the window ]*
   *If AC = LIMIT(IC) then goto step-5;*
   *otherwise set CC <-- SPAN(IC,AC);*
   *AC <-- AC+1; SKEY <-- 3; and exit.*

5. *[ Set the indexes for the movement of the window ]*
   *Set IC <-- IC+1; and*
   *if LIMIT(IC) < (AC+1) then repeat step-4;*
   *otherwise set CC <-- SPAN(IC,AC);*
   *AC <-- AC+1; SKEY <-- 2 and exit.*

```
6. [ Check and set the left cursor movement indexes ]
     If AC = 1 then SET SKEY <-- 3; and exit.
     otherwise set AC <-- AC-1;
     If CC = CLIMIT then
     set IC <-- IC-1; SKEY <-- 2; and exit.
     otherwise
     set CC <-- CC-WIDTH[AC]; SKEY <-- 3; and exit.

7. [ Check and set the down cursor movement indexes ]
     If AR = RLIMIT then set SKEY <-- 3; and exit.
     otherwise set AR <-- AR+1;
     If CR = REND then
     set IR <-- IR+1; SKEY <-- 1; and exit.
     otherwise
     set CR <-- CR+1; SKEY <-- 3; and exit.

8. [ Check and set the up cursor movement indexes ]
     If AR = 1 then set SKEY <-- 3; and exit.
     otherwise set AR <-- AR-1;
     If CR = Rstart then
     set IR <-- IR-1; SKEY <-- 1; and exit.
     otherwise
     set CR <-- CR-1; SKEY <-- 3; and exit.

9. [ Finished]
     Exit.
```

Algorithm LIMIT

```
1. [ Initialize the pointer and the width buffer ]
     Set i <-- IC-1; and w <-- 0;

2. [ Calculate the maximum limit within the window ]
     Set i <-- i+1; and  w <-- WIDTH[i];
     If  w+WIDTH[i+1]  < CEND-CSTART  then repeat step-2;
     otherwise set LIMIT <-- i;

3. [ Finished]
     Exit.
```

Algorithm SPAN

```
1. [ Initialize the pointer and the width buffer ]
     Set i <-- IC-1; and CC <-- 0;
```

2. [ Calculate the maximum limit within the window ]
   Set i <-- i+1; and SPAN <-- SPAN + WIDTH[i];
   Repeat step-2 until i <= AC;

3. [ Finished ]
   Exit.

CHAPTER-7

# DATA REPRESENTATION AND COMMANDS

## 7.1 INTRODUCTION

Since the selection of data structures significantly influences the speed and efficiency of an implementation of an algorithm, hence it is also one of the most important part of the system design. As the data structure, arrays are used here because of the following advantages:

# Arrays are helpful in organizing sets of data into meaningful groups.

# Array names with subscripts minimize the need for keeping track of many data items with different names.

# The use of subscripts allows for instant and automatic access to any element of an array.

# Subscripting also allows for automatic, fast, and efficient processing of all data or selected subsets of data stored in arrays. Such processing includes initialyzing, searching, storing, and updating.

The cell system is an aggregate of concurrently active objects, organized in a rectangular array of cells, similar to the paper spreadsheet. Thus in this system design, data structures of the cell system are mainly defined by rectangular arrays.

Entire cell system can be defined as a rectangular grid containing a number o of cells with their absolute addresses. In this grid system each cell must have two unique address representing a row and a column. A two dimensional array is best suited for this representation. Internally each cell has four types of information as:

# cell value,
# cell content,
# display format and data type, and
# reference table for external reference.

As described earlier, a cell value is the result obtained by evaluating the contents of the cell . Thus the array representing the values of the grid system must be of numeric type. The cell contents are the formula, text, repeating text etc. Thus these must be represented by character type of storage and hence defined by the string type of array. Data type and display format informations can be represented by integer numbers with signs. Hence it is defined by the integer type of array. Reference table contains all the cell addresses for external reference to use imformation. Two type of reference can be possible; one the value reference of a cell by other cells containing formula and the other is the formula reference of a cell by

other cells containing values. This means that a cell value may be referenced by a formula in another cell and such a reference is the value of the original cell, not to its content( formula).

## 7.2 REFERENCE TABLE GENERATION

This table corresponds to a cell who has a value or a formula with external reference for recalculation or evaluation of formula. In this case of a cell having a value, the reference table has all the cell addresses who use this cell value in their formula. In the case of a cell having a formula, the reference table represents all the cell addresses whose values are used by this cell. The linking between the host cell and the referenced cell can be distinguised by two ways: one is the direct linking, where the host cell directly uses the value of the referenced cell and other is the indirect linking, where the host cell uses a cell value who used the value of another cell. In the indirect linking the host cell reference table has the address of the cell having a value of value type and the referenced cell have a reference table with the address of the indirect host cell. For example, we consider the following cell-informations.

| Cell ID | Value | Formula | Reference table |
|---|---|---|---|
| A1 | 10 | - | B5,A4 |
| B5 | 30 | A1+20 | A1 |
| C2 | 40 | - | A4 |
| A4 | 1200 | B5*C2 | A1,C2 |

Here A1 and C2 have values of 10 and 40 respectively and these cells are of value type. B5 and A4 have values of 30 , 1200 and they are of formula type. The values of B5 and A4 are the evaluated values of their formulas. Now the reference table of A1 indicates the name of the cell B5 and A4 because these cells use the value of the cell A1. The reference table of B5 has the reference of cell A1 only because the cell B5 uses the value of A1 only. Similarly the reference table of A4 shows the addresses of cells A1 and C2 because the cell A4 uses indirectly the value of cell A1 and directly the value of the cell C2. Thus any change in value of A1 causes the change of the evaluated values of the cells B5 and A4. In this way a long chain of indirect reference of a value can be made possible.

### 7.2.1 Algorithm CELLREF

The input parameters of this algorithm are the active cell address (here active cell is the host cell ) and the referenced cell address. In this Algorithm a recalculation key is checked for the execution of the algorithm. Here active cell address is represented as AR and AC. Similarly the referenced cell address is represented by RR and RC. Recalculation key is denoted by RECKEY. CRET is the reference table array. Here CCON is the procedure used to convert a cell address to the corresponding cell identifier.

```
1.[ Check the recalculation key for exit or not ]
     If RECKEY = true then goto step-8;

2.[ Initialize the reference table index ]
     Set i <-- 1;

3.[ Check the data type of the cell ]
    If ABS( FMT[AR,AC]) = vt then  set
    CRET[AR,AC]<--CRET[AR,AC]+JUSTIFICATION(CCON[RR,RC],5,0);
    and goto step-5;

4.[ Isolate the identifier for the reference table ]
    Set CID <-- SUB(CRET[RR,RC],i,5) and call CELLID(CID);

5.[ Check the existance of the host cell address ]
    If INDEX(JUSTIFICATION(CCON[AR,AC],5,0),CRET[R,C],1)<>0
    then goto step-6;
    otherwise set
    CRET[R,C] <-- CRET[R,C]+JUSTIFICATION(CCON[AR,AC],5,0);

6.[ Check the existance of the cell address in the table ]
    If INDEX(CID,CRET[AR,AC],1) <> 0 then goto step-7;
    otherwise set CRET[AR,AC] <-- CRET[AR,AC]+CID;

7.[ Increment the index and check for further processin ]
    Set i <-- i+1; and
    if I <= LENGTH(CRET[RR,RC] then repeat from step-4;

8. [ Finished ]
     Exit.
```

### 7.2.2 Algorithm RECALCULATION

This algorithm uses a table for recalculation of formulas referenced in the table. Here STABLE is the source table given to represent the cell's list for recalculation. In this algorithm each element of the source table is translated and the corresponding formula is evaluated by the expression interpreter named EXPR_TER. FORM is the array

representing the system formulas and CVAL is the array representing the system values. CVAL is of string type so the RESULT returned by the expression interpreter must be converted to string before storing. Recalculation key is initially set to true and finally set to false at the end of the recalculation algorithm.

*1. [ Initialize the recalculation key and the table index ]*
*Set RECKEY <-- true; i <-- 1;*

*2. [ Isolate the cell address and translate the address ]*
*Set CID <-- SUB( STABLE,i,5); and [r,c] <-- CELLID(CID);*

*3. [ Assign the cell indexes returned ]*
*Set RI <-- r; and CJ <-- c;*

*4. [ Load the identified formula and call the interpreter ]*
*Set SOURCE <-- FORM[RI,CJ]; and call EXPR_TER;*

*5. [ Store the new value of the cell ]*
*Set CVAL[RI,CJ] <-- STRING(RESULT);*

*6. [ Advance the pointer and check for further looping ]*
*Set i <-- i+1;*
*If i <= LENGTH(STABLE) then repeat fron step-2;*

*7. [ reset the recalculation key ]*
*Set RECKEY <-- false;*

*8. [ Finished ]*
*Exit.*

## 7.3 Algorithm HPRINT

This algorithm prints the display contents of the cells defined by the user. The contents may be of value type or text type or formula type. The contents are printed by the hard-copy printer according to the formats specified.

This algorithm first sets the printing width which must be within the physical width of the printer. Then the column border string is calculated if the border key is found on and the infomations corresponding to this string are sent to the printer buffer for printing. This column border string is set (as defined before in the algorithm COLUMN) by considering the printing width and the individual width of each cell defined within the range. Then if the border key is found on , row border string is taken into consideration otherwise not for the printing of the cell contents. Then the cells of each row defined within the range for printing are arranged according to their specified formats. This calculated row with or without the row identifier string is sent to the printer buffer for printing. Then the current row is compared with the given row range and the result determines whether the next row will be calculated or not. If found not then exits from the algorithm otherwise continues the printing process by taking another row. Each parameters and the variables used in this algorithm have the same meaning as described in the previous algorithms.

*1.[ Initialize ]*
*Set j <-- LP; k <-- WIDTH(j); CSPAN <-- (CEND-CSTART);*
*CBORDER <-- SPACE(4); i <-- NP; and r <-- RSTART;*

*2.[ Check the column number and set the border ]*
*If j<27 then set FIRST <--CHR(32);1<--j and goto step-3;*
*otherwise if J<53 then*
*set FIRST <-- CHR(65); 1 <-- (j-26); and goto step-3;*
*otherwise ifj<79 then*
*set FIRST <-- CHR(66); 1 <-- (j-52); and goto step-3;*
*otherwise print error message and exit*

```
3.[ Set the width and check for odd or even ]
    Set w <-- WIDTH(j);
    If MOD(w,2) = 0 then
    set w <-- (w-1); and NAME <-- CHR(32)+FIRST;
    otherwise set NAME <-- FIRST;

4.[ Set the blank character number for each cell ]
    Set m <-- INTEGER(w*0.5)-1;

5.[ Set the column border string ]
    Set CBORDER <-- CBORDER+CHR(d)+SPACE(m-1)+NAME+CHR(64+1)
                    +SPACE(m)+CHR(d);

6.[ Increment the letter pointer & set the width & check ]
    Set j <-- j+1; and k <-- k+WIDTH(j);
    If k <= CSPAN then repeat from step-2;

7.[ Print the border string ]
    PRINT (ROW-1,COLUMN-4,JUSTIFICATION(CBORDER,CSPAN+4,0));

8.[ Set the row pointer ]
    Set i <-- i+1;

9.[ Check the border key and set the string ]
    If BKEY = true then set
    ROW <-- JUSTIFICATION(STRING(r),3,1)+CHR(b);
    otherwise set ROW <-- SPACE(4);

10.[ Reset the column number and set the cell width ]
    Set j <-- LP; and k <-- 0;

11.[ Check the column number and set the row ]
    Set k <-- k+WIDTH(j);
    If k > CSPAN then goto step-14;

12.[ Check the cell format of the row and adjust string ]
    If SIGN( FMT[i,j]) = -1 then set
    ROWSTR <-- ROWSTR+JUSTIFICATION(CVAL[i,j],WIDTH[j],1);
    otherwise set
    ROWSTR <-- ROWSTR+JUSTIFICATION(CVAL[i,j],WIDTH[j],0);

13.[ Increment the column pointer and repeat the loop ]
    Set j <-- j+1 and repeat from step-11;

14.[ Print the row line ]
    PRINT (   ROWSTR)

15.[ Increment the row pointer and check for looping ]
    Set r <-- r+1; and
    If r < ROWEND then repeat from step-8;

16.[ Finished ]
    Exit.
```

## 7.4 Algorithm SAVE

This algorithm stores all the necessary informations related to the cell system including the contents defined by the user, in an auxilliary storage unit. All the saved information can be retrieved for modification or extension. This algorithm also saves some system setting informations related to the cells. Each column width is important for loading a file which had been saved before. A reference cell idetification is also important to resume to the previous condition. Now the most important parameter to save is the table containing all the cell's address having some kind of information.This table is subdivided into three subtables depending on the type of the information stored into the cells. So from this table we can get information about the empty and nonempty cell and also get information about the type of data hold by each nonempty cell. Finally the informations to be save are the contents of each nonempty cell.Here the "cell contents" means all type of informations each cell contains.

Here width of the cells are stored in a string delimited by a character code(say "¦"). This method of saving reduces the storage area. Here in this algorithm WSTR is the string created by the concatenated column widths with a delimiter character dc. COLNUM is the number of columns defined by the user or system. TOPCELL is the top-left cell identifier of the current window as the reference cell. TTABLE, VTABLE and FTABLE are the tables containing the cell identifiers corresponding to the text, value and formula type of cells. Here PUT is a procedure to print the argument content to a file just created. And we assume that the informations are

stored sequentially into the created file and each PUT statement represents the creation of each record of the created file. CVAL, CELL, FMT, and CREF are the rectangular arrays to represent the value, content, format and the reference table informations respectively.

```
1. [ Initialize the width string to null ]
     Set WSTR <-- NUL; i <-- 1;

2. [ Concatenate the cell with with a delimiter character ]
     Set WSTR <-- WSTR + dc + STRING(WIDTH[i]);

3. [ Increment the column pointer and check for the last ]
     Set i <-- i+1; and
     If i <= COLNUM then repeat step-2;

4. [ Create the file and store the infomations ]
     CREATE FILENAME;
     PUT (WSTR, TOPCELL);
     PUT (TTABLE);
     PUT (VTABLE);
     PUT (FTABLE);

5. [ Check the text cell table whether it is empty or not ]
     If LENGTH(TTABLE) <> 0 then goto step-7;
     otherwise set i <--1;

6. [ Store the text cell's informations ]
     Set CELL <-- SUB(TTABLE,i,i+5);
        [r,c] <-- CELLID(CELL);
     PUT (CVAL[r,c], CELL[r,c]);
     Set    i <-- i+5; and
     repeat step-6 until i < LENGTH(TTABLE);

7. [ Check the value cell table whether it is empty or not ]
     If LENGTH(VTABLE) <> 0 then goto step-9;
     otherwise set i <--1;
```

```
8. [ Store the value cell's informations ]
   Set CELL <-- SUB(VTABLE,i,i+5);
      [r,c] <-- CELLID(CELL);
   PUT (CVAL[r,c], CELL[r,c], CREF[r,c]);
   Set   i <-- i+5; and
   repeat step-8 until i < LENGTH(VTABLE);

9. [ Check the formula cell table whether it is empty or not ]
   If LENGTH(FTABLE) <> 0 then goto step-7;
   otherwise set i <--1;

10. [ Store the formula cell's informations ]
    Set CELL <-- SUB(FTABLE,i,i+5);
       [r,c] <-- CELLID(CELL);
    PUT (CVAL[r,c], CELL[r,c], FMT[r,c], CREF[r,c]);
    Set   i <-- i+5; and
    repeat step-10 until i < LENGTH(FTABLE);

11. [ Close the file and exit ]
    CLOSE ;
    Exit.
```

## 7.5 Algorithm LOAD

This algorithm is the reverse procedure as defined by the algorithm SAVE. All parameters are the same as defined in the algorithm SAVE. Here GET is a procedure to retrieve information from a file just opened. Here we also assume that the file is of sequential type and each GET statement reads one record from the opened file.

```
1. [ Open the file and read the data ]
   OPEN FILENAME;
   GET (WSTR, TOPCELL);
   GET (TTABLE);
   GET (VTABLE);
   GET (FTABLE);

2. [ Check the text cell table whether it is empty or not ]
   If LENGTH(TTABLE) <> then goto step-4
   otherwise set i <-- 1;
```

```
3. [ Read the informations of each text cell ]
   Set CELL <-- SUB(TTABLE,i,i+5); and
      [r,c] <-- CELLID(CELL);
   GET (CVAL[r,c], CELL[r,c]);
   Set i <-- i+5 and repeat step-3 until i < LENGTH(TTABLE);

4. [ Check the value cell table whether it is empty or not ]
   If LENGTH(VTABLE) <> then goto step-6
   otherwise set i <-- 1;

5. [ Read the informations of each text cell ]
   Set CELL <-- SUB(VTABLE,i,i+5); and
      [r,c] <-- CELLID(CELL);
   GET (CVAL[r,c], CELL[r,c]);
   Set i <-- i+5 and repeat step-3 until i < LENGTH(VTABLE);

6. [ Check the formula cell table whether it is empty or not ]
   If LENGTH(FTABLE) <> then goto step-8
   otherwise set i <-- 1;

7. [ Read the informations of each text cell ]
   Set CELL <-- SUB(FTABLE,i,i+5); and
      [r,c] <-- CELLID(CELL);
   GET (CVAL[r,c], CELL[r,c]);
   Set i <-- i+5 and repeat step-3 until i < LENGTH(FTABLE);

8. [ Initialize the column with array pointer ]
   Set i <-- 1;

9. [ Initialize the column widths from the width string ]
   Set    j <-- INDEX(WSTR,dc,1);
   WIDTH[i] <-- VALUE(SUB(WSTR,1,j-1)); and
       WSTR <-- SUB(WSTR,j+1,LENGTH(WSTR));
      Set i <-- i+1; and repeat step-9 until i < COLNUM;

10. [ Initialize the window parameters ]
    Set [r,c] <-- CELLID(TOPCELL);
           AR <-- IR <-- r; AC <-- IC <-- c;
           CR <-- RSTART; and CC <-- CSTART;

11. [ Finished ]
    Exit.
```

CHAPTER-8

# CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

## 8.1 CONCLUSION

Algorithms and a modular software program have been developed for processing a large number of information cells interrelated with high level program statements. The processing of the cells involves analysis of high level program statements including assignments, arithmetic expressions, logical and conditional expressions. The information or data types handled by the cells consist of value, text string, formula etc. The developed system is a powerful tool for solving many types of mathematical, business, and financial problems. In designing the system, the structured approach has been followed which integrates a number of seperate algorithms.

The developed algorithms may be used for the development of sophisticated application program softwares such as,

- Electronic spread-sheet developing programs,
- Text editor or word processor developing programs,
- Compiler writting programs,
- Interpreter writting programs etc.

The developed system is an aggregate of concurrently active objects, organized into a rectangular array of cells similar to the paper spreadsheet used by an accountant. Each cell has a rule specifying how its value is to be determined. Every time a value is changed anywhere in the cell system, all values dependent on it are recomputed instantly and the new values are stored and displayed. The system can be defined as a simulated pocket universe that continuously maintains its fabric; it is a kit for a surprising range of applications. Here the user illusion is simple, direct and powerful. There are few mystifying surprises because the only way a cell can get a value is by having the cell's own value rule put in there. Thus the developed software module acts as a simple means to tap the power of a computer to do time-consuming, repetative calculations.

This is a microcomputer based system and the program has been developed by the CBASIC compiler language. In the program, errors are trapped and the proper messages are appeared on the display screen where necessary. The system commands are completely menu driven and thus it is user friendly. A user with minimum knowledge can use the system. User need not to know the inside processing activity of the system program. He or she should only understand the meaning of the error message sent by the system.

The developed program module may be used for the following applications,

. Balance sheets,
. Cash flow analysis/ forecasting,
. Job cost estimates,
. Patient records,
. Profit/sales projections/statements,
. General ledger,
. Inventry control,
. Market share analysis and planning,
. Project budgeting and control,
. Salary records,
. Tax estimation etc.

## 8.2 SUGGESTIONS FOR FUTURE WORK

A generalized system program software has been developed to facilitate the basic control functions and the most common used features. No new system can claim as a perfect one. This system may have some difficulties and undetected errors. But the system has been developed keeping a large number of flexibilities which can facilitate some addition or changes by giving a minimum effort. For the expansion and enhancment of the system to incorporate more complex functions and facilities, the following points can be taken into consideration for priority basis;

# Automatic adjustment of the formulas during insertion, deletion or arrangemrnt of a subset of cells.
# Special data format facilities by defining different format codes.
# Complex logical expressions handling by developing a complex logical expression handler routine.

# Calender function facilities by developing a procedure to interface with the system clock routine.
# And many other facilities according to the user's requirements.

(5-10)

The developed algorithms have been implemented in the form of the program modules by using a comprehensive and versatile programming language called CBASIC compiler. The main limitation of this compiler language is the lack of structural environment. The algorithms are written in a structured manner irrespective to the coding language. By using other high level compiler language one can implement the algorithms in a more structured and convenient manner.

# APPENDIX–A

## ASCII CODES

| ASCII Value | Character | Control Character |
|---|---|---|
| 000 | (null) | NUL |
| 001 | ☺ | SOH |
| 002 | ● | STX |
| 003 | ♥ | EXT |
| 004 | ♦ | EOT |
| 005 | ♣ | ENQ |
| 006 | ♠ | ACK |
| 007 | (beep) | BEL |
| 008 | | BS |
| 009 | (tab) | HT |
| 010 | (line feed) | LF |
| 011 | (home) | VT |
| 012 | (form feed) | FF |
| 013 | (carriage return) | CR |
| 014 | ♫ | SO |
| 015 | ☼ | SI |
| 016 | ► | SLE |
| 017 | ◄ | DC1 |
| 018 | ↕ | DC2 |
| 019 | !! | OC3 |
| 020 | ¶ | DC4 |
| 021 | § | NAK |
| 022 | ▬ | SYN |
| 023 | ↨ | ETB |
| 024 | ↑ | CAN |
| 025 | ↓ | EM |
| 026 | → | SUB |
| 027 | ← | ESC |
| 028 | (cursor right) | FS |
| 029 | (cursor left) | GS |
| 030 | (cursor up) | RS |
| 031 | (cursor down) | US |

| ASCII Value | Character |
|---|---|
| 032 | (space) |
| 033 | ! |
| 034 | " |
| 035 | # |
| 036 | $ |
| 037 | % |
| 038 | & |
| 039 | ' |
| 040 | ( |
| 041 | ) |
| 042 | * |
| 043 | + |
| 044 | , |
| 045 | - |
| 046 | . |
| 047 | / |
| 048 | 0 |
| 049 | 1 |
| 050 | 2 |
| 051 | 3 |
| 052 | 4 |
| 053 | 5 |
| 054 | 6 |
| 055 | 7 |
| 056 | 8 |
| 057 | 9 |
| 058 | : |
| 059 | ; |
| 060 | < |
| 061 | = |
| 062 | > |
| 063 | ? |

| ASCII Value | Character |
|---|---|
| 064 | @ |
| 065 | A |
| 066 | B |
| 067 | C |
| 068 | D |
| 069 | E |
| 070 | F |
| 071 | G |
| 072 | H |
| 073 | I |
| 074 | J |
| 075 | K |
| 076 | L |
| 077 | M |
| 078 | N |
| 079 | O |
| 080 | P |
| 081 | Q |
| 082 | R |
| 083 | S |
| 084 | T |
| 085 | U |
| 086 | V |
| 087 | W |
| 088 | X |
| 089 | Y |
| 090 | Z |
| 091 | [ |
| 092 | \ |
| 093 | ] |
| 094 | ^ |
| 095 | - |

| ASCII Value | Character |
|---|---|
| 096 | ` |
| 097 | a |
| 098 | b |
| 099 | c |
| 100 | d |
| 101 | e |
| 102 | f |
| 103 | g |
| 104 | h |
| 105 | i |
| 106 | j |
| 107 | k |
| 108 | l |
| 109 | m |
| 110 | n |
| 111 | o |
| 112 | p |
| 113 | q |
| 114 | r |
| 115 | s |
| 116 | t |
| 117 | u |
| 118 | v |
| 119 | w |
| 120 | x |
| 121 | y |
| 122 | z |
| 123 | { |
| 124 | \| |
| 125 | } |
| 126 | ~ |
| 127 | Δ |

# APPENDIX-B

## INTERNAL DATA REPRESENTATION

CBASIC machine-level representation varies somewhat for real numbers, integers, strings, and arrays.

REAL NUMBERS are stored in binary coded decimal (BCD) floating-point form. Each real number occupies eight bytes of memory storage space, as shown in Figure B-1. The high-order bit in the first byte (byte 0) contains the sign of the number. The remaining seven bits in byte 0 contain a decimal exponent. The exponent is a binary number representing a power of ten. The number is biased by 40H. Therefore, an exponent value of 42H represents an actual exponent of 2. Bytes 1 through 7 contain the mantissa. Two BCD digits occupy each of the seven bytes in the mantissa. The most significant digit of the number is stored in byte 7, furthest from the exponent. The floating decimal point is always situated to the left of the most significant digit.

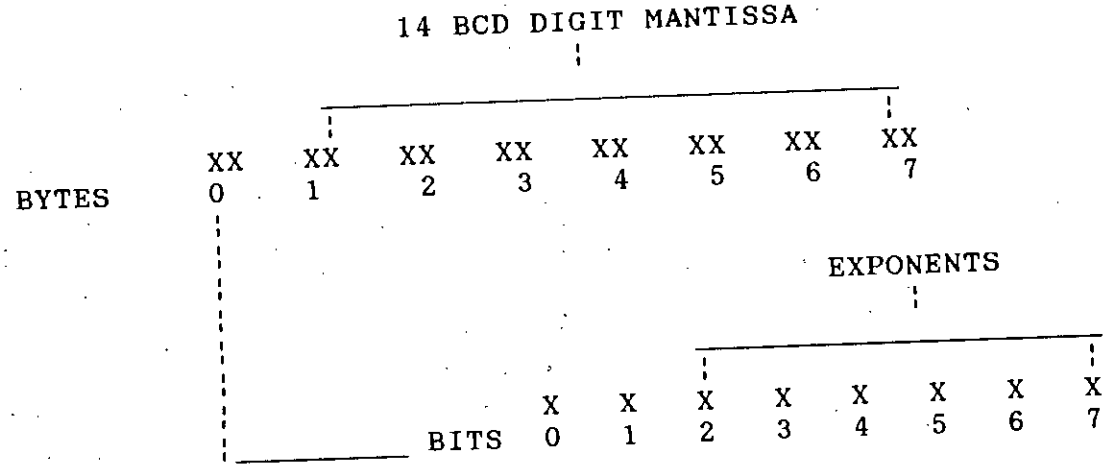


Figure-B-1 Real Number Storage

INTEGERS are stored in two bytes of memory space with the low order byte first, as shown in Figure B-2. Integers are represented as 16-bit, two's complement binary numbers. integer values range from -32768 to +32767, inclusive.

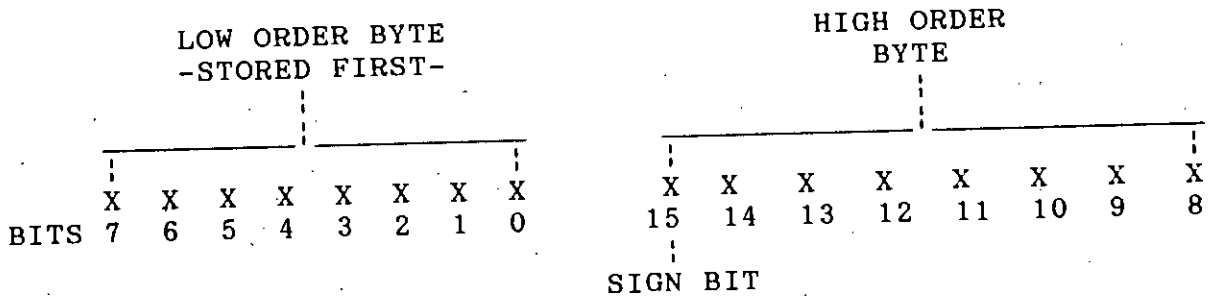


Figure-B.2 Integer Storage

STRINGS are stored as a sequence of ASCII characters. The length of a string is stored in the first two bytes followed by the actual ASCII values as shown in the following figure. The high-order length byte is stored first. The maximum number of character in a string is 32,762. CBASIC Compiler allocates space in the Dynamic Storage Area for strings. A pointer in the Data Area is an address in the DSA for the actual string.

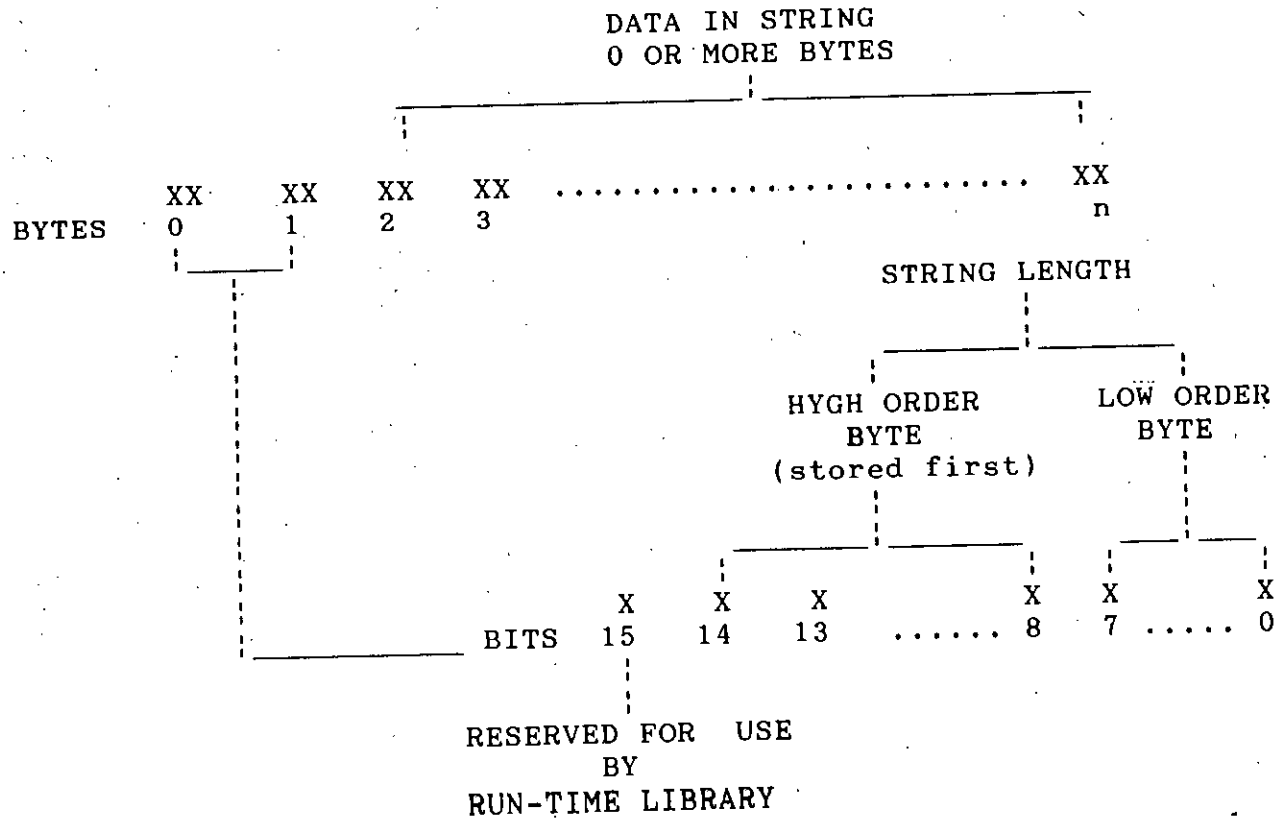


Figure-B.3 String storage

ARRAYS, both numeric and string, are allocated space in the Dynamic storage Area as required. Eight bytes are reserved for each element of an array containing real numbers and two bytes for each element of an integer array. String arrays are allocated two bytes for each entry plus the sum of all the string elements.

At some point in a program it might be necessary to free memory space allocated to arrays that are no longer needed in the program. Freeing numeric array space requires the redimension the array to zero. However, freeing string array space is a two step process. First, all the string array elements are setted to null. This is done by setting all the string array elements equal to a string variable that has never been assigned a value. A variable named NULLS can be used in such a case and on the assumption that NULLS has never been used as an array variable.

# APPENDIX-C

## CBASIC COMPILER

A compiler is a computer program that translates high-level programming language instructions into machine-executable code. The compiler takes as input a user-written source program and produces as output a machine-level object program. Some compilers translate a user-written source program into a program that a computer can execute directly. The CBASIC Compiler system, however, uses a link editor and a libary in addition to the compiler. Together the three components translate the CBASIC source-code file into a directly executable program. This approach uses the microcomputer's memory space as efficiently as possible. The system enables us to modularize programs for quick and easy maintenance. The result is a programming system that rivals the performance of systems based on much larger machines.

The primary advantage that compilers provide over other methods of translation is speed. Compiled applications programs execute faster than interpreted programs because the compiler creates a program that the computer can execute diredtly.

The compiler, CB86, translates CBASIC source code into relocatable machine code modules. Source programs default to a .BAS filetype unless otherwise specified. CB86 generates .OBJ files. CB86 consists of an executable program and three overlays.

The link editor, LINK86, combines the relocatable object modules that the compiler creates and routines from the indexed library into a directly executable program with optional overlays. LINK86 generates .CMD files.

The indexed libary, CB86.L86 , provides routines that allocate and release memory, detrermine available memory space, and perform input/output processing. CBASIC Compiler provides a library manager utility program, LIB86.

# APPENDIX–D

## COMPLETE PROGRAM LISTING

```
rem ************************************************************
rem *                                                          *
rem *              STORAGE AND DATA DECLARATIONS               *
rem *                                                          *
rem ************************************************************
rem **** rectangular array range initialization
         CLIMIT% = 35: RLIMIT% = 110
rem **** storage declaration for the data structure
         DIM X$(RLIMIT%,CLIMIT%),XY$(RLIMIT%,CLIMIT%)
         DIM CD$(RLIMIT%,CLIMIT%),YY%(RLIMIT%,CLIMIT%)
         DIM W%(CLIMIT%),WS$(30),M$(4)
rem **** initialize the default cell widths
         W%(1) = 9: FOR I% = 2 TO CLIMIT%: W%(I%) = W%(1): NEXT I%
rem **** set the control code strings
         CURSTR$ = "23 19 26 1  "
         DTYSTR$ = "128129130131"
         COMSTR$ = "47 33 61 3  133 134 135 136"
         INCODE$ = "13 8  137 138 1  19 136 133"
rem **** set the data type indicator string
         M$(1) = "VALUE": M$(2) = "TEXT": M$(3) = "R-TEXT"
         M$(1) = "FORMULA"
          FM$ = "FORMULA: ": VTAB$ = "": FTAB$ = ""
rem **** define the delimiter character
         CLON$ = ":": SLON$ = ";": COM$ = ",": LPR$ = "("
         RPR$ = ")"
rem **** initialize the data structure pointers
         IIR% = 1: IIC% = 1: R2% = 1: C2% = 1
rem **** initialize the window setting parameters
         RST% = 2: RED% = 20: CST% = 5: CED% = 81
rem **** initialize the various key setting
         MR% = 2: MC% = 5: RECKEY% = 0
         KEY% = 4: MKEY% = 1: RT% = 62: SPLIT% = 0
         RJUS% = 0: CSEC% = 0: KEY% = 0: BOR% = 1: SW% = 0
rem ************************************************************
rem **** declarations for various arrays
         DIM O$(7),B$(20),TS(10),S$(9),BFUN$(5),LFUN$(6),LRE$(9)\
         TK%(10),FN%(10),MX$(20),RLMT$(5),LLMT$(5),B$(25),STK%(3)
rem **** define strings for letters, digits and operators
         LETTER$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
         DIGIT$ = "1234567890."
         OPERATOR$ = "-+/*^"
rem **** define the library function keyword string
         LFUNCTION$ = "ABSEXPINTLOGRNDSINCOSTANCSCSECCOT"
rem **** table for operators and the parantheses
         FOR I% = 1 TO 7: READ O$(I%): NEXT I%
         DATA -,+,/,*,^,(,)
rem **** table for temporary storage variables
         FOR I% = 1 TO 9: READ S$(I%): NEXT I%
         DATA TS1,TS2,TS3,TS4,TS5,TS6,TS7,TS8,TS9
rem **** table for the built-in function keywords
         FOR J% = 1 TO 5: READ BFUN$(J%):NEXT J%
         DATA MAX,MIN,SUM,AVG,CNT
```

```
rem **** table for the boolean function keywords
         FOR I% = 1 TO 6 : READ LFUN$(I%): NEXT I%
         DATA IF,NOT,AND,OR,EQU,NEQ
rem **** table for the relational operators
         FOR I% = 1 TO 9 : READ LRE$(I%): NEXT I%
         DATA <>,><,>=,=>,<=,=<,=,>,<
rem ************************************************************
rem *                    FUNCTION DEFINITIONS                  *
rem ************************************************************
rem **** key initialization
         DEF KESET(KEY,COD)
         PRINT CHR$(27)+CHR$(58)+CHR$(KEY)+CHR$(COD)+CHR$(00)
         FEND
rem **** clearing the screen
         DEF CLS
         PRINT CHR$(12)
         FEND
rem **** back ground color setting
         DEF BCOL(TPE%)
         PRINT CHR$(27)+CHR$(99)+CHR$(TPE%);
         FEND
rem **** fore ground color setting
         DEF FCOL(TYP%)
         PRINT CHR$(27)+CHR$(98)+CHR$(TYP%);
         FEND
rem **** blank space string  setting
         DEF BK$(NO%)
         BK$ = STRING$(NO%,CHR$(32))
         FEND
rem **** string printing at a particular position of the screen
         DEF P(YCOR%,XCOR%,S$)
         PRINT CHR$(27)+CHR$(89)+CHR$(31+YCOR%)+CHR$(31+XCOR%)+S$;
         FEND
rem **** data type indication string printing
         DEF SP%(MK%)
         CALL P(22,1,BK$(7)):CALL FCOL(4):CALL BCOL(10)
         CALL P(22,1,M$(MK%)):CALL FCOL(7):CALL BCOL(0)
         FEND
rem **** back spacing of the cursor
         DEF BS$(WQ$)
         L% = LEN(WQ$): IF L% = 0 THEN RETURN
         BS$ = LEFT$(WQ$,L%-1)
         FEND
rem **** replace a character in a string
         DEF RECH$(SR$,RS1$,IQ1%)
         I% = LEN(SR$)
         RECH$ = LEFT$(SR$,IQ1%-1)+RS1$+MID$(SR$,IQ1%+1,I%)
         FEND
rem **** delete a character fron a string
         DEF DECH$(SD$,IQ2%)
         I% = LEN(SD$)
         DECH$ = LEFT$(SD$,IQ2%-1)+MID$(SD$,IQ2%+1,I%)
         FEND
rem **** insert a character from a string
         DEF INCH$(SI$,RS2$,IQ3%)
```

```
        I% = LEN(SI$)
        INCH$ = LEFT$(SI$,IQ3%-1)+RS2$+MID$(SI$,IQ3%,I%)
        FEND
rem **** set the function keys
        CALL KESET(77,19):CALL KESET(59,128):CALL KESET(60,129)
        CALL KESET(75,01):CALL KESET(61,130):CALL KESET(62,131)
        CALL KESET(72,23):CALL KESET(63,132):CALL KESET(71,133)
        CALL KESET(80,26):CALL KESET(73,134):CALL KESET(81,135)
        CALL KESET(79,136):CALL KESET(82,137):CALL KESET(83,138)
rem ************************************************************
rem *                    ROW PRINTING PROCEDURE                *
rem ************************************************************
        DEF ROWP(COLUMN%,ROWST%,ROWED%,NPOINTER%)
        CALL FCOL(2222): CALL BCOL(1)
        FOR I% = ROWST% TO ROWED%
          CALL P(I%,COLUMN%-4,J$(STR$(NPOINTER%),3,1)+CHR$(124))
          NPOINTER% = NPOINTER%+1
        NEXT I%
        CALL FCOL(07): CALL BCOL(16)
        FEND
rem ************************************************************
rem *                 COLUMN PRINTING PROCEDURE                *
rem ************************************************************
        DEF COLUMNP(ROW%,COLSTART%,COLEND%,LPOINTER%,PTER%)
        IF PTER% = 0 THEN CALL FCOL(2222): CALL BCOL(1)
        COLDIFF% = COLEND%-COLSTART%
        JJ% = W%(LPOINTER%): H$ = "     "
        WHILE JJ% <= COLDIFF%
          IF LPOINTER% < 27 THEN E$ = CHR$(32): CHP% = LPOINTER%\
          ELSE E$ = CHR$(65): CHP% = LPOINTER%-26
          WITH% = W%(LPOINTER%)
          IF MOD(WITH%,2) = 0 THEN WITH%=WITH%-1: SP$ = CHR$(32)\
          +E$ ELSE SP$ = E$
          B% = INT((WITH%)*0.5)-1
          H$ = H$+CHR$(124)+BK$(B%-1)+SP$+CHR$(CHP%+64)+ \
          BK$(B%)+CHR$(124)
          LPOINTER% = LPOINTER%+1
          JJ% = JJ%+W%(LPOINTER%)
        WEND
        IF PTER% = 1 THEN PRINT J$(H$,COLDIFF%+4,0): RETURN
        CALL P(ROW%-1,COLSTART%-4,J$(H$,COLDIFF%+4,0))
        CALL FCOL(11): CALL BCOL(16)
        FEND
rem ************************************************************
rem *                   WINDOW SCANNING PROCEDURE              *
rem ************************************************************
        DEF ARRAY(CSA%,CEA%,RSA%,REA%,CA%,RA%)
        CDIF% = (CEA%-CSA%): ICA% = CA%: CALL FCOL(11)
        FOR R% = RSA% TO REA%
          CA% = ICA%:C% = W%(CA%):T$ = ""
          WHILE C% <= CDIF%
            IF SGN(YY%(RA%,CA%)) = -1 THEN \
            T$ = T$+J$(X$(RA%,CA%),W%(CA%),1) ELSE \
            T$ = T$+J$(X$(RA%,CA%),W%(CA%),0)
            CA% = CA%+1: C% = C%+W%(CA%)
```

```
            WEND
            CALL P(R%,CSA%,J$(T$,CDIF%,0))
            RA% = RA%+1
          NEXT R%
          CALL FCOL(7)
          FEND
rem ***************************************************************
rem *                  CELL IDENTIFIER PROCEDURE                  *
rem ***************************************************************
          DEF CCON$(XC%,YC%)
          IF YC% < 27 THEN FC$ = "": GOTO CC
          IF YC% < 53 THEN FC$ = CHR$(65): GOTO CC
          IF YC% < 79 THEN FC$ = CHR$(66)
CC:      CCON$ = FC$+CHR$(YC%+64)+STR$(XC%)
          FEND
rem ***************************************************************
rem *                   CURSOR PRINTING PROCEDURE                 *
rem ***************************************************************
          DEF CURSOR$(NPC%,LPC%,RP%,CP%)
          IF SGN(YY%(NPC%,LPC%)) = -1 THEN  \
          CSOR$ = J$(X$(NPC%,LPC%),W%(LPC%),1) ELSE \
          CSOR$ = J$(X$(NPC%,LPC%),W%(LPC%),0)
          CALL P(RP%,CP%,CSOR$)
          FEND
rem ***************************************************************
rem *                     DATA INPUT PROCEDURE                    *
rem ***************************************************************
         INDATA: V$ = ""
69.1     CALL P(23,3,CHR$(62)+V$)
         VV% = INKEY
         IF VV% = 13 THEN 69.3
         IF VV% = 8 THEN 69.2
         V$ = V$+CHR$(VV%): GOTO 69.1
69.2     IF LEN(V$) = 0 THEN BACK%=1: RETURN
         V$ = LEFT$(V$,LEN(V$)-1): GOTO 69.1
69.3     BACK% = 0: RETURN
rem ***************************************************************
rem *                 WIDTH CALCULATION PROCEDURE                 *
rem ***************************************************************
SETWIDTH:I% = IIC%: PS% = W%(I%)
          WHILE PS%+W%(I%+1) <= (CED%-CST%)
             PS% = PS%+W%(I%)
             I% = I%+1
          WEND
          LIMIT% = I%
          RETURN
rem ***************************************************************
rem *                    RECULCULATION PROCEDURE                  *
rem ***************************************************************
RECAL:    RECKEY% = 1: VP% = 1: LLL% = LEN(DUM$)
          WHILE VP% <= LLL%
             BB$ = MID$(DUM$,VP%,5)
             GOSUB CELLID: RI% = II%: CJ% = JJ%
             AA$=XY$(RI%,CJ%): GOSUB EXPTR
             X$(RI%,CJ%) = STR$(RESULT)
```

```
rem ************************************************************
rem *                  BLANK REMOVAL PROCEDURE                 *
rem ************************************************************
        DEF REBK$(S$)
        K% = 1
RE10:   I% = MATCH(" ",S$,K%)
        IF I% = 0 THEN REBK$ = S$: RETURN
        S$ = LEFT$(S$,I%-1)+RIGHT$(S$,LEN(S$)-I%)
        K% = I%: GOTO RE10
        FEND
rem ************************************************************
rem *                  JUSTIFICATION PROCEDURE                 *
rem ************************************************************
        DEF J$(IN$,SL%,K%)
        LENGTH% = LEN(IN$)
        IF K% = 0 THEN GOTO LJ0
        IF LENGTH% < SL% THEN J$ = STRING$(SL%-LENGTH%," ")+IN$\
        :RETURN
        IF LENGTH% > SL% THEN J$ = RIGHT$(IN$,SL%): RETURN
        J$ = IN$: RETURN
LJ0:    IF LENGTH% < SL% THEN J$ = IN$+STRING$(SL%-LENGTH%," ")\
        :RETURN
        IF LENGTH% > SL% THEN J$ = LEFT$(IN$,SL%): RETURN
        J$ = IN$
        FEND
rem ************************************************************
rem *                  NUMBER ROUNDING FUNCTION                *
rem ************************************************************
        DEF ROUND(NUM)
        TEM = NUM-INT(NUM)
        IF TEM >= 0.5 THEN ROUND = INT(NUM)+1: RETURN
        ROUND = INT(NUM)
        FEND
rem ************************************************************
rem *                  DIGIT CHECKING PROCEDURE                *
rem ************************************************************
        DEF DGCK%(C$)
        I% = 1
        WHILE I% <= LEN(C$)
           IN$ = MID$(C$,I%,1)
           IF MATCH(IN$,"0123456789.E+-",1)=0 THEN DGCK% = 0 \
           :RETURN
           I% = I%+1
        WEND
        DGCK% = 1
        FEND
rem ************************************************************
rem *                CELL IDENTIFICATION PROCEDURE             *
rem ************************************************************
CELLID: BB$ = REBK$(BB$): SPAN% = LEN(BB$)
        IF SPAN% > 5 THEN DERROR% = 1: GOTO EMES
        I% = MATCH(LEFT$(BB$,1),LETTER$,1)
        IF I% = 0 THEN DERROR%=1: GOTO EMES
        J% = MATCH(MID$(BB$,2,1),LETTER$,1)
        IF J% = 0 THEN JJ%=I%: II% = VAL(MID$(BB$,2,SPAN%))\
```

```
        ELSE JJ% = 26*I%+J%: II% = VAL(MID$(BB$,3,SPAN%))
        IF II% > RLIMIT% OR II% < 1 THEN DERROR%=2: GOTO EMES
        IF JJ% > CLIMIT% OR JJ% < 1 THEN DERROR%=3: GOTO EMES
        RETURN
rem ************************************************************
rem *                        ERROR ROUTINE                       *
rem ************************************************************
EMES:   ON DERROR% GOTO 555.1,555.2,555.3,555.4
555.1   E$ = "WRONG ENTRY":GOTO 555.0
555.2   E$ = "ROW ERROR":GOTO 555.0
555.3   E$ = "COLUMN ERROR":GOTO 555.0
555.4   E$ = "BRAKET MISSING"
555.0   CALL P(22,65,E$):GOTO LEV3
rem ************************************************************
rem *                  SCREEN MANIPULATING PROGRAM               *
rem ************************************************************
rem **** clear the screen and print the data type title
        CALL CLS : CALL SP%(1)
rem **** initially set the row and the column border
LEV0:   CALL ROWP(CST%,RST%,RED%,IIR%)
        CALL COLUMNP(RST%,CST%,CED%,IIC%,0)
rem **** switch to different levels for displaying the window
LEV:    ON KEY% GOTO LEV1, LEV2, LEV3, LEV4
rem **** print the row border if the border key is on
LEV1:   IF BOR%=1 THEN CALL ROWP(CST%,RST%,RED%,IIR%): GOTO LEV3
rem **** print the column border if the border key os on
LEV2:   IF BOR%=1 THEN CALL COLUMNP(RST%,CST%,CED%,IIC%,0)
rem **** scan the whole window with cell contents
LEV3:   CALL ARRAY(CST%,CED%,RST%,RED%,IIC%,IIR%)
rem **** set the colors and display the active cell cursor
LEV4:   CALL BCOL(16):CALL FCOL(11)
        CALL CURSOR$(R1%,C1%,MMR%,MMC%)
rem **** identify the reference cell of the window
        TOPCELL$ = CCON$(IIR%,IIC%)
rem **** erase the previous cursor of the window
        CALL BCOL(4): CALL CURSOR$(R2%,C2%,MR%,MC%)
        CALL BCOL(16)
rem **** identify the active cell and display on the status line
        CELL$ = CCON$(R2%,C2%): CALL P(21,2,CELL$)
rem **** set the active cell identifier for reference table
        CELLRF$ = J$(CELL$,5,0)
rem **** print the active cell cursor direction on the window
        CALL FCOL(12): CALL P(21,1,CHR$(RT%)+BK$(5))
        CALL FCOL(2222)
rem **** erase and print the cell content on the status line
        CALL P(21,39,BK$(LEN(XY$(R1%,C1%))))
        CALL P(21,39,XY$(R2%,C2%))
rem **** print the content heading on the status line
        CALL (12): CALL P(21,30,FM$): CALL FCOL(11)
rem ************************************************************
rem *                       SELECT PROCEDURE                     *
rem ************************************************************
SELT:   CALL P(23,1," 1"+CHR$(62)+BK$(L%))
        CALL P(23,4,""): CODE% = INKEY
        IF CODE% = 13 THEN 9.4 ELSE CK$ = J$(STR$(CODE%),3,0)
```

```
        IF CODE% = 13 THEN 9.4 ELSE CK$ = J$(STR$(CODE%),3,0)
        CURCODE% = MATCH(CK$,CURSTR$,1)
        IF CURCODE% <> 0 THEN  9.224
        DTYCODE% = MATCH(CK$,DTYSTR$,1)
        IF DTYCODE% <> 0 THEN  9.223
        COMCODE% = MATCH(CK$,COMSTR$,1)
        IF COMCODE% <> 0 THEN  9.222
        IF CODE% >= 32 OR CODE% <= 127 THEN 9.221
        GOTO SELT
rem ************************************************************
rem *                    DATA INPUT PROCEDURE                  *
rem ************************************************************
9.221   V$ = CHR$(CODE%): CALL SP%(MKEY%)
        J% = 2: INSK% = 0: D% = 1
37      UP$ = J$(STR$(J%),2,1): CALL P(23,1,UP$+">")
        IF D% = 1 THEN CALL P(23,4,V$+BK$(ABS(L%-LEN(V$))))
        L% = LEN(V$): CALL P(23,J%+3,"")
        D% = 1: CCODE% = INKEY: CHK$ = J$(STR$(CCODE%),3,0)
        M% = MATCH(CHK$,INCODE$,1)
        IF M% = 0 THEN 39.9 ELSE M% = (M%-1)/3+1
        ON M% GOTO 39.1,39.2,39.3,39.4,39.5,39.6,39.7,39.8
39.1    COMKEY% = 0: GOTO 9.3
39.2    IF J% = 1 THEN V$ = "": GOTO SELT  ELSE \
        V$ = MID$(V$,1,LEN(V$)-1): J% = J%-1: GOTO 37
39.3    IF INSK% = 1 THEN INSK% = 0 ELSE INSK% = 1: GOTO 37
39.4    V$ = DECH$(V$,J%): GOTO 37
39.5    D% = 0: IF J% > 1 THEN J% = J%-1: GOTO 37 ELSE GOTO 37
39.6    D% = 0
39.6    IF J% <= L% THEN J% = J%+1: GOTO 37 ELSE CCODE% = 32
39.7    J% = L%+1: GOTO 37
39.8    J% = 1: GOTO 37
39.9    IF CCODE% < 32 OR CCODE% > 127 THEN 37
        R$ = CHR$(CCODE%)
        IF J% = L% THEN V$ = V$+R$: J% = J%+1: GOTO 37
        IF INSK% = 1 THEN V$ = INCH$(V$,R$,J%) \
        ELSE V$ = RECH$(V$,R$,J%)
        J% = J%+1: GOTO 37
rem ************************************************************
rem *                    COMMAND PROCEDURE                     *
rem ************************************************************
9.222   I% = ((COMCODE%-1)/3)+1
        ON I% GOTO COMMAND,49.2,49.3,49.4,49.5,49.6,49.7,\
                   49.8,49.9
49.2    DUM$ = FTAB$: GOSUB RECAL: KEY% = 3: GOTO LEV
49.3    GOSUB CJUMP: KEY% = 3: GOTO LEV0
49.4    GOTO 5.0
49.5    GOSUB HOME: KEY% = 4: GOTO LEV
49.6    GOSUB PGUP: KEY% = 1: GOTO LEV
49.7    GOSUB PGDN: KEY% = 1: GOTO LEV
49.8    GOSUB SEND: KEY% = 1: GOTO LEV
49.9    GOTO XCHANGE
rem ************************************************************
rem *                  DATA TYPE SET PROCEDURE                 *
rem ************************************************************
9.223   MKEY% = ((DTYCODE%-1)/3)+1
```

```
        CALL SP%(MKEY%)
        GOTO SELT
rem ************************************************************
rem *                CURSOR CONTROL KEY SET PROCEDURE          *
rem ************************************************************
9.224   I%=((CURCODE%-1)/3)+1
        ON I% GOTO 44.1,44.2,44.3,44.4
44.1    RT% = 94: GOTO 9.4
44.2    RT% = 62: GOTO 9.4
44.3    RT% = 86: GOTO 9.4
44.4    RT% = 60: GOTO 9.4
rem ************************************************************
rem *                     MAIN PROCEDURE                       *
rem ************************************************************
rem **** input data assigned by the buffer variable
9.3     AA$=V$
rem **** branch for processing the data
        ON MKEY% GOTO VENTRY,TENTRY,RENTRY,FENTRY
rem ************************************************************
rem *                   TEXT DATA PROCESSING                   *
rem ************************************************************
TENTRY: XY$(R2%,C2%) = AA$: GOSUB TINDEX
        X$(R2%,C2%) = AA$: YY%(R2%,C2%) = 1: GOTO 9.4
rem ************************************************************
rem *              REPEATING TEXT DATA PROCESSING              *
rem ************************************************************
RENTRY: FOR I%=C2% TO 26
          XY$(R2%,I%) = AA$: X$(R2%,I%) = AA$
        NEXT I%
        GOSUB TINDEX: YY%(R2%,C2%) = 1
        CALL ARRAY(CST%,CED%,RST%,RED%,IIC%,IIR%)
        GOTO 9.4
rem ************************************************************
rem *                  VALUE DATA PROCESSING                   *
rem ************************************************************
VENTRY: IF DGCK%(REBK$(AA$)) = 0 THEN  DERROR% = 1: GOTO EMES
        X$(R2%,C2%)=AA$:IF YY%(R2%,C2%)=-2 THEN \
        :DUM$ = CD$(R2%,C2%): GOSUB RECAL \
        :CALL ARRAY(CST%,CED%,RST%,RED%,IIC%,IIR%): GOTO 9.4
        YY%(R2%,C2%) = -2: GOSUB VINDEX: GOTO 9.4
rem ************************************************************
rem *                   FORMULA PROCESSING                     *
rem ************************************************************
FENTRY: GOSUB EXPTR
        XY$(R2%,C2%) = V$: X$(R2%,C2%) = STR$(RESULT)
        YY%(R2%,C2%) = -3: GOSUB FINDEX
rem ************************************************************
rem *                 CURSOR CONTROL PROCEDURE                 *
rem ************************************************************
9.4     R1% = R2%: C1% = C2%: MMR% = MR%: MMC% = MC%
        CALL P(23,4,BK$(75))
        IF RT% = 60 THEN GOTO MOVEL
        IF RT% = 86 THEN GOTO MOVED
        IF RT% = 94 THEN GOTO MOVEU
MOVER:  IF C2% = CLIMIT% THEN KEY% = 4: GOTO LEV
```

```
        GOSUB SETWIDTH: MC% = CST%: CALL P(23,4,BK$(75))
        IF C2% = LIMIT% THEN GOTO MOVER1
        FOR I% = IIC% TO C2%: MC% = MC%+W%(I%): NEXT I%
        C2% = C2%+1: KEY% = 4: GOTO LEV
MOVER1: IIC% = IIC%+1: GOSUB SETWIDTH
        IF LIMIT% < C2%+1 THEN MOVER1
        FOR I% = IIC% TO C2%: MC% = MC%+W%(I%): NEXT I%
        C2% = C2%+1: KEY% = 2: GOTO LEV
MOVEL:  IF C2% = 1 THEN KEY% = 4: GOTO LEV
        IF MC% = CST% THEN IIC% = IIC%-1: C2% = C2%-1:\
        KEY% = 2: GOTO LEV
        MC% = MC%-W%(C2%-1): C2% = C2%-1: KEY% = 4: GOTO LEV
MOVED:  IF R2% = RLIMIT% THEN KEY% = 4: GOTO LEV
        IF MR% = RED% THEN IIR% = IIR%+1: R2% = R2%+1:\
        KEY% = 1: GOTO LEV
        MR% = MR%+1: R2% = R2%+1: KEY% = 4: GOTO LEV
MOVEU:  IF R2% = 1 THEN KEY% = 4: GOTO LEV
        IF MR% = RST% THEN IIR% = IIR%-1: R2% = R2%-1:\
        KEY% = 1: GOTO LEV
        MR% = MR%-1: R2% = R2%-1: KEY% = 4: GOTO LEV
rem ************************************************************
rem *                    SLASH COMMAND PROCEDURE                 *
rem ************************************************************
COMMAND: CALL P(22,1,BK$(80))
        CALL P(22,1,"/B,C,D,F,G,I,L,M,O,Q,R,S,W,X,Z")
        CALL P(23,4,BK$(PP%)):CALL P(23,3,">/")
        VC%=INKEY :CALL P(22,1,BK$(30))
        IF VC%=67 OR VC%=(67+32) THEN GOTO SPWDRAW
        IF VC%=70 OR VC%=(70+32) THEN GOTO FORMAT
        IF VC%=71 OR VC%=(71+32) THEN GOTO GLOBAL
        IF VC%=76 OR VC%=(76+32) THEN GOTO LOAD
        IF VC%=81 OR VC%=(81+32) THEN GOTO QUIT
        IF VC%=83 OR VC%=(83+32) THEN GOTO SAVE
        IF VC%=87 OR VC%=(87+32) THEN GOTO SPLITSC
        IF VC%=88 OR VC%=(88+32) THEN GOTO COMMAND
        IF VC%=79 OR VC%=(79+32) THEN GOTO HPRINT
        IF VC%=8 THEN CALL SP%(MKEY%):KEY%=4:GOTO LEV
        GOTO COMMAND
rem ************************************************************
rem *                      SAVE PROCEDURE                        *
rem ************************************************************
SAVE:   CALL P(22,1,"Enter the file name "):CALL P(23,5,CHR$(32))
        GOSUB INDATA: IF BACK = 1 THEN GOTO COMMAND
        CALL P(22,1,"Saving process continuing")
        WD$ = STR$(W%(1)): CALL P(23,5,BK$(LEN(V$)))
        CREATE V$ AS 1
        FOR I% = 2 TO 30: WD$ = WD$+"¦"+STR$(W%(I%)): NEXT I%
        WD$=WD$+"¦"
        PRINT #1; WD$,TOPCELL$
        PRINT #1; TTAB$
        PRINT #1; VTAB$
        PRINT #1; FTAB$
        TTT$ = TTAB$: GOSUB SV1
        TTT$ = VTAB$: GOSUB SV1
        TTT$ = FTAB$: GOSUB SV1
```

```
          CLOSE 1
          CALL P(22,1,BK$(26))
          KEY% = 4: GOTO LEV
REM ****************** SAVE CELL'S INFOMATION ********************
SV1:      FOR MP% = 1 TO LEN(TTT$) STEP 5
          BB$ = MID$(TTT$,MP%,5): GOSUB CELLID
          PRINT #1; XY$(II%,JJ%),X$(II%,JJ%),\
          CD$(II%,JJ%),YY%(II%,JJ%)
          NEXT MP%: RETURN
rem ***************************************************************
rem *                        LOAD PROCEDURE                       *
rem ***************************************************************
LOAD:     CALL P(22,1,"Enter the file name "):CALL P(23,5,CHR$(32))
          GOSUB INDATA: IF BACK = 1 THEN GOTO COMMAND
          CALL P(22,1,"Loading process continuing")
          CALL P(23,1,BK$(LEN(V$)))
          IF END #2 THEN 234
          OPEN V$ AS 2
          READ #2; WD$,TOPCELL$
          READ #2; TTAB$
          READ #2; VTAB$
          READ #2; FTAB$
          IF TTAB$ <> "" THEN TTT$ = TTAB$: GOSUB LD1
          IF VTAB$ <> "" THEN TTT$ = VTAB$: GOSUB LD1
          IF FTAB$ <> "" THEN TTT$ = FTAB$: GOSUB LD1
234       FOR I% = 1 TO 30
          J% = MATCH(";",WD$,1): W%(I%) = VAL(LEFT$(WD$,J%-1))
          WD$ = RIGHT$(WD$,LEN(WD$)-J%): NEXT I%
          BB$ = TOPCELL$: GOSUB CELLID: MR% = RST%: MC% = CST%
          IIR% = II%: IIC% = JJ%: R2% = II%: C2% = JJ%
          CALL P(22,1,BK$(26))
          KEY% = 1: GOTO LEV
rem **** load the cell's informations
LD1:      FOR MP% = 1 TO LEN(TTT$) STEP 5
          BB$ = MID$(TTT$,MP%,5): GOSUB CELLID
          READ #2; XY$(II%,JJ%),X$(II%,JJ%),\
          CD$(II%,JJ%),YY%(II%,JJ%)
          NEXT MP%: RETURN
rem ***************************************************************
rem *                    SPLIT SCREEN PROCEDURE                   *
rem ***************************************************************
SPLITSC:  IF SPLIT%=1 THEN \
          CALL P(22,1,"Split Screen Mode Active:Press any key")\
          :CALL P(23,3,">"):FAL%=INKEY:GOTO 9.2
          CALL P(22,1,"Vertical Or Horizontal:Press V Or H")
          CALL P(23,3,">/Split Screen,")
          VC71%=INKEY:CALL P(22,1,BK$(40)):CALL P(23,3,BK$(15))
          IF VC71%=72 OR VC71%=(72+32) THEN SPLIT%=1:GOTO 5.711
          IF VC71%=86 OR VC71%=(86+32) THEN SPLIT%=1:GOTO 5.712
          IF VC71%=8 THEN GOTO COMMAND
          GOTO SPLITSC
5.711     RST%=MR%:CSPLIT%=0:TOPC$=TOPCELL$
          IIR%=R2%:CALL P(RST%-1,1,BK$(3)):KEY%=3:GOTO LEV0
5.712     MC%=MC%+4:CST%=MC%:CSPLIT%=1:TOPC$=TOPCELL$
          IIC%=C2%:KEY%=3:GOTO LEV0
```

```
rem ************************************************************
rem *            SPLIT SCREEN WITHDRAW PROCEDURE                 *
rem ************************************************************
SPWDRAW: CALL P(22,1,"Withdraw Split Mode:Press Y or N")
         CALL P(23,4,""):VC72%=INKEY:CALL P(22,1,BK$(35))
         IF VC72%=89 OR VC72%=(89+32) THEN SPLIT%=0:GOTO 5.721
         IF VC72%=78 OR VC72%=(78+32) THEN KEY%=4:GOTO LEV
         IF VC72%=8 THEN GOTO COMMAND ELSE GOTO SPWDRAW
5.721    MR%=2:MC%=5:BB$=CELL$:GOSUB CELLID
         IIR%=II%:IIC%=JJ%:R2%=II%:C2%=JJ%
         RST%=2:CST%=5:RED%=20:CED%=81
         KEY%=3:GOTO LEV0
rem ************************************************************
rem *                      EXIT PROCEDURE                        *
rem ************************************************************
QUIT:    CALL P(22,1,"Exit from the spread-sheet ? Press Y or N")
         CALL P(23,3,">/Quit,")
         VC73=INKEY:CALL P(22,1,BK$(50)):CALL P(23,3,BK$(8))
         IF VC73=89 OR VC73=(89+32) THEN 5.0
         IF VC73=78 OR VC73=(78+32) THEN 9.2
         IF VC73=8 THEN GOTO COMMAND ELSE GOTO QUIT
rem ************************************************************
rem *                BORDER ON/OFF PROCEDUER                     *
rem ************************************************************
GLOBAL:  IF SPLIT%=1 THEN GOTO SPLITSC
         CALL P(22,1,"Border off ? Yes or No")
         CALL P(23,3,">/Global,")
         VC74%=INKEY:CALL P(22,1,BK$(25)):CALL P(23,3,BK$(10))
         IF VC74%=78 OR VC74%=(78+32) THEN BOR%=1:KEY%=4:GOTO LEV0
         IF VC74%=89 OR VC75%=(89+32) THEN BOR%=0:GOTO 5.741
         IF VC74%=8 THEN GOTO COMMAND ELSE GOTO GLOBAL
5.741    CALL P(1,1,BK$(80))
         FOR I%=1 TO 20:CALL P(I%,1,BK$(4)):NEXT I%
         KEY%=4:GOTO LEV0
rem ************************************************************
rem *       EIDTH SETTING AND JUSTIFICATION PROCEDURE           *
rem ************************************************************
FORMAT:  CALL P(22,1,"Width,Right justification,Left justification")
         CALL P(23,3,">/Format,")
         VC75%=INKEY:CALL P(22,1,BK$(50)):CALL P(23,3,BK$(10))
         IF VC75%=87 OR VC75%=(87+32) THEN 5.751
         IF VC75%=82 OR VC75%=(82+32) THEN 5.752
         IF VC75%=76 OR VC75%=(76+32) THEN 5.753
         IF VC75%=8 THEN GOTO COMMAND
         GOTO FORMAT
5.751    CALL P(22,1,"All cells or active column only ? Press G or <cr>")
         CALL P(23,3,">/Width,")
         VC751%=INKEY:CALL P(22,1,BK$(50)):CALL P(23,3,BK$(10))
         IF VC751%=71 OR VC751%=(71+32) THEN G%=1:GOTO 5.7511
         IF VC751%=13 THEN G%=0:GOTO 5.7511
         IF VC751%=8 THEN GOTO FORMAT
         GOTO 5.751
5.7511   CALL P(22,1,"Set width between 6 to 75")
5.75112  GOSUB INDATA:WD%=VAL(V$):CALL P(22,1,BK$(50))
         IF BACK%=1 THEN 5.751
```

```
          IF BACK%=1 THEN 5.751
          IF (WD% >= 6) AND (WD% <= 75) THEN 5.75111
          CALL P(22,1,"Width range between 6-75: Set the correct width")
          GOTO 5.75112
5.75111   CKEY%=1:IF G%=0 THEN 5.75113
          FOR I%=1 TO CLIMIT%
          W%(I%)=WD%:NEXT I%
          MC%=CST%:C2%=IIC%:MR%=RST%:R2%=IIR%:KEY%=3:GOTO LEV0
5.75113   W%(C2%)=WD%:KEY%=2:GOTO LEV
5.752     PROMPT$="Global,range or active cell only? Press G,R or<cr>"
          CALL P(22,1,PROMPT$)
          CALL P(23,3,">/Right Justification,")
          VC752%=INKEY:CALL P(22,1,BK$(60)):CALL P(23,3,BK$(20))
          IF VC752%=71 THEN RJUS%=1:KEY%=3:GOTO LEV
          IF VC752%=88 THEN RJUS%=0:GOTO 5.7521
          IF VC752%=13 THEN RJUS%=0:GOTO 5.7522
          IF VC752%=8 THEN GOTO FORMAT
5.7521    KEY%=1:GOTO LEV
5.7522    IF YY%(R2%,C2%)=0 THEN KEY%=1:GOTO LEV
          YY%(R2%,C2%)=-ABS(YY%(R2%,C2%)):KEY%=1:GOTO LEV
5.753     RJUS%=0:KEY%=1:GOTO LEV
rem ************************************************************ *
rem *                  SCREEN EXCHANCE PROCEDURE                 *
rem ************************************************************
XCHANGE:  CALL CURSOR$(R2%,C2%,MR%,MC%):KEY%=3
          BB$=TOPC$:TOPC$=TOPCELL$
          GOSUB CELLID:IIR%=II%:IIC%=JJ%:R2%=II%:C2%=JJ%
          IF CSPLIT%=1 THEN 5.761
          IF RST%=2 THEN RST%=RED%+2:RED%=20:MR%=RST%:MC%=5:GOTO LEV
          RED%=RST%-2:RST%=2:MR%=2:MC%=5:GOTO LEV
5.761     IF CST%=5 THEN CST%=CED%+4:CED%=81:MC%=CST%:MR%=2:GOTO LEV
          CED%=CST%-4:CST%=5:MC%=5:MR%=2:GOTO LEV
rem ************************************************************ *
rem *                     PRINT PROCEDURE                        *
rem ************************************************************
HPRINT:   CALL P(22,1,"Printer or Disk ? "):CALL P(23,3,"/Output,")
          VC77%=INKEY:CALL P(22,1,BK$(18)):CALL P(23,3,BK$(8))
          IF VC77%=80 OR VC77%=112 THEN 5.771
          IF VC77%=68 OR VC77%=100 THEN 5.772
          IF VC77%=8 THEN GOTO COMMAND ELSE GOTO HPRINT
5.771     CALL P(22,1,"LINE WIDTH ? UPTO 132")
          CALL P(23,3,"/Printing width ?")
          INPUT PWD%:T%=CED%:CED%=PWD%
          CALL P(22,1,"Printing with border ? Yes or No")
          VC771=INKEY:IF VC771=89 THEN PBOR%=1 ELSE PBOR%=0
          IF PBOR%=1 THEN  CALL COLUMNP(RST%,CST%,CED%,IIC%,1)
          I0%=IIR%:CDIF%=CED%-CST%
          FOR R%=RST% TO RED%
          IF PBOR%=1 THEN T$=J$(STR$(I0%),3,1)+CHR$(124) ELSE T$=BK$(4)
          J0%=IIC%:C%=W%(J0%)
          WHILE C%<=CDIF%
          IF SGN(YY%(I0%,J0%))=-1 THEN T$=T$+J$(X$(I0%,J0%),W%(J0%),1) \
          ELSE T$=T$+J$(X$(I0%,J0%),W%(J0%),0)
          J0%=J0%+1:C%=C%+W%(J0%)
          WEND
```

```
        LPRINTER
        PRINT J$(T$,CDIF%,0)
        I0%=I0%+1:NEXT R%
        CONSOLE
        CED%=T%:GOTO COMMAND
rem ************************************************************
rem *                      GOTO CELL PROCEDURE                 *
rem ************************************************************
CJUMP:  CALL P(23,4,"=>"):BB$=""
5.61    CJ%=INKEY:IF CJ%=13 THEN 5.6
        IF CJ%=8 THEN BB$=BS$(BB$) ELSE BB$=BB$+CHR$(CJ%)
        CALL P(23,7,BB$+" "):GOTO 5.61
5.6     BB$=REBK$(BB$):GOSUB CELLID
        IIR%=II%:IIC%=JJ%
        R2%=IIR%:C2%=IIC%:MR%=RST%:MC%=CST%:RETURN
rem ************************************************************
rem *                      HOME PROCEDURE                      *
rem ************************************************************
HOME:   BB$=TOPCELL$:GOSUB CELLID
        R2%=II%:C2%=JJ%:MR%=RST%:MC%=CST%:RETURN
rem ************************************************************
rem *                    PAGE UP PROCEDURE                     *
rem ************************************************************
PGDN:   BB$=TOPCELL$:GOSUB CELLID
        IF II%=RLIMIT%-19 THEN KEY%=4:GOTO LEV
        IF II%+19 > RLIMIT% THEN IIR%=RLIMIT%-19 ELSE IIR%=II%+19
        IIC%=JJ%:R2%=IIR%:C2%=IIC%:MR%=RST%:MC%=CST%:RETURN
rem ************************************************************
rem *                    PAGE DOWN PROCEDURE                   *
rem ************************************************************
PGUP:   BB$=TOPCELL$:GOSUB CELLID
        IF II%=1 THEN KEY%=4:GOTO LEV
        IF II%-19 < 1 THEN IIR%=1 ELSE IIR%=II%-19
        IIC%=JJ%:R2%=IIR%:C2%=IIC%:MR%=RST%:MC%=CST%:RETURN
rem ************************************************************
rem ************************************************************
rem *                  EXPRESSION INTERPRETER                  *
rem *                         MODULE                           *
rem ************************************************************
EXPTR:  FKEY% = 0:  M1% = 0:  NOPR% = 0
rem **** remove the blanks from the source statement string
        AA$ = REBK$(AA$)
rem **** check for the boolean function keyword
        FOR LTYPE% = 1 TO 6
            FPOS% = MATCH(LFUN$(LTYPE%),AA$,1)
            IF FPOS% <> 0 THEN GOSUB LFUN: GOTO BCHECK
        NEXT LTYPE%
rem **** check for the built-in function keyword
BCHECK: FOR BTYPE%= 1 TO 5
            FPOS% = MATCH(BFUN$(BTYPE%),AA$,1)
            IF FPOS% <> 0 THEN GOSUB BFUN: GOTO BCHECK
        NEXT BTYPE%
rem **** check for the library function keyword
        IF MATCH(LPR$,AA$,1) <> 0 THEN GOSUB LIBF
rem **** check for the precedence delimiter
```

```
LOOP:     M1% = M1%+1: PPOS% = MATCH(LPR$,AA$,1)
          IF PPOS% <> 0 THEN GOSUB BRAKET ELSE NOPR% = 1: A$=AA$
rem **** call the arithmetic expression interpreter
          GOSUB LEX: GOSUB POST: GOSUB VALUEEX: GOSUB EVALUTE
          TS(M1%) = RESULT
rem **** library function evaluation
          IF SWF% = 1 THEN GOSUB EVA.LIB: SWF% = 0: IV% = IV%-1
rem **** loop back for further interpretation
          IF NOPR% <> 1 THEN GOTO LOOP
rem **** back to the boolean expression interpreter
          IF FKEY% = 1 THEN GOSUB LFUN
          IF FKEY% = 1 THEN GOTO BCHECK
rem **** end of the interpretation
          RETURN
          STOP
rem ************************************************************
rem *                      LEXICAL ANALYZER                    *
rem ************************************************************
rem **** initialize the token and the buffer index
LEX:      K% = 1:  M% = 1
rem **** search for the operator in the expression
LEX1:     FOR I% = 1 TO 5
             IF O$(I%) = MID$(A$,K%,1) THEN GOTO LEX2
          NEXT I%
rem **** check for the final token
          IF LEN(A$) = K% THEN GOTO LEX3
          K% = K%+1: GOTO LEX1
rem **** check for the unary operator and exponent token
LEX2:     IF K% = 1 THEN K% = 2: GOTO LEX1
          IF MID$(A$,K%-1,1) = "E" THEN K% = K%+1: GOTO LEX1
rem **** isolate the tokens from the expression string
          B$(M%) = LEFT$(A$,K%-1)
          B$(M%+1) = O$(I%)
          A$ = RIGHT$(A$,LEN(A$)-K%)
rem **** increment the token table index and set the buffer index
          M% = M%+2: K% = 1: GOTO LEX1
rem **** isolate the final token
LEX3:     B$(M%) = A$: N% = M%
rem **** set the single token key
          IF M% = 1 THEN SVAL% = 1 ELSE SVAL%=0
          RETURN
rem ************************************************************
rem *                      SYNTAX ANALYZER                     *
rem ************************************************************
rem **** initialize the token table index and the buffer index
POST:     K% = 2: M% = 1: L% = 0
rem **** exchange the elements
POST1:    TM$ = B$(K%-L%): B$(K%-L%) = B$(K%+1): B$(K%+1) = TM$
rem **** check for the final token under consideration
POST2:    K% = K%+1: IF K% > N% THEN GOTO POST3
rem **** find the operator number
          I% = MATCH(B$(K%),OPERATOR$,1)
          IF I% = 0 THEN POST2
rem **** load the operator buffer
          STK%(M%) = I%: M% = M%+1
```

```
        IF M% = 2 THEN GOTO POST2  ELSE M% = 1
rem **** check the precedence of the operator for arrangement
        IF STK%(2) = 1 OR STK%(2) = 2 THEN L% = 0: GOTO POST1
        IF STK%(2) = 5 THEN L% = 1: GOTO POST1
        IF STK%(1) = 1 OR STK%(1) = 2 THEN L% = 1: GOTO POST1
        L% = 0: GOTO POST1
rem **** exit from the procedure
POST3:  RETURN
rem ************************************************************
rem *                    VALUE TABLE GENERATOR                 *
rem ************************************************************
rem **** initialize the table and set the table index
VALUEEX: DIM Z(50): I1% = 1
rem **** check for the final token for consideration
        WHILE I1% <= N%
rem **** check and process for the number token
          IF MATCH(LEFT$(B$(I1%),1),DIGIT$,1)<>0 THEN \
                                  Z(I1%) = VAL(B$(I1%)): GOTO VALEX3
rem **** check and process for the operator token
          IF MATCH(B$(I1%),OPERATOR$,1)<>0 THEN GOTO VALEX3
          FOR J% = 1 TO 5
rem **** check and process for the temporary storage token
          IF S$(J%) = B$(I1%) THEN Z(I1%) = TS(J%): GOTO VALEX3
          NEXT J%
rem **** check  for the token with unary operator
          FOR OP% = 1 TO 2
              IF O$(OP%) = LEFT$(B$(I1%),1) THEN GOTO VALEX1
          NEXT OP%
rem **** process for the identifier token
          BB$ = B$(I1%): GOSUB CELLID: GOSUB CRF
          Z(I1%) = VAL(X$(II%,JJ%)): GOTO VALEX3
rem **** process for the token with unary operator
VALEX1:   BB$ = RIGHT$(B$(I1%),LEN(B$(I1%))-1)
          IF MATCH(LEFT$(BB$,1),DIGIT$,1)<>0 THEN \
                                  VUE = VAL(BB$): GOTO VALEX2
          FOR J% = 1 TO 5
              IF S$(J%) = BB$ THEN VUE = TS(J%): GOTO VALEX2
          NEXT J%
          GOSUB CELLID: GOSUB CRF: VUE = VAL(X$(II%,JJ%))
VALEX2:   IF OP% = 1 THEN Z(I1%) = -VUE ELSE Z(I1%) = VUE
rem **** increment the token table index
VALEX3:   I1% = I1%+1
        WEND
rem **** exit from the procedure
        RETURN
rem ************************************************************
rem *                      VALUE EVALUATION                    *
rem ************************************************************
rem **** check for the single token
EVALUTE: IF SVAL% = 1 THEN RESULT = Z(1): RETURN
rem **** check the table pointer with unity
        WHILE N% > 1
rem **** search for an operator
```

```
            FOR I% = 1 TO N%
                J% = MATCH(B$(I%),OPERATOR$,1)
                IF J% <> 0 THEN GOTO EVALTE1
            NEXT I%
EVALTE1:    FIR% = I%-1: SED% = I%-2
rem **** branch to perform operation according to the operator
            ON J% GOTO ST,AD,DV,ML,EP
rem **** perform substraction operation
ST:         Z(SED%) = Z(SED%)-Z(FIR%): GOTO EVALTE2
rem **** perform addition operation
AD:         Z(SED%) = Z(SED%)+Z(FIR%): GOTO EVALTE2
rem **** perform division operation
DV:         Z(SED%) = Z(SED%)/Z(FIR%): GOTO EVALTE2
rem **** perform multiplication operation
ML:         Z(SED%) = Z(SED%)*Z(FIR%): GOTO EVALTE2
rem **** perform exponentiation operation
EP:         Z(SED%) = Z(SED%)^Z(FIR%)
rem **** shifts the elements to eliminate the operands
EVALTE2:    WHILE I% < N%
               B$(I%-1) = B$(I%+1): Z(I%-1) = Z(I%+1)
            I% = I%+1: WEND
rem **** decrement the the table pointer  and  back for looping
        N% = N%-2: WEND
rem **** store the result and exit from the procedure
        RESULT = Z(1): RETURN
rem ************************************************************
rem *              BOOLEAN EXPRESSION INTERRETER                 *
rem ************************************************************
rem **** check the key for new entry or not
LFUN:   IF FKEY% = 1 THEN GOTO LC21
rem **** initialize the tables and the indexes
        DIM ZZ(6), TABLE$(6): K% = 1: ED% = 0: AUG% = 0
rem **** identify the argument delimiter
        L1% = MATCH(LPR$,AA$,1)
rem **** isolate the function argument string
        TM$ = MID$(AA$,L1%+1,LEN(AA$)-L1%-1)
rem ************************************************************
rem *                BOOLEAN LEXICAL ANALYSER                   *
rem ************************************************************
rem **** identify the token delimiter character
LC1:    L3% = MATCH(COM$,TM$,1)
rem **** load the buffer variable for further processing
        IF L3% = 0  THEN TTM$ = TM$: ED% = 1: GOTO LC11
        TTM$ = LEFT$(TM$,L3%-1)
rem **** identify the relational operator if any
LC11:   FOR I% = 1 TO 9
            RPOS% = MATCH(LRE$(I%),TTM$,1)
            IF RPOS% <> 0 THEN GOTO LC12
        NEXT I%
rem **** store the token having no relational operator
        TABLE$(K%) = TTM$: K% = K%+1: GOTO LC13
rem **** store the operands and the operator tokens
LC12:   TABLE$(K%) = LEFT$(TTM$,RPOS%-1)
        TABLE$(K%+1) = LRE$(I%): ZZ(K%+1) = I%
        TABLE$(K%+2) = MID$(TTM$,RPOS%+LEN(LRE$(I%)),LEN(TTM$))
```

```
rem **** increment the token table index
         K% = K%+3
rem **** check for the final token
LC13:    IF ED% <> 1 THEN TM$=RIGHT$(TM$,LEN(TM$)-L3%): GOTO LC1
rem **** set the token range
         RANGE% = K%-1
rem ************************************************************
rem *       BOOLEAN EXPRESSIN VALUE TABLE GENERATION           *
rem ************************************************************
rem **** initialize the token table and the displacement index
         TP% = 1: DIS% = 2
rem **** call the expression interpreter for evaluation
LC2:     AA$ = TABLE$(TP%): FKEY% = 1: RETURN
rem **** store the evaluated value and increment the index
LC21     ZZ(TP%) = RESULT: TP% = TP%+DIS%
         IF TP% = 3 THEN GOTO LC2
rem **** check for the final token
         IF TABLE$(TP%) = "" THEN TP% = 1: GOTO LC3
         IF ED% = 0 THEN GOTO LC2
         TP% = 4: ED% = 0
         IF RANGE% = 6 THEN DIS% = 2 ELSE DIS% = 1
         GOTO LC2
rem ************************************************************
rem *              BOOLEAN EXPRESSION EVALUATION               *
rem ************************************************************
rem **** load the operands and the operator buffer variables
LC3:     OPD1 = ZZ(TP%): OPR = ZZ(TP%+1): OPD2 = ZZ(TP%+2)
rem **** switch to a particular label according to the operator
         ON OPR GOTO 58.51,58.51,58.52,58.52,58.53,58.53,58.54,\
                     58.55,58.56
         58.51 IF OPD1 <> OPD2 THEN GOTO TRUE ELSE GOTO FALSE
         58.52 IF OPD1 >= OPD2 THEN GOTO TRUE ELSE GOTO FALSE
         58.53 IF OPD1 <= OPD2 THEN GOTO TRUE ELSE GOTO FALSE
         58.54 IF OPD1  = OPD2 THEN GOTO TRUE ELSE GOTO FALSE
         58.55 IF OPD1  > OPD2 THEN GOTO TRUE ELSE GOTO FALSE
         58.56 IF OPD1  < OPD2 THEN GOTO TRUE
rem **** switch to a label according to the function type
FALSE:   ON LTYPE% GOTO 58.61,58.62,58.63,58.64,58.63,58.65
rem **** store the result for the conditional IF operation
 58.61   RESULT = ZZ(5): GOTO EXIT
rem **** set the result for the logical NOT operation
 58.62   RESULT = 1: GOTO EXIT
rem **** perform functions for the logical AND & XNOR operation
 58.63   IF ED% = 0 THEN ED% = 1: TP% = 4: GOTO LC3
         IF AUG% = 1 THEN RESULT = 1 ELSE RESULT = 0
         GOTO EXIT
rem **** set the result for the logical OR operation
 58.64   RESULT = 0: GOTO EXIT
rem **** perform functions for the logical XOR operation
 58.65   IF ED% = 0 THEN ED% = 1: TP% = 4: GOTO LC3
         IF AUG% = 1 THEN RESULT = 0 ELSE RESULT = 1
         GOTO EXIT
rem **** switch to a label according to the function type
TRUE:    ON LTYPE% GOTO 58.71,58.72,58.73,58.74,58.75,58.74
rem **** store the result for the conditional IF operation
```

```
rem **** store the result for the conditional IF operation
 58.71    RESULT = ZZ(4): GOTO EXIT
rem **** set the result for the logical NOT operation
 58.72    RESULT = 0: GOTO EXIT
rem **** set the result for the logical AND operation
 58.73    RESULT = 1: GOTO EXIT
rem **** perform functions for the logical OR & XNOR operation
 58.74    IF ED% = 0 THEN ED% = 1: TP% = 4: AUG% = 1: GOTO LC3
          IF AUG% = 1 THEN RESULT = 1 ELSE RESULT = 0
          GOTO EXIT
rem **** perform functions for the logical XOR operation
 58.75    IF ED% = 0 THEN ED% = 1: TP% = 4: AUG% = 1: GOTO LC3
          IF AUG% = 1 THEN RESULT = 0 ELSE RESULT = 1
rem **** set the key and exit from the procedure
EXIT:     FKEY% = 0: RETURN
rem ************************************************************
rem *                BUILT-IN FUNCTION INTERPRETER               *
rem ************************************************************
rem **** initialize the temporary storage index
BFUN:     M1% = M1%+1
rem **** identify the function argumenr string
          I1% = MATCH(LPR$,AA$,FPOS%): I2% = MATCH(RPR$,AA$,I1%)
rem **** isolate the function argument string
          A$ = MID$(AA$,I1%+1,I2%-I1%-1)
rem **** replace the function by the temporary storage identifier
          AA$ = LEFT$(AA$,FPOS%-1)+S$(M1%)+RIGHT$(AA$,LEN(AA$)-I2%)
rem **** call the procedures for the evaluation of the  function
          GOSUB EXPAND: GOSUB BEVALUTE
          ON BTYPE% GOTO 23.1, 23.2, 23.3, 23.4, 23.5
rem **** load the temporary storage by the function value & exit
23.1      GOSUB MX: TS(M1%) = MAX: RETURN
23.2      GOSUB MN: TS(M1%) = MIN: RETURN
23.3      GOSUB SM: TS(M1%) = SUM: RETURN
23.4      GOSUB AG: TS(M1%) = AVG: RETURN
23.5      TS(M1%) = CNT%: RETURN
rem ************************************************************
rem *          FUNCTION ARGUMENT LEXICAL ANALYZER                *
rem ************************************************************
rem **** initialize the indexes
EXPAND:   M% = 0: J% = 0
rem **** check for the final token for isolation
          WHILE A$<>""
rem **** identify the token delimiter
            K2% = MATCH(SLON$,A$,1)
            IF K2% = 0 THEN BUFF$ = A$: A$ = "": GOTO BIN1
rem **** isolate the token and load the buffer for  checking
            BUFF$ = LEFT$(A$,K2%-1)
rem **** reduce the argument string
            A$ = RIGHT$(A$,LEN(A$)-K2%)
rem **** identify the range or list delimiter
BIN1:       K2% = MATCH(CLON$,BUFF$,1)
            IF K2% <> 0 THEN J% = J%+1: GOTO BIN2
rem **** store the individual cell token
            M% = M%+1: MX$(M%) = BUFF$: GOTO BIN3
rem **** isolate and store the tokens representing list or range
```

```
BIN2:       LLMT$(J%) = LEFT$(BUFF$,K2%-1)
            RLMT$(J%) = RIGHT$(BUFF$,LEN(BUFF$)-K2%)
rem **** looping back for further checking
BIN3:     WEND
rem **** exit from the procedure
          RETURN
rem ************************************************************************
rem *                  FUNCTION VALUE TABLE GENERATOR                      *
rem ************************************************************************
rem **** initialize the value table and set the indexes
BEVALUTE:DIM Z(100): NR% = J%: NC% = M%: NT% = 0
rem **** check for the individual token index limit
          IF NC% = 0 THEN BEV4
          FOR K% = 1 TO NC%
rem **** identify the cell by calling a procedure
             BB$ = MX$(K%): GOSUB CELLID
rem **** extract the value from the data structure and store
             Z(K%) = VAL(X$(II%,JJ%))
          NEXT K%
rem **** check for the list or range index limit
BEV4:     IF NR% = 0 THEN LMT% = NC%: RETURN
rem **** increment the token table index
          NT% = NT%+1
rem **** identify the cell representing the lower limit
          BB$ = LLMT$(NT%): GOSUB CELLID
          KK1% = II%: LL1% = JJ%
rem **** identify the cell representing the upper limit
          BB$ = RLMT$(NT%): GOSUB CELLID
          KK2% = II%: LL2% = JJ%
          IF KK1% = KK2% THEN GOTO BEV5
          IF LL1% = LL2% THEN GOTO BEV6
rem **** extract and store the values from a block of cells
          FOR II% = KK1% TO KK2%
              FOR JJ% = LL1% TO LL2%
                  M% = M%+1: Z(M%) = VAL(X$(II%,JJ%)): GOSUB CRF
              NEXT JJ%
          NEXT II%
          GOTO BEV7
rem **** extract and store the values from a row of cells
BEV5:     II% = KK1%
          FOR JJ% = LL1% TO LL2%
              M% = M%+1: Z(M%) = VAL(X$(II%,JJ%)): GOSUB CRF
          NEXT JJ%
          GOTO BEV7
rem **** extract and store the values from a column of cells
BEV6:     JJ% = LL1%
          FOR II% = KK1% TO KK2%
              M% = M%+1: Z(M%) = VAL(X$(II%,JJ%)): GOSUB CRF
          NEXT II%
rem **** check for further processing over any more token
BEV7:     IF NT% <> NR% THEN GOTO BEV4
rem **** set the value table index limit and the count
          LMT% = M%: CNT% = M%
rem **** exit from the procedure
          RETURN
```

```
SM:       SUM = 0
          FOR I% = 1 TO LMT%
              SUM = SUM+Z(I%)
          NEXT I%
          RETURN
rem **** procedure to find the maximum value
MX:       MAX = Z(1)
          FOR I% = 2 TO LMT%
              IF MAX <= Z(I%) THEN MAX = Z(I%)
          NEXT I%
          RETURN
rem **** procedure to find the minimum value
MN:       MIN = Z(1)
          FOR I% = 2 TO LMT%
              IF MIN >= Z(I%) THEN MIN = Z(I%)
          NEXT I%
          RETURN
rem **** procedure to calculate the average
AG:       GOSUB SM: AVG = SUM/LMT%: RETURN
rem ************************************************************
rem *                    PROCEDURE  BRAKET                     *
rem ************************************************************
BRAKET:   J1% = 1
BLOOP:    I1% = MATCH(LPR$,AA$,J1%)
          IF I1% <> 0  THEN  J1% = I1%+1: GOTO BLOOP
          IF J1%-1 = TK%(IV%) THEN  SWF% = 1
          J2% = J1%: I2% = MATCH(RPR$,AA$,J2%)
          IF I2% = 0 THEN  DERROR = 4: GOTO  555
          J2% = I2%-1:  A$ = MID$(AA$,J1%,J2%-J1%+1)
          AA$ = LEFT$(AA$,J1%-2)+S$(M1%)+ \
                                  +RIGHT$(AA$,LEN(AA$)-J2%-1)
          RETURN
rem ************************************************************
rem *            LIBRARY FUNCTION EVALUATION                   *
rem ************************************************************
EVA.LIB:  X = TS(M1%)
          ON FN%(IV%) GOTO 2.1,2.2,2.3,2.4,2.5,2.6,2.7,2.8,2.9,\
                                                  2.10,2.11,2.12
2.1       X = ABS(X): GOTO  2.01
2.2       X = EXP(X): GOTO  2.01
2.3       X = INT(X): GOTO  2.01
2.4       X = LOG(X): GOTO  2.01
2.5       X = ROUND(X): GOTO  2.01
2.6       X = SIN(X): GOTO  2.01
2.7       X = COS(X): GOTO  2.01
2.8       X = TAN(X): GOTO  2.01
2.9       X = 1/SIN(X): GOTO  2.01
2.10      X = 1/COS(X): GOTO  2.01
2.11      X = 1/TAN(X): GOTO  2.01
2.12      X = ATN(X): GOTO 2.01
2.01      TS(M1%) = X
          RETURN
rem ************************************************************
rem *      LIBRARY FUNCTION KEYWORD IDENTIFIER PROCEDURE        *
rem ************************************************************
```

```
LIBF:     IV% = 1: SWF% = 0: J% = 1: MK% = 3
LIB1:     BR% = MATCH (LPR$,AA$,MK%)
          IF BR% < 4 THEN GOTO LIB2
          TEST% = MATCH(MID$(AA$,BR%-3,3),LFUNCTION$,1)
          IF TEST% = 0 THEN MK% = BR%+1 : GOTO LIB1
          I% = ((TEST%-1)/3)+1
          FN%(IV%) = I%: TK% (IV%) = BR%-3
          AA$ = LEFT$(AA$,BR%-4)+RIGHT$ (AA$,LEN(AA$)-BR%+1)
          IV% = IV%+1: MK% = BR%-2: GOTO LIB1
LIB2:     IV% = IV%-1: RETURN
rem ************************************************************************
```

## REFERENCES


[1]. Wakerley, J. F.
Microcomputer Architecture and Programming,
John Wiley and Sons, N.Y (1981)

[2]. Tremblay J. P. and Sorenson P. G.
An Introduction to Data Structures with Applications,
McGRAW-Hill, N.Y (1976)

[3]. Goodman S. E. and Hedetniemi S. T.
Introduction to the design and analysis of Algorithms,
TOKYO, JAPAN (1977)

[4]. Gries David,
Compiler Construction for Digital Computers,
John Wiley and Sons, N.Y (1971)

[5]. Digital Research,
CBASIC Compiler Language Reference Manual,
U.S.A. (1983)

[6]. Digital Research,
CBASIC Compiler Language Programming Guide,
U.S.A. (1983)

[7]. Monroe System for Business,
CP/M-86 DPX Operating System Supplement,
U.S.A. (1983)

[8]. Monroe System for Business,
CP/M-86 DPX User's Guide,
U.S.A. (1983)

[9]. Monroe System for Business,
CP/M-86 DPX Programmer's Guide,
U.S.A. (1983)

[10]. Monroe System for Business,
CP/M-86 DPX System Guide,
U.S.A. (1983)