ALEX - an Alexical Programming Language

Dexter Kozen*
Tim Teitelbaum†
Wilfred Chen
John Field
William Pugh
Brad Vander Zanden

87-835
May 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*Supported by NSF grant DCR-8602663.
†Supported by NSF grant DCR-8514862

ALEX—an Alexical Programming Language

Dexter Kozen* Tim Teitelbaum† Wilfred Chen
John Field William Pugh Brad Vander Zanden

Computer Science Department
Cornell University
Ithaca, New York 14853, USA

Abstract

ALEX is an experimental language for high-level parallel programming. It is a testbed for exploring various non-traditional ways of expressing algorithmic ideas, making extensive use of high-resolution color graphics. The language itself is not a programming language in the traditional sense, since there is no lexical syntax. This paper discusses the basic design of the ALEX user interface.

1 Introduction

ALEX is an experimental graphical language for high-level parallel programming. Data structures are represented graphically on the screen and manipulated in much the same way that an algorithm would be described to a colleague on the blackboard. Extensive use is made of high-resolution color graphics and the mouse.

ALEX is not a programming language in the conventional sense, because *there is no syntax* in the traditional lexical sense.[1] The programmer creates an internal program representation directly and interactively. A display manager maintains a window into the internal program, providing a customized view of a selected portion of the program structure. Shape, color, and other visual cues are used extensively and may be modified to taste. Powerful primitives are provided for creating and revising the program and for customizing the display. Especially important are primitives to suppress undesired information.

It is tempting to think of the graphics appearing on the screen as the language ALEX. However, the interactive or conversational language that is used to associate functions with

*supported by NSF grant DCR-8602663

†supported by NSF grant DCR-8514862

[1]This rather controversial idea is due to Snyder [8]; we coin the term *alexical*. Just as functional programming aspires to "liberate programming from the von Neumann style" [1], alexical programming aspires to liberate programming from an obsolete style of communication that harks back to the days of teletypes and card punches.

their inputs and outputs, position functions and data objects on the screen, and perform the myriad other tasks associated with specifying a program, is every bit as much a part of ALEX, and a serious design consideration. In this respect, ALEX is more accurately described as a programming environment.

Building a production-quality compiler is a secondary goal of our project; our primary interest is in exploring various alexical ways of expressing algorithmic ideas. For this reason, we have resisted the temptation to create a general purpose language, but have instead restricted our attention to a particular application domain, namely numerical matrix algorithms. These algorithms tend to use simple data structures and exhibit a high degree of inherent parallelism of a certain type well suited to our approach. At present, the only data structures ALEX supports are multidimensional arrays of integers, reals, and complex numbers, but very powerful primitives are provided within this limited domain. We believe that the ideas and techniques we are developing will apply more generally.

In the Section 2, we review some related work. In Section 3, we describe the principal features of the ALEX user interface. In Section 4, we discuss implementation considerations. The description of the internal representation of programs and compilation issues will be the subject of a forthcoming paper.

2 Related work

Previous work on graphical programming has concentrated on representing the control flow, dataflow, or topology of a program. PECAN provides features such as flowcharts and Nassi-Schneiderman diagrams that give pictorial representations of control flow [7]. ALEX differs from these languages in that they emphasize the graphical representation of control flow, whereas ALEX emphasizes the graphical representation of data objects. Apart from hardware design tools, we are aware of no other graphical language with this emphasis; this is corroborated by a recent article in Computer [6].

In general, parallel languages in current use require the programmer either to specify parallelism at the processor level, explicitly specifying message routing; or to use sequential structures that a parallelizing compiler then attempts to remove. Dataflow languages such as VAL and ID define control flow in terms of data dependencies [4,5]. These languages exploit fine-grain parallelism at a low level, and are lexical and not graphical [3]. The Program Visualization (PV) system attempts to specify the overall structure of a program graphically [2]. Languages like OCCAM and parallel FORTRAN address the physical processors explicitly. POKER [8] embodies the idea of alexical syntax, but is intended primarily for low-level parallel programming and circuit design. ALEX differs from these approaches in that it allows the programmer to express the natural parallelism of a parallel algorithm at a high level, independent of the number or names of processors.

3 Principal features of ALEX

ALEX is a functional language [1]. Of conventional languages, it is closest to FP or a dataflow

language with recursion. ALEX allows the graphical representation of two types of objects: *data objects* and *functions.*

3.1 Data objects

ALEX's data objects are integer, real, or complex scalars and multidimensional arrays of scalars. They are represented on the screen by rectangles of various sizes, shapes, and colors. A data object should not be thought of as a block of storage locations, but rather a conceptual organization of data as it is flowing through the program.

Data objects may be created anew by menu selection, or copied from an already existing data object. If a data object is copied, then a data dependency is automatically established between the old and the new data object. Data dependencies are represented by *color matching* (see Section 3.4 below). Colors may be selectively suppressed to reduce visual complexity and then later redisplayed if desired.

The *type* of a data object is either specified through menu selection or inherited from a related data object. Limited polymorphism is permitted, but polymorphic types must be instantiated before compilation. Dimensions of arrays are available as data.

Data objects may have selected subobjects, which appear as smaller rectangles superimposed on the original data object. For example, one or more rows or columns may be selected from a matrix. Selection of subobjects is discussed below in Section 3.5.

3.2 Temporal arrays

A dimension of an array may be regarded as either *spatial* or *temporal.* In general, a temporal dimension is one in which there are data dependencies among the elements, and a spatial dimension is one in which there are no such dependencies. For example, in the sample parallel prefix program,[2] the vertical dimension of the large matrix is temporal and the horizontal dimension is spatial. Although the distinction is conceptually useful, ALEX does not formally distinguish between them; all data manipulation operations apply equally to both.

An array with a temporal dimension is called a *temporal array.* Temporal arrays are most useful in accumulating the partial results of loops, with each entry in the array representing the output of one iteration of the loop. Each iteration of a loop can make use of the results of prior iterations by accessing the appropriate entries of the temporal array. This representation of loops, in conjunction with the selection of a "typical" array element as described in Section 3.5 below, gives a convenient way of specifying a loop by describing a single (typical) iteration. This mirrors very closely the way we think about and describe loops informally. Although we are usually only interested in the data produced by the last iteration of the loop, occasionally the output is the entire temporal array, as for example in the sample QR decomposition program.[2] Such examples distinguish temporal arrays from ordinary **for** loops.

[2] Example omitted from this version for lack of space.

3.3 Functions

A *program* or *function* is a tree-like hierarchical structure. There is a library of primitive system-defined functions, and compound functions may be built by the user from data objects and previously defined functions. Each function has an associated set of typed inputs and outputs, which are data objects.

Functions may appear on the screen in *expanded* or *contracted* form. While a function is being programmed, it is displayed in expanded form and takes up the entire screen. Programming is performed by creating or copying data objects and functions, positioning them on the screen, and establishing data dependencies between them. Once programmed, a function can be contracted to a small box containing an icon created by the user. It can then be saved in a library for later use, copied, etc.

In addition to the standard numeric functions, system-defined functions are provided for conditional evaluation, sorting, merging, and selection, and permutation. Functions can be polymorphic in the sense that their types need not be completely specified; but all types must be instantiated before compilation.

There are no *global* objects. All data used inside a function must be passed to it as an input parameter or created. This is consistent with the philosophy of functional programming [1].

3.4 Representing data dependencies

Data dependencies within a function are represented by *color matching*, or coloring the source and destination data object the same color. An example of color matching is shown in the Appendix. Many other graphical dataflow languages use lines or arrows, which we have avoided for two reasons: in complicated programs, lines quickly begin to look like spaghetti on the screen; and lines require sophisticated routing algorithms in the display manager.

Colors are local to the function; they can be reused outside the function or in a subfunction to indicate a different dependency. A data object receives its color when it is created, either inheriting it from the parent object from which it was copied, or receiving a unique color if it was created anew.

Because of the scoping of colors, a data dependency that crosses a function boundary cannot be indicated by color matching. It is instead indicated by relative position, as follows. When a function is displayed in expanded form, the inputs are so designated by attaching them to the ceiling, and the outputs are so designated by attaching them to the floor. When the function is contracted, the inputs appear as tiny lobes along the top of the function box *in the same order as in the expanded form*, and similarly for the outputs. The dependencies between the formal (inside) and actual (outside) parameters are indicated by relative position along the floor or ceiling.

3.5 Manipulation of objects

ALEX has powerful primitives for moving and reshaping data objects and selecting subob-

jects. We feel that the primitives for selecting subobjects are quite novel and interesting. For example, one commonly wants to select the first or last element of an array or row of a matrix. To select the first row of a matrix, one first creates a floating row of the matrix, which can be carried by the cursor up and down the length of the matrix, but may not leave it. Slamming the row up against the top of the matrix twice in quick succession causes the row to stick to the top of the matrix and become the first row. This will also work for the $i+1$st row, if the ith is already selected; one just slams the floating row up against the already-selected row. One can also select a row of a matrix whose index is designated by a computed arithmetic expression.

It is also possible to sweep out an interval between two selected elements or the top or bottom of an array, and select a "typical" element in that interval. This is used with a spatial dimension to specify parallelism, as described below in Section 3.6. It is used with a temporal dimension to specify a **for** loop, as follows. A temporal array is created of length $n+1$, where n is the desired number of iterations. The first element is selected and an initial value inserted. Then a typical element (say the ith) is selected from the interval $1 \ldots n$, some operation performed on it (the value of i is available in this computation), and the result inserted in the temporal array at the position following the typically selected element. This computation is replicated automatically and invisibly for all i in the interval $1 \ldots n$. The final result is then extracted from the last element of the temporal array.

Input and output parameters, as mentioned in Section 3.4, are so designated by attaching them to the ceiling and floor, respectively, of the function box. That a particular data object is to be an input is indicated by slamming it twice in quick succession against the ceiling, at which point it sticks. Similarly, two arrays of length n can be coalesced into a $2 \times n$ matrix or an array of length $2n$ by slamming them together.

3.6 Parallelism

ALEX is well-suited to the tightly coupled, synchronous, single-instruction-multiple-data parallelism that occurs frequently in scientific and numerical computations. When a "typical" element, row, or column from an array is selected, then any operation subsequently performed on that typically selected element will be replicated automatically and invisibly across the entire array. This corresponds to the "apply-to-all" functional form in functional programming. The matrix multiplication example in the Appendix embodies these features: the programmer selects a typical row of the first matrix and a typical column of the second, chooses scalar multiplication and vector addition function boxes from the program library and applies them to those two vectors, and finally inserts the resulting scalar into the appropriately selected element of the output array.

3.7 Filtering information

It is difficult enough poring over a printed listing of a sequential program, flipping through several pages of **then** clause to find the **else**, etc. The complexity is further compounded in the presence of parallelism. Some programming environments support elision (...), but

it is usually considered an extra nicety to be incorporated into a pretty-printing algorithm, and certainly not a first-class part of the language.

We consider the suppression of unwanted information utterly essential in large-scale parallel programming. ALEX provides primitives for suppression of color and encapsulation. In the latter, a family of subfunctions and data objects can be collected into a new function. Inputs and outputs are created automatically to account for existing data dependencies crossing the boundary of the new function.

3.8 Other features

The language incorporates primitives for restructuring flow of data, for redimensioning arrays, for splitting arrays according to Boolean conditions, for shifting, rotating, permuting, and transposing arrays. There are means for moving about the program tree, for saving and retrieving functions from libraries, and for customizing the display. These features will be described in more detail in the full paper.

4 Implementation considerations

ALEX seeks to provide a medium in which problems can be expressed concisely and naturally—a user's heaven. This creates an equally unearthly challenge for the designer of an efficient compiler, which is subject to the constraints of hardware economy. Our ultimate implementation will therefore have two goals: a user interface that meets the requirements sketched above; and a compiler that produces efficient code on existing or feasible hardware. We have chosen to let the design considerations of the former dominate the design considerations of the latter.

We have implemented a rough black-and-white prototype of the user interface. It is written in INTERLISP and runs on a XEROX Dandelion workstation. The examples in the appendix are actual screen images produced by the prototype. A full scale implementation to run on a SUN3/160C-4 color graphics workstation is planned.

Acknowledgements

We thank Larry Carter, Jens Dill, Bruce Esrig, Jeanne Ferrante, David Loshin, Alex Nicolau, Prakash Panangaden, Frank Schmuck, and Larry Snyder for valuable discussions.

References

[1] Backus, J. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *CACM* 21:8, August 1978.

[2] Brown, G.P., Carling, R.T., Herot, C.F., Kramlick, D.A. and Souza, P. "Program visualization: graphical support for software development," *Computer* 18:8, August

1985, 27-35.

[3] Dennis, J.B. "Dataflow computation," in: *Control Flow and Data Flow: Concepts of Distributed Programming.* M. Broy, ed., Springer-Verlag, Berlin, 1985, 346-398.

[4] Gostelow, K.P. and Thomas, R.E. "A view of dataflow," in: *Proceedings of the 1979 National Computer Conference*, New York, June, 1979. AFIPs Conference Proceedings, v. 48, AFIPS Press, Arlington, Va., 1979, pp. 1079-1086.

[5] McGraw, J.R. "The VAL language: description and analysis," *TOPLAS* 4:1, January 1982, 44-82.

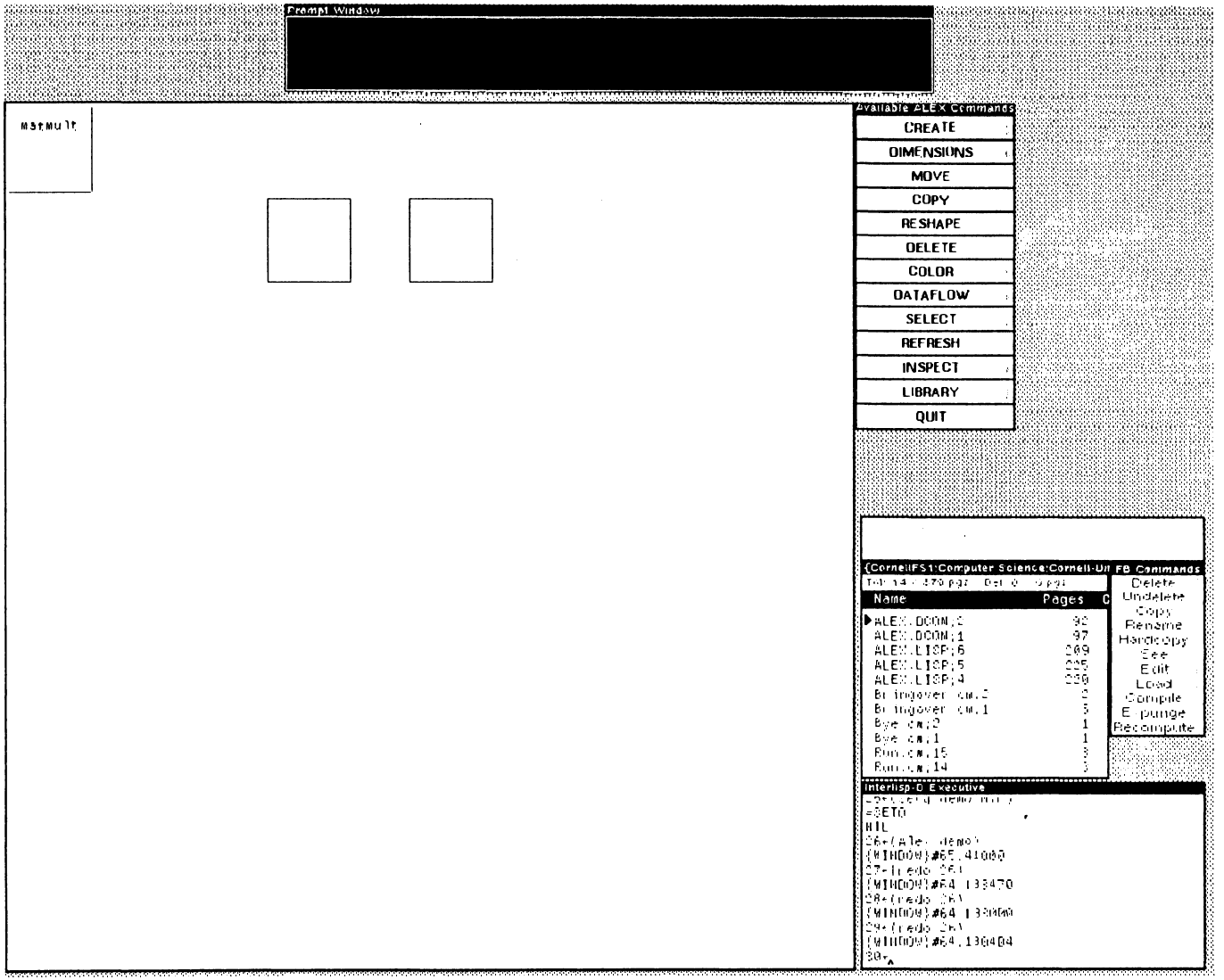
[6] Raeder, G. "A survey of current graphical programming techniques," *Computer* 18:8, August 1985, 11-25.

[7] Reiss, S.P. "Graphical program development with PECAN program development systems," *Proceedings of the ACM Sigsoft-Sigplan Software Engineering Symposium on Practical Software Development Environments*, April 1984, 30-41.

[8] Snyder, L. "Parallel programming and the POKER programming environment," *Computer* 17:7, July 1984, 27-36.

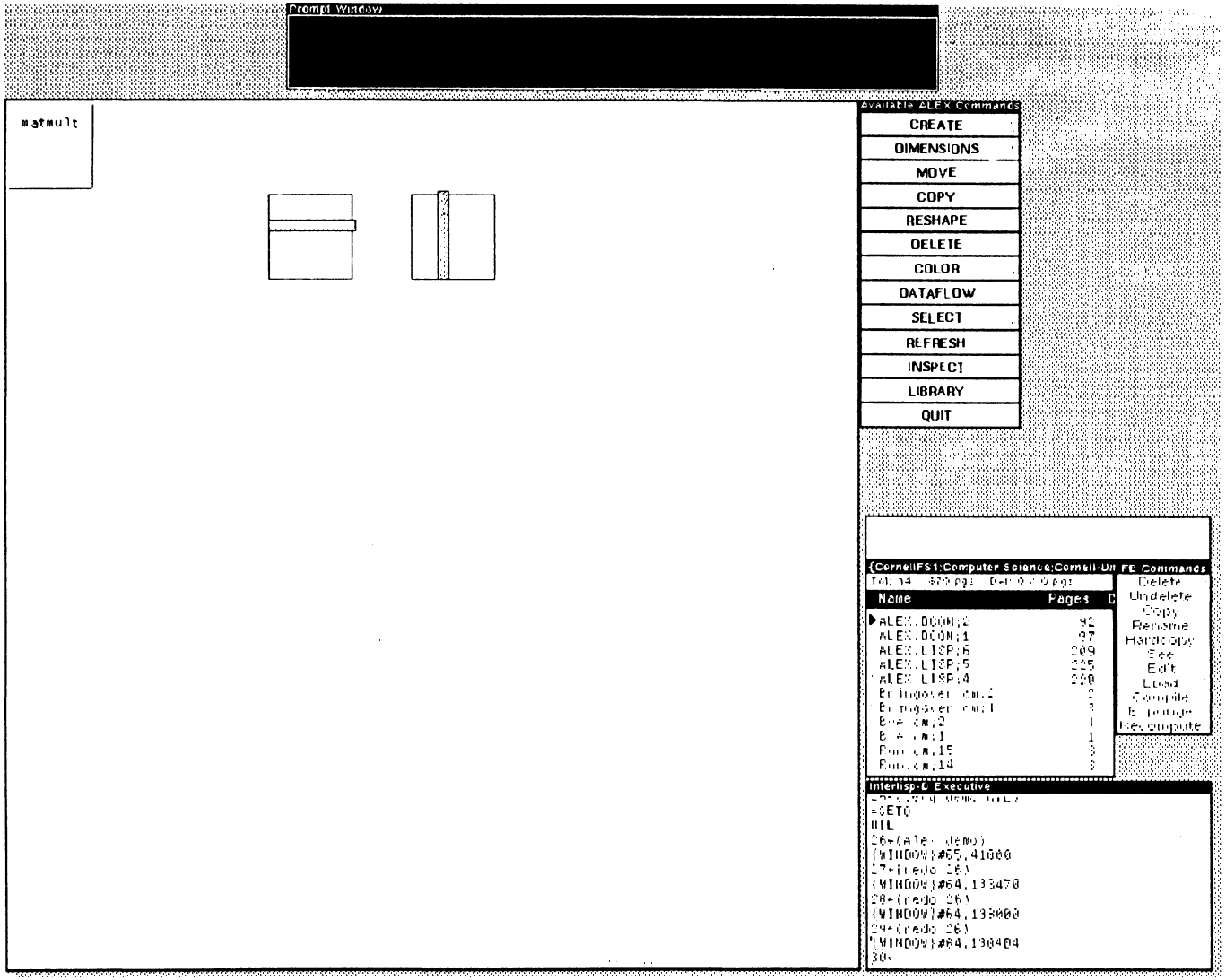
Appendix

The following sequence of figures show how matrix-matrix multiplication would be programmed in ALEX. These figures are actual screen images produced by our prototype. In the full paper we will include example programs for a polymorphic parallel prefix computation and QR decomposition.



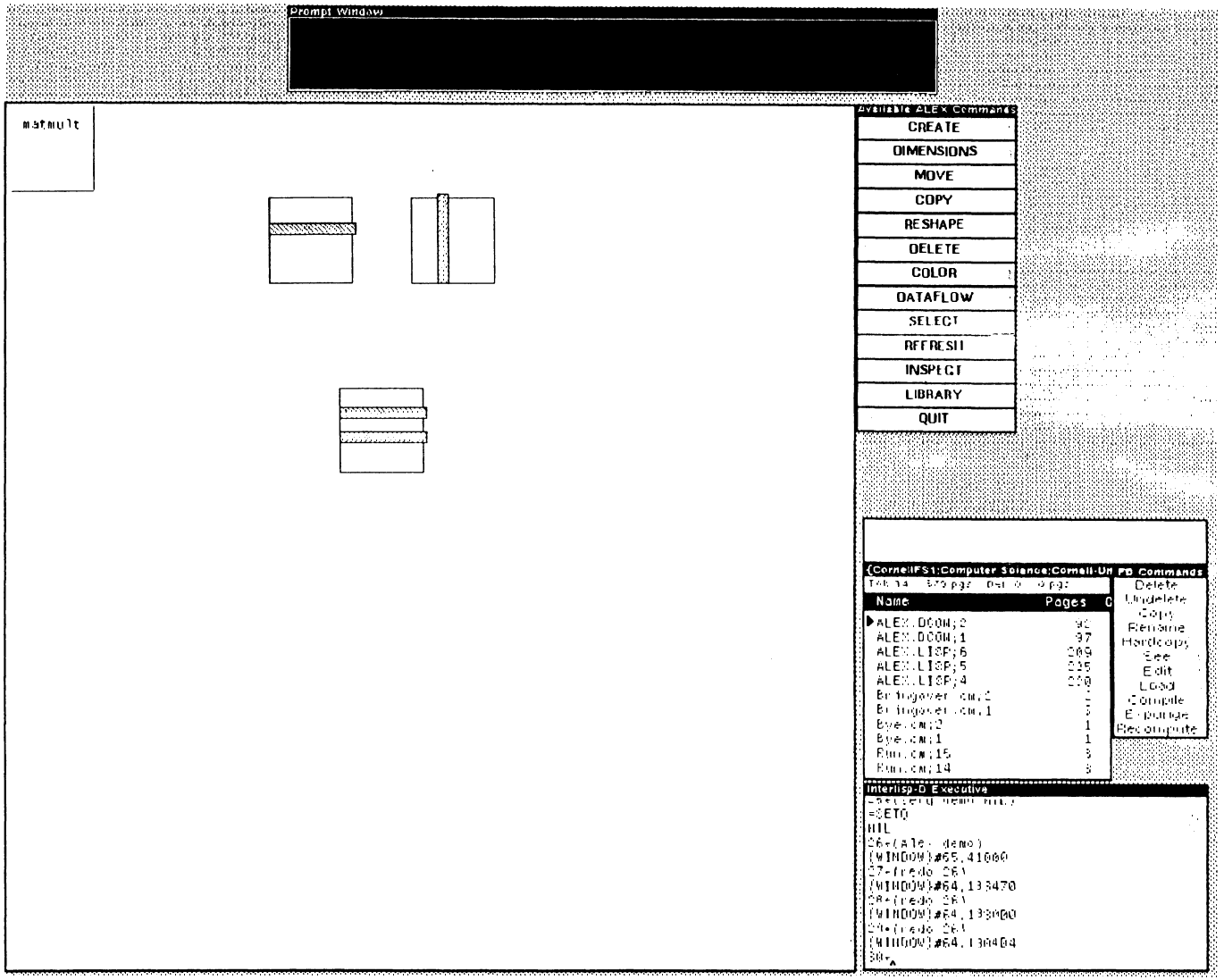
(1) Two matrices are created using the CREATE menu entry. Dimension constraints are established using the DIMENSION menu entry (not yet implemented).

Window Image



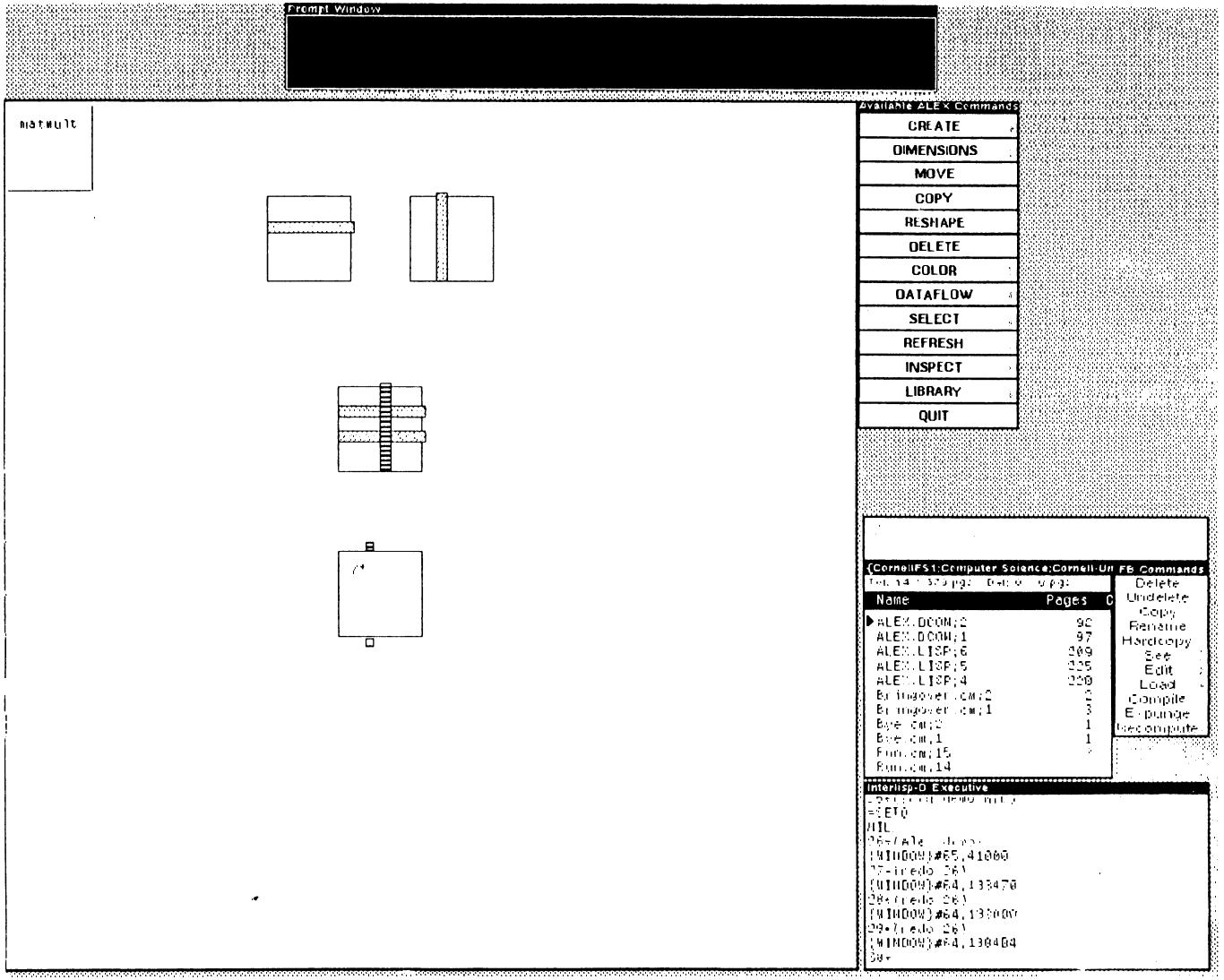
Window Image

(2) A typical i^{th} row of the first matrix is selected using SELECT. A typical j^{th} column of the second matrix is selected. Further operations on these vectors will be replicated automatically for all i and j.



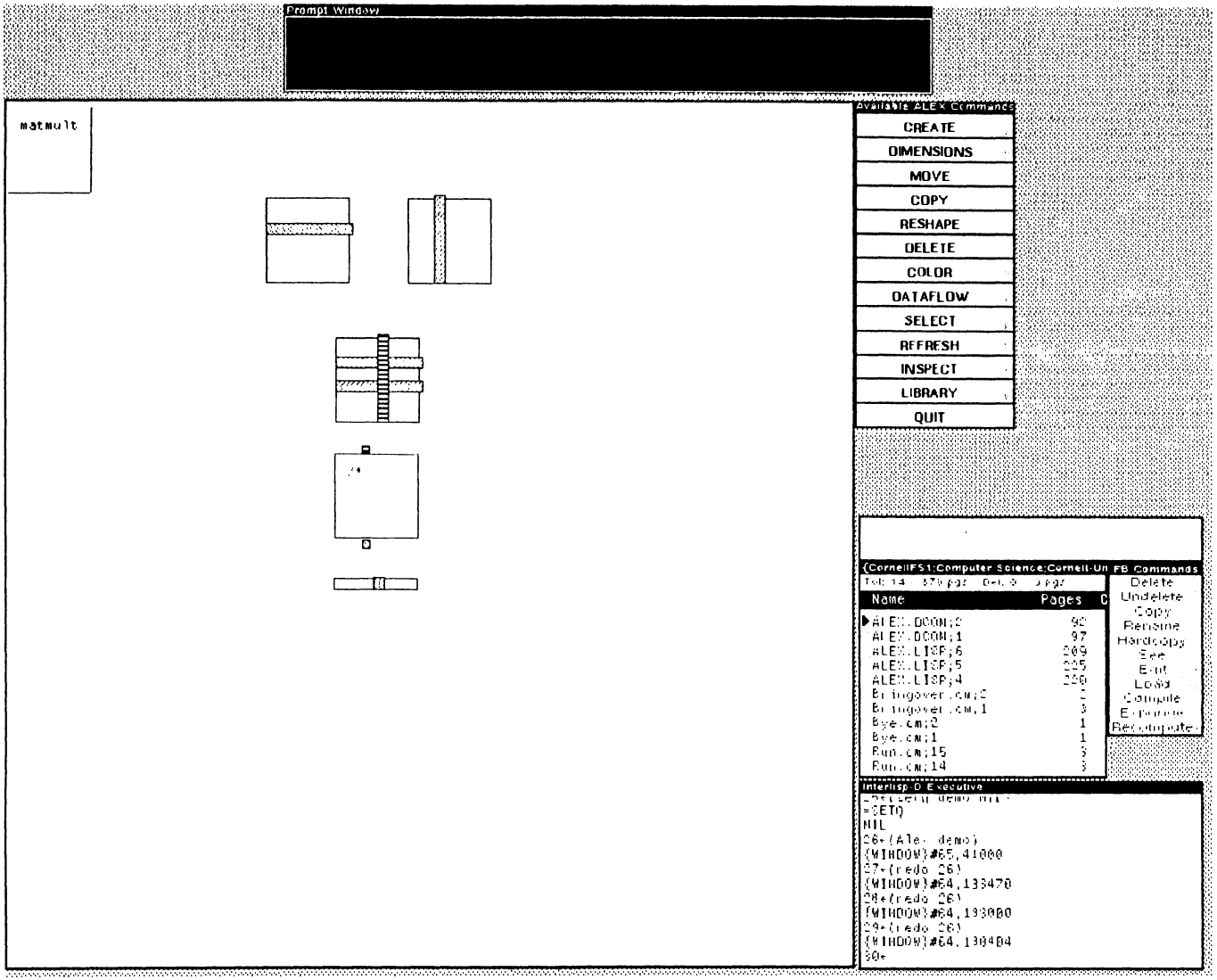
(3) The data from these two vectors is brought down and inserted into a new $2 \times n$ array. The $2 \times n$ matrix could be reshaped to look more rectangular, using RESHAPE (not yet implemented).

Window Image



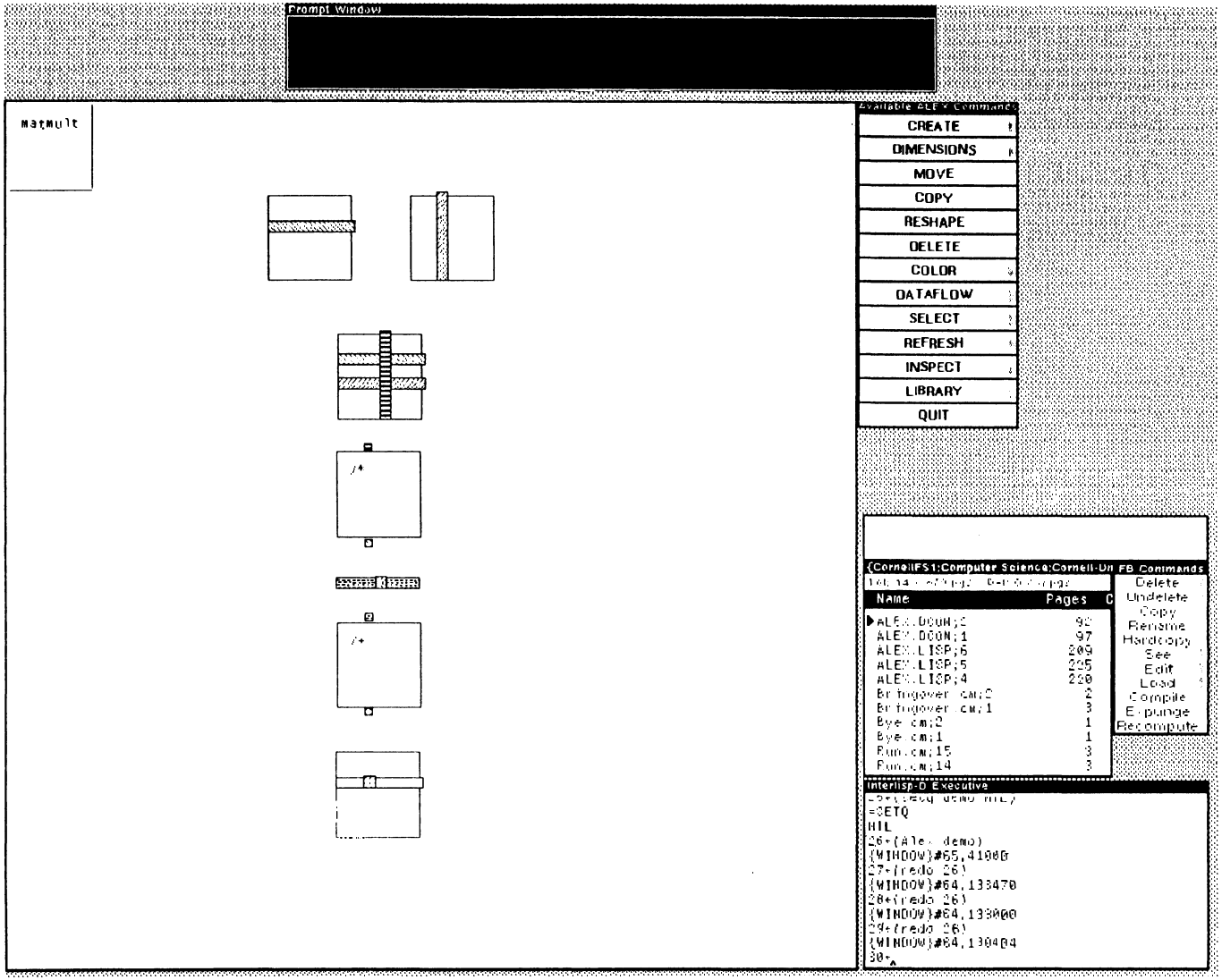
Window Image

(4) A typical k^{th} column is selected and inserted into the input of a /* function, which has been programmed previously and retrieved from the library. The /* function computes the product of elements of a vector.



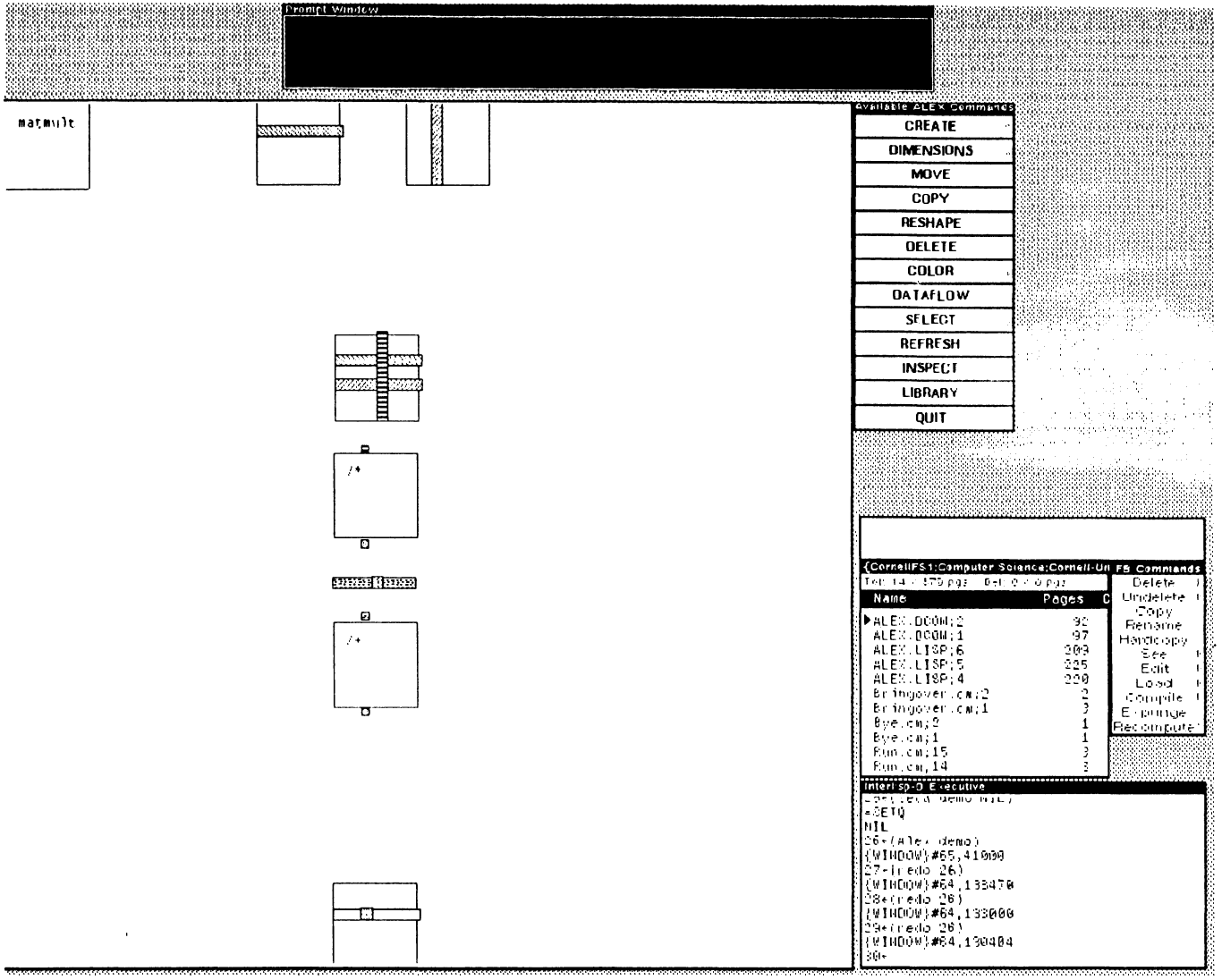
Window Image

(5) The output of /* is a scalar which is inserted into the k^{th} element of a $1 \times n$ array. k is the same as in (4) and is selected using SELECT.



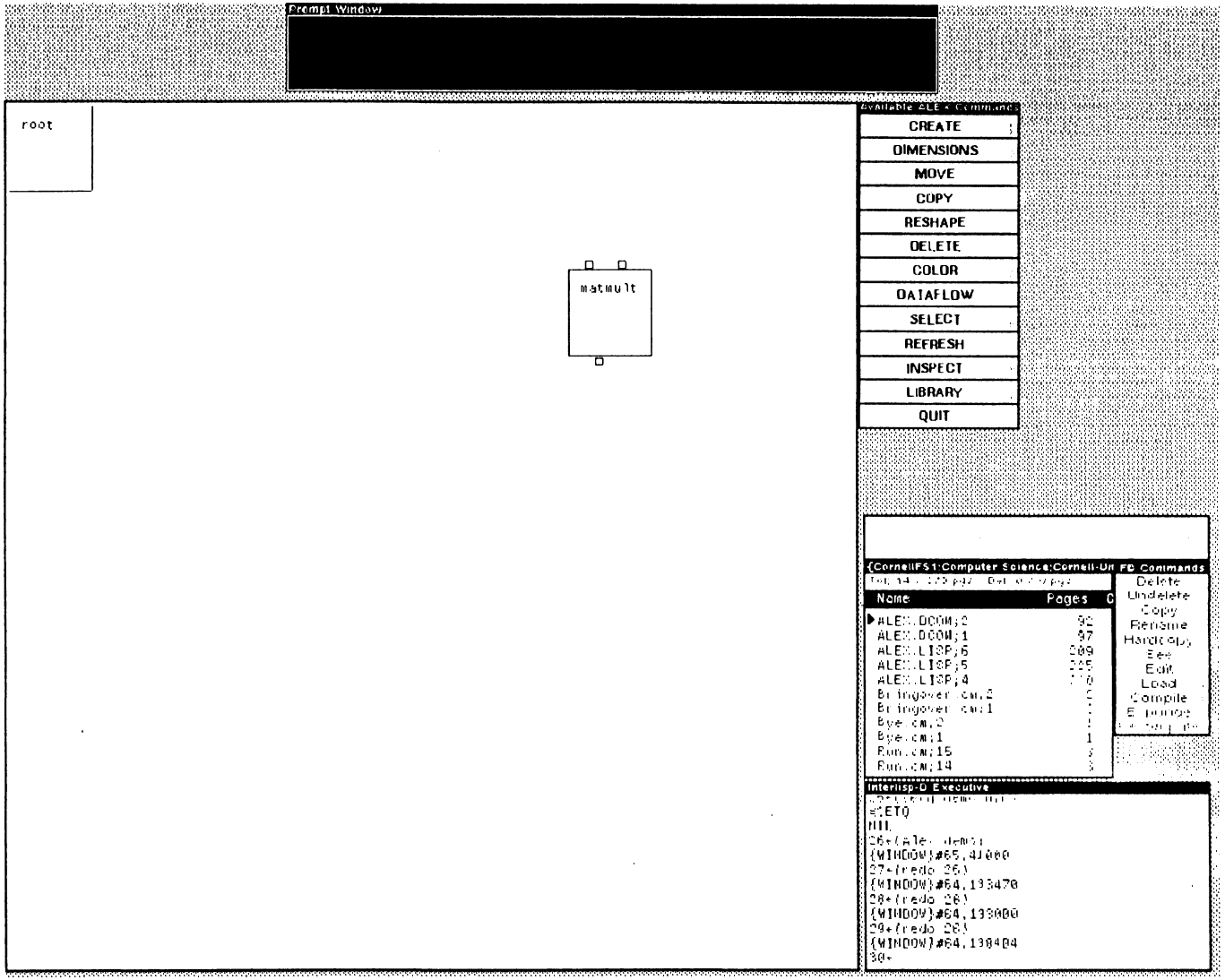
Window Image

(6) That array is then inserted into the input of a /+ function, which sums a vector. The function /+ has been previously programmed and retrieved from the library. The output is a scalar and is inserted into the i, j^{th} element of the output matrix. i and j are the same as in (2).



(7) The two input matrices are moved up against the ceiling to indicate that they are inputs to the function. The output matrix is moved down against the floor to indicate that it is an output of the function.

Window Image



(8) The function we just programmed has been contracted. The inputs and outputs show up as lobes. This function can now be saved in the library and retrieved as needed.

Window Image