DREAMS: Display REpresentation for Algebraic Manipulation Systems

Gregg Foster

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
March 1984

ABSTRACT

DREAMS is an elaborated expression syntax tree display system for mathematical expressions on bitmap computer displays. To support current display technology, we need to handle screen position, font information, and classes of displayable objects . DREAMS is a lisp-based system. DREAMS is designed as the basis of an output and input feedback system for a user interface for Algebraic Manipulation.

April 30, 1984

DREAMS: Display REpresentation for Algebraic Manipulation Systems

Gregg Foster

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
March 1984

ABSTRACT

DREAMS is an elaborated expression syntax tree display system for mathematical expressions on bitmap computer displays. To support current display technology, we need to handle screen position, font information, and classes of displayable objects . DREAMS is a lisp-based system. DREAMS is designed as the basis of an output and input feedback system for a user interface for Algebraic Manipulation.

1. Introduction

Algebraic manipulation systems (AMSs) have been developed to put some of the burden of mathematical expression understanding and manipulation onto machines. The display (the way a user sees the expressions) is useful if it can display a good quantity of mathematical notation using a readily understandable representation of the expression. The power of a notation tool can be judged by the richness of the symbols and their ease of manipulation.

DREAMS is a way of representing mathematical expressions. The representation supports display and manipulation of typeset-quality mathematical expressions on a computer bitmap display. In a display environment that supports variable font/symbol sizes the display representation must include screen position, expression sizes, and font information.

DREAMS* is designed as the display method for a sophisticated interface to an Algebraic Manipulation system. The display representation allows expressions to be quickly and accurately displayed in an aesthetically pleasing form. Since subexpressions are manipulable objects, it facilitates interactive user manipulation of the expression (see section 5 for more on sub-expression indicating). An

* We use the term 'DREAMS' to refer both to the "ExpBox" data-structure and to the display system that uses it.

interface using DREAMS would be an improvement over the traditional display technology, paper and pencil, since the user could not only see his work in clear form but also command the underlying AMS to *perform* the computation an expression denotes, or some other transformations incidental to the object being displayed.*

2. Previous Mathematical Display Systems

AMS display techniques have improved with advances in display technology, but in the case of widely distributed access, the quality has usually been tied to typical time-sharing terminal equipment and fixed-character displays. Figure 1 shows the improvement in mathematical display on computer screens in the last 25 years. The "display" of a mathematical expression in FORTRAN is simply a code listing. The only expression display consideration given in FORTRAN was the (vast) improvement of using linearized infix over program segments. William Martin used vector-based (stroke) display devices[1], but this system was not exploited because of cost and technical complications. Robert Anderson developed syntax rules for expression dimensioning in the course of his work in parsing hand-printed mathematics[2]. CHARYBDIS, MathLab, and Project MAC (Macsyma) brought expression display to the masses by using standard fixed-character CRTs[3, 4, 5]. Michael Genesereth explored the semantics of subexpression indication[6]. John Foderaro made use of high-resolution hard copy display technology[7].

2.1. CHARYBDIS

CHARYBDIS (CHARacter-composed sYmBolic DISplay) was the output interface of MathLab[3, 4]. It was designed for use on typewriter-like devices (tele-typewriters, line-printers, and fixed-character CRTs). It used a small character set (the "FORTRAN set": upper case, decimal digits, and +-=/.,:()*). Large symbols, like sigma for summation, were built up of several other characters. Alternate symbols, like the Greek alphabet (see figure 2), were spelled out.

* There is generally an ambiguous mapping between objects displayed and mathematical intentions. In text this is conveyed by comments. AMSs often miss this point. For example, $f'(x)=0$ could be a statement of fact concerning specific f and x, or an equation to be solved for f and x, or both. It could denote the class of functions f satisfying the equation, or only one of them. The prime could denote differentiation with respect to x or another variable, or something else entirely. A proper solution to the display problem must handle not only expressions, but functions, domains, etc.

```
alpha*sum(x(i),i,0,n)+beta
```

```
                n
               ====
               \
alpha *        >      x   + beta
               /        i
               ====
               i = 0
```

$$\alpha \sum_{i=0}^{n} x_i + \beta$$

FORTRAN (c. 1960) -vs- Macsyma (c. 1968 - present) -vs- DREAMS (1984)

Figure 1.

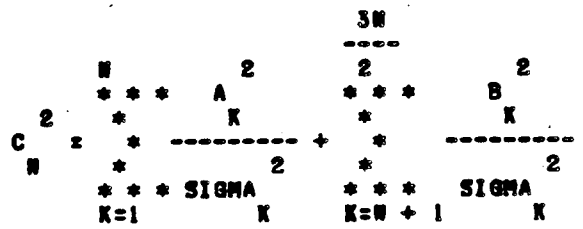
CHARYBDIS also displayed binary tree structures. CHARYBDIS was a precursor to the display systems of Macsyma, Reduce, and other systems.

2.2. William Martin and Symbolic Mathematical Laboratory

William Martin, at about the same time as Anderson's expression dimensioning work and Millen's CHARYBDIS Lisp display (for MathLab), built the Symbolic Mathematical Laboratory (SML). SML had the most ambitious display system of recent AMSs. While building SML Martin addressed most of the basic problems in displaying mathematical expressions. He formalized mathematical expression dimension structure (see figure 3)[8]. His CRT display was quite sophisticated: He handled several special symbols (non ASCII), displayed different font sizes, and had good character positioning. He also addressed multi-line dimensioning and expression breakpointing*. Martin explored input/output for various devices (teletype, crt, plotter, light pen) and introduced

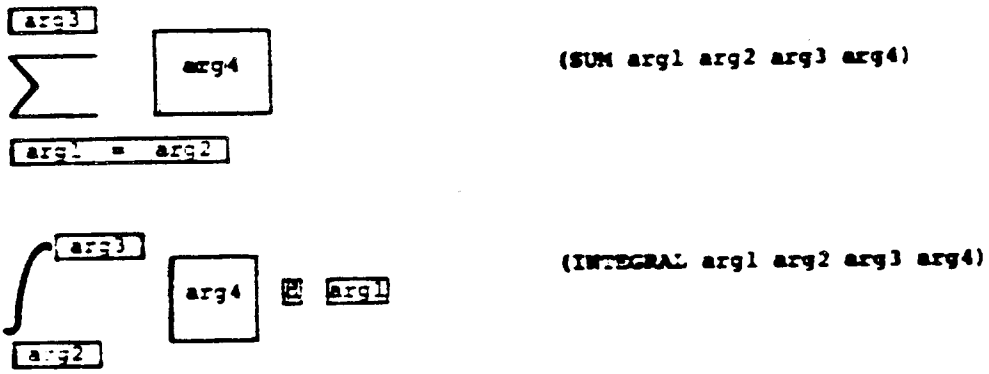
* Breakpoints are the "sensible" places to break a long expression (see section 5.1.1).

```
CHARYBDIS((EQUAL(EXPT(SUB C N)2)(PLUS(EXT(SIGMA)K 1 N
(QUOTIENT(EXPT(SUB A K)2)(EXPT(SUB SIGMA K)2)))(EXT
(SIGMA)K(PLUS N 1)(QUOTIENT(TIMES 3 N)2)(QUOTIENT(EXPT
(SUB B K)2)(EXPT(SUB SIGMA K)2))))) 1 60)
```



A CHARYBDIS Summation

Figure 2.



Martin's summation and integral dimension formats

Figure 3.

the powerful idea that lists (i.e. Lisp trees) are equivalent to expressions (and sub-lists are equivalent to subexpressions).

Martin's subexpression selection technique involved a light pen and tree traversal by numeric ordering of expression nodes.

Some limitations of SML were the allowance for only two different font sizes for characters (a base font and a font half the size), different font types (like Greek, Roman, bold, script, ...) were not used, and SML also ran slowly using a data link between expensive machines (in 1968 -- performance would be much better and cheaper in 1984)[1].

2.3. Macsyma

Martin's SML went through several iterations after being transferred to the Macsyma system. While Macsyma's expression display is a distinct improvement over 1 dimensional displays (like FORTRAN), it should perhaps be categorized as a "1.5 dimensional" display. Macsyma uses fixed character size terminals for output/display, and so cannot properly display large symbols (like integral signs or Sigmas). It is forced to build up large symbols using fixed size characters (see figure 1). Similarly, Macsyma's display cannot shrink symbols in exponents or subscripts.

Other AMS displays of the 1970's and early 1980's (Reduce and Maple, for instance) are similar.

Macsyma users select sub-expressions by a fairly painful encoded walk down the expression tree[5] (see figure 4).

2.4. Michael Genesereth and Circling Subexpressions

Michael Genesereth proposed using a light pen [6] to indicate sub-expressions on a screen by drawing circles around them. This avoids the 'part hopping' of Macsyma and is flexible to the extent that you can circle anything you want and then let the program try to make sense out of it. It certainly seems easy to use, although it appears it was never fully implemented. Genesereth planned to have the expression editing program figure out which sub-expression was indicated by taking the largest sub-expression completely inside the circle. Users would have to be careful not to cut any corners when circling an expression. He later suggested heuristics that would assume that if a certain percentage of an expression was circled then that must be what the user had in mind[6].

```
(c2)  a+b/c+d^2/(e+f+g^(h/(i^3)+j)-k^4);
                                    2
                                   d                     b
(d2)                  -------------------------------- + - + a
                                     h                   c
                                 j + --
                                      3
                   4                 i
                - k  + g                 + f + e

(c3)  dpart(%,1,2,2,2,2,2);
                                    2
                                   d                     b
(d3)                 -------------------------------- + - + a
                                     h                   c
                                j + ----
                                    """"
                                    " 3"
                                    "i "
                   4                """"
               - k  + g                  + f + e
```

Part selection in Macsyma

Figure 4.

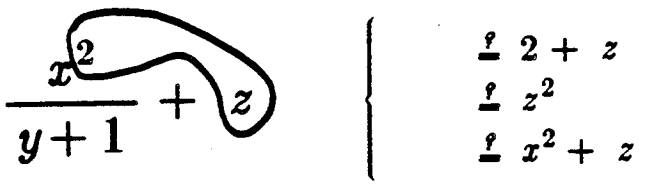
Circling is too free (see figure 5). It became necessary to develop (many) heuristics to figure out what the circled subexpressions mean.

The freedom of circling creates topological problems, although alternate selection systems seem to create other topological problems.

2.5. John Foderaro and Photot

John Foderaro built *Photot,* a system that takes Macsyma internal forms and generates EQN* commands that can be used to "typeset" (on a Benson-Varian, for instance) a Macsyma session (see figure 6). The only drawback is that the user at the terminal during the Macsyma session has a difficult time seeing what is going on (he sees a stream of EQN commands instead of graphical mathematics notation)[7]. The main use of Photot has been in preparing already

* EQN is a UNIX preprocessor for converting a linear mathematical notation language to typesetter instructions.



Circling semantics

Figure 5.

tested command sequences for use on slides and for inclusion in papers.

```
(c2) x+y/z-w^3;

.EQ (d2)
 y over  z  ^+^  x  ^-^  w sup  3
.EN

(c3) 'integrate(x^3/y,x,0,%inf);

.EQ (d3)
 {  int from  0 to  inf ( x sup  3 ) ~"d"  x }  over  y
.EN
```

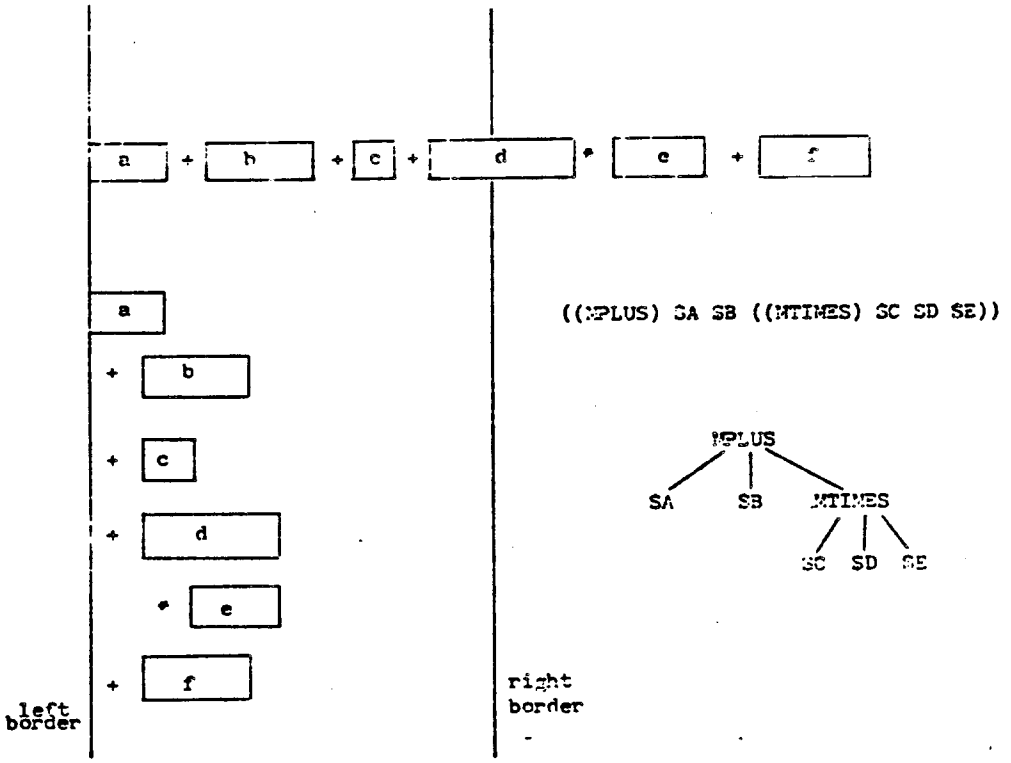
Photot EQN stream for Macsyma expression typesetting

Figure 6.

In theory one could interpose the EQN and associated processors between the Macsyma system and the user display for real-time typesetting. This has until recently been difficult because most display terminals have had insufficient

resolution for the use of font and size variations, and EQN is poorly suited to an interactive environment – it is usually used via UNIX "pipes". A system with similar formula manipulation , T_EX, had similar problems. T_EX is comparable to EQN plus TROFF[9, 10, 11].

Another part of Foderaro's work was his implementation of a more sophisticated expression-breaking that takes into account binding powers at possible break points (see figure 7). We discuss breakpoints later (see section 5.1).



Foderaro's binding powers break points

Figure 7.

2.6. Exploratory work

This section describes two of our experiments which familiarized us with Macsyma, and lead to our final design.

2.6.1. Numbering Convention Interface

Macsyma has a numbering convention that allows a user to traverse the expression tree and pick subexpressions to operate on (see figure 3)[5]. We built a simple interface to Macsyma's number convention subexpression selection. The interface allowed the user to traverse the tree semi-graphically by using keyboard keys to advance along the expression (right), retreat (left), go in (deeper into the tree), or go out (toward the root expression). The subexpression in current focus was displayed in reverse video on a conventional CRT. Substitution of subexpressions was also allowed. These routines used the UNIX curses package[12] and were usable by any standard terminal with cursor addressing. This program used existing Macsyma functions (dpart, substpart,...). This interface was an improvement over the tedious and obscure Macsyma functions it was using, but still suffered from the lack of flexibility inherent in its tools (the Macsyma functions).

2.6.2. EQN on the BBN BitGraph

We also built a crude expression displayer specific to the BBN BitGraph. The routines used John Foderaro's Photot program. Photot converted Macsyma internal forms into EQN command sequences (see section 2.5). We piped the Photot forms through EQN and then attempted to display the generated bit patterns on the fly. This was definitely the wrong approach. It was slow -- 1 to 5 minutes per page. The selection of troff fonts, including scientific symbols, was useful (though new glyphs would have been painful to introduce).

3. Criteria for a Display for an Algebraic Manipulation System

Before going further, we would like to lay out design criteria for display systems. These are somewhat vague but provide some guidance. Any display system (whatever the technological constraints) should address the following four criteria.

3.1. Adequate Mathematical Representation

The display, at a minimum, must be capable of displaying things that are recognized as mathematical expressions by users. Ideally, the display will be seen as an example of excellent mathematical representation: clear style, aesthetically pleasing, and highly readable.

3.2. Traditional Mathematical Forms

This is obvious but necessary. Traditional mathematical forms are the standard mathematics found in journals and textbooks. There is less than total agreement about what is *standard;* one journal's standard is another's variant. But the display should use some reasonably accepted standard and should generally not require users to acquire a new formalism. In cases of ambiguity, new notations may appear.

The bitmap screen display and mouse pointing device we use with DREAMS, could be considered a new display/manipulation formalism, replacing the movement of pencil over paper. This is non-traditional and therefore a disadvantage -- however, we hope the benefits of this new manipulation formalism will prevail[13].

3.3. Completeness

Ideally, the display system will display *any* form of mathematical expression *any* user desires. Realistically, the system should be able to display a large enough subset of mathematics so that a user isn't told very often that what he wants to do is impossible or is given an unreadable display. Furthermore, the display should provide a level of formality (at least as an option) which removes some of the ambiguity of traditional mathematics.

3.4. Usability and Interaction

The display internals must be maintainable and extendible to new forms. The interface should be flexible enough to allow extension and user preferences. The display should be sufficiently fast as to impose an inconsequential delay for the user. A user should be able to specify his favorite display conventions (see The Future, section 7).

4. A High Level Look at DREAMS

This section explains our program in increasing levels of detail.

4.1. Overview

DREAMS expects expression input in the form of Lisp symbolic expressions (S-expressions)*. These S-expressions could be provided by a separate parser of user input or generated by machine computation. The input expression is processed to form a tree of print-boxes. Once the print-box tree is formed, DREAMS must figure out how to display the expression sensibly on the screen. A strategy for displaying the expression is chosen using the size and extent of the outermost enclosing box. The tree of print-boxes is traversed printing the leaves (and doing any other chores).

4.1.1. The Input Expression Tree

As mentioned above, the expected input to DREAMS is an S-expression representing the expression to be displayed. The S-expression is a Lisp (list) version of an expression tree. The passed S-expression can have other information attached to it (unique labels, font requests, type information, etc). The S-expression represents the *form* the expression will have, not necessarily the semantics. That is, ***dx/dy*** will be given to DREAMS as a quotient form, *(// dx dy)*, not something like *(derivative x y)*.

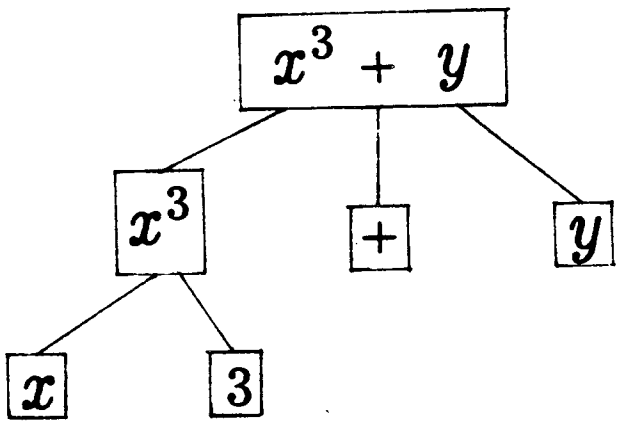
DREAMS currently follows the Macsyma convention and displays products involving negative exponents as rationals (if possible); from the "what you see is what you get" point of view, this is a mistake – negative exponents should be displayed as negative exponents and rationals as rationals. Form decisions at that level should be made by the the AMS/parser. DREAMS should know as little mathematics as possible. DREAMS can work given Macsyma forms, like *((mplus simp) x y)* or given Xi "genrep" forms, like *(+ x y)*. Other expression formats can be accommodated easily.

4.1.2. Forming The ExpBox tree

The input S-expression is converted into a tree of "ExpBoxes", by a data-driven process. Each possible operator, like "+ "' or "integrate", has a formatting function attached to it, like "plus-box" or "integrate-box". This process works recursively (depth first), boxing up subexpressions until it gets to atomic sub-expressions, like "1" or "x".

* For example: an input, *x+ y*, would be given to DREAMS as one of the two current implemented forms, *(+ x y)* or *((mplus simp) x y)*.

Consider a short example using *(+ x y)* as input. The "+" indicates that DREAMS must call a special function, say *plus-box* *, that arranges its arguments with "+"s (and little spaces) separating them. Each argument can be an arbitrarily complicated expression. A skeleton ExpBox is formed enclosing the entire expression. It can't be fleshed out until more is known about the arguments/sub-expressions inside it ("+", "x", and "y" in this case). ExpBoxes for the arguments are created. In this simple case all but the real screen position can be immediately specified. Now that the dimensions of the InnerBoxes (the boxes enclosing "x", "+", and "y") are known, the OuterBox (the box containing "x + y") can figure out its dimensions and exact screen position. The OuterBox can then tell the InnerBoxes their screen offsets relative to the OuterBox. Of course, had "x" been a more complicated expression, the process would have needed to recurse down to the atomic level with InnerInnerBoxes telling InnerBoxes about their dimensions and InnerBoxes telling InnerInnerBoxes about their screen offsets (see figure 8).



Tree of boxes for Expression

Figure 8.

* What really happens is the "+" calls a more general function, *n-ary-op-box*, with "+" as an additional argument (see section 4.1.1).

4.1.3. Screen Display Strategy

Once the ExpBox tree is complete it must be fit in its allotted space on the screen. If the whole expression fits on a single line then we simply write it out. Often, expressions will be too long to fit on a single line; then they must be broken into several lines. The screen offsets of some subexpressions must be changed. If the entire (possibly broken) expression will fit in the allocated window then everything is fine. If it is still too large then we need to hide some of its subexpressions inside a single symbol, shrink the expression, reformat it, and/or page through it. For a discussion of these strategies, see section 5.3 (Figuring out How to Fit It on the Screen).

4.1.4. Traverse the ExpBox Tree and Spray Bits on the Screen

Once an ExpBox tree that will fit on the screen is formed, the expression can finally be displayed. The ExpBox Tree is traversed adding InnerBox screen offsets to OuterBox screen offsets to get the proper screen position (see figure 3). All subexpression ExpBoxes are logically represented on the screen (and therefore available for selection, highlighting etc.) but only the atomic subexpressions at the leaves of the ExpBox tree are printable strings.

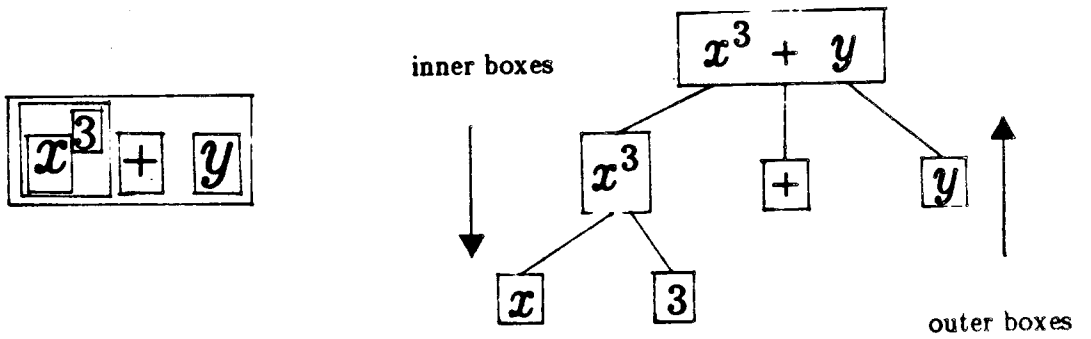
In the above example, the *(+ x y)* subexpression box exists, but subexpression boxes associated with "x", "+ ", and "y" are the ones used for printing. If *(+ x y)* were the entire expression being displayed then the OuterBox would have offsets from the screen base coordinates. The InnerBoxes, "x", "+ ", and "y", would then have offsets from the OuterBox.

4.2. The Hierarchical DREAMS Representation for Mathematical Expressions

Trying to make sense of a random string of symbols (that may represent a mathematical expression) without an underlying structure to organize them would be difficult. The central idea that DREAMS expresses is the correspondence between sublists (in a Lisp representation) and sub-expressions (in a mathematical expression).

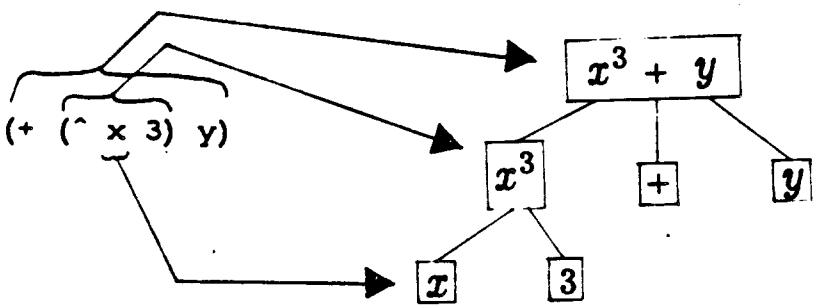
4.2.1. DREAMS Objects

DREAMS is a hierarchical tree of ExpBoxes (sub-expression boxes). A subexpression ExpBox is considered the "offspring" of the expression ExpBox that contains it and it inherits default values from its ancestors. If a slot (or



InnerBoxes and OuterBoxes

Figure 9.



Sub-lists and sub-expressions

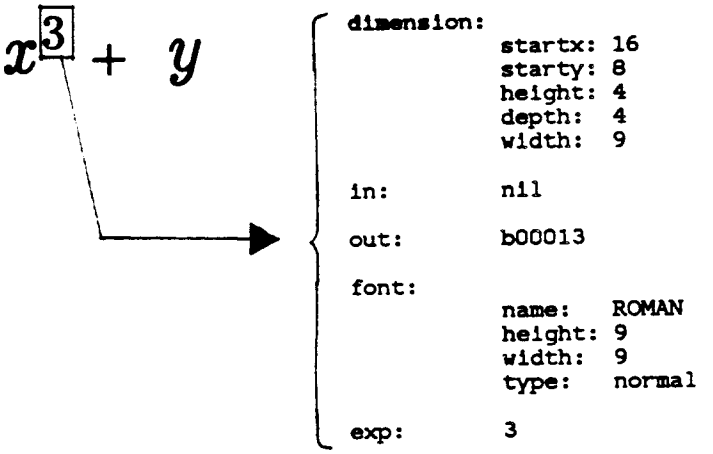
Figure 10.

expected data-field) in the current ExpBox of interest is empty then its ancestors are checked to find the inherited slot value*. For example, all expressions are handed the font information of their parents; this font information may be used as given or used as a base from which to grow (for an integral sign), to shrink (for an exponent or subscript), or to coherently change alphabets.

4.2.2. The Tree of ExpBoxes

Each displayable expression and sub-expression have corresponding ExpBoxes. Each ExpBox is an object that can be referred to (both on the screen and internally). Any level of subexpression, from individual atoms to the entire displayed expression, can be referred to.

These are the basic ExpBox slots (see also figure 11).



An ExpBox

Figure 11.

An ExpBox:

Dimension contains expression size and screen position information:

start_x – the horizontal offset of the left side of the center line relative

* This is not currently true. The ancestors are not checked; slots are given the values of their parents and permission to change them should circumstances warrant.

to the enclosing ExpBox.

start_y -- the vertical offset of the left side of the center line relative to the enclosing ExpBox.

height -- the height of the box above the center line.

depth -- the depth of the box below the center line.

width -- the width of the box.

In is a list of the labels of ExpBoxes contained by the current ExpBox (subexpressions contained by the current expression).

Out is the label of the ExpBox that contains the current ExpBox.

Font contains font information:

name -- the name of the font this ExpBox thinks is active.

dimensions -- the basic width and height of characters for this font.

type -- any special things to be done with this font (like making it bold).

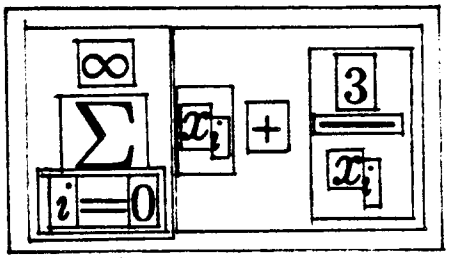
Expression is the lisp representation of the expression enclosed by the current ExpBox. Different representation formats are accepted (currently, Macsyma forms and Xi forms).

Figure 12 shows a small expression and its associated ExpBoxes.

5. Details of Expression Handling in DREAMS

The general strategy, as explained in the previous section, is to create a tree of dimensioned (boxed) expression-objects, figure out how to fit the expression represented by the tree of expression-objects clearly on the screen (possibly breaking, unrevealing, or otherwise altering the appearance of the expression), and traverse the expression-object tree actually putting the bits of the display up on the screen. While putting the expression on the screen, other things (like attaching actual screen properties to the sub-expressions) can be done.

This section covers details of expression display such as how to fit large expressions on the screen, how precedences are dealt with, special characters, and subexpression selection.



A small expression all 'boxed up'

Figure 12.

5.1. Figuring out How to Fit Expressions on the Screen

Given a neatly packaged expression we must figure out some way to put it nicely on the screen. There is no single best way to deal with all cases of large expressions. Screen real-estate and human comprehension are limited, so the expression may have to displayed in one of several different forms. Even assuming the existence of a display screen that is somehow expandable to accommodate an expression of any size and complexity, it is unlikely that anyone would want to see a wall sized display of small characters -- complete perhaps, but hardly accessible. The display must sometimes choose between clarity and completeness of expression.

If the expression is less than a screenful in width and height then it can be simply displayed. If the expression is more than a screenful in width then it may be broken up (at breakpoints) and fit on the screen. If an expression takes more than a screenful to display then there are several options. The expression can be shrunk, unrevealed, paged, or scrolled[14].

5.1.1. Expression breaking

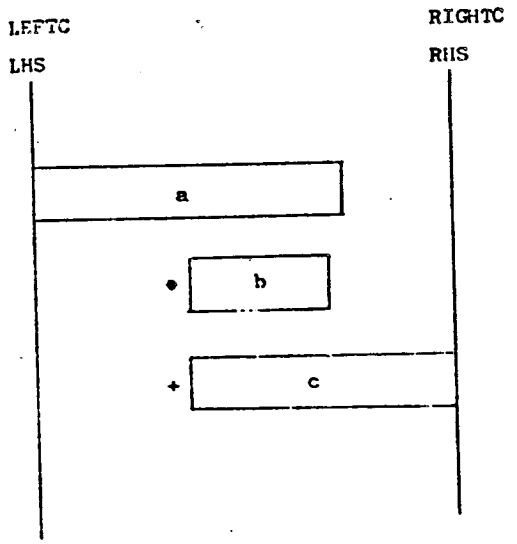
The goal of large expression breaking (other than to fit the expression on the screen) must be to maintain clarity of expression. There is no perfect break algorithm because different things are clear to different people. As far as the details

are concerned it is largely a matter of taste. But some general principles can be applied[7, 10].

"Breakpoints" refer to places where an expression may be broken creating a minimum of confusion. Low-level/High-binding operators like +, -, and * are obvious choices. A break near the root of the expression tree of a high-level operator is generally preferable.

A *major* is the first line of a multi-line expression. The *minors* are the rest of the lines (all indented from the major). The major, in general, establishes the maximum width for all following lines. The minors are arranged under the major with appropriate breaks.

Macsyma has undergone various changes in display. Its current one is an altered version of Martin's algorithm. [5] Briefly, if the algorithm can fit the expression on one line it centers it, otherwise the algorithm breaks the expression at the first acceptable point beyond 2/3 of a page width. All minors are indented the same amount so Macsyma can generate its display on the fly (see figure 13), avoiding a potentially costly back-track display algorithm.

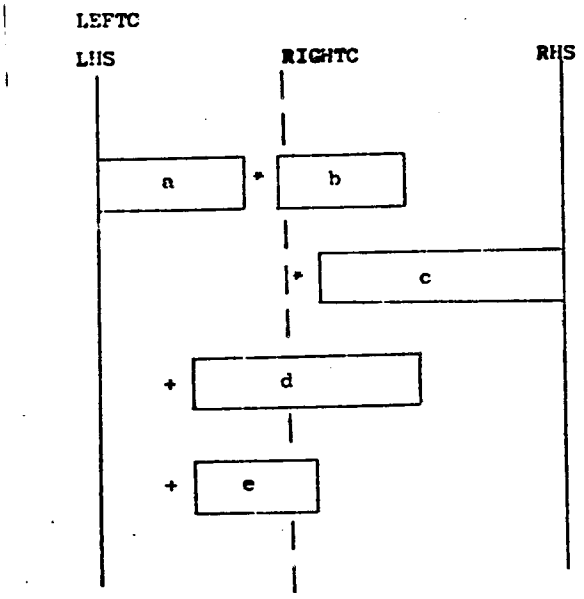


Macsyma Expression Breaking (Majors and Minors)

Figure 13.

Foderaro uses a slightly different algorithm. [7] Foderaro's improvement, briefly, is that operator binding powers are taken into account. Figure 14 shows how Foderaro's breaking algorithm would break the same expression into major

and minor lines.



Foderaro Expression Breaking (Majors and Minors)

Figure 14.

This produces better display but may be more costly. In the context of off-line batch typesetting, the quality is worth the expense.

Neither display program affects the order of expressions much from the Macsyma simplifier output. This can lead to some moderately peculiar expression display, even though Macsyma is generally better than alternatives.

5.1.2. Expression Shrinking

On devices supporting many font sizes, shrinking may be a feasible way to display large expressions. If the expression is only a little larger than a screenful, shrinking would be almost unnoticable. Even massive shrinking of expressions might be useful as a mnemonic or preview in an interactive system.

5.1.3. Expression revealing

In some cases, expressions that turn out to need more than one line (or more than one page) may best be displayed by summarizing them. This is also called *unrevealing*. This is (perhaps) a precise form of the way people talk about

unwieldy expressions. A mathematician, talking about a sum of many terms, might well refer to it as "a sum of many terms". It might be well to display an enormous sum of terms:

sum_of_97_terms (completely unrevealed)

or as

a + b + c + 94_more_terms (head revealed)

or as

94_terms + x + y + z (tail revealed)

rather than running the expression over several lines or pages. The choice of methods might be determined by what the user or the system thinks the interesting part of the expression will be. Head unrevealed, perhaps, if the sum was expected to be dominated by the first few terms. Tail unrevelation might be suggested if the error term is the interesting part.

The user, of course, should be able to force expansion or alteration of any summarizations (possibly creating multi-page display). In journal articles, really large expressions are sometimes dealt with by re-naming common subexpressions. This is beyond the scope of our project. Sometimes these expressions have enough structure that an alternate display format is used – for example, a 2D table of coefficients in a bivariate polynomial.

5.1.4. Multi-page display

There are three versions of multi-page display: Paging, scrolling, and scanning. Combinations of the three are a possible.

Displays are often long and may take several times the size of a screen to completely display an expression. In *paging* the large expression is displayed one page a time, waiting for the user to request the next page (--more--). Macsyma at MIT-MC is initially set for this mode. Paging is normally uni-directional. *Scrolling* may be bi-directional. The broken expression rolls by the user (probably with a mechanism for the user to halt or pause the scrolling). Macsyma on VAX UNIX initially set for this mode. *Scanning* is left/right scrolling on an unbroken expression. The large expression is viewed a piece at a time as if through a moveable window. This has not been implemented on any system we are aware of. Unrevealing might be used to avoid multi-page displays altogether (though at the cost of expression details).

Commands are usually short, but not always (e.g. definitions of programs). The system should be designed to support short commands while allowing long commands with a minimum of fuss.

5.2. Special Characters

Currently, characters in the available fonts are simply drawn by the graphics package in the desired size and location. Special characters like horizontal and vertical lines and radical symbols are drawn as (a sequence) of appropriately sized lines using the drawing routines in the graphics package. Boldface is achieved by redrawing the symbol one pixel to the right.

5.3. Precedences and Parentheses

All operators have special precedence properties attached to them. Needed parentheses can be inferred by comparing the precedence of the InnerBox operator with the precedence of the OuterBox operator.

5.4. Subexpression Selection

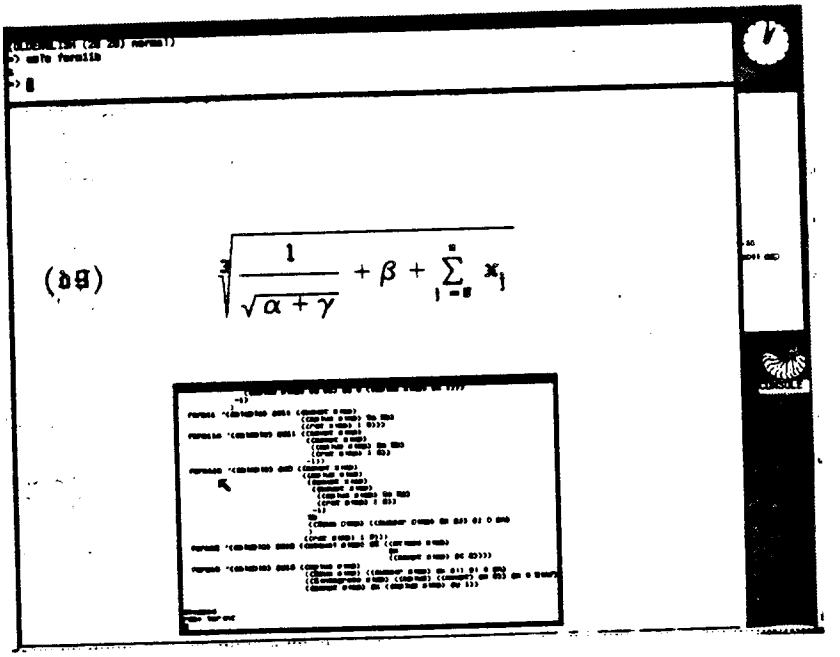
Each ExpBox that contains a valid subexpression is selectable and, therefore, manipulable/highlightable. Each ExpBox has its window coordinates attached as a property to the ExpBox name. For more on subexpression selection see section 7 (The Future).

Non-highlightable boxes (like the outer box containing parentheses inserted to preserve operator precedences) are only indirectly manipulable by focusing on the expression containing it or the expression inside it.

6. Implementation

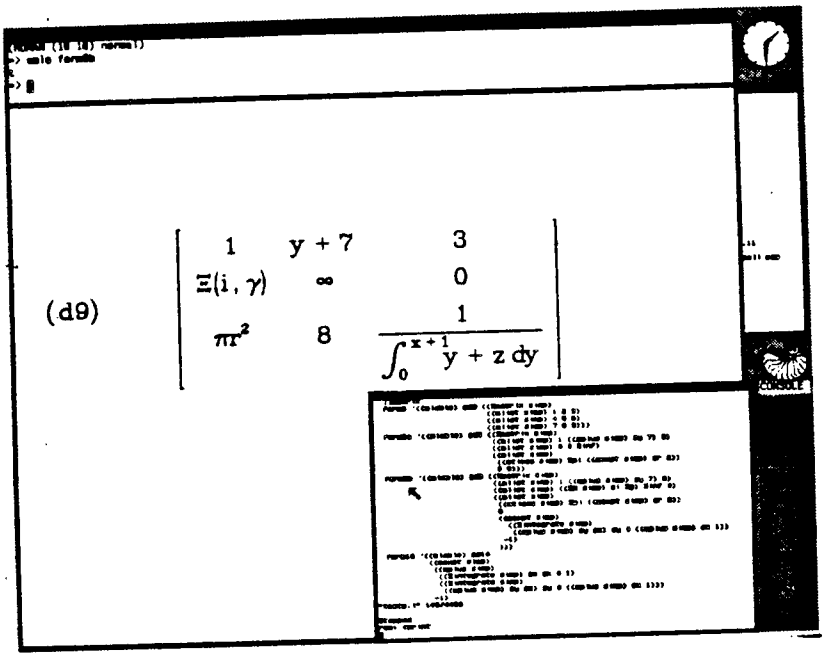
6.1. DREAMS in Action

Figure 15 shows an expression on a Sun Workstation (with other processes present). In Figure 16 DREAMS displays an unlikely matrix. Unrevealing is the basic way DREAMS handles large expressions (see figure 17). DREAMS can be used in place of Macsyma's usual display (see figure 18).



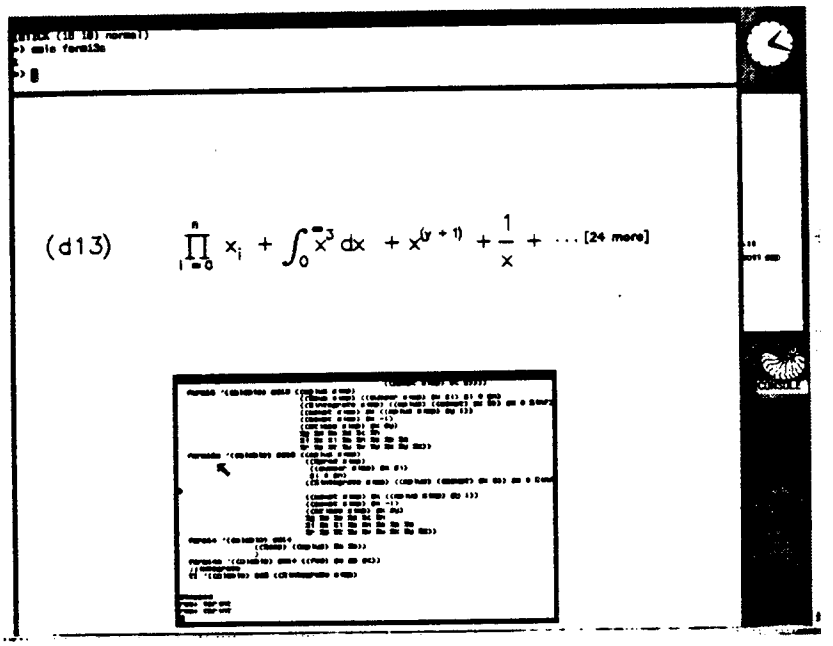
DREAMS

Figure 15.



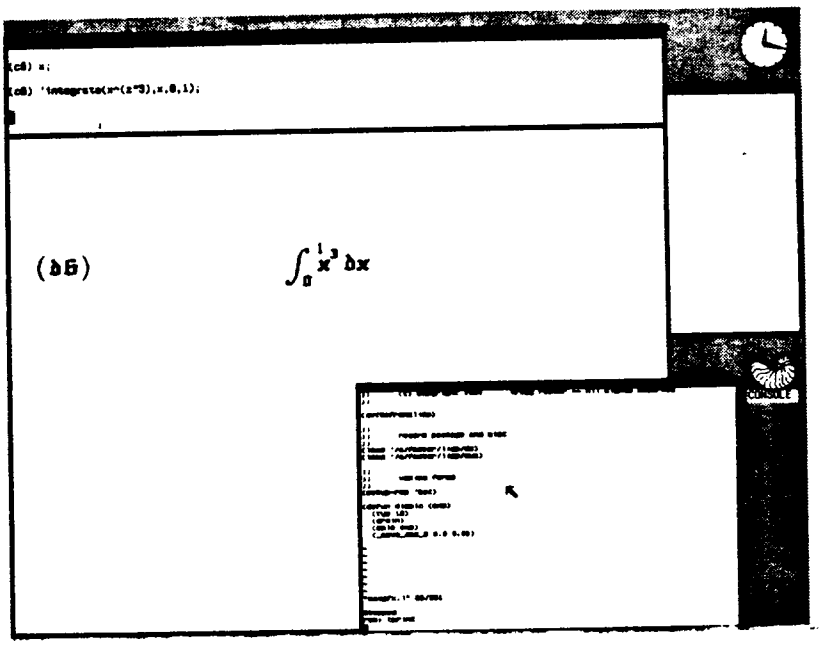
DREAMS

Figure 16.



DREAMS with 'unrevealing'

Figure 17.



DREAMS attached to Vaxima (Macsyma)

Figure 18.

6.2. Tools

A benefit of Lisp as a programming language is that it encourages the design of tailor-made higher-level data-types and functions. To create a higher level development environment a record package for the DREAMS/ExpBox data-structure, an ExpBox pretty printer, and many macros were written.

6.2.1. Record package

The record package was designed to meet the needs of this implementation. It automatically defines the appropriate field access macros when a record type is declared.

6.2.2. Debugging aid(s)

ExpBoxes are represented internally as nested lists. A box pretty printer was designed to allow easy inspection of the slot contents of each ExpBox by displaying the box information in a 2-dimensional form rather than in the 1-dimensional Lisp format (see figure 19). The pretty printer works equally well on ordinary CRTs and bitmap displays.

6.3. The Sun Workstations

DREAMS was implemented using Sun Workstations* with Core graphics software and Franz Lisp[15]. Sun Workstations are Motorola 68000-68010-based machines with optical mice and bitmap displays (1000x800). The current main memory configuration is 2 megabytes.

We had hoped that this would be a good test environment to evaluate DREAMS. This turned out to be partially true. Franz Lisp worked well almost immediately, but the Sun workstations and the Suncore graphics were somewhat frustrating. As Berkeley was a Beta-test site for Sun, we suffered through several bugs and software changes. The Sun Workstations seem fairly stable now (March 1984).

* Sun Microsystems Incorporated.

```
=> mle form3
(b00046 b00013)
=> b00013
((364 18 17 0 0)
 (b00017 b00030 b00031)
 nil
 (ROMAN (16 16) normal)
 nil
 ((mplus simp)
  ((%integrate simp) $x $x 0 %inf)
  ((%integrate simp)
   ((mplus simp) $y $z)
   $y
   0
   ((mplus simp) ((mexpt simp) $x 2) 1))))
=> see b00013

        +---------+       font: ROMAN      (normal)
      h=18      |         height: 16
        |       |         width:  16
  ~[0,0]~     ~|~
        |       |       out: nil
      d=17      |       in: (b00017 b00030 b00031)
        +---------+
          w=364         exp:
                          ((mplus simp) ((%integrate simp) $x $x 0 %inf) ((%inte

nil
=> b00017
((106 17 17 0 0)
 (b00018 b00022 b00024 b00023 b00025 b00026)
 b00013
 (ROMAN (16 16) normal)
 nil
 ((%integrate simp) $x $x 0 %inf))
=> see b00017

        +---------+       font: ROMAN      (normal)
      h=17      |         height: 16
        |       |         width:  16
  ~[0,0]~     ~|~
        |       |       out: b00013
      d=17      |       in: (b00018 b00022 b00024 b00023 b00025 b00026)
        +---------+
          w=106         exp:
                          ((%integrate simp) $x $x 0 %inf)

nil
=> b00018
((24 12 12 0 0) nil b00017 (GREEK (24 24) normal) nil {)
=> see b00018

        +---------+       font: GREEK      (normal)
      h=12      |         height: 24
        |       |         width:  24
  ~[0,0]~     ~|~
        |       |       out: b00017
      d=12      |       in: nil
        +---------+
          w=24          exp:
                          {
```

a list representation of ExpBoxes and a pretty-printed ExpBoxes

Figure 19.

6.3.1. Suncore

The SunCore graphics routines[16], an augmented version of the ACM core specification is a decent minimal graphics package. SunCore is written in C and, for our purposes, it had to be accessible from Franz Lisp. Now, the entire SunCore graphics library is usable from Lisp on a Sun.

Probably the worst SunCore non-bug fault, from DREAMS' point of view, is that, in complying with ACM standards, SunCore does all computation using floating point operations. This slows things down considerably. The SunCore documentation *looked* pretty good, but did not always correspond with the way functions really worked. A few documented functions turned out not to exist.

Being a test site, we went through experimental releases of SunCore. Each release usually involved several function and argument changes. This forced us to fix bugs in formerly bug-free code.

6.3.2. SunFonts

At first the vector fonts provided in SunCore looked good. There were five different basic fonts (including Greek/Scientific) and each was displayable in any desired height, width, and orientation. They worked pretty well in toy applications and can be used (carefully) for demonstrations.

The font stretchability didn't come free. The font quality was poor (thickness, spacing, and size were all uneven). Parentheses were particularly unruly: they displayed much larger than they were told to. Character size inquiry functions in SunCore did not correspond accurately to text screen postions. Since characters were generated on the fly, the display was slow. Characters didn't seem to be constrained to fit into the space they are assigned. It was usual for characters to crowd together or to be too far apart. Over all, the Sun fonts convinced us that we needed some high quality, pre-generated, character fonts for speed and beauty.

6.3.3. Franz Liszt

Franz Liszt on the Suns (Motorola 68000/68010) worked well. Lisp code developed on Vaxes (780/750) ran and compiled on the Suns without alteration.

7. The Future

DREAMS is intended as the basis for a new user interface for algebraic manipulation systems.

7.1. Subexpression Selecting

Subexpression selecting is the foot-in-the-door leading to expression editing. The expression trees are further decorated to include the actual window position of each subexpression ExpBox. This is done after the ExpBox-tree is built (and after any large-expression alterations have been made). Screen coordinate properties are added in a separate traversal (though it makes sense to do this during the display phase).

An ExpBox is chosen with a pointing device (a mouse on the Suns). The uppermost ExpBox under the mouse cursor is highlighted as feedback. The highlighted/chosen ExpBox is changed by moving the mouse or by issuing tree traversal commands.

7.1.1. Mice as Pointing Devices

Mice are admirably suited as pointing devices in AMS systems. Since subexpression ExpBoxes are objects and know their screen positions they can be selected by mouse cursors (or other pointing techniques/devices).

7.1.2. EXED

Using a data structure similar to DREAMS (but limited to a few types of mathematical structures), Richard Anderson wrote a mouse-driven skeleton of an expression editor. It successfully demonstrated the usefulness of subexpression highlighting and the mouse as a subexpression selector. He extended the representation and subexpression highlighting to include basic expression evaluation and editing. He relied on a prototype version of SunCore that is no longer supported[17].

The first use of DREAMS is as a display structure. A logical extension is to use DREAMS as the basis for editing expressions. There two kinds of editor to consider. A structural editor that does what the user tells it to regardless of whether the resultant expression makes any mathematical or other sense, or an editor that has a semantic component (or access to one) and limits the user to mathematically sensible expressions.

7.1.3. Expression Re-arrangement

If subexpressions/ExpBoxes can be cut and pasted onto each other, then only the changed portions (sub-trees) of the display should have to be redimensioned. Their true (new) screen positions will be re-figured as they are put up on the screen.

Should the original expression semantics be preserved? This is limiting but safe. It also would require some (simple?) mathematical knowledge in the display system[18].

7.1.4. Editing Problems

Invisible operators (^,*,...) have no symbol on the screen and may or may not have an associated Expbox. This make them difficult to indicate. Re-arrangement of expressions is also a problem. How are the arguments of n-ary operators shuffled around? Distributivity of the operator must be checked at some point. If a symbol is selected, say x, and altered, should that change be global or local?

7.1.5. Menus and Commands

An editing interface should use both menus and a command language and allow free movement between them. We believe novices would be the primary users of menus. Menus might be used less and less as a user became conversant with the command language. Alternatively, command-language-written modules could be incorporated into the menus -- giving users some control over the system abstraction level. An expert might use menus as an occasional memory aid or to explore an unfamiliar part of the system[13].

7.2. Integration into a Window Package

There are many uses for windows in mathematical expression display and especially in mathematical expression editing. A graphics tool[16] that displayed and manipulated expressions in graphics sub-windows while accepting input in a command sub-window is a first cut. Of course pop-up windows for interesting sub-expressions and unreveals would be nice. Menus will have a place in the command/manipulation language[13].

7.3. System Support

A *status* area that reports system load, garbage collection, and other information would be useful.

A *history* display of the last several commands (or the last several 'interesting' commands) would also be useful. History would probably have a shortened version of the relevant commands in a window. The history area could be used to re-issue previous commands as well as a mnemonic.

Related to the history is the *Audit Trail.* An audit trail is a record of the user's every move (probably in executable form). Synthetic audit trails or user command files could be created and loaded.

7.4. Input/Output Options

Eventually other forms of input (other than keyboard and mouse/select) will be explored. Possibilites are light pen and finger for pointing, and hand written input handling.

Displaying input is another area to look at. Input might be parsed character by character, formatting and displaying the system's guess of what the expression is as it parses. Aesthetics advantages aside, this technique avoids errors since the user gets immediate feedback of what the system thinks he typed.

Various forms of output should be integrated into a DREAMS-spawned system. EQN file, screen bit-dump, Laser printers, multi-monitors.

7.5. Personalization and Extensibility

The system should accept an initialization with various user options. Journal/personal display preferences (font style, indentation/break conventions) could be reflected. Users should be able to define thresholds for dealing with large expressions (request reveal/unreveal levels, force breakpoints,...).

There should be a mechanism for introducing new (alternate) symbols/functions and their associated display routines.

7.6. Parting Shots (not really the future -- comments on the present)

Running interpreted on a Sun Workstation, DREAMS will display a simple one line expression in about 5 seconds, more complicated expressions take corresponding longer. Considering that an AMS takes seconds or minutes (or hours ...) to create a displayable expression, this is "real time". DREAMS should

be easily portable to machines other than Sun Workstations (assuming bitmap displays). Only a few percent of the routines are system or machine dependent, and those few percent are carefully modular.

8. Conclusions

DREAMS is a representation for displaying mathematical expressions on bitmap displays. It decorates an expression tree with display information, including screen position (relative and absolute), expression (and sub-expression) size, and font information -- all necessary in a variable sized character/symbol display environment. Each sub-expression is a manipulable object. Each operator is an object with associated display/formatting routines. DREAMS forms the basis for an intelligent display environment for mathematics using current advances in display technology.

9. Acknowledgements

I am grateful to Richard Fateman, my research advisor, for his advice and support. I would like to thank Keith Sklower for his frequent help with Franz Lisp, especially in porting the SunCore graphics routines. My thanks also to John Foderaro and Kevin Layer who patiently answered questions on Franz and UNIX idiosyncracies. I'd also like to acknowledge the helpful criticisms and suggestions of Carl Andersen, Richard Anderson, John Elvey, and Ken Rimey. Finally, I would like to thank Chris Herdell for improving my prose.

References

1. William A. Martin, "Symbolic Mathematical Laboratory," MIT (MAC-TR-36), January 1967.
2. Robert H. Anderson, "Syntax-Directed Recognition of Hand Printed Two-Dimensional Mathematics," Harvard (PhD. Thesis), January 1968.
3. J. K. Millen, "CHARYBDIS: A Lisp Program to Display Mathematical Expressions on Typewriter-like Devices," *ACM Symposium on Interactive Systems for Experimental Applied Mathematics*, Washington, D.C., August 1967.
4. Carl Engelman, "MathLab: A Program for On-Line Machine Assistance in Symbolic Computations," MTP-18, The Mitre Corporation, Bedford, Massachusetts, October 1965.
5. MathLab, *MACSYMA Reference Manual*, January 1983. The MathLab Group, Laboratory for Computer Science, MIT
6. Michael R. Genesereth, "The Use of Semantics in a Tablet-Based Program for Selecting Parts of Mathematical Expressions," *Proceedings of the 1979 MACSYMA Users Conference*, 1979.
7. John K. Foderaro, "Typesetting Macsyma Equations," MS report (UC Berkeley), December 1978.
8. William A. Martin, "Computer Input/Output of Mathematical Expressions," *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, 1971.
9. Michael Spivak, *The Joy of TEX*, 1980. American Mathematical Society.
10. Brian W. Kernighan, "A System for Typesetting Mathematics – User's Guide," Bell Laboratories Computing Science Technical Report 17, 1977.
11. Joseph F. Ossanna, "NROFF/TROFF User's Manual," Bell Laboratories Computing Science Technical Report 54, 1976.
12. Ken Arnold, "Screen Updating and Cursor Movement Optimization," UC Berkeley, date?.
13. Gregg Foster, *User Interface Considerations for Algebraic Manipulation Systems*, University of California, Berkeley, December 1983.
14. Gregg Foster, *Display of Large Expressions*, UC Berkeley, 1984.

15. John K. Foderaro, Keith L. Sklower, and Kevin Layer, *The Franz Lisp Manual*, 1983. CS Division, EECS Department, UC Berkeley.

16. Sun, *SunCore Programmer's Reference Manual*, May 15, 1983. Sun Microsystems, Inc.

17. R. J. Anderson, "EXED: A Prototype Environment for Algebraic Computation," Unpublished Manuscript, UC Berkeley, 1983.

18. Carl Andersen, "A Cosmetic Expression Editor," Unpublished Manuscript, UC Berkeley, 1983.