



# Visual Abstraction in an Interactive Programming Environment

Michael L. Powell  
Mark A. Linton

Computer Science Division  
Department of Electrical Engineering and Computer Sciences  
University of California  
Berkeley, CA 94720

## ABSTRACT

We are designing a software development system that implements "what you see is what you get" for programming. The system, called OMEGA, allows software to be displayed, processed, and modified, using pictorial representations to convey the structure and levels of abstraction of the program.

OMEGA takes advantage of the interactive user interface to provide syntax-free input, user selectable display format, and incremental semantic analysis. By distinguishing input specification from output display, and exploiting interaction in semantic analysis, we are able to unify the different abstraction mechanisms present in traditional programming environments.

## 1. Introduction

Ideas in programming languages (data abstraction, overloading, type parameterization), user interfaces (menus, pointing devices, graphics), and database systems (relational data models, recursive data, views) are converging on the problem of managing large software system development. We have combined these ideas in the design of a programming system, called OMEGA, that provides powerful mechanisms for constructing and manipulating software. OMEGA will use a high-resolution, color graphics display with a pointing device to view and modify program structures that are stored in a general purpose database system.

To simplify the construction and manipulation of software, programmers abstract recurring concepts into reusable parts. Current programming languages provide builtin parts, (e.g., statements, variables, data types,

Research supported by NSF grant MCS-8010686, a State of California MICRO grant, and Defense Advance Research Projects Agency (DoD) ARPA Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

modules) and mechanisms for creating new constructs (by, e.g., writing a procedure, declaring a variable, defining an abstract data type, or instantiating a module). These mechanisms allow programs to be modified easily, since a change to the definition of a part affects all its uses.

Due to the independent evolution of program structures and their different requirements for parsing in conventional programming systems, each has its own way (syntax and visual representation) for programmers to specify abstractions in terms of simpler elements. For example, in some languages, a program may define a new kind of integer that can be used just as easily (with overloaded operators), efficiently (with inline expansion of procedures), and cleanly (with implementation details hidden) as the native integer type. However, in most languages, it is not possible to define a new kind of **for** loop. Another inconsistency is in overloading of identifiers. Although it is often possible to overload procedure names based on type, it is not possible to overload variable names in the same way.

OMEGA is an interactive programming environment that provides a single form of abstraction that supports the language and database facilities of a software environment. The user interface to OMEGA provides a simple and powerful mechanism for creating, viewing, and modifying abstractions.

In OMEGA, we employ the concept of "what you see is what you get", which has been applied in many applications. OMEGA users define visual representations of their programs' objects and structure. Thus they can directly manipulate objects and immediately observe the results of those manipulations. This is in contrast to the idea of conventional software development, which builds a description of the desired computation that is subsequently compiled.

## 2. Other Programming Systems

The design of OMEGA has been influenced by the positive and negative aspects of existing systems. Concepts such as abstraction, extensibility, strong typing, integration, and background compilation are important in supporting the programming process. In contrast, we believe many features of existing systems, such as syntax errors, reserved words, identifier scopes, and tree-structured program representations, are usually impediments to programming.

In tool-based programming environments such as UNIX† [Kernighan and Mashey 81], each tool has its own abstractions that are often not compatible with the programming language or other tools. For example, the UNIX command interpreter provides string variables and the concatenation operator; the C language does not. The C language provides subroutines with variables as parameters; the UNIX command interpreter does not. Yet each of these features would be useful in both environments.

Integrated programming environments such as Interlisp [Teitelman and Masinter 81] do provide uniform interfaces to programming facilities. This provides the user a consistent way to view and modify programs. However, it provides only a single way of viewing program structures: as LISP lists. The most limiting problem with LISP systems, though, is the difficulty of static analysis and checking of the abstraction mechanisms due to the weak and dynamic types.

Other systems such as the Cornell Program Synthesizer [Teitelbaum and Reys 81] and the Incremental Programming Environment [Medina-Mora and Feiler 81] support richer language semantics, but the range of semantics is fixed. Both these systems are tree-oriented, in that program construction consists of adding or changing nodes in a tree. Neither permits definition of new types of nodes, only instantiation and composition of the builtin ones. Moreover, a tree structure is awkward for describing type information and module dependencies.

### 3. Goals and Ideas of OMEGA

OMEGA is intended for large software system programming using strongly typed language semantics and executing compiled code. In an interactive system, it is desirable to have errors caught as soon as possible. Furthermore, the system should help users produce correct software, not simply prevent them from producing incorrect software. One of the goals of OMEGA is to take advantage of an interactive user interface by having

- no input syntax
- multiple output formats
- interactive semantic analysis
- multi-threaded program organization

No input syntax means that the user is not required to cast the program in one particular form for a compiler. Program construction should be a conversation between the programmer and OMEGA, with OMEGA occasionally asking questions and making suggestions. Although the database command language will have a syntax, it is not a programming language. Thus it does not place constraints on the structure of a program, but only specifies the kinds of operations that may be done to create one.

Support for multiple output formats means that the user may have program structures displayed in a variety of ways, depending on the aspect of the program of interest at the moment. Programming systems typically

use the language as both the input specification and the displayed form of the program. As a result, compromises must be made between what can be parsed and what information should be displayed. Graphical output and icons should be exploited to convey the most information in an easily assimilated way. At the minimum, output formats must support the multiple ways of building programs, to allow the user to work without mentally translating between points of view.

Interactive semantic analysis means that a program is examined as it is being built. Just as oral communication is more effective than written communication, because the speaker can adjust to the response of the listener, the system should provide feedback to the programmer as the program is built. Errors due to inconsistency or ambiguity should be resolved immediately. In addition, by displaying the structure of the program as it is being built, it may help the programmer see higher-level problems that the programming system cannot detect.

Multi-threaded program organization means that there may be multiple threads through the program representation along which manipulations may take place. Conventional programming systems provide only one view of a program. The programmer, however, may see the program in different ways when it is being built, modified, or debugged. For example, a group of statements might be edited as a unit because they appear in the same procedure, because they all reference the same variable, or because they will be executed consecutively even though they are in different procedures.

The remainder of this paper describes in more detail the interactive user interface to the OMEGA system. We first describe the features of the interface that allow the user to create programs. Then we discuss how this interface enables the user to control the display of program constructs. Next we describe how this interface interacts with the program being manipulated. Finally, we give some indication of how a database could help support abstraction in this environment.

### 4. User Interface

The key to lifting the burden of syntax from a programming environment is to stop using text as the medium of program construction. Conventional programming languages and systems represent program constructs as *text*, sequences of characters forming words, usually grouped in lines.

Text hampers human understanding because it is not unique visually; "free format" languages allow tokens to be placed in many different positions. Text is not an optimal internal representation for a program, since semantic properties cannot easily be determined without converting it to some other form such as a parse tree and a symbol table. Text is also not a good representation for editing. Logically one wishes to operate on program structures (e.g., statements, variables, types, etc.); using a text editor one must manipulate some combination of lines, words and characters.

OMEGA resolves the different needs for program representation by allowing the program to be entered, displayed, edited, and analyzed in different formats. This flexibility is provided by separating the pictorial representation of an object from the object itself, by

---

†UNIX is a registered trademark of Bell Laboratories.

pointing rather than typing to identify objects, and by using multiple windows to allow pieces of programs to be constructed independently.

#### 4.1 Pictographs

Most programming environments do not distinguish between an object and the pictorial representation of that object. In OMEGA, program structures are displayed consistently as *pictographs*. A pictograph is a view of the object displayed on the screen. Pictographs may be arbitrarily assigned to objects; different pictographs for the same object may be selected when different aspects of the object are to be emphasized.

A pictograph consists of letters or icons arranged in a two-dimensional area. The optimal display device is a high-resolution color graphics device, which would allow color, intensity, and non-character graphics to be used. The principles of pictographs apply to lower resolution, black and white, or character-only displays. However, existing 1920 character CRTs probably hold too little information for these ideas to be used on any significant scale.

A pictograph is the visual object that a programmer sees and manipulates. Shapes and spatial relationships help convey structural information. An important feature of a pictograph is that parts of it can be used to represent slots into which parameters are placed.

Figure 1 shows an example pictograph for a table search. A table search is a two-exit control structure, since the desired element may or may not be in the table. The slots in the pictograph show places where parameters may be inserted for the table to be searched (*Table*), the key for the desired entry (*Key*), and the variable to point to the object desired (*Element*). Note that *Element* has a default value; use of the pictograph defines an object if no other one is substituted.

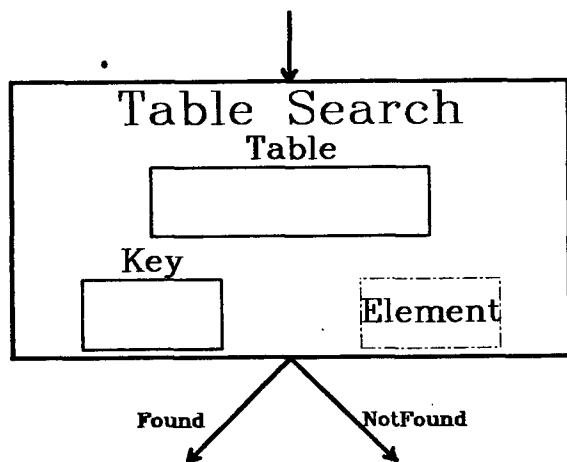


Figure 1. Graphical Table Search Pictograph

Figure 2 shows another pictograph for the same control structure. This shows more details of the implementation and is in the traditional text form. This lower-level view of the control structure reveals aspects that are hidden by the higher-level view. In Figure 1, the parameters to the pictograph are represented by boxes; in Figure 2, by italicized words.

label *NotFound, Found*

var *element* : subscript of *Table*

if empty(*Table*) then goto *NotFound*  
*element* := first(*Table*)

loop

if *element*.key = *Key* then goto *Found*

if *element* = last(*Table*) then goto *NotFound*

*element* := next(*Table*,*element*)

endloop

Figure 2. Text-like Table Search Pictograph

An important collection of pictographs are those representing objects in the program. These pictographs may appear in the program structure, but also may appear in a *glossary*. A glossary is simply a list of pictographs and their meanings. Figure 3 shows a glossary that might exist in a program using the table search of Figure 2.

<i>Employees</i>	array of <i>EmployeeRecord</i> , table of all employees
<i>InputName</i>	<i>EmployeeName</i> , name of employee just read
<i>CurrentEmployee</i>	<i>EmployeeIndex</i> , points to the record of the current employee

Figure 3. A sample glossary

#### 4.2 It /s Polite To Point

Our alternative to entering text is to display relevant pictographs on the screen and have the user point at, pick up, and put down the corresponding objects using a pointing device (e.g., a mouse, light pen, finger, etc.). "Picking up" and "putting down" generally mean pointing at something and pressing a key or button. The act of picking up an object and putting it down someplace may have different effects based on the objects and the parts of the pictograph pointed at. Picking up the *EmployeeRecord* pictograph in the glossary in Figure 3 and pressing the "what is this?" button would cause a description of the type *EmployeeRecord* to be displayed. Picking up the *Employees* pictograph and putting it down in the *Table* box of Figure 1 makes *Employees* the actual parameter of the *Table Search* pictograph.

For example, consider the search procedure in Figure 1. The pictograph in Figure 1 might be displayed as a result of a query asking for search procedures. To use the *TableSearch* control structure in the program, we first pick up a copy of it by moving the mouse to the pictograph and pressing the pick up button. We place it at the desired point in the statement list we are working on by moving the mouse just below the statement we wish it to follow and pressing the put down button. This causes the entry line of the pictograph to be connected to the previous statement.

The parameters are filled in by picking up the objects and putting them down in the boxes. The two possible exits are now sites for additional statements to be connected. In this manner, the *TableSearch* control structure is inserted into the program.

#### 4.3 What Is In A Name?

Identifiers in programs serve two functions: they provide a visual tag that the reader uses to associate together different instances of the same object, and they provide a mnemonic description of some properties of the object. In traditional systems, these two purposes run against each other. Shorter, more distinct identifiers are easier to resolve visually, yet longer identifiers that often may be similar are more descriptive. In OMEGA, these two functions are separable. Pictographs may be assigned to objects arbitrarily to improve the visual representation of the program; properties of the object are instantly accessible (and may be displayed on part of the screen as a glossary) from the database.

The ability to name by pointing adds significant power to the programming environment. For instance, it is not necessary for displayed pictographs to be unique. If it is necessary to disambiguate a name, the user simply points to the intended pictograph in the glossary (or somewhere else on the screen). Since the system always references objects and merely displays pictographs for the convenience of the user, the same pictograph may be used in different parts of the program without causing confusion about what object they refer to.

In conventional programming systems, the case often arises that the best name for an instance of a data structure is the name of the type of the data structure. This must usually be solved by adding a prefix or suffix to the one or the other of the names. A similar problem occurs here; when pointing to a pictograph, it may be meaningful to pick up either the actual object or a new instance of the object. Such problems are easily avoided by allowing several pick up keys. For example, after pointing to a variable, the user might choose to pick up the variable itself, the variable's type or value, or even a new variable of the same type as that variable.

#### 4.4 Rome Was Not Built In A Day

One of the advantages text-oriented interfaces have had in the past is the support of partially-formed programs. Since no examination of the program occurs until the user requests it, it is easy to leave loose ends to be fixed up later. Tree-oriented systems, in particular, often have restrictions, for example, that nodes must be added top-down. Moreover, the transformations possible on text are limited only by the power of the text editor

and the imagination of the user. Structure-oriented editors often make some transformations difficult; for example, it may not be possible to change one kind of a node to another without first deleting and then recreating the node's children.

There are some transformations that can be accomplished only with text-oriented systems. For example, moving delimiters to make what used to be a string or comment into program statements requires parsing. "Commenting out" code is a meaningful and straightforward transformation in OMEGA, however, and it is not necessary to resort to text tricks to accomplish it.

Programs are not represented linearly on the screen in OMEGA. It is possible to build several program fragments independently in different windows and connect them together by picking up and moving around pictographs. For instance, in the previous section, it would have been equally possible to assign the parameters to the *TableSearch* construct before inserting it into the program as a statement.

One freedom a pointing interface does not allow is that of referring to an object that is not yet defined. This is not so bad since the parameters of an operation can be defined without defining its implementation. For example, one cannot create a call to procedure *f* before creating the procedure, but one can create *f* and refer to it before specifying its body. Eventually, the program will reach a state in which all necessary objects and attributes have been specified, and then be ready to run.

#### 4.5 An Example

Figure 4 shows what the screen might look like during an OMEGA session.

Catalog windows are the primary means of searching for information in the database. Standard queries will allow users to locate previously defined operations, objects, and program fragments that they can use. Things in the catalog may be displayed in different ways. For example, the lower right window shows operations on booleans; the middle right window shows operations used to read from a file.

Glossary windows are created in conjunction with program windows. The glossary is the place where the two functions of traditional identifiers, tags and descriptions, are brought together. It displays the pictograph for objects and descriptions of what the objects are. Normally, the glossary associated with a program window will contain entries for each object displayed in the program.

A window typically belongs to one of four classes. A *catalog* window displays a subset of the available operations that are defined, including objects, control structures, operations, etc. A *program* window displays possibly partially assembled program fragments. A *glossary* window displays information about pictographs on some part of the screen, usually in a program construction window. A *response* window displays output from some command or program, e.g., an error message.

The details of window management are beyond the scope of this paper. However, we are developing automatic window management strategies, and expect nearly all decisions about where to put information to be made by the window manager.

















procedure readdata	files
initialize 	read from file
while not full  do	write to file
tmp := new 	<b>data structures</b>
{ set contents of tmp here }	fill data structure
end while	sort 
	sort 
B and not eof 	<b>read from file</b>
read tmp from 	eof  : boolean
	eoln  : boolean
	read  from 
<b>glossary for readdata</b>	readln 
 : aggregate	reset 
 : file of text	<b>boolean operations</b>
tmp : element of 	B and B : B
B : boolean	B or B : B
	not B : B

Figure 4: Sample OMEGA screen.

gram window. Of course, it is possible to have some of the well-known entries omitted.

The top three windows on the left show program fragments under construction. The third window shows a statement that will probably be moved to replace the comment in the top window.

Thus far, we have relied on the reader's intuition for an understanding of what will happen when pictographs are put together. In the next section, we describe more details of the abstractions that pictographs represent.

## 5. Abstractions

We use the term *abstraction* to refer to the general class of things that pictographs represent. An abstraction may be a program object such as a variable, type, control structure, or operation; it may be a program constructor such as a variable declarator, procedure template, or type former; or it may be a program manipulation command such as a query, configuration definition, or directive. Abstractions are defined using other abstractions.

There will be abstractions called *variable*, *package*, *procedure*, and *type* that are used to create typical program objects. An operation that places or instantiates an abstraction causes some semantic changes to the program database. For example, instantiating a variable abstraction causes entries to be made in the database to indicate that a new variable of the specified type has been created.

An abstraction has three parts: the pictograph that represents it, the parameters (and how they appear in the pictograph), and the semantics of the operation on the database. The pictograph determines what the user will see, and what the visual interaction is. The parameters specify what kinds of objects can be connected to the abstraction and how that is done using the pictograph. We will not discuss the semantics of the database in this paper, except to give an idea of how the database will be manipulated by abstractions. The operations performed are similar to those done during syntactic and semantic analysis of conventional programming languages.

### 5.1 Defining and Using Abstractions

Consider the following simple abstraction for creating variables.

Abstraction: declare a variable  
Pictograph: var *name* : *type*  
Parameters: *name* is a pictograph  
              *type* is a type object

Actions: Create a new variable object  
Set the variable's pictograph to *name*  
Set the variable's type to *type*

The pictograph in the example is similar to declarations in conventional languages. Note that simply by changing the pictograph in the *declare a variable* abstraction to be "*type name*;", declarations would be displayed in a C-like format instead of a Pascal-like one.

Suppose we wish to define the exponentiation operator. The following abstraction would be used:

Abstraction: declare a function  
Pictograph: function *name* (*parameters*) : *type*  
                  *body*

Parameters: *name* is a pictograph  
              *parameters* is a parameter list object  
              *type* is a type object  
              *body* is a statement list object

Database: Create a function object  
Set its parameter list to *parameters*  
Set its return type to *type*  
Set its body to *body*  
Define its database semantics to insert a call to the function body

As one might expect, there are also abstractions for statements, parameter lists, and other program structures. If we wish to define the exponentiation abstraction, we would perform the following steps:

- Create a new function by pointing at the "declare a function" pictograph and pushing the "new" button. The pictograph for the definition of the new function will be displayed in a newly allocated window.
- Construct its parameter specifications using the "build a parameter list" abstraction. It would presumably contain a real parameter called *base* and an integer parameter called *exponent*.
- Connect the parameter list to the *parameters* part of the function definition.
- Pick up a reference to the data type "real" pictograph and place it on *type*.
- Construct the function body in the *body* slot by creating and connecting the necessary declarations and statements.
- Build a pictograph for exponentiation referencing the *base* and *exponent* pictographs and place it in the *name* slot.

Once the exponentiation function has been defined, we may install it in the catalog. This would be done using the "create catalog entry" abstraction, which might have parameters such as the pictograph for the function and a list of attributes on which to index the function. A subsequent reference to the function creates an instance of the function abstraction, which will cause the specified database operations to be performed when all of the parameters have been bound.

## 5.2 Semantic Error Detection

As the user manipulates abstractions, updates are made to the database. Note that this does not necessarily imply a change to the resulting program. Any change, such as defining a variable or creating a new statement, modifies the database. The program will be altered only when the statement or variable is connected to the program. Moreover, the program will be changed only when a complete, consistent, and correct modification has been made.

Once the abstraction has been completed (i.e., all parameters are specified), the updates specified by the abstraction are attempted. This updating takes place as a transaction on the database system. Erroneous transactions will not complete and improper objects will not appear as part of the program. For example, a statement may refer to variable objects whose type has not yet been specified. The insertion of such a statement would not take effect until the type is defined. When the type gets defined, all references to the variable are checked to be sure they are consistent with the type. If they are, the statements are added to the program; otherwise, the statements, though in the database, do not yet affect the program.

Each time an object is connected to a parameter, a check is made to see if the object meets the parameter's specifications. If it does not, the object is not connected and an error message is generated. For example, connecting a variable object to the *type* parameter in "define a function" would result in an error. This is sort of a "square peg into a round hole" approach: the user cannot bind an object to a parameter if doing so would result in a type violation.

An application-level database transaction mechanism is used to manage partial updates to the program. Since the completion of one update may trigger the initiation of others, it is essential that multiple transactions be allowed at once. Note that these transactions are built on top of the standard lower-level transaction mechanism, which ensures the reliable and consistent storage of the state of the programming environment, even if that state describes a partial or incorrect program.

The semantic analysis necessary to determine if a parameter "fits" is equivalent to that done in a compiler after names have been resolved to objects. Although the user may give an object a name by putting an identifier in its pictograph, references to a pictograph lead directly to the associated abstraction. This eliminates the problem of resolving overloading for procedures since the user points at the actual procedure, not the name of a procedure.

Because the semantic error detection is done as the program is constructed, errors are detected and fixed by the user in the context in which they occurred, not after some period of time during which the user has forgotten why the mistake was made. Many sorts of errors (missing parameters, undefined variables) simply cannot occur due to the sequence of operations necessary to create the program.

Global changes that affect many parts of the program may be performed more reliably because OMEGA can detect incomplete changes. If it is necessary to add a parameter to an operation, the system can find and request modification of each instance. Of course, it is not required that all instances be fixed immediately. Such temporary inconsistencies or abstraction invocations without all of their parameters forms a task list of work to be performed by the user.

The implementation of abstractions depends critically on the underlying database system. In the next section, we describe some properties of the database necessary to support pictographs and abstractions.

## 6. Multi-Threaded Database

The user sees a program as a collection of objects (variables, procedures, types, statements), which are the traditional program components, and a collection of abstractions (declare something, compose something, create something), which are "parameterized recipes" for program construction.

Unlike conventional programming languages and environments that enforce a strong structure on how these objects and abstractions are manipulated, OMEGA permits the user to arrange them in the manner most suited to the task at hand. Although it is possible for programs to be organized by procedures according to some hierarchy, it is also possible for other organizations to be used.

One example of such organization is that used in most layered network protocols. Each layer is usually divided into different functions. It is difficult, however, to edit similar functions in different layers together, since traditional program organization would make each layer a module, and each function a submodule of its layer. Such cross-sectional organization is important when modifications are being made to large, multi-function software systems.

In OMEGA, the same program may be manipulated according to several different organizations. If we are changing the whole link-level protocol, we will work with a "horizontal" thread; if we are changing error processing, we will work with a "vertical" thread; if we are changing the buffer data structure, we will want a thread through all modules that manipulate the buffers.

For users of conventional programming languages, a hierarchical structure may be most familiar. Recent developments in programming languages have favored modular structures, with restrictions on which objects and operations are available to which modules. OMEGA not only makes such constraints easy to describe and check, but allows auditing of usage in a natural way.

Languages such as Ada† require the programmer to describe modules twice – once from the perspective of the implementor, and once from the perspective of a user. OMEGA needs only one description, plus indications of what parts should be available to users. In fact, it is easy to generalize the notion to allow different classes of users to have different levels of access to the implementation of the module.

The details of the program database are presented in [Linton 83]. Difficult problems that are being resolved include the storage of graphical data for pictographs, convenient and efficient storage and access of program data structures, as well as a more complete implementation of the mechanisms described above.

## 7. Implementation Status

We have begun implementing OMEGA by building a program to view and modify objects in the database. To initially create a database of programs, we have built a parser that takes text for the MODEL programming language [Morris 80] and stores the internal representa-

tion in the database. The current prototype displays textual pictographs, processes simple queries and makes simple updates to the program. Colleagues are defining database semantics and figuring out how to generate code from the database.

The overall goals of OMEGA eliminate the possibility of a complete working system for several years. Our approach is to identify important subproblems and build prototypes to experiment with solutions to those particular problems. As work by others in database systems, graphics, and program semantics progresses, we will incorporate their results into the system. We are also watching for developments in hardware systems that will provide an appropriate vehicle for a production OMEGA system.

A problem that we are partially addressing is the introduction of existing software and programmers to OMEGA. Though we have proposed a "syntax-free" form of input, a traditional textual interface could be provided using an incremental parser with associated semantic actions. One alternative would be to try to match the syntax of existing pictographs; this would require the definition of a set of pictographs that are parsable. Another approach would be to have a simple specification language such as is found in most LISP systems. Regardless of the format in which a program is entered, it can be displayed subsequently according to any available pictographs.

Our current solution completely parses a program and enters it into the database. This permits us to start with substantial programs, and to rapidly enter programs into the database.

## 8. Conclusions

The fundamental problems of supporting software development are communication and information management. The communication aspect involves interactions with a user. The information being managed includes various properties of the program as well as its structure and contents.

Graphical input and output provides efficient and effective ways of expressing and representing the relationships between different program elements. Rather than forcing the programmer to express the program in terms of character-string tokens that are easy to parse, we provide a structural interface to allow the programmer to build the program. Instead of using unique identifiers for objects, which requires rules for resolution, we separate the picture of an object from the object itself.

OMEGA provides a simple model for how program information is manipulated and uses a general-purpose database system to store it. By leaving issues such as consistency, error recovery, query optimization, and efficient storage management to the database system (whose authors spend most of their time worrying about them), we can concentrate on the difficult issues facing software developers. Because the partially constructed program is stored in a database, it is possible to immediately check for programmer errors. Moreover, because the program is built rather than typed, a variety of common errors cannot be made. Storing the program in a

---

†Ada is a registered trademark of the Department of Defense.

database also allows it to be viewed differently depending on how it is being manipulated.

Abstraction is the mechanism that humans use to organize and manage information. By designing OMEGA around a powerful abstraction mechanism, we provide tools that mimic and support human programming processes rather than simply helping the human programmer cope with the machine.

## 9. References

[Kernighan and Mashey 81]

Kernighan, B., and Mashey, J., "The Unix Programming Environment", *Computer*, Vol. 14, No. 4, April 1981.

[Linton 83]

Linton, M. A., "Queries and Views of Programs Using a Relational Database System", Ph.D. Thesis, in progress, Computer Science Division, Univ. of Cal., Berkeley.

[Morris 80]

Morris, J. B., *A Manual for the MODEL Programming Language*, February 1980.

[Medina-Mora and Feiler 81]

Medina-Mora, R., and Feiler, P., "An Incremental Programming Environment", *IEEE Transactions of Software Engineering*, Vol. SE-7, No. 5, September 1981.

[Teitelbaum and Reps 81]

Teitelbaum, T., and Reps, T., "The Cornell Program Synthesizer: A Syntax-directed Programming Environment", *Communications of the ACM*, Vol. 24, No. 9, September 1981.

[Teitelman and Masinter 81]

Teitelman, W., and Masinter, L., "The Interlisp Programming Environment", *Computer*, Vol. 14, No. 4, April 1981.