10-21-1988

IFE, an interactive formula editor

Teresa C. Chau

Florida International University

DOI: 10.25148/etd.FI14060149

Follow this and additional works at: <https://digitalcommons.fiu.edu/etd>

Part of the [Computer Sciences Commons](#)

Recommended Citation

Chau, Teresa C., "IFE, an interactive formula editor" (1988). *FIU Electronic Theses and Dissertations*. 2117.
<https://digitalcommons.fiu.edu/etd/2117>

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

ABSTRACT

IFE: AN INTERACTIVE FORMULA EDITOR

by

Teresa C. Chau

IFE, the Interactive Formula Editor is an experimental system designed, developed and implemented to propose a different approach to handle mathematical formulae. Its main characteristics are: (1) Interactive creation and edition of a mathematical formula, (2) Tex-form output for a printed version of a formula, and (3) Ability to generate a new set of characters by means of a character editor.

Mathematical symbols are provided and adjusted by the system, automatically. The system also, guides the user during the formula description because it knows the syntax of the graphical representation.

The system seems to be complete and perform well. The listing of the program is included, as are suggestions for further development.

To Professors Masoud Milani, Toby Berk and Raimund Ege

This thesis, having been approved in respect to form and mechanical execution, is referred to you for judgment upon its substantial merit.

Dean James A. Mau
College of Arts and Sciences

The thesis of Teresa C. Chau is approved.

Professor Toby Berk

Professor Raimund Ege

Major Professor Masoud Milani

Date of Examination:

October 21, 1988

IFE: AN INTERACTIVE FORMULA EDITOR

by

Teresa C. Chau

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

at

FLORIDA INTERNATIONAL UNIVERSITY

1988

ACKNOWLEDGEMENTS

To my parents, whose courage, spirit and love got me through the pages of this thesis.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS	v
CHAPTER	
I. INTRODUCTION: IDENTIFYING THE PROBLEM AND THE SOLUTION	1
Difficulties	2
Classification	3
Characteristics of the Solution	5
II. DEVELOPING THE SOLUTION	7
The Structure of the System	7
Scope and Purpose	7
Interface Principles	7
Graphical Representation	8
Grammar	8
Internal Representation	11
Objects	12
The Tree	14
Nodes	18
Creating and Editing	21
Creating A Formula	21
Editing A Formula	23
The Cursor	23
Formula Modification	24
Selection	25
Operations	25
III. IMPLEMENTING THE SOLUTION	27
The Interactive Formula Editor	27
Configuration	27
The Screen	28
The Cursor	29
The IFE Main Menu	29
Option 1: Create a Formula	30
Option 2: Edit an Existing Formula	34
Selecting an Entity	35
Selecting an Operation	36

Option 3: Save Formula into Disk	37
Fonts and Characters	38
Option 4: Load Extra Font	38
Option 5: Create an Alphabet Set	39
Option 6: Save Alphabet into Disk	41
Option 7: Load Alphabet from Disk	41
The Implementation	41
Improvements	45
Conclusion	46
APPENDIX	47
BIBLIOGRAPHY	122

LIST OF ILLUSTRATIONS

1.	Using the R Formatter	4
2.	Using the EQN System	4
3.	Using the TeX System	4
4.	A Formula and its Tree Representation	15
5.	Some Mathematical Objects and their Boxes ...	17
6.	The leftmost-child, right-sibling, parent representation of a tree	19
7.	Insert After Operation	26
8.	Insert Before Operation	26
9.	The Edit Mode Cursor	29
10.	The Input Mode Menu	30
11.	The Menu of Function 4	31
12.	The Menu of Function 9	31
13.	The Greek Set	32
14.	The Scientific Set	33
15.	The Edit Mode Menu (Part I)	35
16.	The Edit Mode Menu (Part II)	35
17.	The Graphics Editor menu	40
18.	Stack of Boxes during Input Phase	43

1.1 DIFFICULTIES

Every computer system is usually provided with some kind of text editor, document production system or a word processor. However, most of these systems have no special feature to handle mathematical formulae.

The difficulties inherent in printing or typing formulae had been recognized by systems like TeX[4] and EQN[3]. In addition, The University of Chicago Press in its "Manual of Style" says:

Mathematics is known in the trade as difficult, or penalty, copy because it is slower, more difficult, and more expensive to set in type than any other kind of copy normally occurring in books and journals[1]

Among the difficulties with mathematical text are the following:

1. The multiplicity of alphabets (Roman, Greek, Italic)
2. Many special symbols of different sizes (line, radical, integral)
3. The two dimensional character of mathematics (horizontal, vertical)
4. The relative positioning of some expressions (index, exponent)
5. The variable sizes for certain symbols (brackets, braces, parentheses)

Therefore, mathematics has its own typographical conventions which are quite different from those of ordinary text.

1.2 CLASSIFICATION

If we consider the broad range of document preparation systems that can manipulate mathematical text, we can clearly distinguish two groups:

A. The first group includes batch systems: the document does not appear until the whole text has been typed, and then processed by the system.

These systems impose a language for describing mathematical text. The description takes the form of a linear expression. Therefore, the final picture can be visualized only with considerable practice. As a consequence, only specialists can use them effectively.

TeX[5], EQN[3], and R[2] are examples of this category. Some formulae and their representations in different text editing systems are depicted in Figures 1, 2 and 3.

B. The Second Group comprises systems that act upon the document almost immediately. These systems feature a menu-based interface or a combination of menu and command-based interface.

WordPerfect[30] and WordStar[31] are examples of combined menu and command-based programs. Packages such as VolksWriter[32] and DisplayWrite[33] are examples of menu-driven systems. XYWrite III Plus[34] is an example of a command-driven system.

Most of these systems offer "what you see is what you get" display as a feature that is becoming a

$$p + \frac{a + \frac{b+c}{d+e}}{f + \frac{g+h}{i}} + \int_{u+v}^{w+x} ydz + qrs$$

```
p+^Xdiv(a+^Xdiv(b+c d+e) f+^Xdiv(g+h i))+
      ^Xinteg(u+v w+x ydz) +qrs
```

Fig. 1. Using the R formatter

$$a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3 + \dots}}}$$

```
a sub 0 + b sub 1 over {a sub 1 + b sub 2 over
      {a sub 2 + b sub 3 over {a sub 3 + ...}}}
```

Fig. 2. Using the EQN system

$$\int_0^{\infty} t dt + \frac{t^2 + s}{k + \frac{1}{t^3}}$$

```
$$\int_0^{\infty} t\,dt + \{t^2+s\over \displaystyle
      \{k+\{1\over\displaystyle t^3\}\}}$$
```

Fig. 3. Using the TeX system

standard with many software packages today.

These systems are easier to use and learn than systems in the first group. However, mathematical capabilities in this group do not always produce mathematical formula. They are limited to a four function calculator, calculations on columns, rows, and combinations of the two as in spreadsheet packages.

In the cases when they produce mathematical formula, they make use of special features such as equation mode, type-over, half-line spacing, space-filling, line-drawing and build-up characters.

The systems in the first group, in accordance with mathematical conventions, automatically draw lines, position expressions, enlarge special symbols and the like. In contrast, the same features are done manually by the user in the systems of the second group.

In either group, typing out a mathematical text calls for a considerable experience or patience. Capturing formulae is very difficult or cumbersome, and it is slower than preparing ordinary text.

1.3 CHARACTERISTICS OF THE SOLUTION

It appears that what users need is an interactive system that has no language description and can handle typographical rules of mathematics automatically.

Such a system should be "natural" so that the author may write mathematical text in the same way he/she does with a pencil on a piece of paper. It

should provide the author with immediate visual feedback--anything that the user does would have an immediate effect on the formula being manipulated, and the screen would present the formula in a form as close as possible to its usual graphical appearance.

This graphical appearance must be the only form that the user needs to know. There should not be a formula description language. Whether capturing a new formula or modifying an old one, all commands should be given using function keys or by making selections from a menu of options.

The user should not worry about positioning parts of the formula, as in subscripts and superscripts, nor the size of special symbols, like the integral sign, delimiters, the extension of a fraction bar, the extension of a radical sign, etc. An ideal system would take care of all these details automatically.

Finally, the user should have available all, or virtually all, of the character sets used in mathematics.

CHAPTER II

DEVELOPING THE SOLUTION

2.1 THE STRUCTURE OF THE SYSTEM

2.1.1 SCOPE AND PURPOSE

Our ultimate goal is to design a system that can handle scientific documents entirely, which includes text, formulae, and even illustrations. The emphasis of this thesis is to create a system that is able to handle mathematical formulae, since there is no need to develop yet another standard text processing system.

This experiment would form the basis for a future system that will manipulate text as well as formulae.

2.1.2 INTERFACE PRINCIPLES

One of the system's most important characteristics is interactivity. Any action by the operator has an immediate effect on the formula, and the system displays the formula on the screen, as close as possible to its normal graphical representation.

There is no special command language, so the user only needs to know this graphical form. By means of function keys and some menus requesting specific values, the formula is described "naturally", similar to the way in which it would be dictated.

Constant guidance is given to the user, by the position of the cursor where the next element is to be entered. This constant movement of the cursor is possible because the system is driven by the syntax of the graphical representation.

The user does not have to worry about the size of the formula because the system itself modifies the size of special symbols and positions parts of the formula when necessary.

2.1.3 GRAPHICAL REPRESENTATION

If we look at a mathematical expression like $y = ax + b$, we can see that it involves an equality, an addition and a product. But, as far as the graphical representation is concerned, the structure is simply a string of characters.

In fact, the only elements that the system has taken into account are those which correspond to graphical representations which are not simple character strings.

Grammar. The grammar of the graphical representation is given next. Words in bold represent terminal symbols. Words in lower case letters are non-terminal symbols. A character is any printable ASCII character including blank.

formula = expression

expression = object

= object expression

```

object = fraction
        = radical
        = integral
        = triple
        = limit
        = vector
        = object exponent
        = object index
        = object exponent index
        = object index exponent
        = block
        = string
string = alphabet text
text = character
      = character text
alphabet = Normal
          = Greek
          = Scientific
          = Other
fraction = numerator FB denominator
numerator = expression
denominator = expression
radical = radical-index RS radicand
        = RS radicand
radical-index = string
radicand = expression

```



```

integral = IS lower-limit upper-limit integrand
          = IS lower-limit integrand
          = IS upper-limit integrand
          = IS integrand
lower-limit = expression
upper-limit = expression
integrand = expression
triple = tsign lower upper triple-body
        = tsign lower triple-body
        = tsign upper triple-body
        = tsign triple-body
lower = expression
upper = expression
triple-body = expression
tsign = Summation
       = Product
       = Union
       = Intersection
limit = LS low-limit
       = LS
low-limit = expression
vector = vlist
vlist = expression expression
       = expression vlist
exponent = expression
index = expression
block = delimiter delimiter expression

```

delimiter = blank

= (

=)

= {

= }

= {

= }

= |

The following terminals represent known mathematical symbols:

FB: fraction bar

RS: radical sign

IS: integral sign

LS: limit sign

Summation, and Product refer to the corresponding mathematical operations. Union and Intersection refer to the corresponding set operators. Normal, Greek, and Scientific refer to the character sets used by our system. Other stands for other character set provided by the user. blank represents one blank space.

2.1.4 INTERNAL REPRESENTATION

The structure of the graphical representation is used for the internal representation. Since a tree can be easily described in a linear form, and it can describe precisely the structure of formula, the tree structure is adopted for the internal representation.

Each formula is represented by a tree structure,

whose nodes are mathematical objects and whose leaves are character strings. There are two kinds of mathematical objects: objects and specific elements.

Objects. Only ten types of objects are represented in our system. Each object corresponds to a specific graphical construction. Almost every formula can be written by combining these constructions.

A mathematical expression is made up of a sequence of these objects. Each of these objects contains one or more specific elements. These specific elements may be optional or required.

Each object has an associated display rule that defines the positioning of the specific elements within the object--the way in which the expressions contained within the object are to be displayed.

Specific elements are themselves made up of a series of objects. Therefore, objects may appear at almost any place in a formula while specific elements appear within a limited context. They depend on the objects they belong to.

The objects and their rules are:

1. Fraction: contains two specific elements, which are required: the numerator and the denominator. The numerator is displayed over the denominator and a horizontal line separates both, which are centered.

2. Radical: contains two specific elements. The radical index, which is optional and the radicand, which

is required. The radical index is displayed outside the radical sign, the radicand is displayed within the radical sign. This radical sign has the same size as the expression it encloses. If the radical index is omitted, then the radical represents the square root of the required expression.

3. Integral: contains three specific elements. Two optional expressions: the lower and upper limits of the integration interval, and one required expression: the integrand. Both limits are displayed at the lower and upper edges of the integral sign. The integrand is displayed centered, and next to the integral sign.

4. Triple: contains three specific elements. Two optional expressions: the lower and upper expressions of a triple sign, and one required expression: the triple body. Both optional expressions are displayed above and below the triple sign (these three are centered in relation to each other). The triple body is displayed next to the triple sign. This object allows for the representation of a summation, a product, a union, or an intersection.

5. Limit: contains one specific element, which is an optional expression. This expression is displayed centered and underneath the limit sign.

6. Vector: contains two or more specific elements. These are the elements of a vector. Elements are displayed one over the other and centered. This object

allows the representation of matrices or vectors when they are presented vertically.

7. Index: requires an expression to be present. The expression is written below and immediately after the preceding object. If the preceding object is an exponent then the rule applies to the object before the exponent.

8. Exponent: requires an expression to be present. The expression is written above and immediately after the preceding object. If the preceding object is an index, then this rule applies to the object before the index.

9. Block: contains one required expression. This expression is displayed between two delimiters (parentheses, brackets, braces, bars, or spaces) which have the same height as the expression they enclose.

10. Character String: consists of a linear sequence of characters belonging to the same alphabet.

The tree. The tree represents the relations between the mathematical objects that make up a formula. Each tree node corresponds to a mathematical object, i.e. an object or a specific element.

The root of the tree represents the formula as a whole. Following the root, the first level of the tree is a sequential series of objects that make up the formula.

$$x - \int_0^{\sqrt{y}} x dx + \sum_{x=0}^n \frac{\sin x}{x}$$

Fig. 4. A Formula and its Tree Representation

Expressions contained in each of these objects are similarly regarded as a series of objects or specific elements, each containing further expressions similarly constituted. Figure 4 gives an example of such a tree (node-types are defined under Nodes in page 18).

In order to closely relate the graphical representation of a formula to its tree structure, we use a concept known as "boxes".

It consists of associating each mathematical object with a rectangular box, invisible to the user, whose purpose is to mark the boundaries of each mathematical object on the screen.

A box is defined by its height, width and reference line (see Figure 5). The reference line of a box is used to align characters horizontally, that is, the expression inside the box, or the main expression that follows a special symbol. For example, in an integral, the integrand box goes aligned to the middle of the integral symbol.

These boxes make it possible to correlate the positions of the mathematical objects on the screen with the tree structure. The relations between the different boxes reflect the relations between the nodes of the tree.

The positioning of the boxes on the screen is determined by reference to the rules of graphical representation presented under Objects in page 12.

a character string

$$\boxed{y = ax + b}$$

a square root

$$\boxed{\sqrt{\boxed{x+y}}}$$

a fraction

$$\left. \begin{array}{|c|} \hline \boxed{1} \\ \hline \boxed{2x} \\ \hline \end{array} \right\} \text{height}$$

an integral

reference
line $\longrightarrow$ $\boxed{\int \frac{\boxed{1}}{\boxed{0}} \boxed{x} \, d\boxed{x}}$

a triple

$$\underbrace{\left[\begin{array}{|c|} \hline \boxed{\infty} \\ \hline \sum \\ \hline \boxed{i=0} \\ \hline \end{array} \boxed{1+i} \right]}_{\text{width}}$$

Fig. 5. Some mathematical objects and their boxes

Nodes. The nodes of the tree contain information on the types of mathematical objects they represent, the node's position in the tree and the node's physical attributes. The node-types are:

M : mathematical formula
 C : character string
 F : fraction
 N : numerator
 D : denominator
 R : radical
 K : radical index
 Q : radicand
 I : integral
 L : lower limit of integral
 U : upper limit of integral
 W : integrand
 T : triple
 G : lower limit of triple
 H : upper limit of triple
 A : triple body
 Z : limit
 O : low expression of limit
 V : vector
 E : vector element
 P : exponent
 S : index
 B : block

Fig. 6. The leftmost-child, right-sibling, parent representation of a tree.

The node's position in the tree is represented by the following relationships:

- (1) node's leftmost-child
- (2) node's right-sibling
- (3) node's parent

Figure 6 shows an example of the relations between the nodes of a tree that corresponds to a formula that consists of a fraction followed by a string.

The node's physical attributes are:

- (1) point of origin (x,y)
- (2) height
- (3) width
- (4) reference line

Node-type T (triple) has one extra piece of information that is used to identify the triple sign. Triple signs are coded as follows:

- 1: summation 2: product
- 3: union 4: intersection

Node-type B (block) has an extra two-digit number that represents the two delimiters combination, from the following list:

- 1: (2:)
- 3: [4:]
- 5: { 6: }
- 7: | 8: space

For example, 12 represents (), 34 represents [], and so on.

Node-type C (character string) has an extra character that denotes the alphabet set the string belong to:

1: normal 2: greek
3: scientific 4: other set

2.2 CREATING AND EDITING

An important function of the system is that it provides for interactive creation and modification of the external representation of formula.

While the user is handling the external representation, the system is manipulating the internal representation according to the user instructions.

2.2.1 CREATING A FORMULA

A new formula or expressions to be included in an existing formula are described similar to the way a formula is manually written. A function key or a menu selection is associated with each object, except for character strings which are specified by using an alphanumeric keyboard.

The formula is described by using these function keys. Every time the user depresses a key, a symbol representing the object, if there is one, is displayed on the screen. The symbol is displayed at the current cursor position, then the cursor moves to the position of the next mathematical object for its input.

An extra key indicating "End of Mathematical Object" is necessary in order to cause the termination of the mathematical object definition.

For example, if the user has called an integral by striking the key that represents integrals, an integral sign shows up on the screen and the cursor moves to the lower limit position, and waits for input. When the lower limit is entered the user has to press the "End" key, indicating that the lower limit is completely entered. The cursor then moves to the upper limit position and again, after this expression and the "End" key are entered, the cursor moves down to the integrand position. After this expression and the "End" key are entered, the integral is fully defined.

This "End of Mathematical Object" key also allows for optional expressions to be omitted. For instance, in the previous example, the "End" key depressed at the time of defining lower and upper limits without any other data, would have produced an integral without limits.

The system takes care of the external representation of the formula by checking that the expressions that make up a mathematical object are properly placed in the formula, and that they have an acceptable appearance.

In our integral example, the integral symbol assumes a basic size, but if the lower and upper limits

together are bigger than the integral sign then a new, taller symbol is redrawn replacing the old one.

2.2.2 EDITING A FORMULA

2.2.2.1 THE CURSOR

In most text-editors, the text cursor is a pointer that indicates position in text, usually between characters. In full-screen editors, the text cursor is usually represented by the screen cursor and can be moved around in the four basic directions by means of cursor keys.

By contrast, the cursor here may be situated on a single character, on a group of characters, or on a node (i.e. a mathematical object, like on the numerator of a fraction, or on a whole fraction).

The cursor is portrayed by a rectangular area delineating a region. Cursor movements include:

UP: to the parent of the current node (a node represents a mathematical object, i.e. a sub formula)

DOWN: to the first child (which is the leftmost-child) of the current node, or if the child is a leaf, to its first character

RIGHT: from a node to the adjacent one (i.e. its right-sibling); from a character to the next one in the present leaf

LEFT: to the previous character in the present leaf; to the previous node (sibling)

For example, in Figure 1, from character p, right movements take the cursor successively to:

+, {fraction}, +, {integral}, +, q, r, s.

From {integral}, down takes the cursor to the lower integral limit {u+v}, and if pressed again, to u; from the lower limit, right takes the cursor to the upper limit, and then to the integrand. So, the cursor moves around the formula-tree and not around the screen.

Notice, that moving the cursor left from the start of a node (subformula) might be used to go upwards, i.e. to the node's parent. However, moving right from the end of a formula does not go downwards.

2.2.2.2 FORMULA MODIFICATION

The system provides for the most common editing functions as in conventional editors. These are: delete, replace and insert.

Minor modifications are made to characters. More complex ones are made to mathematical objects. Any modification in an existing formula is performed in two steps. The user first designates the entity to be modified, and then the action to be performed on the selected entity.

Entities are either characters or mathematical objects. These entities are selected using the cursor and a rectangular box, which must be placed within the mathematical object in question or over the appropriate character.

2.2.2.2.1 SELECTION

Two function keys are needed: one to designate mathematical objects and another to designate characters.

To select a mathematical object the user moves the cursor to any place within the mathematical object. By depressing a "Select a mathematical object" key a rectangular box highlights the smallest mathematical object the cursor is in. By depressing the same key again and again, the system will successively highlight higher levels of nested mathematical objects.

To select a character string, the user moves the cursor to the first character of the sequence; then, by depressing a "Select a character" key, the character on which the cursor is positioned is highlighted. Successive strokes of the same key will allow the selection of subsequent characters.

2.2.2.2.2 OPERATIONS

Four operations can be performed on the portion selected:

Delete, redisplay the formula with the selected part erased. It causes the whole formula to be reconstructed as if the selected area had never existed.

Replace, deletes the selected part, enters input mode and replaces the new expression for the old one.

Insert Before/After, the system enters input mode and adds the new expression into the current formula

right before or after the selected part.

Fig. 7. Insert After operation

For example, the formula in Figure 7, is modified by inserting the expression: "+ c". Care should be taken to obtain the result wanted, in the first case the string "a + b" was selected, and in the second case the mathematical object radical was specified.

Fig. 8. Insert Before operation

Figure 8 displays a formula and two ways of using insert before. The boxed area indicates the selected part before inserting the character "3".

CHAPTER III

IMPLEMENTING THE SOLUTION

3.1 THE INTERACTIVE FORMULA EDITOR

3.1.1 CONFIGURATION

The system was developed in Turbo Pascal ver. 3.0 and Turbo Graphix Toolbox on an IBM PC with an IBM Graphics Card installed, running under DOS 2.10.

The Pascal language was chosen mainly because of its dynamic creation of records that is needed for the implementation of the tree, and Toolbox for its window management system.

This configuration has a resolution of 640 pixels wide by 200 pixels tall. The alphanumeric keyboard has 10 function keys and a 15-key keypad. These keys are defined by our editor. Keys have been assigned to functions that match their standard legends (such as Ins, Del, and End) when possible.

It is obvious that a higher resolution display would allow more accurate portrayal of the various symbols and characters. However, a configuration less ideal but more feasible to acquire was accepted, mainly because this one is commonly available at low cost.

The system is a large program (4,000 lines) so overlays are used to overcome the 64K code barrier.

But, even with the use of this technique we could not provide a more sophisticated editing tool,¹ like the Undo operation, which consists of reversing the last operation performed, say Delete, Insert or Replace.

The ultimate objective of the formula editor is to produce a good quality printed version of the formula, just as it appears on the screen. To accomplish this objective, we have included a module that translates the formula internal representation generated by our system into its TeX form (see section 3.1.7). TeX is presently being used in our lab with a laser printer to produce fine quality manuscripts.

3.1.2 THE SCREEN

Basically, the screen is divided in two halves: the top window is for the SYSTEM and the bottom window is for the USER AREA. Although this is not the only way the screen is divided, this is the main division of the screen. The user will see this separation most of the time.

System Area. The system area is intended for all the system's menus and instructions.

User Area. Here is where the formula is drawn by

¹ The compiler can handle up to 64k of code, 64k of data, 64k of stack and unlimited heap. The object code however cannot exceed 64k. An object code greater than 64k causes a compiler error 98. The error 98 message is the following: "Memory Overflow. You are trying to allocate more storage for variables than is available."

the actions of the function and character keys.

3.1.3 THE CURSOR

There are two kinds of cursors: the INPUT MODE cursor symbolized by the "*", and the EDIT MODE cursor, represented by a rectangular box.

The Input Mode cursor indicates a position in the formula, where the next character is going to be. In contrast with other cursors, it cannot be moved around by the user. The system keeps moving the cursor because it knows the syntax of the formula.

The Edit Mode cursor, which is handled by the user, is used to point to a part of the formula. In order to distinguish the cursor position and the actual selection of a mathematical object or character string, the rectangular frame is drawn in two styles: a dotted line style for the cursor movement and a unbroken line style for selection (see Figure 9).

Fig. 9. The Edit Mode Cursor

3.1.4 THE IFE MAIN MENU

When invoked, the Interactive Formula Editor responds with a menu of choices. These choices are:

1. Create a formula
2. Edit an existing formula

3. Save formula into disk
4. Load extra font
5. Create an alphabet set
6. Save alphabet into disk
7. Load alphabet from disk
8. Exit system

The first three options deal with the formula. The next four are about character and alphabet sets. The last option is self-explanatory.

3.1.5 OPTION 1: CREATE A FORMULA

This option displays a new menu labeled "INPUT MODE". It presents a selection of objects plus a change font feature. The function keys and what they represent are shown in Figure 10.

F1 fraction	F2 radical
F3 integral	F4 triple
F5 limit	F6 vector
F7 index	F8 exponent
F9 block	F10 change font

Fig. 10. The INPUT MODE menu

Function keys, 1 through 9, represent the first nine objects as presented under Objects in Chapter II, page 12.

Function keys 1, 2, 3, 4, 5 and 9 display symbols that corresponds to the objects they represent. All these symbols except function key 5, the limit, may increase in size as the context requires it.

The formula editor centers expressions in relation

to their symbol, like in function 1: Numerator and Denominator; function 4: lower and upper limits of triple sign; and function 5: lower expression of limit. It also centers expressions in relation to other expressions, like in function 6: vector elements.

```

        sum: 1
      product: 2
        union: 3
intersection: 4
Selection:

```

Fig. 11. The menu of Function 4

Function key 4, representing the triple object, calls up a menu with 4 choices (see Fig. 11).

Function key 9, representing the block object, calls up a menu of delimiters (see Fig. 12) and requests a pair of digits, one for each delimiter (the open delimiter and the close delimiter).

```

(: 1      ): 2
[: 3      ]: 4
{: 5      }: 6
|: 7  space: 0
Pair (##):

```

Fig. 12. The menu of Function 9

Function key 10 features the change font option. It changes the current font type in the following order: Normal (default), Greek (see Figure 13), Scientific (see Figure 14), and an extra font set.

The extra font type is provided by the user (see option 5) and loaded by selecting option 4 from the main

!		@	#	\$	%	^	&	*	(	)	-	+
1		2	3	4	5	6	7	8	9	0	-	=
Q	Θ	W	E	R	T	Y	U	I	O	P	{	}
	θ											]
A	Σ	S	D	F	G	H	J	K	L	:	"	-
	σ										,	,
	α											
	Z	Z	X	C	V	B	N	M	<	>	?	
\			X	X		β		μ	,	.	/	
			ζ	ξ								

Fig. 13. The Greek Set

!	@	#	\$	%	^	&	*	(	)	-	+
1	2	3	4	5	6	7	8	9	0	-	=
Q	W	E	R	T	Y	U	I	O	P	{	}
A	S	D	F	G	H	J	K	L	:	"	~
	Z	X	C	V	B	N	M	<	>	?	/
\											

Fig. 14. The Scientific Set

menu. Changing font types in a circular way like this, takes us from Normal to Greek, or from the extra font back to the Normal set again.

Next to the Input Mode menu, information is given about the following:

1. The name of the current font in use
2. The availability of two more keys: the "END" key, to finish the definition of a mathematical object and the "BACK SPACE" key, to erase the last character entered
3. The font template option, which is called by hitting the [Ctrl T] keys, shows the configuration of the keyboard with the characters of the current font set.

3.1.6 OPTION 2: EDIT AN EXISTING FORMULA

Before going into the Edit Mode, the system requests the name of the formula previously created. If the formula files are not present a "..not found.." message is displayed.

All formulas are stored in two files and both must be present: a "formula-name.DTA" file and a "formula-name.STR" file.

After the formula files are read, the system displays the formula on the User Area and the Edit Mode cursor is placed on the first object (leftmost-child) of the formula (root).

An Edit Mode menu shows up on the system area (See Figures 15 and 16). This menu has two parts: the first part is for the process of selecting an entity to work

on, and the second part is for selecting an operation to be performed on the entity just selected.

```
I. SELECT SUBFORMULA by typing...
  ^O  to select a math object
  ^S  to select a character
  ^K  to cancel last selection
and after selection is made, PRESS ENTER
```

Fig. 15. The Edit Mode Menu (Part I)

```
II. CHOSE ONE OPERATION
DEL      to delete
INS      to insert before
^A       to insert after
^R       to replace
```

Fig. 16. The Edit Mode Menu (Part II)

3.1.6.1 SELECTING AN ENTITY

The process of selecting an entity is done by placing the rectangular cursor on the mathematical object or character, and then proceed to: hit [Ctrl O] keys to select a mathematical object or hit [Ctrl S] keys to select a character.

Successive strokes of these keys get bigger frames and higher levels of inclusion.

For example, in Figure 4, after placing the cursor within the string "xdx", depressing the "select a math object" key will highlight the integrand with a continuous line box. By depressing the same key again, it will draw a bigger box around the integral that includes the integrand. If the same key is depressed one more time then the entire formula is highlighted.

The system can remember all the steps in its way

up, so the user may go back to the previous framed mathematical object by canceling the last selection. This is done by hitting the [Ctrl K] keys.

3.1.6.2 SELECTING AN OPERATION

There are four options to chose from. These are:

Delete Part Selected. The user is asked to press the [Enter] key to delete it or may hit [Esc] to cancel the operation.

The ? Mark. Trying to erase a required expression (i.e. a required specific element) in an object instead of the entire object causes the system to display a "?" mark in place of the required expression, to suggest the necessary presence of this element. The user may later come back to this point and replace it with something else.

For example, trying to erase a numerator from a fraction construction brings a question mark in place of the numerator.

Insert Before/After Selected Part. Causes the system to divide the top window in half. The System enters Input Mode, and this quarter of the screen is ready to accept the new subformula.

The same Input Mode principles applies here. When the creation of the new subformula is finished, the user has to hit the [Enter] key to finally insert it or hit the [Esc] key and cancel the insertion operation.

The current formula is then redisplayed with the

new addition included.

Replace Selected Part. The system behaves in the same pattern as in the Insertion operations, except that this time the system replaces the selected part with the new formula created.

3.1.7 OPTION 3: SAVE FORMULA INTO DISK

A formula written using the IFE editor can be saved into disk by selecting option 3 from the main menu.

The system keeps the formula in two files: "formula-name.DTA", which holds information about the internal structure of the formula (the tree), and "formula-name.STR", which has all the character strings that make up the formula. These two files are stored on disk when this option is called.

In addition to these two files the system generates a file named "formula-name.TEX". It is the formula expressed in its TeX form. Note that this formula output can be easily edited into manuscript.

The system can recognized most of the mathematical symbols used in TeX. Upper and lower cases Greek letters, some of the miscellaneous symbols of type Ord, five large operators in their display styles, half of the binary operations, almost all of the relations and negated relations, some of the arrows, all of the openings and closings, and two punctuation symbols.

All the objects and their specific elements of the system are translated into TeX commands.

If any of these files already exist then the system destroys them and generates new file entries in the directory.

"Formula-name" is the name of the formula given by the user, and it may be up to 8 characters long.

3.2 FONTS AND CHARACTERS

3.2.1 OPTION 4: LOAD EXTRA FONT

The system supplies three character fonts. These fonts are loaded into the computer's memory right after the editor is invoked.

However, these three fonts may not be enough to cover all the user needs, hence the system allows for one extra font set, which can be created as described below in section 3.2.2.

The editor asks for the name of the font set, which can be up to 8 characters long. A file called: "name.FON" must be present or a "..not found.." message shows up on the screen.

If this step is successful the user now has four character sets available. The name identifying the set currently in use, during input mode, is displayed in the system area.

To write characters from the extra font set, one should make sure the desired font name appears on the screen. If it does not, the font can be changed with the [F10] change font key.

As with the three font sets, the extra font set

loaded is also brought over the keyboard map of characters. The user may always look at this map during the input mode stage.

To create a "name.FON" file, select option 5 from the main menu. After creating it, one must save it into disk by calling option 6 from the same menu.

3.2.2 OPTION 5: CREATE AN ALPHABET SET

The characters that appear on the screen are created by lighting dots or pixels on the CRT screen. Which dots are lit to make up any particular character is determined by that character's dot matrix definition.

An alphabet set contains up to 94 characters. In order to use a character from an alphabet set created in this way, it must be first saved in disk and then loaded as an extra font set (see options 6 and 4).

Hence, if the character sets available are not sufficient, it is possible to create and add one's own characters.

To create a character, the system displays a matrix of empty squares (14 by 12). Each square represents a dot or a pixel. To design a character the cursor is moved around and the squares are filled in by depressing a key.

In the same way, a square may be emptied, the entire dot matrix definition can be cleared up, each character can be seen in its real size, and the character dot definition can be saved or retrieved.

The character generator commands and their corresponding keys to call them up are shown next.

```
^D Draw a pixel
^E Erase a pixel
^L Clear Window
^V View character in real size
^S Save character
^I Display character
End Leave Graphic Editor
```

Fig. 17. The Graphics Editor Menu

Cursor movement is by arrow keys on the keypad. The Graphic Editor works on a buffer, twenty-four bytes long, to hold one character dots definition.

By striking the [Ctrl S] keys, the user can copy the current definition in the buffer to the corresponding key in the font file. The system asks the user to strike the key corresponding to the character desired.

By striking the [Ctrl I] keys, the user can reverse the save character process. In this way, a character dot definition in a font file is copied into the buffer. Again, the system requests for the corresponding key to be displayed.

The system's own font sets were designed with the Graphic Editor. In the same way, the user can modify them by loading the alphabet from disk (option 7), changing it to the user's taste (option 5) and then saving it (option 6).

3.2.3 OPTION 6. SAVE ALPHABET INTO DISK

The character set generated by option 5, which is in the computer's memory, can be stored in disk.

The memory area in which this character set rests differs from the area in which the loaded extra font set is (from option 4).

The system request the font name (up to 8 characters long) and it is stored in disk. The system then generates a file named "name.FON". If a file with the same name exists, the system destroys it and generates a new file entry in the directory.

The font file created with this option cannot be used with TeX. Since there is no translation between such a font and the TeX commands. Such a feature could be added at a later date, however.

3.2.4 OPTION 7. LOAD ALPHABET FROM DISK

This is the reciprocal of option 6. System asks for a font name, which must be present or a "..not found.." message is displayed.

The character set is loaded in memory, for examination by the system's character generator.

3.3 THE IMPLEMENTATION

The size of a formula is computed by the system, from the size of its mathematical objects. That is, the dimensions of the root of the tree are computed from the dimensions of the root's children.

In addition to the tree structure, the IFE system keeps a stack of boxes corresponding to mathematical objects which are on the path from the root to the current tree node.

The bottom of the stack contains the main box (the root of the tree, corresponding to the whole formula). The top of the stack contains the current box (the current node of the tree).

From bottom to top, each box encloses the next one. So that, if the current box expands or contracts, then all enclosing boxes (all boxes of the stack) may be expanded or contracted very quickly.

During input mode these structures are kept consistent and built dynamically. Function keys (representing objects and specific elements) push boxes into the stack, while the "End of mathematical object" key pops boxes out of the stack (see Figure 18).

When the user selects part of a formula, the system builds a stack of all boxes enclosing the cursor position. This process is done by scanning the tree of the formula. This stack is then handled according to the "Select a math object" or "Select a character" keys.

When displaying a formula, the system displays boxes sequentially after all boxes have their final dimensions and locations.

$$x - \frac{1}{2 - x}$$

USER TYPES	EXPLANATION	STACK
1	Option 1: Creates a Formula	<div><div>M</div></div>
x-	This is a character string	<div><div>C</div><div>M</div></div>
F1	Function Key 1 for fraction object	<div><div>M</div><div><div>N</div><div>F</div><div>M</div></div></div>
1	The numerator string	<div><div>C</div><div>N</div><div>F</div><div>M</div></div>

Fig. 18. Stack of boxes during INPUT phase

**USER
TYPES**
EXPLANATION
STACK

End End key for the end of the
 numerator

2-x The denominator string

End End key for the end of the
 denominator

End End key for the end of the
 formula

Fig. 18. Continuation

3.4 IMPROVEMENTS

At present IFE knows only a limited number of objects, and therefore only allows formula construction using these objects.

The system could include the following two objects:

1. An OVER object, for constructions with one required expression and a symbol on top of it. This symbol would expand horizontally up to the width of the expression.
2. An UNDER object, same as above but with the symbol below the expression.

The vector object keeps its vector elements centered. Adding right and left justification, would give more flexibility to the appearance of this object.

If a formula is very long, it is not possible to fit it on the screen at one time. One feature could be left and right scrolling as well as up and down scrolling. A further facility would allow the user to gain an overall view of the formula by replacing, on the screen, parts of the formula with an ellipsis (...).

An Undo operation within the Edit Mode would enhance the system's ability to recuperate from mistakes.

After erasing or replacing a subformula, this subtree could be saved so that it could be recalled and included in the main formula.

3.5 CONCLUSION

The system was design, developed and implemented in fulfillment of my thesis requirement. Although the system only handles mathematical formulae, it could serve as the basis for a future system that will manipulate text as well as formulae.

Our system is highly interactive, it also has the capability of producing a formula in its TeX form. This seems to make our system a good tool for writting mathematical text.

Although the system seems to be complete, the true utility of our system can not be measured without putting it to a real use. This would expose the limitations of the system as well as its powers. Such evaluation will point out the further extensions that may be necessary for the system.

To the best of our knowledge, our system seems to be the only software running on PC's with the same capabilities described thorough this thesis. This definitely makes our effort worthwhile.

APPENDIX

```

program IFE;
  ( * * * * * )
  ( *
  ( *   INTERACTIVE FORMULA EDITOR   *
  ( *   -----   *
  ( *
  ( * * * * * )
{$I dummies.inc}

{$I b:TYPEDEF.FIL}
{$I b:PRIMITIV.FIL}
{$I b:OVERLAY1.FIL}
{$I b:OVERLAY2.FIL}
{$I b:SAVEFMLA.FIL}
{$I b:DISPLAYS.FIL}
{$I b:CREATES.FIL}
{$I b:EDITS.FIL}

var system code  : char;          { action to take }
    SetFile      : file of FontSet; { I/O alphabet file }
    formulaname,  : f8name;         { external formula name }
    font name     : f8name;         { external font name }
    fileFont      : fname;          { ext. name + "fon" }
    fflag         : sentinel;       { old formula existence }

const
    exxit : Sentinel = false;      { sentinel }

    (* MAIN PROGRAM *)
BEGIN
    InitGraphic;

    (* loads normal, greek & scientific fonts *)
    Clear Alpha Area;
    GoToXY(33,3); writeln ('loading fonts...');
    for aflag := 1 to 3 do
        begin
            LoadAlphabetSet (fontname [aflag]);
            if aflag <> 1 then
                Draw_template (fntname [aflag]);
        end;

    (* displays the System's menu, *)
    (* which is stored in window 1 *)
    Display System_Menu;
    Do Window3;
    Init_Values;

    (* system's main loop *)
    WHILE (NOT exxit) DO
        begin
            buflen:=1; GoToXY(27,12);
            read (system code);
            CASE system_code OF

                (* creates a new formula *)
                '1' : begin
                    Init Values;
                    Display Input Menu;
                    Do Window12;
                    Write CurrentFont;
                    Do UserWindow;

```

```

        MakeT Root (Root);
        push(Root);
        Get Input (Root);
        pop;
        Display_Formula (Root);
    end;

    (* edits an old formula *)
    '2': begin
        RestoreWindow (3,0,0);
        GoToXY(56,4); write ('Formula name (xxxxxxx):');
        GoToXY(56,5); buflen:=8; read (formulaname);
        Do UserWindow;
        Edit Formula (formulaname,fflag);
        if NOT fflag then
            begin
                GoToXY(56,6); Write('... not found ...');
            end;
        end;

    (* saves formula tree & tex commands into disk *)
    '3': begin
        RestoreWindow (3,0,0);
        GoToXY(56,4); write ('Formula name (xxxxxxx):');
        GoToXY(56,5); buflen:=8; read (formulaname);
        Save_Formula (formulaname);
    end;

    (* loads an x-tra font into the system *)
    '4': begin
        RestoreWindow (3,0,0);
        GoToXY(56,4); write ('Font name (xxxxxxx):');
        GoToXY(56,5); buflen:=8; read (font_name);
        filefont := font_name + '.FON';
        aflag:=4;
        LoadAlphabetSet (filefont);
        if xflag then
            begin
                Draw Template (font_name);
                fntname [4] := font_name;
            end
        else
            begin
                GoToXY(56,6); write ('... not found...');
            end;
        end;

    (* character generator *)
    '5': Graph_Editor;

    (* saves alphabet set from memory into disk *)
    '6': begin
        RestoreWindow (3,0,0);
        GoToXY(56,4); Write('Font name (xxxxxxx): ');
        GoToXY(56,5); Buflen:=8; read (font_name);
        filefont := font_name + '.FON';
        Assign(SetFile,filefont);
        {$I-} Reset(SetFile); {$I+}
        If IOresult = 0 then
            write(SetFile,Char_Set)
        else
            begin
                Rewrite(SetFile);
                write(SetFile,Char_Set);
            end;
        end;
    end;

```

```

        end;
        Close(SetFile);
    end;

    (* loads alphabet set from disk to memory *)
    '7': begin
        RestoreWindow (3,0,0);
        GoToXY(56,4); Write('Font name (xxxxxxx): ');
        GoToXY(56,5); buflen:=8; read (font_name);
        filefont := font_name + '.FON';
        Assign(SetFile,filefont);
        {$I-} Reset(SetFile); {$I+}
        If IOresult = 0 then
            begin
                read(SetFile,Char_Set);
                Close(SetFile);
            end
        else
            begin
                GoToXY(56,6); Write('... not found ...');
            end;
        end;

    (* exits main routine *)
    '8': exxit := true;
end; { case }

system code := ' ';
RestoreWindow(1,0,0);
end; { end of main routine loop }
LeaveGraphic;
END.

```



```

(* * * * * * * * * * * * * * * * *)
(*                                     *)
(*      TYPE DEFINITION FILE      *)
(*                                     *)
(* * * * * * * * * * * * * * * * *)

```

```

type
  Font3byte = array [0..2] of byte;           { matrix with 3 bytes }
  Font5byte = array [0..4] of byte;           { matrix with 5 bytes }
  Font8byte = array [0..7] of byte;           { matrix with 8 bytes }
  FontBit   = array [0..23] of byte;           { matrix with 24 bytes }

  FontSet   = array [33..126] of FontBit;      { Set of Characters type }
  Alphabet_Flag = integer;                     { Alphabet Set Number type }

  comand = (stringcm, fractioncm, rootcm, integralcm, triplecm,
            limitcm, vectorcm, blockcm, exponentcm, indexcm,
            leftky, rightky, upky, downky, backspky, endiky,
            chgefont, Tkey,
            SAO, SAC, CANCEL, entercm, ESCcm,
            insaftercm, insbeforecm, deletcm, replacecm, savecm,
            othercm);

  par = (open, close);                         { block delimiters type }
  Sentinel = boolean;                          { sentinel type }
  fname = string[12];                          { file name type }
  f8name = string[8];                          { 8 characters file name type }

  Cptr = ^Clist;
  Clist = record                               { list of strings }
    Value: array [1..40] of char;
  end;

  Pointer = ^Tree;
  Tree = record                                { tree of boxes }
    rstr flag: integer;
    Sibling, Parent: Pointer;
    Typecode: integer;                        { Alphabet set/Block par/
                                              Triple choice }
    Xpo, Ypo, Height, Width,                 { Point of Origin }
    Refline: integer;                        { Height & Width }
    Case Node: Char of                      { Reference line }
      'C': (CChild: Cptr);                  { String }
      'M',                                     { Formulae Root }
      'F', 'N', 'D',                          { Fraction }
      'I', 'L', 'U', 'W',                     { Integral }
      'T', 'G', 'H', 'A',                     { Triple }
      'Z', 'O',                               { Limit }
      'R', 'K', 'Q',                          { Root }
      'P',                                     { Exponent }
      'S',                                     { Index }
      'V', 'E',                               { Vector }
      'B': (Child: Pointer);                 { Block }
    end;

  Stackptr = ^SStack;
  SStack = record                             { stack of boxes }
    element: pointer;
    next: stackptr;
  end;

const
  fontname: array [1..3] of fname = ('NORMAL.FON ',

```

```

                                'GREEK.FON      ',
                                'SCIENTFC.FON');
fntname: array [1..4] of f8name = ('NORMAL ', 'GREEK ',
                                'SCIENTFC', ' ');
xflag: boolean = false;      { x-tra font flag }

Xinit: integer = 2;           { initial value of coordinates }
Yinit: integer = 5;           { or margin of user's area }
Yini: integer = 2;           { initial Y margin for the replace world coord.}
glue: integer = 2;

Cursor : Fontbit = ($00,$00,$00,$00,$00,$00,$00,$00,
                    $0C,$0C,$07,$80,$1F,$E0,$07,$80,
                    $0C,$0C,$00,$00,$00,$00,$00,$00);

ZERO: FontBit = ($00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,
                 $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00);

mask : Font8byte = ($80,$40,$20,$10,$08,$04,$02,$01);
mask0: Font8byte = ($7F,$BF,$DF,$EF,$F7,$FB,$FD,$FE);

    { integral }
int1: Font8byte = ($7E,$C3,$C3,$C0,$C0,$C0,$C0,$C0);
int2: Font8byte = ($03,$03,$03,$03,$03,$03,$C3,$7E);
I_sym width:integer = 14;      { default }
I_sym height:integer = 26;

    { triple }
ts1: Font8byte = ($C0,$C0,$C0,$C0,$C0,$60,$38,$0F);
ts2: Font8byte = ($03,$03,$03,$03,$03,$06,$1C,$F0);
ts3: Font8byte = ($0F,$38,$60,$C0,$C0,$C0,$C0,$C0);
ts4: Font8byte = ($F0,$1C,$06,$03,$03,$03,$03,$03);
T_sym height: integer = 20;    { default }
T_sym width: integer = 25;

    { limit }
zlimit: Fontbit = ($70,$30,$00,$30,$00,$00,$30,$70,
                  $CC,$30,$30,$FE,$30,$30,$FE,$30,
                  $30,$D6,$78,$78,$C6,$00,$00,$00);
Z_sym height :integer = 8;     { default }
Z_sym width :integer = 24;
Z_sym_refline :integer = 7;

    { root }
R_sym height :integer = 13;    { default }
R_sym width :integer = 30;
R_sym_refline :integer = 10;
R_sym :integer = 8;

    { block }
B_sym height: integer = 12;    { default }
B_sym width: integer = 8;
bd11: Font3byte = ($0E,$1C,$38);
bd12: Font3byte = ($38,$1C,$0E);
bd13: Font3byte = ($70,$38,$1C);
bd14: Font3byte = ($1C,$38,$70);
bd15: Font3byte = ($0F,$18,$18);
bd16: Font5byte = ($18,$18,$F0,$18,$18);
bd17: Font3byte = ($18,$18,$0F);
bd18: Font3byte = ($F0,$18,$18);
bd19: Font5byte = ($18,$18,$0F,$18,$18);
bd110: Font3byte = ($18,$18,$F0);

```

var

```

VideoBox: FontBit;           { screen image of a basic box }
Char Set: FontSet;           { memory space for one extra Font }
AlphaSet: array [1..4] of FontSet; { alphabet sets (4) }
Leaving: Sentinel;           { F10 routine sentinel }
Charter: char;               { input: ascii }
comand code: comand;         { input: command }
ChWidthH,ChHeight,ChReflne:integer;
                             {Character's width,height,and reference line }
Xcol, Ylin: integer;         { where system starts drawing }
I: integer;                  { counts # of characters in a string }
current font: f8name;        { font currently in use : name }
aflag: Alphabet_Flag;        { font index }
Root: pointer;               { pointer to root of the tree }
top_stack: Stackptr;         { top of stack }
top: pointer;                { top of stack, pointer type }
SubRoot: pointer;            { pointer to a subformulae tree }
cuw: integer;                { current user window }
Ybot: integer;               { Y bottom line to draw cursor }
Ynt: integer;                { Y initial value to start drawing }
F: File of Tree;
G: File of Clist;

```

```
(* ** * ** * ** * ** * ** * ** *)
(*
(*      PRIMITIVE routines FILE      *)
*)
(* ** * ** * ** * ** * ** * ** *)

(* fetch next keyboard input          *)
(* and translates it into a comand *)
procedure GetComand (Var ccode:comand);
begin
    read (kbd,Charter);
    case Charter of
        #1: ccode := insaftercm;
        #8: ccode := backspky;
        #11: ccode := CANCEL;
        #13: ccode := ENTERcm;
        #15: ccode := SA0;
        #18: ccode := replacecm;
        #19: ccode := SAC;
        #20: ccode := Tkey;
        #22: ccode := savecm;
        #27: begin
                read (kbd,Charter);
                case Charter of
                    #59: ccode := fractioncm;
                    #60: ccode := rootcm;
                    #61: ccode := integralcm;
                    #62: ccode := triplecm;
                    #63: ccode := limitcm;
                    #64: ccode := vectorcm;
                    #65: ccode := indexcm;
                    #66: ccode := exponentcm;
                    #67: ccode := blockcm;
                    #68: ccode := chgefont;
                    #72: ccode := upky;
                    #75: ccode := leftky;
                    #77: ccode := rightky;
                    #79: ccode := endiky;
                    #80: ccode := downky;
                    #82: ccode := insbeforecm;
                    #83: ccode := deletecm;
                end; (* case *)
            end; (* #27 *)
        #32..#126: ccode := stringcm;
        else ccode := othercm;
    end; (* case *)
end; (* getcommand *)

(* reads a pixel from a matrix of 12X2=24 bytes *)
function pix2 (imagen:FontBit;m,n:integer):integer;
var bytt,bitt:integer;
begin
    bytt := (n-1)*2 + (m-1) shr 3;
    bitt := (m-1) AND 7;
    if (imagen [bytt] AND Mask [bitt]) = 0 then
        pix2 := 0
    else
        pix2 := 1;
    end;
end;

(* Draws the cursor at [Xcol,Ylin] *)
procedure DrawCursor;
var k,x,y:integer;
    Temporal:FontBit;
```

```

by2, bt2: integer;
(* set value of pixel in memory *)
procedure Pixel_Drawn;
begin
  by2 := (y-1) * 2 + (x-1) shr 3;
  bt2 := (x-1) and 7;
  if PointDrawn (Xcol+(x-1), Ylin+(y-1)) then
    VideoBox [by2] := VideoBox [by2] or mask [bt2]
  else
    VideoBox [by2] := VideoBox [by2] and mask0 [bt2];
end;
(* draws an actual pixel on screen *)
procedure Draw_Pixel;
begin
  if pix2(Temporal, x, y) = 1 then
    DrawPoint(Xcol+(x-1), Ylin+(y-1))
  else
    begin
      SetColorBlack;
      DrawPoint(Xcol+(x-1), Ylin+(y-1));
      SetColorWhite;
    end;
end;
(* Copy a box of pixels from screen to a Videomemory *)
procedure copy_videobox;
begin
  for k := 0 to 23 do VideoBox [k] := 0;
  for y := 1 to 12 do
    for x := 1 to 14 do
      Pixel_Drawn;
    end;
end;

begin
  if ((Xcol < 1) or (Xcol > 626)) then
    Xcol := 2;
  if ((Ylin < 1) or (Ylin > Ybot)) then
    Ylin := 2;
  copy videobox;
  for k := 0 to 23 do
    Temporal[k] := Videobox[k] XOR Cursor[k];
  for y := 1 to 12 do
    for x := 1 to 14 do
      Draw_Pixel;
    end;

    (* Displays Cursor at [Xcol, Ylin], *)
    (* right after the top-object *)
  procedure Display_Cursor;
  begin
    if top <> nil then
      case top^.Node of
        'M', 'N', 'D', 'L', 'U', 'W', 'G', 'H', 'A', 'O', 'K', 'Q', 'P', 'S', 'E':
          begin
            if top^.Child = nil then
              Xcol := top^.Xpo
            else
              Xcol := top^.Xpo + top^.Width;
              Ylin := top^.Ypo + top^.Reflines - 9;
              DrawCursor;
            end;
          'B': begin
            if top^.Child = nil then
              Xcol := top^.Xpo + B_sym_width
            else

```

```

        Xcol := top^.Xpo + top^.Width - B_sym_width;
        Ylin := top^.Ypo + top^.Refline - 9;
        DrawCursor;
      end;
    'C':begin
      Xcol := top^.Xpo + top^.Width;
      Ylin := top^.Ypo;
      DrawCursor;
    end;
  end;
end;

(* push box down stack *)
procedure Push (box: pointer);
var t:stackptr;
begin
  New (t);
  t^.element := box;
  t^.next := top_stack;
  top_stack := t;
  top_ := top_stack^.element;
end;

(* pop box up stack *)
procedure Pop;
var t:stackptr;
begin
  if top_stack <> nil then
    begin
      t := top_stack;
      top_stack := top_stack^.next;
      top_ := top_stack^.element;
      Dispose (t);
    end;
end;

(* Creates a tree record that represents *)
(* a basic or specific object *)
procedure MakeT_Record (Var rcrd: pointer);
begin
  New (rcrd);
  rcrd^.Node := ' ';
  rcrd^.Child := nil;
  rcrd^.Sibling := nil;
  rcrd^.Parent := nil;
  rcrd^.Typecode := 0;
  rcrd^.Xpo := 0;
  rcrd^.Ypo := 0;
  rcrd^.Width := 0;
  rcrd^.Height := 0;
  rcrd^.Refline := 0;
end;

(* creates a string value record *)
procedure Create_Cstring (var str: Cptr);
var x:integer;
begin
  New (str);
  for x:=1 to 40 do
    str^.Value [x] := ' ';
  end;
end;

(* creates a string leave record *)
procedure Create_Leave (var ptr: pointer);

```

```

begin
  New (ptr);
  ptr^.Node:='C';
  ptr^.CChild:=nil;
  ptr^.Sibling:=nil;
  ptr^.Parent:=nil;
  ptr^.Typecode:=1;
  ptr^.Xpo:=0;
  ptr^.Ypo:=0;
  ptr^.Width:=0;
  ptr^.Height:=0;
  ptr^.Refline:=0;
end;

(* * * * * *)
(*
(*      Save & Restore tree (into & from disk)      *)
(*
(* * * * * *)

type
  que_ptr = ^que node;
  que_node = record
    info      : pointer; {tree pointer to be held by queue }
    left,right: que_ptr;  { Double linked queue }
  end;
  que_type = record
    front: que_ptr;
    rear : que_ptr;
  end; { hold queue front and rear }

procedure NewQ(var Que: que_type);
begin
  with Que do
  begin
    new(front);
    new(rear);
    front^.right:=rear;
    rear^.left:=front;
    rear^.right:=nil;
    front^.left:=nil;
  end;
end;

function EmptyQ(Que: que_type): boolean;
begin
  EmptyQ := (que.front^.right = que.rear);
end;

procedure InsertQ(var Que: que_type; P: pointer);
var Q: que_ptr;
begin
  new(Q);
  Q^.info := P;
  with Que.rear^ do
  begin
    left^.right := Q;
    Q^.right := que.rear;
    Q^.left := left;
    left := Q;
  end;
end;

procedure RemoveQ(var Que: que_type; var P: pointer);

```

```

var    Q: que_ptr;
begin
  if NOT EmptyQ(Que) then
    begin
      Q := que.front^.right;
      Q^.right^.left := que.front;
      que.front^.right := Q^.right;
      P := Q^.info;
    end
  else
    writeln('Queue is empty');
  end;

procedure Link_To_Parent(R: pointer; Parnt: pointer);
begin
  if R <> nil then
    begin
      R^.Parent := Parnt;                { Point to my parent }
      if R^.Node <> 'C' then
        Link To Parent (R^.Child, R);    { Child's parent is me }
      Link_To_Parent (R^.Sibling, Parnt); { Sibling's parent is my parent }
    end;
  end;
end;

```



```

      (* * * * * *)
      (*          big OVERLAY1 FILE          *)
      (* * * * * *)

      (* displays Introductory Menu *)
overlay PROCEDURE DISPLAY_SYSTEM_MENU;
BEGIN
  DefineWindow (1, 0,0, 54,YMaxGlb div 2);
  DefineHeader (1, 'SYSTEM');
  SelectWindow (1);
  SetHeaderOn;
  SetBackground(0);
  DrawBorder;
  GoToXY(15,3);   Writeln (' 1. Create a formula      ');
  GoToXY(15,4);   Writeln (' 2. Edit an existing formula');
  GoToXY(15,5);   Writeln (' 3. Save formula into disk ');
  GoToXY(15,6);   Writeln (' 4. Load extra font      ');
  GoToXY(15,7);   Writeln (' 5. Create an alphabet set ');
  GoToXY(15,8);   Writeln (' 6. Save alphabet into disk ');
  GoToXY(15,9);   Writeln (' 7. Load alphabet from disk ');
  GoToXY(15,10);  Writeln (' 8. Exit System          ');
  GoToXY(15,12);  Writeln (' Selection: ');
  StoreWindow (1);
END;

      (* defines physical window for main formula *)
overlay PROCEDURE DO_USERWINDOW;
var Xmin,Xmax, Ymin,Ymax: integer;
BEGIN
  Xmin := 1;  Xmax := 638;           { World margins main menu }
  Ymin := 1;  Ymax := 89;           { User window }
  DefineWindow (2, 0,(YMaxGlb div 2) + 1, XMaxGlb,YMaxGlb);
  { world definition for window #2 }
  DefineWorld (1, Xmin,Ymax, Xmax,Ymin);

  DefineHeader (2, 'U S E R   A R E A');
  SelectWorld(1);
  SelectWindow(2);
  SetBackground(0);
  DrawBorder;
  StoreWindow(2);
END;

      (* displays a blank space next to system's window *)
      (* it's used to ask for formula/font name          *)
overlay PROCEDURE DO_WINDOW3;
BEGIN
  DefineWindow (3, 55,0, XMaxGlb,YMaxGlb div 2);
  SelectWindow (3);
  SetBackground (0);
  DrawBorder;
  StoreWindow (3);
END;

      (* displays the input menu : object functions *)
overlay PROCEDURE DISPLAY_INPUT_MENU;
BEGIN
  DefineWindow (11, 0,0, 30,YMaxGlb div 2);
  DefineHeader (11, 'INPUT MODE');
  SelectWindow (11);
  SetHeaderOn;

```

```

SetBackground(0);
DrawBorder;
GoToXY(2,3);   Writeln ('F1 fraction F2  radical   ');
GoToXY(2,4);   Writeln ('F3 integral F4  triple    ');
GoToXY(2,5);   Writeln ('F5 limit   F6  vector    ');
GoToXY(2,6);   Writeln ('F7 index   F8  exponent  ');
GoToXY(2,7);   Writeln ('F9 block   F10 change font ');
StoreWindow (11);
END;

(* continuation of input menu window *)
(* it's next to input menu window *)
overlay PROCEDURE DO_WINDOW12;
BEGIN
  DefineWindow (12, 31,0, XMaxglb,YMaxGlb div 2);
  SelectWindow (12);
  SetBackground (0);
  DrawBorder;
  GoToXY(32,3); write ('Current Font: ');
  GoToXY(32,4); write ('^T      Font Template ');
  GoToXY(32,5); write ('END      ends object  ');
  GoToXY(32,6); write ('BACKSPACE del. last char');
  StoreWindow (12);
END;

(* defines window for selection/edition operations *)
overlay PROCEDURE DO_WINDOW14;
BEGIN
  DefineWindow (14, 0,0, XMaxglb,YMaxGlb div 2);
  DefineHeader (14,'EDIT MODE');
  SelectWindow (14);
  SetHeaderOn;
  SetBackground (0);
  DrawBorder;
  StoreWindow (14);
END;

(* displays edit menu: selection phase *)
overlay PROCEDURE DISPLAY_SELECT_MENU;
BEGIN
  RestoreWindow (14,0,0);
  GoToXY(2,3); write('I. SELECT SUBFORMULA by typing... ');
  GoToXY(2,4); write('  ^O  to select a math object ');
  GoToXY(2,5); write('  ^S  to select a character ');
  GoToXY(2,6); write('  ^K  to cancel last selection ');
  GoToXY(2,7); write('and after selection is made, PRESS ENTER ');
  GoToXY(45,3); write('      use arrowkeys to move around ');
  GoToXY(45,4); write('      the formula or, ');
  GoToXY(45,5); write('press ENTER to go back to main menu');
END;

(* displays edit menu: operation phase *)
overlay PROCEDURE DISPLAY_OPERATE_MENU;
BEGIN
  RestoreWindow (14,0,0);
  GoToXY(2,3); write('II. CHOSE ONE OPERATION ');
  GoToXY(2,4); write('  DEL    to delete ');
  GoToXY(2,5); write('  INS    to insert before ');
  GoToXY(2,6); write('  ^A     to insert after ');
  GoToXY(2,7); write('  ^R     to replace ');
  GoToXY(2,8); write('  any key to Escape ');
END;

(* defines window for subformula *)

```

```

    (* it's used at insert/replace operations *)
overlay PROCEDURE DO_REPLACEWINDOW;
var Xmin,Xmax, Ymin,Ymax: integer;
BEGIN
    Xmin := 1;  Xmax := 638;
    Ymin := 1;  Ymax := 42;
    DefineWindow (15, 0,57, XMaxGlb,YMaxGlb div 2);
    DefineWorld (2, Xmin,Ymax, Xmax,Ymin);
    SelectWorld(2);
    SelectWindow(15);
    SetBackground(0);
    DrawBorder;
    StoreWindow(15);
END;

    (* Clears memory space for all alphabet sets *)
overlay PROCEDURE CLEAR_ALPHA_AREA;
var
    k,l: integer;
BEGIN
    for k:=1 to 4 do
        for l:=33 to 126 do
            AlphaSet [k][l] := ZERO;
        for l:=33 to 126 do Char_Set [l] := ZERO;
        ChWidth := 14;
        ChHeight := 12;
        ChRefline := 9;
    END;

    (* Loads an Alphabet Set into memory *)
overlay PROCEDURE LOADALPHABETSET (FileFontName:fname);
var
    FileFont : file of FontSet;
    AlphabetSet: FontSet;
    l:integer;
BEGIN
    for l:=33 to 126 do AlphabetSet [l] := ZERO;
    Assign(FileFont,FileFontName);
    {$I-} Reset(FileFont);  {$I+}
    If IOresult = 0 then
        begin
            read(FileFont,AlphabetSet);
            close(FileFont);
            AlphaSet [aflag] := AlphabetSet;
            if aflag = 4 then xflag:=true;
        end
    else
        xflag:=false;
    END;

    (* Shows template of alphabet set other than normal *)
overlay PROCEDURE TEMPLATE_SHOW;
var answer: char;
BEGIN
    CopyScreen;
    ClearScreen;
    case aflag of
        2: RestoreWindow(7,0,0);
        3: RestoreWindow(8,0,0);
        4: RestoreWindow(10,0,0);
    end;
    repeat
        read(Kbd,answer);
    until answer = #27;

```

```

ClearScreen;
SwapScreen;
END;

(* identifies current font and writes *)
(* its name on screen : window 12 *)
overlay PROCEDURE WRITE_CURRENTFONT;
BEGIN
  case aflag of
    1: begin
        current font := fntname [1];
        GoToXY(46,3);
        Write(current_font);
      end;
    2: begin
        current font := fntname [2];
        GoToXY(46,3);
        Writeln(current_font);
      end;
    3: begin
        current font := fntname [3];
        GoToXY(46,3);
        Writeln (current_font);
      end;
    4: begin
        current font := fntname [4];
        GoToXY(46,3);
        Writeln (current_font);
      end;
  end; (* case *)
END;

(* initial values at input time *)
overlay PROCEDURE INIT_VALUES;
BEGIN
  cuw := 2;
  Ybot := 88;
  Ynt := Yinit;
  Xcol := Xinit;
  Ylin := Yinit;
  Root := nil;
  top_stack := nil;
  top := nil;
  I := 0;
  aflag := 1;
END;

(* Creates the root of the formulae *)
overlay PROCEDURE MAKET_ROOT (var ptr:pointer);
BEGIN
  New (ptr);
  ptr^.Node := 'M';
  ptr^.Child := nil;
  ptr^.Sibling := nil;
  ptr^.Parent := nil;
  ptr^.Xpo := Xcol;
  ptr^.Ypo := Ylin;
  ptr^.Height := ChHeight;
  ptr^.Width := ChWidth;
  ptr^.Refline := ChRefline;
END;

(* initial values at input time but, *)
(* for a subformula *)

```

```
overlay PROCEDURE INI_VALUE;
```

```
BEGIN
```

```
  cuw := 15;
  Ybot := 35;
  Ynt := Yini;
  Xcol := Xinit;
  Ylin := Yini;
  SubRoot := nil;
  top_stack := nil;
  top := nil;
  I := 0;
  aflag := 1;
```

```
END;
```

```
(* displays triple options on top of input menu *)
```

```
overlay PROCEDURE DO_MENU_TRIPLE (var t_number:integer);
```

```
BEGIN
```

```
  t_number := 0;
  while (t_number < 1) OR (t_number > 4) do
    begin
      DefineWindow (9, 3,15,23,57);
      SelectWindow (9);
      SetBackground(0);
      DrawBorder;
      GotoXY(5,3); Writeln ('      sum : 1 ');
      GotoXY(5,4); Writeln ('      product : 2 ');
      GotoXY(5,5); Writeln ('      union : 3 ');
      GotoXY(5,6); Writeln ('intersection : 4 ');
      GotoXY(5,7); Write (' Selection: ');
      buflen:=1; read(t_number);
    end;
    RestoreWindow (11,0,0);
    SelectWindow (cuw);
```

```
END;
```

```
(* displays block options on top of input menu *)
```

```
overlay PROCEDURE DO_MENU_BLOCK (var b_number:integer);
```

```
BEGIN
```

```
  b_number := -1;
  while (b_number < 0) OR (b_number > 77) do
    begin
      DefineWindow (9, 3,15,23,57);
      SelectWindow (9);
      SetBackground(0);
      DrawBorder;
      GotoXY(5,3); Writeln (' ( : 1 ) : 2 ');
      GotoXY(5,4); Writeln (' [ : 3 ] : 4 ');
      GotoXY(5,5); Writeln (' { : 5 } : 6 ');
      GotoXY(5,6); Writeln (' | : 7 space: 0 ');
      GotoXY(5,7); Write (' Pair (##) : ');
      buflen:=2; read(b_number);
    end;
    RestoreWindow (11,0,0);
    SelectWindow (cuw);
```

```
END;
```

```
(* restores tree from disk *)
```

```
overlay PROCEDURE RESTORE_TREE (var R: pointer);
```

```
var
```

```
  P,Parnt : pointer;
  S : Cptr;
  Que : que_type;
```

```
BEGIN
```

```
  R := nil;
```

```

newq(Que);
if NOT EOF(F) then
begin
    New(P);                      { Allocate memory for a new tree node }
    read(F,P^);                  { Read next node from the file }
    R := P;                      { Take the first node as a root }
    Parnt := P;                  { Hold it as a parent to which one or two son }
                                { will be attached }
end;

while NOT EOF(F) do
begin
    New(P);                      { Read next tree node }
    read(F,P^);                  { If a parent has no son (indicate by rstr_flag = 0) }
                                { get next node as parent }

    if P^.Node = 'C' then
    begin
        New(S);
        read(G,S^);
        P^.CChild := S;
    end;
    while (Parnt^.rstr_flag=0) AND NOT EmptyQ(Que) do
        RemoveQ(Que,Parnt);

    case Parnt^.rstr_flag of
        0 : writeln('Extra record left when restore tree !!!!! ');
        1,3: begin
            Parnt^.Child := P;    { The parent has one left son }
            Parnt^.rstr_flag := Parnt^.rstr_flag - 1; { Attached the new node to it }
            { Indicate no more left sons }
        end;
        2 : begin
            Parnt^.Sibling := P;  { The parent has one right son }
            Parnt^.rstr_flag := 0; { Attached the new node to it }
            { No more sons for parent }
        end;
    end; (* case *)

    InsertQ(Que,P);              { Insert the new node into queue and
                                { it will be taken out from the queue
                                { as next parent }

end; (* while *)
END;                             (* end of procedure restore_tree *)

(* Stores formula tree into disk *)
overlay PROCEDURE STORE_TREE (R: pointer);
var
    Que : que type;
    P : pointer;
    S : Cptr;
BEGIN
    newq(Que);                  { Initialize the queue }
    InsertQ(Que,R);             { Put root into the queue }
    while NOT EmptyQ(Que) do
    begin
        RemoveQ(Que,P);
        P^.rstr_flag := 0;      { get next node of the tree }
                                { Start to set the flag value }
        if P^.CChild <> nil then
            if P^.Node <> 'C' then
            begin
                P^.rstr_flag := P^.rstr_flag + 1;
                InsertQ(Que,P^.Child);
            end
        else
            begin

```

```

        S := P^.CChild;
        write(G,S^);
    end;
    if P^.Sibling <> nil then
    begin
        P^.rstr_flag := P^.rstr_flag + 2;
        InsertQ(Que,P^.Sibling);
    end;
    write(F,P^);                                { put the node into file }
end;
END;      { STORE-TREE proc }

(* changes current font, called from input menu *)
procedure Change_Font;
begin
    if aflag < 4 then
        aflag := aflag + 1
    else
        aflag := 1;
    if aflag = 4 then
        if not xflag then
            aflag := 1;
        Write_CurrentFont;
    end;
end;

```

```

      (* * * * * * * * * * * * * * * *)
      (* * * * * * * * * * * * * * * *)
      (*          big  OVERLAY2  FILE          *)
      (* * * * * * * * * * * * * * * *)
      (* * * * * * * * * * * * * * * *)

      (* Draws a template of a font          *)
      (* mapped with the standard keyboard *)
      overlay PROCEDURE DRAW_TEMPLATE (alphabet_name:f8name);
      var wdow_number:integer;

      (* Displays Keyboard arrengment on screen *)
      procedure showkybd (Window_number:integer);
      type
        ASCII_table=array[1..94] of integer;
      const
        { order in which key characters are drawn }
        Symcod:ASCII_table=(33,64,35,36,37,94,38,42,40,41,95,43,
          49,50,51,52,53,54,55,56,57,48,45,61,
          81,87,69,82,84,89,85,73,79,80,123,125,
          113,119,101,114,116,121,117,105,111,112,91,93,
          65,83,68,70,71,72,74,75,76,58,34,126,
          97,115,100,102,103,104,106,107,108,59,39,96,
          124,90,88,67,86,66,78,77,60,62,63,
          92,122,120,99,118,98,110,109,44,46,47);
      var
        e:integer;                                { ASCII-table index }
        m,n:integer;
        Xs,Ys:integer;
        X temp:integer;
        aIndx:alphabet_flag;

      procedure DrawSymbol(Ch_Def:FontBit; X_1,Y_1:integer);
      var k,l:integer;

      function Pixel 2(s,t:integer):integer;
      var byte2,bit2:integer;
      begin
        byte2:=(t-1)*2 + (s-1) SHR 3;
        bit2:=(s-1) AND 7;
        if (Ch_Def [byte2] AND Mask [bit2] = 0) then
          Pixel_2 := 0
        else
          Pixel_2 := 1;
        end;

      begin
        for k:=1 to 14 do                          { Chwidth }
          for l:=1 to 12 do                          { Chheight }
            if Pixel_2(k,l) = 1 then DrawPoint(X_1+k,Y_1+l);
          end;
        end;

      procedure DrawLineofSymbols(Code:integer);
      var mmax:integer;
      begin
        if Code < 4 then
          mmax:=12
        else
          mmax:=11;
        X temp:=Xs;
        for n:=1 to 2 do
          begin
            for m:=1 to mmax do
              begin

```



```

        DrawSymbol(AlphaSet[aindx][Symcod[e]],Xs,Ys);
        Xs:=Xs+40;
        e:=e+1;
    end;
    Ys:=Ys+17;
    Xs:=X_ temp;
end;
end;

begin
    (* Shows Keyboard template *)
    { whole screen for template }
    DefineWindow (Window number,0,0,XMaxGlb,YMaxGlb);
    { world definition for window #7 }
    DefineWorld (2, 1,200,640,1);
    ClearScreen;
    SelectWorld(2);
    SelectWindow(Window number);
    GoToXY(14,2); write('KEY ASSIGNMENT FOR FONT: ');
    GoToXY(39,2); write(alphabet name);
    GoToXY(55,2); write('PRESS <ESC> TO RETURN ');
    GoToXY(57,3); write('TO EDITING');
    (* Draws First Alphabet Set *)
    e:=1; aindx:=1;
    DrawLine(76,36,556,36); { First line }
    Xs:=76;
    for m:=1 to 13 do
    begin
        DrawLine(Xs,36,Xs,76);
        Xs:=Xs+40;
    end;
    Xs:=80; Ys:=41;
    DrawLineofSymbols(1);
    DrawLine(76,76,572,76); { Second line }
    Xs:=92;
    for m:=1 to 13 do
    begin
        DrawLine(Xs,76,Xs,116);
        Xs:=Xs+40;
    end;
    Xs:=96; Ys:=81;
    DrawLineofSymbols(2);
    DrawLine(92,116,580,116); { Third line }
    Xs:=100;
    for m:=1 to 13 do
    begin
        DrawLine(Xs,116,Xs,156);
        Xs:=Xs+40;
    end;
    Xs:=104; Ys:=121;
    DrawLineofSymbols(3);
    DrawLine(92,156,580,156); { Fourth line }
    Xs:=92;
    for m:=1 to 12 do
    begin
        DrawLine(Xs,156,Xs,196);
        Xs:=Xs+40;
    end;
    Xs:=96; Ys:=161;
    DrawLineofSymbols(4);
    DrawLine(92,196,532,196); { Fifth line }
    (* draws second alphabet set *)
    e := 1; aindx := aflag;
    Xs:=97; Ys:=41;
    DrawLineofSymbols(1);

```

```

    Xs:=113; Ys:=81;
    DrawLineofSymbols(2);
    Xs:=121; Ys:=121;
    DrawLineofSymbols(3);
    Xs:=113; Ys:=161;
    DrawLineofSymbols(4);
end;      (* end of showkeyboard template *)

BEGIN
  SelectScreen(2);
  case aflag of
    2: wdw_number:=7;
    3: wdw_number:=8;
    4: wdw_number:=10;
  end;
  ShowKybd(wdw_number);
  StoreWindow(wdw_number);
  SelectScreen(1);
END;      { DRAW-TEMPLATE proc }

(* * * * * * * * * * * * * * * * * * * * * * * * * *)
(*
(*          GRAPHIC EDITOR
(*          -----
(*      It is the Special Character Generator.
(*      Occupies the whole screen, it generates a white window
(*      where the creation of characters takes place.
(*      A list of graphic commands is given and a view of the
(*      character in real size can be seen next to the window.
(*
(* * * * * * * * * * * * * * * * * * * * * * * * *)
overlay PROCEDURE GRAPH_EDITOR;

procedure graphedit;
const
  WrealX=16; WrealY=12;           { real-size world coord. }
  WminX=1;  WmaxX=112;           { char-def.world coord. }
  WminY=1;  WmaxY=48;           { " " " " " " " " " " }
  limx1=1;  limx2=14;           { logical limits that represents }
  limy1=1;  limy2=12;           { number of pixels in a char }
type
  Char_Box=array[0..3,0..7] of 0..1;
var
  SetName:fname;                 { Symbol File Name (name + FON) }
  Set8Name:f8name;               { Symbol File Name (8 characters) }
  CharKey:char;                  { Character Key }
  ChFlag:integer;                { Character Flag }
  CharCode:integer;              { Character Code Number }

  procedure CharHandler;
  const
    Prompt:Char_Box = ((0,0,0,0,0,0,0,0),(0,1,1,1,1,1,1,0),
                       (0,1,1,1,1,1,1,0),(0,0,0,0,0,0,0,0));
  var
    ChK:char;                    { char read from keyboard }
    Gx,Gy,                       { x,y coordinates in world system }
    XM,YM:integer;               { logic coordinates }
    byte2,bit2:integer;          { byte and bit #s from char-matrix }
    CharMem:FontBit;             { character-matrix }
    PMem:Char_Box;               { cursor bit-matrix }

    procedure ClearSmallWindow; { clears window for real-size picture }
    var k,l:integer;
    begin

```

```

SetColorBlack;
for k := 1 to 14 do
  for l := 1 to 12 do
    DrawPoint(k,l);
  SetColorWhite;
end;

{ Fills screen_box with current data }
procedure DrawPrompt (xi,yj:integer);
var k,l:integer;
begin
  for l := 0 to 3 do
    for k := 0 to 7 do
      if (PMem[l,k] XOR Prompt[l,k]) = 1 then
        DrawPoint(xi+k,yj+l)
      else
        begin
          SetColorBlack;
          DrawPoint(xi+k,yj+l);
          SetColorWhite;
        end;
    end;
  end;

  { Fills screen_box memory with current data }
  procedure FillPMem(xi,yj:integer);
  var k,l:integer;
  begin
    for l := 0 to 3 do
      for k := 0 to 7 do
        begin
          if PointDrawn(xi+k,yj+l) then
            PMem[l,k] := 1
          else
            PMem[l,k] := 0;
        end;
      end;
    end;

    { Fills screen_box with previous data }
    procedure FillScreenBox(xi,yj:integer);
    var k,l:integer;
    begin
      for l := 0 to 3 do
        for k := 0 to 7 do
          begin
            if PMem[l,k] = 1 then
              DrawPoint(xi+k,yj+l)
            else
              begin
                SetColorBlack;
                DrawPoint(xi+k,yj+l);
                SetColorWhite;
              end;
          end;
        end;
      end;

      procedure FillBox(xi,yj:integer);
      var k,l:integer;
      begin
        for l := 0 to 3 do
          for k := 0 to 7 do
            DrawPoint(xi+k,yj+l);
          end;
        end;

        procedure EraseBox(xi,yj:integer);
        var k,l:integer;
        begin
          SetColorBlack;

```

```

    for l := 0 to 3 do
      for k := 0 to 7 do
        DrawPoint(xi+k,yj+1);
        SetColorWhite;
      end;
    end;

function Pixel_2(Ch_Def:FontBit; xi,yj:integer):integer;
begin
  byte2:=(yj-1)*2 + (xi-1) SHR 3;
  bit2:=(xi-1) AND 7;
  if Ch_Def [byte2] AND Mask [bit2] = 0 then
    Pixel_2 := 0
  else
    Pixel_2 := 1;
  end;
end;

procedure RedrawCharacter;
var k,l:integer;
begin
  for k:=1 to 14 do
    for l:=1 to 12 do
      if Pixel_2(CharMem,k,l) = 1 then
        DrawPoint(k,l);
      end;
    end;
  end;

procedure RedrawSymlMatrx;
begin
  for XM := 1 to 14 do
    for YM := 1 to 12 do
      if Pixel_2(CharMem,XM,YM) = 1 then
        begin
          Gx := (XM * 8) - 7;
          Gy := (YM * 4) - 3;
          Fillbox(Gx,Gy);
        end;
      end;
    end;
  end;

procedure InitMemories;
var k,l:integer;
begin
  for k:=0 to 3 do
    for l:=0 to 7 do
      PMem[k,l]:=0;
    end;
    for k:=0 to 23 do
      CharMem[k]:=0;
    end;
  end;

procedure InitPicture;
begin
  Gx := 49; XM := 7;
  Gy := 21; YM := 6;
  FillPMem (Gx,Gy);
  DrawPrompt (Gx,Gy);
end;

procedure InitDrawing;
begin
  DrawBorder;
  DrawSquare(25,9,88,40,false);
  DrawLine(1,36,112,36);
  DrawLine(57,1,57,48);
end;

begin
  InitMemories;
  { char-handler routine }

```

```

InitPicture;
repeat
  InitDrawing;
  read(kbd,ChK);
  case ChK of
    #79: begin                                { exit char-handler }
      LEAVING := true;
    end;
    #12: begin                                { clear window area }
      SetBackground(0);
      InitDrawing;
      InitMemories;
      InitPicture;
    end;
    #4:  begin                                { draw a pixel }
      Fillbox(Gx,Gy);
      byte2:=(YM-1)*2 + (XM-1) SHR 3;
      bit2:=(XM-1) AND 7;
      CharMem[byte2] := CharMem[byte2] OR Mask [bit2];
      FillPMem(Gx,Gy);
      DrawPrompt (Gx,Gy);
    end;
    #5:  begin                                { erase a pixel }
      EraseBox(Gx,Gy);
      byte2:=(YM-1)*2 + (XM-1) SHR 3;
      bit2:=(XM-1) AND 7;
      CharMem[byte2] := CharMem[byte2] AND Mask0 [bit2];
      FillPMem(Gx,Gy);
      DrawPrompt (Gx,Gy);
    end;
    #22: begin                                { view picture in normal size }
      SelectWorld(4);
      SelectWindow(6);
      ClearSmallWindow;
      RedrawCharacter;
      SelectWorld(3);
      SelectWindow(5);
    end;
    #72: begin                                { move cursor up }
      FillScreenBox(Gx,Gy);
      if (YM = limy1) then
        begin
          Gy := WmaxY - 3;
          YM := limy2;
        end
      else
        begin
          Gy := Gy - 4;
          YM := YM - 1;
        end;
      FillPMem(Gx,Gy);
      DrawPrompt (Gx,Gy);
    end;
    #80: begin                                { move cursor down }
      FillScreenBox(Gx,Gy);
      if (YM = limy2) then
        begin
          Gy := WminY;
          YM := limy1;
        end
      else
        begin
          Gy := Gy + 4;
          YM := YM + 1;
        end
      end;
  end;
end;

```

```

        end;
        FillPMem(Gx,Gy);
        DrawPrompt (Gx,Gy);
    end;
#75: begin { move cursor left }
        FillScreenBox(Gx,Gy);
        if (XM = limx1) then
            begin
                Gx := WmaxX - 7;
                XM := limx2;
            end
        else
            begin
                Gx := Gx - 8;
                XM := XM - 1;
            end;
        FillPMem(Gx,Gy);
        DrawPrompt (Gx,Gy);
    end;
#77: begin { move cursor right }
        FillScreenBox(Gx,Gy);
        if (XM = limx2) then
            begin
                Gx := WminX;
                XM := limx1;
            end
        else
            begin
                Gx := Gx + 8;
                XM := XM + 1;
            end;
        FillPMem(Gx,Gy);
        DrawPrompt (Gx,Gy);
    end;
#19: begin { save dot-matrix definition in memory }
        GoToXY(50,22); write('Character Key = ');
        BufLen:=1; read(CharKey);
        CharCode:=Ord(Charkey);
        Char Set [CharCode] := CharMem;
        repeat until Keypressed;
        GoToXY(50,22); write(' ');
    end; {end of Saving Character procedure}

#9: begin { displays dot-matrix definition from memory }
        GoToXY(50,22); write('Character Key = ');
        BufLen:=1; read(CharKey);
        CharCode:=Ord(Charkey);
        CharMem := Char Set [CharCode];
        SetBackground(0);
        InitDrawing;
        RedrawSymb1Matrx;
        InitPicture;
        SelectWorld(4);
        SelectWindow(6);
        ClearSmallWindow;
        RedrawCharacter;
        SelectWorld(3);
        SelectWindow(5);
        repeat until Keypressed;
        GoToXY(50,22); write(' ');
    end; { end of Display Character procedure }

end; { end of case }

```

```

    until LEAVING = true;
end;                                     { end of charhandler routine }

begin      (* Graph Edit proc *)
  DefineWindow (4,2,8,77,191);           { frame window for F10 routine }
  DefineWindow (5,5,24,18,71);           { character-definition window }
  DefineWindow (6,40,24,41,35);          { real-size picture window }
  DefineWorld (3,WminX,WmaxY,WmaxX,WminY); { world definition for window #5 }
  DefineWorld (4,1,WrealY,WrealX,1);     { world definition for window #6 }
                                         { draw major frame for graph editor }

  ClearScreen;
  SelectWindow(4);
  SetBackground(0);
  SetColorWhite;
  DrawBorder;

  { list of commands is displayed next }
  GoToXY(6,13); writeln('                GRAPHIC EDITOR      ');
  GoToXY(6,14); writeln('                -----      ');
  GoToXY(6,16); writeln('^D Draw a pixel');
  GoToXY(6,17); writeln('^E Erase a pixel');
  GoToXY(6,18); writeln('^L Clear window');
  GoToXY(6,19); writeln('^V View character in real size');
  GoToXY(6,20); writeln('^S Save character');
  GoToXY(6,21); writeln('^I Display character');
  GoToXY(6,22); writeln('END Leave Graphic Editor');
  GoToXY(6,23); writeln('Use arrow keys for cursor movement');
                { character-definition starts here }

  SelectWorld(3);
  SelectWindow(5);
  CharHandler;
  ClearScreen;
end;      (* end of graph-edit proc *)

BEGIN
  Leaving := false;
  CopyScreen;
  GraphEdit;
  SwapScreen;
END;      { GRAPH-EDITOR proc }

  (* Every time a char is added, or deleted,      *)
  (* the tree is traverse for new dimensions      *)
  (* from the current node back to the root      *)
  (* C may have one of the following parent nodes: *)
  (* basic objects:      M,B,S,P,                  *)
  (* specific objects:   N,D,L,U,W,G,H,A,O,K,Q,E *)
  (* these cannot be a parent of C: F,I,T,Z,R,V *)
overlay  PROCEDURE UPDATE_BOX_STACK (dummy:pointer);

  var dum,before dum,last dum:pointer;
    { all dum pointers are siblings and, dummy is their only parent }
    { before dum is a pointer called just before dum }
    { last dum is a pointer to the last occurrence of a 'C' or a 'B' }
    { that would be affected by a subsequent 'P' or 'S' node }

    sumW,maxH,maxR,maxD,lastdeep:integer;
    sumH,maxW: integer;      { vector only }
    { maxD is the maximun deep of the current subformulae }
    symH,symW,symR, part1,part2,
    aaa,bbb,wH,xxx,yyy,zzz,wR,maxim_width: integer;

  BEGIN
    while dummy <> nil do

```

```

begin
  case dummy^.Node of
    'C':begin
      dummy^.Height := ChHeight;
      dummy^.Refline := ChRefline;
      dummy^.Width := I * ChWidth;
    end;      (* end of string update *)
    'M','N','D','L','U','W','G','H','A','O','K','Q','P','S','E','B':
      begin
        if dummy^.Child = nil then
          begin
            sumW := ChWidth;
            maxH := ChHeight;
            maxR := ChRefline;
          end
        else
          begin
            sumW := 0;
            maxH := 0;
            maxR := 0;
          end;
        dum := dummy^.Child;
        while dum <> nil do
          begin
            if (dum^.Node = 'C') OR (dum^.Node = 'B') then
              last dum := dum;
            case dum^.Node of
              'P': begin
                if maxR < (last_dum^.Refline + dum^.Height) then
                  begin
                    maxD := maxH - maxR;
                    maxR := last_dum^.Refline + dum^.Height;
                    maxH := maxR + maxD;
                  end;
                if before dum^.Node <> 'S' then
                  sumW := sumW + dum^.Width
                else
                  if dum^.Width > before_dum^.Width then
                    sumW := sumW +
                      (dum^.Width - before_dum^.Width);
                  end;
              'S': begin
                { a 'S' node does not change maxR }
                maxD := maxH - maxR;
                lastdeep := last dum^.Height - last dum^.Refline;
                if maxD < (lastdeep + dum^.Height) then
                  begin
                    maxD := lastdeep + dum^.Height;
                    maxH := maxR + maxD;
                  end;
                if before dum^.Node <> 'P' then
                  sumW := sumW + dum^.Width
                else
                  if dum^.Width > before_dum^.Width then
                    sumW := sumW +
                      (dum^.Width - before_dum^.Width);
                  end;
              end;
            else begin
              if maxH < dum^.Height then
                maxH := dum^.Height;
              if maxR < dum^.Refline then
                maxR := dum^.Refline;
            end;
          end;
        end;
      end;
    end;
  end;
end

```



```

        sumW := sumW + dum^.Width;
    end;
    end;      (* end of case *)
    before dum := dum;
    dum := -dum^.Sibling;
end;      (* end of while *)

if dummy^.Node = 'B' then
begin
    if maxH < B_sym height then
        dummy^.Height := B_sym_height
    else
        dummy^.Height := maxH;
        dummy^.Width := B_sym_width + sumW + B_sym_width;
    end
else
begin
    dummy^.Height := maxH;
    dummy^.Width := sumW;
end;
dummy^.Reflines := maxR;

end;      (* end of specific objects update *)

'F': begin
    dummy^.Height := dummy^.Child^.Height + 1 +
                    dummy^.Child^.Sibling^.Height;
    if dummy^.Child^.Width > dummy^.Child^.Sibling^.Width then
        dummy^.Width := dummy^.Child^.Width
    else
        dummy^.Width := dummy^.Child^.Sibling^.Width;
    dummy^.Reflines := dummy^.Child^.Height + 1 + 4;
end;      (* end of fraction update *)

'R': begin
    aaa := 0; bbb := 0;
    xxx := 0; yyy := 0; wR := 0;
    dum := dummy^.Child;
    while dum <> nil do
    begin
        case dum^.Node of
            'K': begin
                    aaa := dum^.Height;
                    xxx := dum^.Width;
                end;
            'Q': begin
                    bbb := dum^.Height;
                    yyy := dum^.Width;
                    wR := dum^.Reflines;
                end;
        end;      (* end of case *)
        dum := dum^.Sibling;
    end;      (* end of while *)
    if (bbb + 1) < R_sym height then
        dummy^.Height := R_sym_height
    else
        dummy^.Height := bbb + 1;
    if (xxx + R_sym + yyy) < R_sym width then
        dummy^.Width := R_sym_width
    else
        dummy^.Width := xxx + R_sym + yyy;
    if wR < R_sym reflines then
        dummy^.Reflines := R_sym_reflines
    else

```

```

        dummy^.Refline := wR;
    end;      (* end of root update *)

'I': begin
    symH := I sym height;      { default }
    symR := (symH div 2) + 4;
    aaa := 0; bbb := 0;  wH := 0;  wR := 0;
    xxx := 0; yyy := 0;  zzz := 0;
    part1 := 0; part2 := 0;
    dum := dummy^.Child;
    while dum <> nil do
    begin
        case dum^.Node of
            'L': begin
                aaa := dum^.Height;
                xxx := dum^.Width;
            end;
            'U': begin
                bbb := dum^.Height;
                yyy := dum^.Width;
            end;
            'W': begin
                wH := dum^.Height;
                zzz := dum^.Width;
                wR := dum^.Refline;
            end;
        end;      (* end of case *)
        dum := dum^.Sibling;
    end;      (* end of while *)
    if (aaa + bbb) >= symH then
        symH := aaa + bbb + ChHeight;
    symR := (symH div 2) + 4;
    if symR < wR then
        part1 := 2 * (wR - 4);
    if part1 > symH then
    begin
        symH := part1;
        symR := (symH div 2) + 4;
    end;
    if (symH - symR) < (wH - wR) then
        part2 := 2 * (wH - wR + 4);
    if part2 > symH then
    begin
        symH := part2;
        symR := (symH div 2) + 4;
    end;
    dummy^.Height := symH;
    dummy^.Refline := symR;
    if xxx >= yyy then
        maxim_width := xxx
    else
        maxim_width := yyy;
    dummy^.Width := I sym width + maxim_width + glue + zzz;
end;      (* end of integral update*)

'T': begin
    symH := T sym height;  symW := T_sym_width;  { default }
    symR := (symH div 2) + 4;
    aaa := 0; bbb := 0;  wH := 0;  wR := 0;
    xxx := 0; yyy := 0;  zzz := 0; part1 := 0; part2 := 0;
    dum := dummy^.Child;
    while dum <> nil do
    begin
        case dum^.Node of

```

```

    'G': begin
        aaa := dum^.Height;
        xxx := dum^.Width;
    end;
    'H': begin
        bbb := dum^.Height;
        yyy := dum^.Width;
    end;
    'A': begin
        wH := dum^.Height;
        zzz := dum^.Width;
        wR := dum^.Reflines;
    end;
    end; (* end of case *)
    dum := dum^.Sibling;
end; (* end of while *)
if (aaa + bbb + symH) < wH then
    symH := wH;
dummy^.Height := aaa + symH + bbb;
dummy^.Reflines := bbb + (symH div 2) + 4;
if dummy^.Reflines < wR then
    part1 := (wR - 4) * 2;
if part1 > symH then
begin
    symH := part1;
    dummy^.Height := aaa + symH + bbb;
    dummy^.Reflines := bbb + (symH div 2) + 4;
end;
if (dummy^.Height - dummy^.Reflines) < (wH - wR) then
    part2 := 2 * (wH - wR + 4);
if part2 > symH then
begin
    symH := part2;
    dummy^.Height := aaa + symH + bbb;
    dummy^.Reflines := bbb + (symH div 2) + 4;
end;
if xxx >= yyy then
    maxim_width := xxx
else
    maxim_width := yyy;
if symW >= maxim_width then
    dummy^.Width := symW + zzz
else
    dummy^.Width := maxim_width + zzz;
end; (* end of triple update *)

'Z': begin
    symW := Z_sym_width;    symH := Z_sym_height;
    aaa := 0; xxx := 0;
    dum := dummy^.Child;
    if dum <> nil then
    begin
        aaa := dum^.Height;
        xxx := dum^.Width;
    end;
    if xxx > symW then
        dummy^.Width := xxx
    else
        dummy^.Width := symW + glue;
    dummy^.Height := symH + aaa;
    dummy^.Reflines := Z_sym_reflines;
end; (* end of limit update *)

'V': begin

```

```

    sumH := 0; maxW := 0;
    dum := dummy^.Child;
    while dum <> nil do
    begin
        sumH := sumH + dum^.Height;
        if maxW < dum^.Width then
            maxW := dum^.Width;
            dum := dum^.Sibling;
        end;
        dummy^.Height := sumH;
        dummy^.Width := maxW;
        dummy^.Reflines := (sumH div 2) + 4;
    end; (* end of vector update *)
end; (* case *)

dummy := dummy^.Parent;
end; (* while *)
END; (* procedure update-box-stack *)

```

```

      (* * * * * *)
      (*          SAVE  ForMuLA  FILE          *)
      (* * * * * *)

(* Save the formula in two ways : *)
(* 1. tree : name.dta & *)
(* file of strings : name.str and *)
(* 2. tex commands : name.tex *)
overlay PROCEDURE SAVE_FORMULA (flaname:f8name);
var
  file name: fname;      { formula name + extension }
  Texfile   : Text;      { tex file }
  Eflag     : Boolean;    { checks for vector elements }

(* print tex commands in a texfile *)
procedure PRINT_TEX_COM (raiz: pointer);
type
  letter = string [16];
  fonttable = array [33..127] of letter;
var
  pter: pointer;
  Blanktable: fonttable;
  Gtable: fonttable;
  Stable: fonttable;

overlay PROCEDURE FILL_GTABLE;
BEGIN
  Gtable[81] := '\Theta ' ;
  Gtable[87] := '\Omega ' ;
  Gtable[69] := 'E ' ;
  Gtable[82] := 'P ' ;
  Gtable[84] := 'T ' ;
  Gtable[89] := '\Psi ' ;
  Gtable[85] := '\Upsilon ' ;
  Gtable[73] := 'I ' ;
  Gtable[79] := 'O ' ;
  Gtable[80] := '\Pi ' ;
  Gtable[65] := 'A ' ;
  Gtable[83] := '\Sigma ' ;
  Gtable[68] := '\Delta ' ;
  Gtable[70] := '\Phi ' ;
  Gtable[71] := '\Gamma ' ;
  Gtable[72] := 'H ' ;
  Gtable[75] := 'K ' ;
  Gtable[76] := '\Lambda ' ;
  Gtable[90] := 'Z ' ;
  Gtable[88] := 'X ' ;
  Gtable[67] := '\Xi ' ;
  Gtable[66] := 'B ' ;
  Gtable[78] := 'N ' ;
  Gtable[77] := 'M ' ;
  Gtable[113] := '\theta ' ;
  Gtable[119] := '\omega ' ;
  Gtable[101] := '\epsilon ' ;
  Gtable[114] := '\rho ' ;
  Gtable[116] := '\tau ' ;
  Gtable[121] := '\psi ' ;
  Gtable[117] := '\upsilon ' ;
  Gtable[105] := '\iota ' ;
  Gtable[111] := 'o ' ;
  Gtable[112] := '\pi ' ;
  Gtable[97] := '\alpha ' ;
  Gtable[115] := '\sigma ' ;
  Gtable[100] := '\delta ' ;
  Gtable[102] := '\phi ' ;
  Gtable[103] := '\gamma ' ;
  Gtable[104] := '\eta ' ;
  Gtable[107] := '\kappa ' ;
  Gtable[108] := '\lambda ' ;
  Gtable[122] := '\zeta ' ;
  Gtable[120] := '\chi ' ;
  Gtable[99] := '\xi ' ;
  Gtable[98] := '\beta ' ;
  Gtable[110] := '\nu ' ;
  Gtable[109] := '\mu ' ;
END;

overlay PROCEDURE FILL_STABLE;
BEGIN
  Stable[81] := '\ge ' ;
  Stable[113] := '\le ' ;
  Stable[87] := '\not\equiv ' ;
  Stable[119] := '\equiv ' ;
  Stable[69] := '\gg ' ;
  Stable[101] := '\ll ' ;
  Stable[82] := '\lor ' ;
  Stable[65] := '\cap ' ;
  Stable[97] := '\cup ' ;
  Stable[83] := '\supset ' ;
  Stable[115] := '\subset ' ;
  Stable[68] := '\supseteq ' ;
  Stable[100] := '\subseteq ' ;
  Stable[70] := '\not\supset ' ;

```

```

Stable[114]:='\\land ' ;
Stable[84]:='\\not> ' ;
Stable[116]:='\\not< ' ;
Stable[89]:='\\not\\approx ' ;
Stable[121]:='\\approx ' ;
Stable[85]:='\\not\\geq ' ;
Stable[117]:='\\not\\leq ' ;
Stable[73]:='\\not\\sim ' ;
Stable[105]:='\\sim ' ;
Stable[79]:='\\not\\simeq ' ;
Stable[111]:='\\simeq ' ;
Stable[80]:='\\not\\cong ' ;
Stable[112]:='\\cong ' ;
Stable[123]:='\\rfloor ' ;
Stable[91]:='\\lfloor ' ;
Stable[125]:='\\rceil ' ;
Stable[93]:='\\lceil ' ;
Stable[90]:='\\nabla ' ;
Stable[122]:='\\partial ' ;
Stable[88]:='\\forall ' ;
Stable[120]:='\\exists ' ;
Stable[67]:='\\emptyset ' ;
Stable[78]:='\\Leftarrow ' ;
Stable[77]:='\\Leftrightarrow ' ;
Stable[60]:='\\neg ' ;
Stable[62]:='\\vdots ' ;
Stable[63]:='\\prime ' ;
Stable[124]:='\\div ' ;

Stable[102]:='\\not\\subset ' ;
Stable[71]:='\\not\\supseteq ' ;
Stable[103]:='\\not\\subseteq ' ;
Stable[72]:='\\propto ' ;
Stable[104]:='\\infty ' ;
Stable[74]:='\\notin ' ;
Stable[106]:='\\in ' ;
Stable[107]:='\\ni ' ;
Stable[76]:='\\Vert ' ;
Stable[108]:='\\vert ' ;
Stable[58]:='\\dashv ' ;
Stable[59]:='\\vdash ' ;
Stable[34]:='\\top ' ;
Stable[39]:='\\bot ' ;
Stable[126]:='\\mp ' ;
Stable[96]:='\\pm ' ;

Stable[86]:='\\gets ' ;
Stable[118]:='\\to ' ;
Stable[66]:='\\uparrow ' ;
Stable[98]:='\\downarrow ' ;
Stable[99]:='\\not=' ;
Stable[110]:='\\rightarrow ' ;
Stable[109]:='\\leftrightarrow ' ;
Stable[44]:='\\mapsto ' ;
Stable[46]:='\\ldots ' ;
Stable[47]:='\\angle ' ;
Stable[92]:='\\times ' ;

```

END;

```

overlay PROCEDURE CHOSE_DELIM (delimiter_number: integer;
                                var del_choice: letter);

```

```

BEGIN

```

```

    case delimiter_number of
        0: del_choice:= ' . ' ;
        1: del_choice:= ' ( ' ;
        2: del_choice:= ' ) ' ;
        3: del_choice:= ' [ ' ;
        4: del_choice:= ' ] ' ;
        5: del_choice:= ' \\{ ' ;
        6: del_choice:= ' \\} ' ;
        7: del_choice:= ' \\vert ' ;

```

```

    end;

```

```

END;

```

```

procedure print_subformula(pnt:pointer);

```

```

var t:pointer;
    snt:Cptr;
    tchoice,bchoice:letter;
    delim1,delim2:integer;
    total,partial,Finx:integer;
    chac:char;

```

```

begin

```

```

    case pnt^.Node of
        'C': begin
            total:=pnt^.Width div ChWidth;
            snt:=pnt^.CChild;
            for partial:=1 to total do
                begin
                    chac:=snt^.Value[partial];
                    if (chac=' ') then
                        if (Eflag) then
                            write(Texfile,'&')
                        else

```

```

        write(Texfile, '\ ')
    else
        case (pnt^.Typecode) of
            1: write(Texfile, chac);
            2: begin
                Finx:=ord(snt^.Value[parcial]);
                write(Texfile, Gtable[Finx]);
            end;
            3: begin
                Finx:=ord(snt^.Value[parcial]);
                write(Texfile, Stable[Finx]);
            end;
        end; (* case *)
    end;
end;
'N', 'D', 'L', 'U', 'W', 'A', 'G', 'H', 'K', 'Q', 'E', 'O':
begin
    if pnt^.Node = 'E' then Eflag:=true;
    t:=pnt^.Child;
    write(Texfile, '{');
    while t <> nil do
        begin
            print_subformula(t);
            t:=t^.Sibling;
        end;
    write(Texfile, '}');
    if pnt^.Node = 'E' then Eflag:=false;
end;
'F': begin
    write(Texfile, '{');
    print_subformula(pnt^.Child);
    write(Texfile, '\over\displaystyle ');
    print_subformula(pnt^.Child^.Sibling);
    write(Texfile, '}');
end;
'R': begin
    t:=pnt^.Child;
    if t^.Node = 'K' then
        begin
            write(Texfile, '\root ');
            print_subformula(t);
            t:=t^.Sibling;
            write(Texfile, '\of ');
        end
    else
        write(Texfile, '\sqrt ');
        if t^.Node = 'Q' then
            print_subformula(t);
        end;
end;
'I': begin
    write(Texfile, '\int');
    t:=pnt^.Child;
    if t^.Node = 'L' then
        begin
            write(Texfile, ' ');
            print_subformula(t);
            t:=t^.Sibling;
        end;
    if t^.Node = 'U' then
        begin
            write(Texfile, '^');
            print_subformula(t);
            t:=t^.Sibling;
        end;
end;

```

```

        if t^.Node = 'W' then
            print_subformula(t);
        end;
'T': begin
    case pnt^.Typecode of
        1: tchoice:='\sum\limits';
        2: tchoice:='\prod\limits';
        3: tchoice:='\bigcup\limits';
        4: tchoice:='\bigcap\limits';
    end;
    write(Texfile,tchoice);
    t:=pnt^.Child;
    if t^.Node = 'G' then
        begin
            write(Texfile,' ');
            print_subformulaā(t);
            t:=t^.Sibling;
        end;
    if t^.Node = 'H' then
        begin
            write(Texfile,'^');
            print_subformula(t);
            t:=t^.Sibling;
        end;
    if t^.Node = 'A' then
        print_subformula(t);
    end;
'Z': begin
    write(Texfile,'\lim\limits');
    t:=pnt^.Child;
    if t <> nil then
        begin
            write(Texfile,' ');
            print_subformulaā(t);
        end;
    end;
'V': begin
    write(Texfile,'\matrix{');
    t:=pnt^.Child;
    while t <> nil do
        begin
            print_subformula(t);
            write(Texfile,'\cr ');
            t:=t^.Sibling;
        end;
    write(Texfile,'}');
    end;
'S': begin
    write(Texfile,'_');
    t:=pnt^.Child;
    write(Texfile,'{');
    while t <> nil do
        begin
            print_subformula(t);
            t:=t^.Sibling;
        end;
    write(Texfile,'}');
    end;
'P': begin
    write(Texfile,'^');
    t:=pnt^.Child;
    write(Texfile,'{');
    while t <> nil do
        begin

```



```

        print subformula(t);
        t:=t^.Sibling;
    end;
    write(Texfile,')');
end;
'B': begin
    delim1:=pnt^.Typecode div 10;
    delim2:=pnt^.Typecode - (delim1*10);
    chose delim(delim1,bchoice);
    write(Texfile,'\left ',bchoice);
    t:=pnt^.Child;
    write(Texfile,'{');
    while t <> nil do
        begin
            print subformula(t);
            t:=t^.Sibling;
        end;
        write(Texfile,')');
        chose delim(delim2,bchoice);
        write(Texfile,'\right ',bchoice);
    end;
end; (* case *)
pnt:=pnt^.Sibling;
end; (* while <> nil *)

begin { PRINT-TEX-COMMANDS proc }
    Eflag:=false;
    for i:=33 to 127 do
        Blanktable[i]:= ' ';
    Gtable := Blanktable;
    fill gtable;
    StabLe := Blanktable;
    fill stable;
    pter:= raiz^.Child;
    write(Texfile,'$$');
    while pter <> nil do
        begin
            print subformula(pter);
            pter:=pter^.Sibling;
        end;
        writeln(Texfile,'$$');
        write(Texfile,'\end');
    end; { PRINT-TEX-COM proc }

BEGIN { SAVE FORMULA procedure }
    IF ROOT <> NIL THEN
        BEGIN
            file name := flaname + '.tex';
            Assign(Texfile,'b:' + file name);
            {$I-} Reset(Texfile); {$I+}
            if IOresult = 0 then
                Erase(Texfile);
            Rewrite(Texfile);
            Print TEX com (Root);
            Close(TexFile);

            file name := flaname + '.dta';
            Assign(F, 'b:' + file name);
            {$I-} Reset(F); {$I+}
            if IOresult = 0 then
                Erase(F);
            Rewrite(F);
            Assign(G, 'b:' + flaname + '.str');

```

```

    {$I-} Reset(G); {$I+}
    if IOresult = 0 then
        Erase(G);
        Rewrite(G);
        store tree (Root);
        Close(F);
        Close(G);
    END;
END; { SAVE-FORMULA proc }

procedure divide2;
begin
    { it separates overlays: save formula overlay }
    { from displays overlay }
end;

```

```

(* * * * *
(*
(*          DISPLAYS  FILE
(*
(*  Displays characters, symbols and menus on screen
(*
(* * * * * *

```

```

(* Display Formula, starts at root *)
overlay PROCEDURE DISPLAY FORMULA (r:pointer);
var wid,baseline:integer;
    n,beforen:pointer;      { beforen is a pointer called before n }
    lastn:pointer;          { lastn points to a node subject to
                             Power or Index }

```

```

(* Reads a pixel in a matrix of 8X1=8 bytes *)
function pix1 (imagen:Font8byte;m,n:integer):integer;
var bytt,bitt:integer;
begin
    bytt := (n-1) + (m-1) shr 3;
    bitt := (m-1) AND 7;
    if (imagen [bytt] AND Mask [bitt]) = 0 then
        pix1 := 0
    else
        pix1 := 1;
end;

```

```

(* reads a pixel from a matrix of 8X3=24 bytes *)
function pix3 (imagen:FontBit;m,n:integer):integer;
var bytt,bitt:integer;
begin
    bytt := (n-1)*3 + (m-1) shr 3;
    bitt := (m-1) AND 7;
    if (imagen [bytt] AND Mask [bitt]) = 0 then
        pix3 := 0
    else
        pix3 := 1;
end;

```

```

(* Reads a pixel in a matrix of 3 bytes *)
function pix4 (imagen:Font3Byte; m,n:integer):integer;
var bytt,bitt:integer;
begin
    bytt := (n-1) + (m-1) shr 3;
    bitt := (m-1) AND 7;
    if (imagen [bytt] AND Mask [bitt]) = 0 then
        pix4 := 0
    else
        pix4 := 1;
end;

```

```

(* Reads a pixel in a matrix of 5 bytes *)
function pix5 (imagen:Font5byte; m,n:integer):integer;
var bytt,bitt:integer;
begin
    bytt := (n-1) + (m-1) shr 3;
    bitt := (m-1) AND 7;
    if (imagen [bytt] AND Mask [bitt]) = 0 then
        pix5 := 0
    else
        pix5 := 1;
end;

```

```

(* Erases a Character at [Xcol,Ylin] *)

```

```

procedure EraseAchar(atype:integer);
var x,y:integer;
begin
  SetColorBlack;
  for y := 1 to 12 do
    for x := 1 to 14 do
      DrawPoint(Xcol+(x-1),Ylin+(y-1));
    SetColorWhite;
  end;

  (* draws a character at [Xcol,Ylin] *)
procedure DisplayAchar (Ctype:integer; Cha: char);
var x,y: integer;
    Code: integer;
begin
  Code := ord(cha);
  If Code = 32 then
    EraseAchar (Ctype)
  else
    begin
      VideoBox := AlphaSet[Ctype][Code];
      for y := 1 to 12 do
        for x := 1 to 14 do
          if pix2 (VideoBox,x,y) = 1 then
            DrawPoint(Xcol+(x-1),Ylin+(y-1));
        end;
      end;
    end;

(* returns maximun width between the two integral limits *)
function Max_width (a,b:pointer): integer;
var aa,bb:integer;
begin
  aa := 0;  bb := 0;
  case a^.Node of
    'L': aa := a^.Width;
    'U': bb := a^.Width;
  end;
  if b <> nil then
    case b^.Node of
      'U': bb := b^.Width;
    end;
  if aa >= bb then
    Max_width := aa
  else
    Max_width := bb;
end;

(* finds the maximun width between all 3 triple children *)
function max3children (child1,child2: pointer): integer;
var m1,m2,m3: integer;
begin
  m1 := 0;  m2 := 0;
  case child1^.Node of
    'G': m1 := child1^.Width;
    'H': m2 := child1^.Width;
  end;
  if child2 <> nil then
    case child2^.node of
      'H': m2 := child2^.width;
    end;
  if m1 >= m2 then
    m3 := m1
  else
    m3 := m2;

```

```

    if T_sym width > m3 then
        max3children := T_sym_width
    else
        max3children := m3;
end;

(* Draws upper or lower part of the integral symbol *)
(* or the union and intersection symbols *)
procedure Draw_limits (lmt: Font8byte);
var x,y:integer;
begin
    for y := 1 to 8 do
        for x := 1 to 8 do
            if pix1 (lmt,x,y) = 1 then
                DrawPoint(Xcol+(x-1), Ylin+(y-1));
            end;
        end;
    end;

    (* Draws the square root symbol *)
    procedure Display_Rsymbol (pp:pointer);
    var px1,px2,px3, py1,py2,py3:integer;
        offx1,offx2:integer;
        Rchild: pointer;
    begin
        offx1 := 0; offx2 := 0;
        Rchild := pp^.Child;
        while Rchild <> nil do
            begin
                case Rchild^.Node of
                    'K': offx1 := Rchild^.Width;
                    'Q': offx2 := Rchild^.Width;
                end;
                Rchild := Rchild^.Sibling;
            end;
            px1 := pp^.Xpo + offx1;
            px2 := pp^.Xpo + offx1 + R_sym;
            px3 := pp^.Xpo + offx1 + R_sym + offx2;
            py1 := pp^.Ypo;
            py2 := pp^.Ypo + ((pp^.Height * 2) div 3);
            py3 := pp^.Ypo + pp^.Height;
            DrawLine (px1,py2, px2,py3);
            DrawLine (px2,py3, px2,py1);
            DrawLine (px2,py1, px3,py1);
        end;

    (* Draws an Integral symbol *)
    procedure Display_Isymbol(pp:pointer);
    var dist,Xline,Ylin1,Yline2:integer;
    begin
        Xcol := pp^.Xpo + 6;
        Ylin := pp^.Ypo;
        Draw_limits (int1);
        Xline := pp^.Xpo + 6;
        dist := pp^.Height - 16;
        Ylin1 := pp^.Ypo + 8;
        Yline2 := pp^.Ypo + 8 + dist;
        DrawLine (Xline,Ylin1, Xline,Yline2);
        DrawLine (Xline+1,Ylin1, Xline+1,Yline2);
        Xcol := pp^.Xpo;
        Ylin := pp^.Ypo + 8 + dist;
        Draw_limits (int2);
    end;

    (* Draws a Triple symbol: sum, product, union, intersection *)
    procedure Display_Tsymbol(pp:pointer; maxi3:integer);

```

```

var h1,h2,h3,Tlenght,offx,
    px1,px2,px3,px4,py1,py2,py3:integer;
Tchild:pointer;
begin
{ compute symbol height: Tlenght }
h1 := 0; h2 := 0; h3 := 0;
Tchild := pp^.Child;
while Tchild <> nil do
begin
case Tchild^.Node of
'G': h1 := Tchild^.Height;
'H': h2 := Tchild^.Height;
'A': h3 := Tchild^.Height;
end;
Tchild := Tchild^.Sibling;
end;
Tlenght := pp^.Height - h1 - h2;
{ compute symbol coordinates }
offx := (maxi3 - T sym width) div 2;
px1 := pp^.Xpo + offx;
py1 := pp^.Ypo + h2;
case pp^.Typecode of
1 : begin { sum }
px2 := px1 + T sym width;
py2 := py1 + Tlenght;
px3 := px1 + (px2 - px1) div 2;
py3 := py1 + (py2 - py1) div 2;
DrawLine (px1,py1, px2,py1);
DrawLine (px1,py1, px3,py3);
DrawLine (px3,py3, px1,py2);
DrawLine (px1,py2, px2,py2);
end;
2 : begin { product }
px2 := px1 + T sym width;
px3 := px1 + (T sym width div 3);
px4 := px1 + ((T sym width * 2) div 3);
py2 := py1 + Tlenght;
DrawLine (px1,py1, px2,py1);
DrawLine (px3,py1, px3,py2);
DrawLine (px3-1,py1, px3-1,py2);
DrawLine (px4,py1, px4,py2);
DrawLine (px4+1,py1, px4+1,py2);
end;
3 : begin { union }
px2 := px1 + 8;
px3 := px1 + (T sym width - 8);
px4 := px1 + T sym width - 1;
py2 := py1 + (Tlenght - 8);
py3 := py1 + Tlenght;
DrawLine (px1,py1, px1,py2);
DrawLine (px1+1,py1, px1+1,py2);
Xcol := px1;
Ylin := py2;
Draw limits (ts1);
DrawLine (px2,py3, px3,py3);
Xcol := px3;
Ylin := py2;
Draw limits (ts2);
DrawLine (px4,py1, px4,py2);
DrawLine (px4-1,py1, px4-1,py2);
end;
4 : begin { intersection }
px2 := px1 + 8;
px3 := px1 + (T sym width - 8);

```

```

        px4 := px1 + T_sym_width - 1;
        py2 := py1 + 8;
        py3 := py1 + Tlenght;
        DrawLine (px1,py2, px1,py3);
        DrawLine (px1+1,py2, px1+1,py3);
        Xcol := px1;
        Ylin := py1;
        Draw limits (ts3);
        DrawLine (px2,py1, px3,py1);
        Xcol := px3;
        Ylin := py1;
        Draw limits (ts4);
        DrawLine (px4,py2, px4,py3);
        DrawLine (px4-1,py2, px4-1,py3);
    end;
end; (* case *)
end; (* procedure display-Tsymbol *)

(* Draws a limit symbol *)
procedure Display Zsymbol (pp: pointer);
var x,y, offx:integer;
begin
    if pp^.Child <> nil then
        if Z_sym_width < pp^.Child^.Width then
            offx := (pp^.Child^.Width - Z_sym_width) div 2
        else
            offx := (Z_sym_width - pp^.Child^.Width) div 2
        else
            offx := 0;
        Xcol := pp^.Xpo + offx;
        Ylin := pp^.Ypo;
        for y := 1 to Z_sym_height do
            for x := 1 to Z_sym_width do
                if pix3 (zlimt,x,y) = 1 then
                    DrawPoint(Xcol+(x-1), Ylin+(y-1));
            end;
        end;

    (* Draws a block delimiter *)
    procedure Display Bsymbol (delimiter:par; pp:pointer);
    var cdel,dlm:integer;
        px1,px2,px3,py1,py2,py3,py4,py5,dist:integer;
        x,y:integer;
    begin
        dlm := -1;
        cdel := (pp^.Typecode div 10);
        py1 := pp^.Ypo;
        if delimiter = open then
            begin
                dlm := cdel;
                px1 := pp^.Xpo;
            end;
        if delimiter = close then
            begin
                px1 := pp^.Xpo + pp^.Width - B_sym_width;
                dlm := pp^.Typecode - (cdel * 10);
            end;
        case dlm of
            0: begin { blank }
                    SetColorBlack;
                    for y:=1 to pp^.Height do
                        for x:=1 to B_sym_width do
                            DrawPoint(px1+(x-1), py1+(y-1));
                        SetColorWhite;
                    end;
                end;

```

```

1: begin      { left parenthesis }
  px2 := px1 + 2;
  py2 := py1 + 3;
  dist := pp^.Height - 6;
  py3 := py2 + dist - 1;
  for y := 1 to 3 do
    for x := 1 to 8 do
      if pix4 (bdl1,x,y) = 1 then
        DrawPoint(px1+(x-1), py1+(y-1));
      DrawLine (px2,py2, px2,py3);
      DrawLine (px2+1,py2, px2+1,py3);
      DrawLine (px2+2,py2, px2+2,py3);
    for y := 1 to 3 do
      for x := 1 to 8 do
        if pix4 (bdl2,x,y) = 1 then
          DrawPoint(px1+(x-1), py3+(y-1));
        end;
2: begin      { right parenthesis }
  px2 := px1 + 3;
  py2 := py1 + 3;
  dist := pp^.Height - 6;
  py3 := py2 + dist - 1;
  for y := 1 to 3 do
    for x := 1 to 8 do
      if pix4 (bdl3,x,y) = 1 then
        DrawPoint(px1+(x-1), py1+(y-1));
      DrawLine (px2,py2, px2,py3);
      DrawLine (px2+1,py2, px2+1,py3);
      DrawLine (px2+2,py2, px2+2,py3);
    for y := 1 to 3 do
      for x := 1 to 8 do
        if pix4 (bdl4,x,y) = 1 then
          DrawPoint(px1+(x-1), py3+(y-1));
        end;
3: begin      { left bracket }
  px2 := px1 + 1;
  px3 := px1 + 6;
  py2 := py1 + pp^.Height - 1;
  DrawLine (px2,py1, px3,py1);
  DrawLine (px2,py1, px2,py2);
  DrawLine (px2+1,py1, px2+1,py2);
  DrawLine (px2,py2, px3,py2);
  end;
4: begin      { right bracket }
  px2 := px1 + 1;
  px3 := px1 + 6;
  py2 := py1 + pp^.Height - 1;
  DrawLine (px2,py1, px3,py1);
  DrawLine (px3,py1, px3,py2);
  DrawLine (px3-1,py1, px3-1,py2);
  DrawLine (px2,py2, px3,py2);
  end;
5: begin      { left brace }
  px2 := px1 + 3;
  py2 := py1 + 3;
  dist := (pp^.Height - 11) div 2;
  py3 := py2 + dist;
  py4 := py3 + 5;
  py5 := py4 + dist;
  for y := 1 to 3 do
    for x := 1 to 8 do
      if pix4 (bdl5,x,y) = 1 then
        DrawPoint (px1+(x-1), py1+(y-1));
      if py2 < py3 then

```



```

        begin
            DrawLine (px2,py2, px2,py3);
            DrawLine (px2+1,py2, px2+1,py3);
        end;
    for y := 1 to 5 do
        for x := 1 to 8 do
            if pix5 (bdl6,x,y) = 1 then
                DrawPoint (px1+(x-1), py3+(y-1));
            if py4 < py5 then
                begin
                    DrawLine (px2,py4, px2,py5);
                    DrawLine (px2+1,py4, px2+1,py5);
                end;
            for y := 1 to 3 do
                for x := 1 to 8 do
                    if pix4 (bdl7,x,y) = 1 then
                        DrawPoint (px1+(x-1), py5+(y-1));
                    end;
        6: begin          { right brace }
            px2 := px1 + 3;
            py2 := py1 + 3;
            dist := (pp^.Height - 11) div 2;
            py3 := py2 + dist;
            py4 := py3 + 5;
            py5 := py4 + dist;
            for y := 1 to 3 do
                for x := 1 to 8 do
                    if pix4 (bdl8,x,y) = 1 then
                        DrawPoint (px1+(x-1), py1+(y-1));
                    if py2 < py3 then
                        begin
                            DrawLine (px2,py2, px2,py3);
                            DrawLine (px2+1,py2, px2+1,py3);
                        end;
                    for y := 1 to 5 do
                        for x := 1 to 8 do
                            if pix5 (bdl9,x,y) = 1 then
                                DrawPoint (px1+(x-1), py3+(y-1));
                            if py4 < py5 then
                                begin
                                    DrawLine (px2,py4, px2,py5);
                                    DrawLine (px2+1,py4, px2+1,py5);
                                end;
                            for y := 1 to 3 do
                                for x := 1 to 8 do
                                    if pix4 (bdl10,x,y) = 1 then
                                        DrawPoint (px1+(x-1), py5+(y-1));
                                    end;
            7: begin          { straight line }
                px2 := px1 + 3;
                py2 := py1 + pp^.Height - 1;
                DrawLine (px2,py1, px2,py2);
                DrawLine (px2+1,py1, px2+1,py2);
            end;
        end;      (* end of case *)
    end;      (* proc display block symbol *)

    (* Displays an object *)
    procedure Display_Node (p:pointer);
    var ptr: pointer;
    beforeptr: pointer;
    { beforeptr is a pointer called before ptr }
    { lastptr is a pointer to a previous object }
    { that is affected by an exponent or a subindex object }

```

```

    lastptr: pointer;
    s:Cptr;
    Many,Count:integer;
    BarY,off:integer;
    base,wi,hei:integer;
    mx3:integer;
begin
    if p <> nil then
        case p^.Node of
            'C': begin
                Many := p^.Width div Chwidth;
                Xcol := p^.Xpo;
                Ylin := p^.Ypo;
                s := p^.CChild;
                for Count := 1 to Many do
                    begin
                        DisplayAchar (p^.typecode,s^.Value [Count]);
                        Xcol := Xcol + Chwidth;
                    end;
                end;
            'N','D','L','U','W','G','H','A','O','K','Q','P','S','E','B':
                begin
                    wi := 0;
                    if p^.Node = 'B' then
                        begin
                            Display_Bsymbol (open,p);
                            wi := B_sym_width;
                        end;
                    base := p^.Ypo + p^.Reflines;
                    ptr := p^.Child;
                    while ptr <> nil do
                        begin
                            if (ptr^.Node = 'C') OR (ptr^.Node = 'B') then
                                lastptr := ptr;
                            case ptr^.Node of
                                'P': begin
                                    ptr^.Ypo := base - lastptr^.Reflines - ptr^.height;
                                    if beforeptr^.Node <> 'S' then
                                        begin
                                            ptr^.Xpo := p^.Xpo + wi;
                                            wi := wi + ptr^.Width;
                                        end
                                    else
                                        begin
                                            ptr^.Xpo := beforeptr^.Xpo;
                                            if beforeptr^.Width < ptr^.Width then
                                                wi := wi + (ptr^.Width - beforeptr^.Width);
                                            end;
                                        end;
                                    end;
                                'S': begin
                                    ptr^.Ypo := base +
                                        (lastptr^.Height - lastptr^.Reflines);
                                    if beforeptr^.Node <> 'P' then
                                        begin
                                            ptr^.Xpo := p^.Xpo + wi;
                                            wi := wi + ptr^.Width;
                                        end
                                    else
                                        begin
                                            ptr^.Xpo := beforeptr^.Xpo;
                                            if beforeptr^.Width < ptr^.Width then
                                                wi := wi + (ptr^.Width - beforeptr^.Width);
                                            end;
                                        end;
                                    end;
                            end;
                        end;
                    end;
                end;
        end;
    end;
end;

```

```

        else begin
            ptr^.Xpo := p^.Xpo + wi;
            ptr^.Ypo := base - ptr^.Reflines;
            wi := wi + ptr^.Width;
        end;
    end; (* end of case *)
    Display Node (ptr);
    beforeptr := ptr;
    ptr := ptr^.Sibling;
end; (* end of while *)
if p^.Node = 'B' then
    Display_Bsymbol (close,p);
end;
'F': begin
    ptr := p^.Child;
    off := (p^.Width - ptr^.Width) div 2;
    ptr^.Xpo := p^.Xpo + off;
    ptr^.Ypo := p^.Ypo;
    Display Node (ptr);
    BarY := p^.Ypo + p^.Child^.Height + 1;
    DrawLine (p^.Xpo,BarY, p^.Xpo + p^.Width,BarY);
    ptr := p^.Child^.Sibling;
    off := (p^.Width - ptr^.Width) div 2;
    ptr^.Xpo := p^.Xpo + off;
    ptr^.Ypo := BarY + 1;
    Display_Node (ptr);
end;
'R': begin
    Display_Rsymbol (p);
    off := 0;
    ptr := p^.Child;
    if ptr^.Node = 'K' then
        begin
            ptr^.Xpo := p^.Xpo;
            ptr^.Ypo := p^.Ypo;
            off := ptr^.Width;
            Display Node (ptr);
            ptr := ptr^.Sibling;
        end;
    if ptr^.Node = 'Q' then
        begin
            ptr^.Xpo := p^.Xpo + off + R_sym + 1;
            ptr^.Ypo := p^.Ypo + 1;
            Display_Node (ptr);
        end;
    end;
end;
'I': begin
    Display_Isymbol (p);
    ptr := p^.Child;
    if ptr^.Node = 'L' then
        begin
            ptr^.Xpo := p^.Xpo + I_sym width;
            ptr^.Ypo := p^.Ypo + p^.Height - ptr^.Height;
            Display Node (ptr);
            ptr := ptr^.Sibling;
        end;
    if ptr^.Node = 'U' then
        begin
            ptr^.Xpo := p^.Xpo + I_sym_width;
            ptr^.Ypo := p^.Ypo;
            Display Node (ptr);
            ptr := ptr^.Sibling;
        end;
    if ptr^.Node = 'W' then

```

```

begin
  ptr^.Xpo := p^.Xpo + I_sym width + glue
             + max width (p^.Child, p^.Child^.Sibling);
  ptr^.Ypo := p^.Ypo + p^.Reflines - ptr^.Reflines;
  Display_Node (ptr);
end;
end;      (* integral *)
'T': begin
  mx3 := max3children (p^.Child, p^.Child^.Sibling);
  Display_Tsymbol (p, mx3);
  ptr := p^.Child;
  if ptr^.Node = 'G' then
    begin
      off := (mx3 - ptr^.Width) div 2;
      ptr^.Xpo := p^.Xpo + off;
      ptr^.Ypo := p^.Ypo + p^.Height - ptr^.Height;
      Display_Node (ptr);
      ptr := ptr^.Sibling;
    end;
  if ptr^.Node = 'H' then
    begin
      off := (mx3 - ptr^.Width) div 2;
      ptr^.Xpo := p^.Xpo + off;
      ptr^.Ypo := p^.Ypo;
      Display_Node (ptr);
      ptr := ptr^.Sibling;
    end;
  if ptr^.Node = 'A' then
    begin
      ptr^.Xpo := p^.Xpo + mx3;
      ptr^.Ypo := p^.Ypo + p^.Reflines - ptr^.Reflines;
      Display_Node (ptr);
    end;
  end;      (* triple *)
'Z': begin
  Display_Zsymbol (p);
  if p^.Child <> nil then
    begin
      ptr := p^.Child;
      off := (p^.Width - ptr^.Width) div 2;
      ptr^.Xpo := p^.Xpo + off;
      ptr^.Ypo := p^.Ypo + Z_sym_height + 1;
      Display_Node (ptr);
    end;
  end;      (* limit *)
'V': begin
  hei := 0;
  ptr := p^.Child;
  while ptr <> nil do
    begin
      off := (p^.Width - ptr^.Width) div 2;
      ptr^.Xpo := p^.Xpo + off;
      ptr^.Ypo := p^.Ypo + hei;
      Display_Node (ptr);
      hei := hei + ptr^.height;
      ptr := ptr^.Sibling;
    end;
  end;      (* vector *)
end;      (* case *)
end;      (* end of procedure *)

BEGIN
  RestoreWindow (cuw, 0, 0);
  beforen:=nil;

```

```

if r^.Child <> nil then
begin
  wid := 0;
  baseline := r^.Refline + Ynt;
  n := r^.Child;
  while n <> nil do
  begin
    if (n^.Node = 'C') OR (n^.Node = 'B') then
      lastn := n;
    case n^.Node of
      'P': begin
        n^.Ypo := baseline - lastn^.Refline - n^.Height;
        if beforen^.Node <> 'S' then
          begin
            n^.Xpo := Xinit + wid;
            wid := wid + n^.Width;
          end
        else
          begin
            n^.Xpo := beforen^.Xpo;
            if beforen^.Width < n^.Width then
              wid := wid + (n^.Width - beforen^.Width);
            end;
          end;
        'S': begin
          n^.Ypo := baseline + (lastn^.Height - lastn^.Refline);
          if beforen^.Node <> 'P' then
            begin
              n^.Xpo := Xinit + wid;
              wid := wid + n^.Width;
            end
          else
            begin
              n^.Xpo := beforen^.Xpo;
              if beforen^.Width < n^.Width then
                wid := wid + (n^.Width - beforen^.Width);
              end;
            end;
          end;
        end; (* end of case *)
        Display_Node (n);
        beforen := n;
        n := n^.Sibling;
      end; (* end of while *)
    end;
  Display_Cursor;
END;

procedure divide3;
begin
  { it separates overlays: displays overlay }
  { from creates overlay }
end;

```

```

(* ***** *)
(*                                     *)
(*          CREATES FILE                *)
(*                                     *)
(*  Builds all the elements that make up the formulae *)
(*  A Tree holds information regarding every object of the *)
(*  formulae (physical dimension) and their tree-like *)
(*  relationship with each other *)
(*  A Stack keeps the current path from the latest object *)
(*  down to the beginning of the tree *)
(* ***** *)

```

```

(* Creates a Formulae *)
overlay PROCEDURE GET_INPUT (R: pointer);
var

```

```

  EndOfFormula: Sentinel;
  Branch: pointer;
  s: Cptr;
  Frtn: pointer;
  Num: pointer;
  Den: pointer;
  Intl: pointer;
  Ilow: pointer;
  Iupp: pointer;
  Intgd: pointer;
  t code: integer;
  Trip: pointer;
  Tlow: pointer;
  Tupp: pointer;
  Tand: pointer;
  Zmain: pointer;
  Zlow: pointer;
  Rmain: pointer;
  Rindx: pointer;
  Rexp: pointer;
  Power: pointer;
  Indice: pointer;
  Vect: pointer;
  Velem: pointer;
  last_elem: pointer;
  Blok: pointer;
  b_code: integer;

  { String object }
  { real String value }
  { fraction object }
  { Numerator object }
  { Denominator object }
  { integral object }
  { integral object, lower limit }
  { integral object, upper limit }
  { integral object, integrand }
  { triple object, triple code }
  { triple object, }
  { triple object, lower limit }
  { triple object, upper limit }
  { triple object, expression inside triple }
  { limit object, main expression }
  { limit object, low limit }
  { root object, }
  { root object, index expression }
  { root object, expression inside root }
  { exponent object }
  { index object }
  { Vector object }
  { Vector object, vector element }
  { last vector element }
  { Block object }
  { block object, delimiter code }

```

```

(* Inserts either child or sibling relationship in the tree *)
procedure Set_Tree_Ptrs (d:pointer);
var e:pointer;

```

```

begin
  if top^.Child = nil then
    top^.Child := d
  else
    begin
      e := top^.Child;
      while e^.Sibling <> nil do
        e := e^.Sibling;
      e^.Sibling := d;
    end;
end;

```

```

(* Deletes either child or sibling from the tree *)

```

```

procedure Del_Tree_Ptrs;
var e:pointer;
begin
  if top^.Parent^.Child = top then

```

```

    top^.Parent^.Child := nil
else
  begin
    e := top^.Parent^.Child;
    while e^.Sibling <> top do
      e := e^.Sibling;
    e^.Sibling := nil;
  end;
end;

(* Checks for the existence of integral/triple/limit *)
(* lower limit or root index, which are optional *)
procedure Check_low_ptr (major:pointer);
begin
  case major^.Node of
    'R': begin
      if Rindx^.Child = nil then
        begin
          Rmain^.Child := Rexp;
          Dispose (Rindx);
        end;
      end;
    'I': begin
      if Ilow^.Child = nil then
        begin
          Intl^.Child := Iupp;
          Dispose (Ilow);
        end;
      end;
    'T': begin
      if Tlow^.Child = nil then
        begin
          Trip^.Child := Tupp;
          Dispose (Tlow);
        end;
      end;
    'Z': begin
      if Zlow^.Child = nil then
        begin
          Zmain^.Child := nil;
          Dispose (Zlow);
        end;
      end;
  end; (* case *)
end;

(* Checks for the existence of integral/triple *)
(* upper limit which are optional *)
procedure Check_upp_ptr (major:pointer);
begin
  case major^.Node of
    'I': begin
      if Iupp^.Child = nil then
        begin
          if Intl^.Child = Ilow then
            Ilow^.Sibling := Intgd
          else
            Intl^.Child := Intgd;
          Dispose (Iupp);
        end;
      end;
    'T': begin
      if Tupp^.Child = nil then
        begin

```

```

        if Trip^.Child = Tlow then
            Tlow^.Sibling := Tand
        else
            Trip^.Child := Tand;
            Dispose (Tupp);
        end;
    end;
end; (* case *)
end;

(* In a String, goes back one character or *)
(* deletes that string *)
procedure Back_String;
begin
    I := I - 1;
    (* first char is erased, so is the node *)
    if I <= 0 then
        begin
            Del Tree Ptrs;
            Dispose (top);
            pop;
        end;
    end;
end;

(* Creates a tree record that represents a string *)
procedure MakeT_LeaveRecord (Var rcrd: pointer);
var Leave: Cptr;
begin
    Create Leave (rcrd);
    rcrd^.Parent := top;
    rcrd^.Typecode := aflag;
    rcrd^.Height := Chheight;
    rcrd^.Refline := ChRefline;
    Create Cstring (Leave);
    I:=1;
    Leave^.Value [I] := Charter;
    rcrd^.CChild := Leave;
end;

(* Adds one character to a String, or creates one *)
procedure Input_String;
begin
    (* C record is created *)
    if (top^.Node <> 'C') then
        begin
            MakeT_LeaveRecord (Branch);
            Set Tree Ptrs (Branch);
            push (Branch);
        end
    else
        (* adding characters, same record *)
        if (top^.Typecode = aflag) then
            begin
                I := I + 1;
                s := top^.CChild;
                s^.Value [I] := Charter;
            end
        else
            (* different alphabet set, so *)
            (* a C record is created *)
            begin
                pop;
                MakeT_LeaveRecord (Branch);
                Set Tree Ptrs (Branch);
                push (Branch);
            end
        end
    end;
end;

```



```

        end;
end; (* end of Input_String *)

(* Creates tree fraction record *)
procedure MakeT_Frtn ;
begin
    MakeT_Record (Frtn);
    Frtn^.Node := 'F';
    Frtn^.Parent := top;
    Set_Tree_Ptrs (Frtn);
end;

(* Creates tree numerator record *)
procedure MakeT_Num;
begin
    MakeT_Record (Num);
    Num^.Node := 'N';
    Num^.Parent := Frtn;
    Frtn^.Child := Num;
end;

(* Creates tree denominator record *)
procedure MakeT_Den;
begin
    MakeT_Record (Den);
    Den^.Node := 'D';
    Den^.Parent := Frtn;
    Num^.Sibling := Den;
end;

(* Creates tree Integral record *)
procedure MakeT_Intl;
begin
    MakeT_Record (Intl);
    Intl^.Node := 'I';
    Intl^.Parent := top;
    Set_Tree_Ptrs (Intl);
end;

(* Creates tree-integral low limit record *)
procedure MakeT_Ilow;
begin
    MakeT_Record (Ilow);
    Ilow^.Node := 'L';
    Ilow^.Parent := Intl;
    Intl^.Child := Ilow;
end;

(* Creates tree-integral upper limit record *)
procedure MakeT_Iupp;
begin
    MakeT_Record (Iupp);
    Iupp^.Node := 'U';
    Iupp^.Parent := Intl;
    Ilow^.Sibling := Iupp;
end;

(* Creates tree-integral integrand record *)
procedure MakeT_Intgd;
begin
    MakeT_Record (Intgd);
    Intgd^.Node := 'W';
    Intgd^.Parent := Intl;
    Iupp^.Sibling := Intgd;
end;

```

```

end;

(* Creates tree Triple record *)
procedure MakeT_Triple(t_class:integer);
begin
  MakeT_Record (Trip);
  Trip^.Node := 'T';
  Trip^.Parent := top;
  Trip^.Typecode := t_class;
  Set_Tree_Ptrs (Trip);
end;

(* Creates tree-triple low limit record *)
procedure MakeT_Tlow;
begin
  MakeT_Record (Tlow);
  Tlow^.Node := 'G';
  Tlow^.Parent := Trip;
  Trip^.Child := Tlow;
end;

(* Creates tree-triple upper limit record *)
procedure MakeT_Tupp;
begin
  MakeT_Record (Tupp);
  Tupp^.Node := 'H';
  Tupp^.Parent := Trip;
  Tlow^.Sibling := Tupp;
end;

(* Creates tree-tripland record *)
procedure MakeT_Tand;
begin
  MakeT_Record (Tand);
  Tand^.Node := 'A';
  Tand^.Parent := Trip;
  Tupp^.Sibling := Tand;
end;

(* Creates limit tree record *)
procedure MakeT_Zmain;
begin
  MakeT_Record (Zmain);
  Zmain^.Node := 'Z';
  Zmain^.Parent := top;
  Set_Tree_Ptrs (Zmain);
end;

(* Creates limit object low limit tree record *)
procedure MakeT_Zlow;
begin
  MakeT_Record (Zlow);
  Zlow^.Node := 'O';
  Zlow^.Parent := Zmain;
  Zmain^.Child := Zlow;
end;

(* Creates {square or any index} root tree record *)
procedure MakeT_Rmain;
begin
  MakeT_Record (Rmain);
  Rmain^.Node := 'R';
  Rmain^.Parent := top;
  Set_Tree_Ptrs (Rmain);
end;

```

```

end;

(* Creates index root tree record *)
procedure MakeT_Rindx;
begin
  MakeT_Record (Rindx);
  Rindx^.Node := 'K';
  Rindx^.Parent := Rmain;
  Rmain^.Child := Rindx;
end;

(* Creates the inside root tree record *)
procedure MakeT_Rexp;
begin
  MakeT_Record (Rexp);
  Rexp^.Node := 'Q';
  Rexp^.Parent := Rmain;
  Rindx^.Sibling := Rexp;
end;

(* Creates exponent tree record *)
procedure MakeT_Power;
begin
  MakeT_Record (Power);
  Power^.Node := 'P';
  Power^.Parent := top;
  Set_Tree_Ptrs (Power);
end;

(* Creates index tree record *)
procedure MakeT_Index;
begin
  MakeT_Record (Indice);
  Indice^.Node := 'S';
  Indice^.Parent := top;
  Set_Tree_Ptrs (Indice);
end;

(* Creates vector tree record *)
procedure MakeT_Vector;
begin
  MakeT_Record (Vect);
  Vect^.Node := 'V';
  Vect^.Parent := top;
  Set_Tree_Ptrs (Vect);
end;

(* Creates vector element tree record *)
procedure MakeT_Velement;
begin
  MakeT_Record (Velem);
  Velem^.Node := 'E';
  Velem^.Parent := Vect;
  Set_Tree_Ptrs (Velem);
end;

(* Creates block tree record *)
procedure MakeT_Block (b_class:integer);
begin
  MakeT_Record (Blok);
  Blok^.Node := 'B';
  Blok^.Parent := top;
  Blok^.Typecode := b_class;
  Set_Tree_Ptrs (Blok);
end;

```

```

end;

BEGIN      (* GET_INPUT routine *)
EndOfFormula := false;
while NOT EndOfFormula do
begin
  Display Formula (R);
  GetComand(comand_code);
  while comand_code in [chgefont,Tkey] do
  begin
    case comand_code of
      chgefont: begin
        Change Font;
        GetComand(comand_code);
      end;
      Tkey: begin
        Template Show;
        GetComand(comand_code);
      end;
    end; (* case *)
  end; (* while *)
  case comand_code of
    Stringcm: begin
      Input String;
      Update_Box_Stack (top);
    end; (*-string *)
    backspky: begin
      if (top^.node <> 'C') then
      begin
        (* this is an error , stay same location *)
      end
      else
      begin
        Back String;
        Update_Box_Stack (top);
      end;
    end; (* backspky *)
    fractioncm: begin
      if top^.Node = 'C' then pop;
      MakeT Frtn;
      push (Frtn);
      MakeT Num;
      push (Num);
      MakeT Den;
      Update_Box_Stack (top);
      Get Input (R);
      pop;
      push (Den);
      Update_Box_Stack (top);
      Get Input (R);
      pop;
      pop;
    end; (* fraction *)
    rootcm: begin
      if top^.Node = 'C' then pop;
      MakeT Rmain;
      push (Rmain);
      MakeT Rindx;
      push (Rindx);
      MakeT Rexp;
      Update_Box_Stack (top);
      Get Input (R);
      Check_low_ptr (Rmain);
      pop;
    end;
  end;
end;

```

```

        push (Rexp);
        Update Box Stack (top);
        Get Input (R);
        pop;
        pop;
    end;      (* square root *)
integralcm: begin
    if top^.Node = 'C' then pop;
    MakeT Intl;
    push (Intl);
    MakeT Ilow;
    push (Ilow);
    MakeT Iupp;
    MakeT Intgd;
    Update Box Stack (top);
    Get Input (R);
    Check_low_ptr (Intl);
    pop;
    push (Iupp);
    Update Box Stack (top);
    Get Input (R);
    Check_upp_ptr (Intl);
    pop;
    push (Intgd);
    Update Box stack (top);
    Get Input (R);
    pop;
    pop;
end;      (* integral *)
triplecm: begin
    if top^.Node = 'C' then pop;
    Do Menu Triple (t code);
    MakeT Triple(t_code);
    push (Trip);
    MakeT Tlow;
    push (Tlow);
    MakeT Tupp;
    MakeT Tand;
    Update Box Stack (top);
    Get Input (R);
    Check_low_ptr (Trip);
    pop;
    push (Tupp);
    Update Box Stack (top);
    Get Input (R);
    Check_upp_ptr (Trip);
    pop;
    push (Tand);
    Update Box stack (top);
    Get Input (R);
    pop;
    pop;
end;      (* triple *)
limitcm: begin
    if top^.Node = 'C' then pop;
    MakeT Zmain;
    push (Zmain);
    MakeT Zlow;
    push (Zlow);
    Update Box Stack (top);
    Get Input (R);
    Check_low_ptr (Zmain);
    pop;
    pop;

```

```

end;      (* limit *)
vectorcm: begin
  if top^.Node = 'C' then pop;
  MakeT Vector;
  push (Vect);
  { makes first element }
  MakeT Velement;
  push (Velem);
  Update_Box_Stack (top);
  Get_Input (R);
  pop;
  { makes second element }
  MakeT Velement;
  push (Velem);
  Update_Box_Stack (top);
  Get_Input (R);
  pop;
  { third and up are optional }
  while (Velem^.Child <> nil) do
  begin
    last elem := Velem;
    MakeT Velement;
    push (Velem);
    Update_Box_Stack (top);
    Get_Input (R);
    pop;
  end;
  Dispose (Velem);
  last elem^.Sibling := nil;
  Update_Box_Stack (top);
  pop;
end;      (* vector *)
exponentcm: begin
  if top^.Node = 'C' then pop;
  MakeT Power;
  push (Power);
  Update_Box_Stack (top);
  Get_Input (R);
  pop;
end;      (* exponent *)
indexcm: begin
  if top^.Node = 'C' then pop;
  MakeT Index;
  push (Indice);
  Update_Box_Stack (top);
  Get_Input (R);
  pop;
end;      (* index *)
blockcm: begin
  if top^.Node = 'C' then pop;
  Do Menu Block (b code);
  MakeT BBlock (b_code);
  push (Block);
  Update_Box_Stack (top);
  Get_Input (R);
  pop;
end;      (* block *)
endiky: begin
  if top^.Node = 'C' then pop;
  EndOfFormula := true;
end;      (* endiky *)
end;      (* end of case *)
end;      (* end of while end-of-formula *)
END;      (* end of procedure GET_INPUT *)

```

```
procedure divide4;  
begin  
    { separates two overlays: creates file from }  
    { edits file }  
end;
```

```

      (* * * * * *)
      (* * * * * *)
      (*      EDITS formula File      *)
      (* * * * * *)
      (* * * * * *)

(* place formula in edit mode      *)
(* it includes two phases:          *)
(* a. selecting a subformula        *)
(* b. performing an edit operation *)
overlay PROCEDURE EDIT_FORMULA (flaname:f8name; var flag:sentinel);
var
  file name: fname;
  node_selected: pointer;
  strg1,strg2: integer;

label Quit;
const E_exit: Sentinel=false;

  (* Selects subformula on which to operate *)
  overlay PROCEDURE SELECT_SUBFORMULA (o:pointer; var oselected:pointer;
                                         var str1,str2:integer);

  type
    listptr = ^Blist;
    Blist = record
      before: listptr;
      box: pointer;
      after: listptr;
    end;

  var toplist,nexthigh,co: listptr;
      j1,j2                : integer;   {char string range}
      la                   : integer;   {records last operation}

  (* computes how many characters are in a string *)
  function Lenght (o_ptr:pointer): integer;
  begin
    if o_ptr^.Node = 'C' then
      Lenght := o_ptr^.Width div ChWidth
    else
      Lenght := 0;
    end;
  end;

  (* put a dotted-line frame around a string of characters *)
  procedure show string (o_ptr:pointer; jx1,jx2:integer);
  var ox1,oy1, ox2,oy2:integer;
  begin
    SetLineStyle(1);
    if (jx1 <> 0) AND (jx2 <> 0) AND (o_ptr^.Node = 'C') then
      begin
        ox1 := o_ptr^.Xpo + ((jx1-1) * Chwidth);
        oy1 := o_ptr^.Ypo;
        ox2 := o_ptr^.Xpo + (jx2 * Chwidth) - 1;
        oy2 := o_ptr^.Ypo + Chheight - 1;
        DrawSquare (ox1,oy1, ox2,oy2, false);
      end;
    end;
  end;

  (* put a dotted-line frame around an object *)
  procedure show object (o_ptr:pointer; D:char);
  var ox1,oy1, ox2,oy2:integer;
  begin
    SetLineStyle(1);
    if o_ptr^.Node = 'C' then

```



```

    if D = 'F' then { F = Foreward }
    begin
        j1:=1;
        j2:=1;
        Show_String (o_ptr,j1,j2);
    end
    else { D = Backward }
    begin
        j1:=Lenght(o_ptr);
        j2:=j1;
        Show_String (o_ptr,j1,j2);
    end
else
begin
    ox1 := o_ptr^.Xpo;
    oy1 := o_ptr^.Ypo;
    ox2 := o_ptr^.Xpo + o_ptr^.Width - 1;
    oy2 := o_ptr^.Ypo + o_ptr^.Height - 1;
    DrawSquare (ox1,oy1, ox2,oy2, false);
end;
end;

(* find the previous left sibling *)
function Find_Prev (o_ptr: pointer):pointer;
var q,previous:pointer;
begin
    q := o_ptr^.parent;
    if q <> nil then
        begin
            q:=q^.Child;
            previous:=q;
            while (q <> nil) AND (q <> o_ptr) do
                begin
                    previous:=q;
                    q:=q^.Sibling;
                end;
            Find_Prev := previous;
            { if there is no previous, then move one level higher }
            if previous = o_ptr then
                Find_Prev := o_ptr^.parent;
        end
    else
        { there is no parent, so returns itself }
        Find_Prev := o_ptr;
    end;
end;

procedure EmptyList;
begin
    nexthigh := nil;
    toplist := nil;
    co := nil;
    co^.after := nil;
    co^.before := nil;
    co^.box := nil;
end;

procedure PushL (o_ptr: pointer);
var t:listptr;
begin
    New (t);
    t^.box := o_ptr;
    t^.after := nil;
    t^.before := nexthigh;
    if nexthigh <> nil then

```

```

    nexthigh^.after := t;
    nexthigh := t;
end;

procedure MakeList (o_ptr:pointer);
begin
    EmptyList;
    PushL (o_ptr);
    toplist := nexthigh;
    co := nexthigh;
    while o_ptr^.parent <> nil do
        begin
            o_ptr := o_ptr^.parent;
            PushL (o_ptr);
        end;
    end;
end;

(* put a solid frame around an object, to symbolize selection *)
procedure Pick object (o_ptr:pointer);
var ox1,oy1, ox2,oy2:integer;
begin
    SetLineStyle(0);
    ox1 := o_ptr^.Xpo;
    oy1 := o_ptr^.Ypo;
    ox2 := o_ptr^.Xpo + o_ptr^.Width - 1;
    oy2 := o_ptr^.Ypo + o_ptr^.Height - 1;
    DrawSquare (ox1,oy1, ox2,oy2, false);
    if o_ptr^.Node = 'C' then
        begin
            j1 := 1;
            j2 := Lenght(o_ptr);
        end;
    end;
end;

(* put a solid frame around a string, to symbolize selection *)
procedure Pick_string (o_ptr:pointer; jx1,jx2:integer);
var ox1,oy1, ox2,oy2:integer;
begin
    SetLineStyle(0);
    if (jx1 <> 0) AND (jx2 <> 0) AND (o_ptr^.Node = 'C') then
        begin
            ox1 := o_ptr^.Xpo + ((jx1-1) * Chwidth);
            oy1 := o_ptr^.Ypo;
            ox2 := o_ptr^.Xpo + (jx2 * Chwidth) - 1;
            oy2 := o_ptr^.Ypo + Chheight - 1;
            DrawSquare (ox1,oy1, ox2,oy2, false);
        end;
    end;
end;

BEGIN      { select subformula }
    oselected := nil;
    str1 := 0; str2 := 0;
    j1 := 1; j2 := 1;
    la := 0;
    EmptyList;
    o := o^.Child;
    StoreWindow (2);
    RestoreWindow(2,0,0);
    Show object(o,'F');
    GetComand(comand code);
    while comand_codē <> entercm do
        begin
            if (comand_code in [upky,downky,leftky,rightky]) OR
                (comand_code in [SAO,SAC,CANCEL]) then

```

```

begin
  RestoreWindow(2,0,0);
  (* cursor movement command *)
  if comand_code in [upky,downky,leftky,rightky] then
    begin
      la:=0;
      EmptyList;
      case comand_code of
        upky: begin
          if o^.parent <> nil then
            o:=o^.parent;
            Show_object(o,'F');
          end; (* upkey *)
        downky: begin
          if o^.node <> 'C' then
            begin
              if o^.Child <> nil then
                o:=o^.Child;
                Show_object(o,'F');
              end
            end
          else
            Show_object(o,'F');
          end; (* downkey *)
        rightky: begin
          if o^.node <> 'C' then
            begin
              if o^.Sibling <> nil then
                o:=o^.Sibling;
                Show_object(o,'F');
              end
            end
          else
            begin
              if Lenght(o) >= j1+1 then
                begin
                  j1:=j1+1; j2:=j1;
                  Show_string(o,j1,j2);
                end
              else
                if o^.Sibling <> nil then
                  begin
                    o:=o^.Sibling;
                    Show_object(o,'F');
                  end
                else
                  Show_string(o,j1,j2);
                end;
            end; (* rightkey *)
        leftky: begin
          if o^.node <> 'C' then
            begin
              o:=Find_Prev(o);
              Show_object(o,'B');
            end
          else
            begin
              if j1 >= 2 then
                begin
                  j1:=j1-1; j2:=j1;
                  Show_string(o,j1,j2);
                end
              else
                begin
                  o:=Find_Prev(o);
                  Show_object(o,'B');
                end
            end
          end
        end
      end
    end
  end
end

```

```

                                end
                                end;
                                end; (* leftkey *)
                                end; (* case *)
                                end (* cursor movement command *)
else
  (* selecting sub-formula command *)
  begin
    case comand_code of
      (* select an object command *)
      SA0: begin
        if toplist = nil then
          begin
            MakeList(o);
            Pick_object(co^.box);
          end
        else
          begin
            if co^.after <> nil then
              co:=co^.after;
              Pick_object(co^.box);
            end;
            la:=1;
          end;
        (* select a character command *)
        SAC: begin
          if o^.node = 'C' then
            begin
              if toplist = nil then
                begin
                  MakeList(o);
                  Pick_string(o,j1,j2);
                end
              else
                begin
                  if j2 < Lenght(o) then
                    j2:=j2+1;
                    Pick_string(o,j1,j2);
                  end;
                  la:=2;
                end
              else
                (* ERROR -- to select a character, *)
                (* enter into a string first -- *)
                begin
                  EmptyList;
                  Show_object(o,'F');
                  la:=0;
                end;
              end;
            end;
          end;
          (* cancel command *)
          CANCEL: begin
            case la of
              (* cancel last SA0 *)
              1: begin
                if co^.before <> nil then
                  begin
                    co:=co^.before;
                    Pick_object(co^.box);
                  end
                else
                  begin
                    EmptyList;
                    Show_object(o,'F');

```

```

        end;
        end;
        (* cancel last SAC *)
    0,2: begin
        EmptyList;
        Show object(o,'F');
        la := 0;
        end;
        end; (* case *)
        end; (* Cancel *)
        end; (* case *)
        end; (* selecting sub-formula command *)
        end; (* if valid comand *)
        GetComand(comand_code);
        end; (* while *)
        if (la <> 0) then
            begin
                oselected := co^.box;
                str1 := j1;
                str2 := j2;
            end;
        END; (* proc select-subformula *)

        (* perform one of the following operations on a sub-formula : *)
        (* delete,replace,insert after or insert before *)
        overlay PROCEDURE CHOSE OPERATION (obj:pointer; str1,str2:integer);
        type operation = (none,dele,rdls,rdlo,repl,insa,insb);
        var range : integer;          { interval of string characters }
            pobj : pointer;            { parent of obj }
            nobj : pointer;            { new string object, after deletion }
            previo : pointer;          { previous sibling }
            NewChild : pointer;        { subroot first child }

        (* computes how many characters are in a string *)
        function Lenght (o_ptr:pointer): integer;
        begin
            if o_ptr^.Node = 'C' then
                Lenght := o_ptr^.Width div ChWidth
            else
                Lenght := 0;
            end;

        (* returns true if this is the only child *)
        function OnlyChild (o_ptr:pointer):boolean;
        var parnt,children: pointer;
            numb: integer;
        begin
            if o_ptr^.parent <> nil then
                begin
                    parnt := o_ptr^.parent;
                    children := parnt^.Child;
                    numb:=0;
                    while children <> nil do
                        begin
                            numb := numb + 1;
                            children := children^.sibling;
                        end;
                    if (numb-1) = 0 then
                        OnlyChild:=true
                    else
                        OnlyChild:=false;
                    end
                end
            else
                OnlyChild:=true;
            end;
    
```

```

end;

(* returns number of siblings *)
function SiblCount (o_ptr:pointer):integer;
var  parnt,children:pointer;
     numb: integer;
begin
  parnt := o_ptr^.parent;
  children := parnt^.Child;
  numb:=0;
  while children <> nil do
    begin
      numb := numb + 1;
      children := children^.sibling;
    end;
  SiblCount := numb;
end;

(* returns previous sibling *)
function PRVS (o_ptr:pointer):pointer;
var  parnt,children:pointer;
begin
  parnt := o_ptr^.parent;
  if parnt <> nil then
    begin
      children := parnt^.Child;
      PRVS := children;
      while (children <> nil) AND (children <> o_ptr) do
        begin
          PRVS := children;
          children := children^.sibling;
        end;
      end
    else
      PRVS := o_ptr;      { o_ptr is root }
    end;
end;

(* returns last sibling *)
function LASTSIBL (o_ptr:pointer):pointer;
var  parnt,children:pointer;
begin
  parnt := o_ptr^.parent;
  if parnt <> nil then
    begin
      children := parnt^.Child;
      while (children <> nil) do
        begin
          LASTSIBL := children;
          children := children^.sibling;
        end;
      end
    else
      LASTSIBL := o_ptr;      { o_ptr is root }
    end;
end;

(* join two strings of the same alphabet *)
procedure Join Strings (o_ptr: pointer);
var A,B: pointer;
     x,y: integer;
begin
  A := o_ptr^.Child;
  B := A^.Sibling;
  while (B <> nil) do
    begin

```

```

    if (A^.Node = 'C') AND (B^.Node = 'C') AND
      (A^.Typecode = B^.Typecode) then
      begin
        y := Lenght(A);
        for x:=1 to Lenght(B) do
          begin
            y := y+1;
            A^.CChild^.Value [y] := B^.CChild^.Value [x];
          end;
        I := y;
        A^.Sibling := B^.Sibling;
        Dispose (B^.CChild);
        Dispose (B);
        Update_Box_Stack(A);
      end
    else
      A := B;
      B := A^.Sibling;
    end;
  end;

  (* replace a node(o_ptr^.Child) *)
  (* with a string that has a marker in it *)
  procedure Make Mark (o_ptr:pointer);
  var marked: pointer;
      mkr: Cptr;
  begin
    Create Leave (marked);
    I := 1;
    Create Cstring (mkr);
    mkr^.Value [I] := '?';
    marked^.CChild := mkr;
    marked^.Parent := o_ptr;
    o_ptr^.Child := marked;
    Update_Box_Stack(marked);
  end;

  (* deletes part of a string *)
  procedure DLTEstring (Var objct:pointer; o_str1,o_str2:integer);
  var x,z: integer;
      v,u: Cptr;
      NewStr, prev :pointer;
  begin
    z:=0;
    Create Leave (NewStr);
    Create Cstring (v);
    NewStr^.CChild := v;
    NewStr^.Typecode := objct^.Typecode;
    NewStr^.Sibling := objct^.Sibling;
    NewStr^.Parent := objct^.Parent;
    prev := PRVS (objct);
    if prev = objct then
      objct^.Parent^.Child := NewStr
    else
      prev^.Sibling := NewStr;
    u := objct^.CChild;
    range := Lenght(objct);
    for x:=1 to range do
      if (x < o_str1) OR (x > o_str2) then
        begin
          z:=z+1;
          v^.Value [z] := u^.Value [x];
        end;
    I := z;
  end;

```

```

    object := NewStr;
end;

(* deletes whole formula *)
procedure DLTEroot (object:pointer);
begin
    Root:=nil;
    Init_Values;
end;

(* using node's parent and information about being *)
(* the only child dislocates node from the tree *)
procedure DLTE (object:pointer);
var po,prev: pointer;
begin
    po := object^.Parent;
    if OnlyChild(object) then
        begin
            case (po^.node) of
                (* deletes root, sends a message *)
                'M': DLTEroot (po);
                (* cannot delete this child, so it replaces *)
                (* it (po^.child) with a marker *)
                'N','D','Q','W','A','B': Make_Mark (po);

                'E': if SiblCount(object) < 2 then
                    Make_Mark (po)
                else
                    begin
                        object:=po;
                        DLTE(object);
                    end;

                'K','L','U','G','H': begin
                    object:=po;
                    DLTE(object);
                end;

                'O': begin
                    po^.parent^.Child:= nil;
                    Update_Box_Stack(po^.parent);
                end;

                'Z': begin
                    po^.Child:=nil;
                    Update_Box_Stack(po);
                end;

                'P','S': begin
                    object:=po;
                    DLTE(object);
                end;
                (* replace object^.child w/string *)
                'R','I','T','V': Make_Mark (object);
            end; (* case *)
        end (* if only-child routine *)
    else
        (* delete sub-formula that *)
        (* has siblings *)
        begin
            prev := PRVS(object);
            if object <> prev then
                if object^.Sibling <> nil then
                    prev^.Sibling := object^.Sibling
                end
            end
        end
    end
end

```



```

        else
            prev^.Sibling := nil
        else
            po^.Child := object^.Sibling;
            Update Box Stack(po);
        end; (* not only-child routine *)
    end; (* dlte routine *)

(* Divides a string-leave into 2 leaves *)
(* and incorporates them in the tree *)
procedure Divide Link (object:pointer; VAR A,B:pointer);
var prev,postr :_pointer;
    u,v: Cptr;      { strings corresponding to leaves }
    nc: pointer;    { dummy pointer }
begin
    prev := PRVS (object);
    Create Leave (A);
    Create_Cstring (v);
    A^.CChild := v;
    A^.Typecode := object^.Typecode;
    A^.Parent := object^.Parent;
    Create Leave (B);
    Create_Cstring (u);
    B^.Typecode := object^.Typecode;
    B^.CChild := u;
    B^.Parent := object^.Parent;
    B^.Sibling := object^.Sibling;
    postr := LASTSIBL (NewChild);
    nc := NewChild;
    while nc <> nil do
        begin
            nc^.Parent := object^.Parent;
            nc := nc^.Sibling;
        end;
    A^.Sibling := NewChild;
    postr^.Sibling := B;
    if prev = object then
        object^.Parent^.Child := A
    else
        prev^.Sibling := A;
    end;

(* replaces a subformula for a new one *)
procedure REPLCE (status:integer; object:pointer;
                                o_str1,o_str2:integer);
var A,B,prev,postr,nc:pointer;
    w,x,z:integer;
begin
    Ini Value;
    Display Input Menu;
    Do Window12;
    Write CurrentFont;
    Do ReplaceWindow;
    MakeT Root (SubRoot);
    Push (SubRoot);
    Get Input (SubRoot);
    pop;
    Display Formula (SubRoot);
    NewChild := SubRoot^.Child;
    GoToXY(32,7);
    write('Press <ENTER> to Replace or any key to Escape');
    GetComand (comand code);
    if (comand code = _ENTERcm) AND (NewChild <> nil) then
        (* if replacement is valid *)

```

```

begin
  case (status) of
    0 : Root := SubRoot;
    1 : begin
        postr := LASTSIBL (NewChild);
        nc := NewChild;
        while nc <> nil do
          begin
            nc^.Parent := object^.Parent;
            nc := nc^.Sibling;
          end;
        prev := PRVS (object);
        if prev = object then
          object^.Parent^.Child := NewChild
        else
          prev^.Sibling := NewChild;
          postr^.Sibling := object^.Sibling;
          Update_Box_Stack(object^.Parent);
        end;
      2 : begin
          object^.Child := NewChild;
          nc := NewChild;
          while (nc <> nil) do
            begin
              nc^.Parent := object;
              nc := nc^.Sibling;
            end;
          Update_Box_Stack(object);
        end;
      3 : begin
          (* replaced part is in front *)
          if o_str1 = 1 then
            begin
              DLTEstring(object,o_str1,o_str2);
              postr := LASTSIBL (NewChild);
              nc := NewChild;
              while nc <> nil do
                begin
                  nc^.Parent := object^.Parent;
                  nc := nc^.Sibling;
                end;
              prev := PRVS (object);
              if prev <> object then
                prev^.Sibling := NewChild
              else
                object^.Parent^.Child := NewChild;
                postr^.Sibling := object;
                Update_Box_Stack(object);
              end;
            end;
          (* replaced part is at the back *)
          if o_str2 = Lenght(object) then
            begin
              DLTEstring(object,o_str1,o_str2);
              postr := LASTSIBL (NewChild);
              nc := NewChild;
              while nc <> nil do
                begin
                  nc^.Parent := object^.Parent;
                  nc := nc^.Sibling;
                end;
              prev := PRVS(object);
              postr^.Sibling := object^.Sibling;
              object^.Sibling := NewChild;
              Update_Box_Stack(object);
            end;
          end;
    end;
  end;
end;

```

```

end;
(* missing part is in the middle of the string, so *)
(* it divides the string into two strings, and *)
(* subformula goes in between *)
if (o_str1 > 1) AND (o_str2 < Lenght(object)) then
begin
  Divide_Link(object,A,B);
  z:=0; w:=0;
  range := Lenght(object);
  for x:=1 to range do
  begin
    if (x < o_str1) then
    begin
      z:=z+1;
      A^.CChild^.Value [z]:=object^.CChild^.Value [x];
    end;
    if (x > o_str2) then
    begin
      w:=w+1;
      B^.CChild^.Value [w]:=object^.CChild^.Value [x];
    end;
  end;
  I := z;
  Update_Box_Stack(A);
  I := w;
  Update_Box_Stack(B);
  Update_Box_Stack(object^.Parent);
end;
end; (* status 3 *)
end; (* case *)
Join Strings (object^.Parent);
end; (* if replacement is valid *)
end; (* replace procedure *)

(* inserts new subformula right before selected object *)
procedure INSERTB (status:integer; object:pointer;
                  o_str1,o_str2:integer);
var A,B,prev,postr,nc:pointer;
    w,x,z:integer;
begin
  Ini Value;
  Display Input Menu;
  Do Window12;
  Write CurrentFont;
  Do ReplaceWindow;
  MakeT Root (SubRoot);
  Push (SubRoot);
  Get Input (SubRoot);
  pop;
  Display Formula (SubRoot);
  NewChild := SubRoot^.Child;
  GoToXY(32,7);
  write('Press <ENTER> to Ins Before or any key to Escape');
  GetComand (comand code);
  if (comand code = ENTERcm) AND (NewChild <> nil) then
    (* if insertion is valid *)
    begin
      case status of
        1 : begin
          postr := LASTSIBL (NewChild);
          nc := NewChild;
          while (nc <> nil) do
            begin
              nc^.Parent := object;
            end;
          end;
        end;
      end;
    end;
  end;
end;

```

```

        nc := nc^.Sibling;
    end;
    postr^.Sibling := object^.Child;
    object^.Child := NewChild;
    Update_Box_Stack (object);
end;
2 : begin
    postr := LASTSIBL (NewChild);
    nc := NewChild;
    while (nc <> nil) do
    begin
        nc^.Parent := object^.Parent;
        nc := nc^.Sibling;
    end;
    prev := PRVS (object);
    if prev = object then
        object^.Parent^.Child := NewChild
    else
        prev^.Sibling := NewChild;
    postr^.Sibling := object;
    Update_Box_Stack (object^.Parent);
end;
3 : begin
    Divide Link(object,A,B);
    range := Lenght (object);
    for x:=1 to range do
    begin
        if (x < o_str1) then
        begin
            z:=z+1;
            A^.CChild^.Value [z] := object^.CChild^.Value [x];
        end
        else
        begin
            w:=w+1;
            B^.CChild^.Value [w] := object^.CChild^.Value [x];
        end;
    end;
    I := z;
    Update_Box_Stack(A);
    I := w;
    Update_Box_Stack(B);
    Update_Box_Stack(object^.Parent);
end; (*status 3 *)
end; (* case *)
Join Strings (object^.Parent);
end; (* if insertion is valid *)
end; (* insert before proc *)

(* inserts new subformula right after object *)
procedure INSERTA (status:integer; object:pointer;
                  o_str1,o_str2:integer);
var A,B,prev,postr,nc:pointer;
    w,x,z:integer;
    answ:Char;
begin
    Ini Value;
    Display Input Menu;
    Do Window12;
    Write CurrentFont;
    Do ReplaceWindow;
    MakeT Root (SubRoot);
    Push T(SubRoot);
    Get_Input (SubRoot);

```

```

pop;
Display Formula (SubRoot);
NewChild := SubRoot^.Child;
GoToXY(32,7);
write('Press <ENTER> to Ins After or any key to Escape');
GetComand (comand code);
if (comand code = 'ENTERcm) AND (NewChild <> nil) then
  (* if insertion is valid *)
  begin
    case status of
      1 : begin
          postr := LASTSIBL(object^.Child);
          nc := NewChild;
          while nc <> nil do
            begin
              nc^.Parent := object;
              nc := nc^.Sibling;
            end;
          postr^.Sibling := NewChild;
          Update_Box_Stack(object);
        end;
      2 : begin
          postr := LASTSIBL(NewChild);
          nc := NewChild;
          while nc <> nil do
            begin
              nc^.Parent := object^.Parent;
              nc := nc^.Sibling;
            end;
          postr^.Sibling := object^.Sibling;
          object^.Sibling := NewChild;
          Update_Box_Stack(object^.Parent);
        end;
      3 : begin
          Divide_Link(object,A,B);
          z:=0; w:=0;
          range := Lenght(object);
          for x:=1 to range do
            begin
              if (x <= o_str2) then
                begin
                  z:=z+1;
                  A^.CChild^.Value [z] := object^.CChild^.Value [x];
                end
              else
                begin
                  w:=w+1;
                  B^.CChild^.Value [w] := object^.CChild^.Value [x];
                end;
            end;
          I := z;
          Update_Box_Stack(A);
          I := w;
          Update_Box_Stack(B);
          Update_Box_Stack(object^.Parent);
        end; (*-status 3 *)
      end; (* case *)
    Join Strings (object^.Parent);
  end; (* if insertion is valid *)
end; (* insert after proc *)

BEGIN { chose operation routine }
  GetComand(comand code);
  case comand_code of

```

```

(* operation delete *)
deletecm: begin
  GotoXY(32,7);
  write('Press <ENTER> to delete or any key to Escape');
  GetComand(comand_code);
  if comand_code = _entercm then
    begin
      case (obj^.node) of
        'M': DLTEroot (obj);
          (* it cannot delete this part so it replaces *)
          (* it (obj^.child) with a marked string *)
          'N','D','Q','W','A': Make_Mark (obj);

        'E': begin
            if SiblCount(obj) < 2 then
              Make_Mark (obj)
            else
              DLTE (obj);
          end;

        'C': begin
            range := str2 - str1 + 1;
            if Lenght(obj) = range then
              DLTE (obj) { entire string }
            else
              begin { or part of it }
                DLTEstring (obj,str1,str2);
                Update_Box_Stack (obj);
              end;
            end;
          else DLTE (obj);
          end; (* case *)
        end;
      end; (* operation delete command *)
    end;
  (* operation replace *)
  replacecm: begin
    case (obj^.Node) of
      'M': REPLCE (0,obj,0,0); { replace root }
      'F','R','I','T','Z','V','S','P','B':
        REPLCE (1,obj,0,0);
      'N','D','K','Q','L','U','W','G','H','A','O','E':
        REPLCE (2,obj,0,0);
      'C': begin
          range := str2 - str1 + 1;
          if range = Lenght (obj) then { replace all }
            REPLCE (1,obj,0,0)
          else
            REPLCE (3,obj,str1,str2); { replace part }
          end;
        end; (* case *)
      end; (* replacecm *)
    end;
  (* operation insert after *)
  insaftercm: begin
    case (obj^.Node) of
      'M','N','D','K','Q','L','U','W','G','H','A','O','E':
        INSERTA (1,obj,0,0);
      'F','R','I','T','Z','V','B','P','S':
        INSERTA (2,obj,0,0);
      'C': begin
          if (str2 = Lenght(obj)) then
            INSERTA (2,obj,0,0)
          else
            INSERTA (3,obj,str1,str2);
          end;
    end;
  end;
end;

```

```

        end; (* case *)
        end; (* insert after command *)
        (* operation insert before *)
    insbeforecm: begin
        case (obj^.Node) of
            'M','N','D','K','Q','L','U','W','G','H','A','O','E':
                INSERTB (1,obj,0,0);
            'F','R','I','T','Z','V','B','P','S':
                INSERTB (2,obj,0,0);
            'C': begin
                if (str1 = 1) then
                    INSERTB (2,obj,0,0)
                else
                    INSERTB (3,obj,str1,str2);
                end;
            end;
        end; (* case *)
        end; (* insert before command *)
    end; (* case *)
    Do UserWindow;
    cuw := 2;
    Ynt := Yinit;
    if Root <> nil then
        Display_Formula (Root);
END;

```

```

BEGIN { EDITS main procedure }
Assign(F, 'b:' + flaname + '.dta');
{$I-} Reset(F); {$I+}
if IOrresult <> 0 then { tree file exists }
    begin
        flag:=false;
        GoTo Quit;
    end;
Assign(G, 'b:' + flaname + '.str');
{$I-} Reset(G); {$I+}
if IOrresult <> 0 then { string file exists }
    begin
        flag:=false;
        GoTo Quit;
    end;
flag := true;
Restore_Tree(Root);
link to parent (Root,nil);
Close(F);
Close(G);
Do Window14;
Do UserWindow;
if Root <> nil then
    Display_Formula (Root);
E exit:= false;
while (NOT E_exit) do
    begin
        (* move cursor and select subformula *)
        Display_Select_Menu;
        SelectWorld(1);
        SelectWindow(2);
        Select_Subformula(Root,nodeselectd,strg1,strg2);
        SetLineStyle(0);
        (* if selected subformula then *)
        (* chose edit mode operation *)
        if nodeselectd <> nil then
            begin
                Display_Operate_Menu;
            end;
        end;
    end;
end;

```

```
        Chose_Operation(nodeselectd,strg1,strg2);
    end
    else
        E_exit:=true;
    end;
Quit:  begin end;
END;   { EDIT-FORMULA }
```


BIBLIOGRAPHY

1. A Manual of Style, 12th ed., The University of Chicago Press, 1969, p. 295.
2. M. Levison, "Editing Mathematical Formulae," Software-Practice and Experience 13 (1983), 189-195.
3. B. Kernighan and L. Cherry, "A System for Typesetting Mathematics," Communications of the ACM 18, no. 3 (1975), 151-157.
4. D. Knuth, TeX and Metafont: New Directions in Typesetting, Digital Press, 1979.
5. _____. The Texbook, Addison Wesley Pub. Co., 1987.
6. M. Hammer et al., "The Implementation of Etude, An Integrated and Interactive Document Production System," Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, ACM SIGPLAN Notices 16, no. 6 (1981), 137-146.
7. V. Quint, "An Interactive System for Mathematical Text Processing," Technology and Science of Informatics 2, no. 3 (1983), 169-179.
8. _____. "An Interactive System for Editing Mathematical Documents," Integrated Interactive Computing Systems, (1983), 153-165.
9. _____. "Editing Mathematics on the Buroviseur," International Workshop on Office Information Systems, St. Maximin, October 1981, (1982), 149-159.
10. J. Milne, "Review of Two Scientific Word-Processing Programs: T3 and Volkswriter Scientific," The Mathematical Intelligencer 18, no. 1 (1986), 66-71.
11. M. Levison, "A Programmable Text-editing System," Software-Practice and Experience 12 (1982), 611-621.
12. "Scientific Word Processors: Formulas for Success," PC Magazine 13 (1988), 251-328.
13. R. Goldstein, J. Loomis and A. Tetewsky, "Technical Wordprocessors for the IBM PC and Compatibles, Report by the Boston Computer Society, Part I - TWP Capabilities and People Needs," Notices of the

- American Mathematical Society, (January 1987), 14-32.
14. _____. "Technical Wordprocessors for the IBM PC and Compatibles, Report by the Boston Computer Society, Part IIA - TWP Summary Tables," Notices of the American Mathematical Society, (February 1987), 262-281.
 15. _____. "Technical Wordprocessors for the IBM PC and Compatibles, Report by the Boston Computer Society, Part IIB - Reviews," Notices of the American Mathematical Society, (April 1987), 462-491.
 16. R. Van Uchelen, Word Processing, a Guide to Typography, Taste, and In-house Graphics, Van Nostrand Reinhold Co., 1980.
 17. G.Cornell and I.Stewart, "Chiwriter," The Mathematical Intelligencer 8, no. 2 (1986) 64-65.
 18. "Volkswriter for the Scientist," Mini-Micro Software 10, no.1 (1985) 5-6.
 19. "Interactive Editing Systems," Computer Surveys 14, no.3 (1982) 321-415.
 20. Datapro Directory of Microcomputer Software, Datapro Research Corporation, 1983.
 21. Datapro Reports on Microcomputers, Datapro Research Corporation, 1984.
 22. Datapro Reports on Office Automation, Datapro Research Corporation, 1984.
 23. Datapro Reports on Word Processing, Datapro Research Corporation, 1984.
 24. Data World, Auerbach Publishers, 1978.
 25. IBM Personal Computer Hardware Reference Library, Technical Reference, 1984.
 26. Turbo Pascal, v. 3.0, Reference Manual, Borland International Inc., 1985.
 27. Turbo Graphix Toolbox, v. 1, Handbook, Borland International Inc., 1985.
 28. Math Text, v. 1.85, Mini Edition.
 29. Brit Scientex, Scientific Communications Corp.

30. WordPerfect, v. 4.2, WordPerfect Corporation, 1987.
31. WordStar Professional, release 4, MicroPro International Corporation, 1987.
32. Volkswriter, LifeTree Software Inc., 1984.
33. DisplayWrite, IBM.
34. XYWrite III Plus, XYQuest, Inc., 1987.