Report 82-34
Stanford -- KSL

Data Abstraction in GLISP.
Gordon S. Novak Jr,
Dec 1982

Heuristic Programming Project **December 1982**
Report No. HPP-82-34 (Revised April 1983)

Data Abstraction in GLISP

Gordon S. Novak Jr.
Heuristic Programming Project
Computer Science Department
Stanford University
Stanford, CA 94305

Reprinted from the Proceedings of SIGPLAN 1983 Symposium on Programming Languages Issues in Software Systems, Data Abstraction in GLISP by Gordon S. Novak, Jr.

To appear in **Proc. SIGPLAN'83 Symposium on Programming**

Language Issues in Software Systems

Data Abstraction in GLISP

Gordon S. Novak Jr.
Heuristic Programming Project
Computer Science Department
Stanford University
Stanford, CA 94305

Abstract

GLISP is a high-level language that is based on Lisp and is compiled into Lisp. It provides a versatile abstract-data-type facility with hierarchical inheritance of properties and object-centered programming. The object code produced by GLISP is optimized, so that it is about as efficient as handwritten Lisp. An integrated programming environment is provided, including editors for programs and data-type descriptions, interpretive programming features, and a display-based inspector/editor for data. GLISP is fully implemented.

1. Introduction

GLISP [8] [9] [10] is a high-level language, based on Lisp and including Lisp as a sublanguage, that is compiled into Lisp (which can be further compiled to machine language by the Lisp compiler). The GLISP system runs within an existing Lisp system and provides an integrated programming environment that includes automatic incremental compilation of GLISP programs, interactive execution and debugging, and display-based editing and inspection of data. GLISP programs are compiled relative to a knowledge base of object descriptions, a form of abstract data types [6] [13]. A primary goal of the use of abstract data types in GLISP is to allow program code to be written in terms of objects but independent of the actual implementations of objects; this allows the same code to be effective for objects that are implemented in different ways and thereby allows the accumulation of programming knowledge in the form of generic programs. A corollary goal is that the code generated for access to objects should be about as efficient as equivalent code written in the underlying language, Lisp. GLISP is fully implemented and is available from the author for several dialects of Lisp.[2]

GLISP program syntax includes Pascal-like control structures and infix arithmetic expressions. Substructures or properties of objects may be referenced in Pascal-like fashion as "<record>:<field>". Object-centered programming is supported for user-defined objects.

The GLISP language is easily extensible. Operator overloading for user-defined objects occurs automatically when arithmetic operators are defined as Messages for those objects. The compiler can compile optimized code for access to objects represented in user-specified *representation languages*, including database systems; GLISP has thus far been interfaced to the representation languages GIRL [7] and LOOPS [2]. GLISP has also been extended as a hardware description language for describing VLSI designs.

2. Object Descriptions

An *object description* describes the actual data structure occupied by an object; in addition, it describes *properties* (values that are computed rather than being stored as data), *adjectives* (used in predicate expressions to test features of the object), and *messages* to which the object can respond. An example of a GLISP object description is given in Figure 2-1. The name of the object type, CIRCLE, is followed by a description of the actual data structure occupied by the object: a Lisp list of the CENTER, which is of type VECTOR, and the RADIUS, which is a REAL number.

[1] This research was supported in part by NSF Grant SED-7912803 in the Joint National Science Foundation - National Institute of Education Program of Research on Cognitive Processes and the Structure of Knowledge in Science and Mathematics and in part by the Defense Advanced Research Projects Agency under Contract MDA-903-80-c-007.

[2] Including Interlisp, Maclisp, Franz Lisp, UCI Lisp, ELISP, and Portable Standard Lisp.

The remaining items describe properties, adjectives, and messages for this object type. As this example illustrates, the syntax of object descriptions makes it easy to define computed properties of objects.

```
(CIRCLE  (LIST (CENTER VECTOR) (RADIUS REAL))

  PROP   ((PI             (3.1415926))
          (AREA           (PI*RADIUS↑2))
          (DIAMETER       (RADIUS*2))
          (CIRCUMFERENCE  (PI*DIAMETER)) )

  ADJ    ((BIG            (AREA>100)))

  MSG    ((DRAW           DRAWCIRCLEFN)
          (GROW           (AREA←+100))        )
```

Figure 2-1: A GLISP object description.

Figure 2-2 shows the object description for a DCIRCLE, which is a different implementation of a circle object. A DCIRCLE has a different storage structure than a CIRCLE, and it stores the DIAMETER of the circle rather than the radius. However, since the RADIUS of a DCIRCLE is defined as a computed property, all of the properties of a CIRCLE that are defined in terms of RADIUS can be inherited by a DCIRCLE.

```
(DCIRCLE (ATOM (PROPLIST (DIAMETER REAL)
                         (CENTER VECTOR)))

  PROP   ((RADIUS   (DIAMETER/2)))

  SUPERS (CIRCLE)) .
```

Figure 2-2: A different implementation of circle objects.

3. Context and Type Inference

One of the design goals of GLISP is that program code should be independent of the implementations of the structures manipulated by the code to the greatest degree possible. Inclusion of redundant type declarations in program code would make the code dependent on the actual implementation of structures; instead, GLISP relies on type inference and its compile-time context mechanism to determine the types of objects.

The *context* is analogous to a symbol table, associating a set of named objects with their types. When a function is compiled, the context is initialized to contain the function's arguments and their types; the other types used within the function can in most cases be derived by type inference. During compilation, the type of each intermediate expression is computed and propagated together with the code that computes the expression. The type of any substructure retrieved from a larger structure is retrieved by the compiler from the structure description of the larger structure. Assignment of a value to an untyped variable causes that variable to be assigned the type of the value assigned to it. Type inference is performed automatically by the compiler for the common "system" functions of Lisp. The type of the value computed by a user function may be declared; usually, the compiler is able to infer the type of the result of a function that is compiled, and it saves a declaration for the result type. Through these mechanisms, the compiler can determine object types without requiring redundant type declarations within program code. Type checking is done during compilation at the point of use; that is, a feature of an object must be defined for the type of the object at the time the feature is referenced or a compilation error will result.

When properties, adjectives, and messages are compiled, the compilation takes place within a context containing the object whose properties are being compiled. Direct reference to the properties and substructures of the object is permitted within the code that defines properties; this is analogous to ***with ... do*** in Pascal. For example, the definition of AREA of a CIRCLE contains direct references to the stored value RADIUS and the property PI.

4. Recursive Compilation

The compilation of properties, adjectives, and messages in GLISP is recursive at compile time. That is, when a property is to be compiled, the definition of the property is taken as a new expression to be compiled and is compiled recursively in the context of the object whose property was referenced. This allows an abstract data type to define its properties in terms of properties that are implemented differently in its subclasses. It also allows expansion of code through multiple levels of implementation description.

5. Compilation of Messages

Object-centered programming, which treats data objects as active entities that communicate by sending messages, was introduced in SIMULA [1] and popularized by Smalltalk [4] [5]. In GLISP, the sending of a message to an object is specified in the form:

```
(SEND <object> <selector> <arguments>)
```

where the function name "SEND" specifies the sending of a message to <object>. The <selector> denotes the action to be performed by the message. When a message is executed at runtime, the <selector> is looked up for the type of the actual <object> to which the message is sent to get the name of the

function that executes the message. This function is then called with the <object> and the actual <arguments> as its arguments. In effect, a message is a function call in which the dynamic <object> type and the <selector> together determine the function name. Interpretive lookup of messages is computationally expensive--often more than an order of magnitude costlier than direct execution of the same code. However, the types of objects can usually be known at compile time [3]. When the response to a message can be uniquely determined at compile time, GLISP compiles in-line code for the response; otherwise, the message is interpreted at run time, as usual. By performing message lookup only once at compile time rather than repeatedly during execution, performance is dramatically improved while retaining the flexibility of object-centered programming.

Associated with each message selector[3] in an object description is a *response* specification that tells how to compile the corresponding message; the response contains code and a property list. There are three basic forms of response code. The first form of response code is simply a function name, and the code that is compiled is a call to that function. The second form is a function name and a flag to specify *open* compilation, in which the named function is "macro-expanded" in place with the actual argument values and types substituted for those of the function's formal arguments. The third form of response is GLISP code, which is recursively compiled in place of the message reference, in the context of the object whose property was referenced.

The last form of response code is a convenient and powerful way of defining computed properties of objects. The more usual way of defining such properties by means of small functions that compute them has several disadvantages. Function-call overhead (and message-lookup overhead in an object-centered system) is expensive for small functions. Since function names must usually be unique, long function names proliferate. The syntactic overhead of writing small functions and calling them discourages their use. In GLISP, function-call overhead is eliminated by expanding the response code in place; the resulting code is then subject to standard compiler optimizations (e.g., constant folding). Names of properties do not have to be unique because they are referenced relative to a particular object type. Response code is easy to write and easy to reference.

[3] And likewise with each property or adjective; property and adjective references are compiled as if they were message calls without any <arguments>.

6. Hierarchical Inheritance

Smalltalk and other object-centered languages organize objects into a hierarchy of classes and subclasses; this provides economy of representation by allowing features that apply to all members of a class to be described only once, at the class level. GLISP treats object descriptions as classes and allows properties, adjectives, and messages to be inherited from parent classes in the hierarchy. Response code that is inherited is compiled recursively *in the context of the original object*; this allows response code in a class to be written in terms of features that are represented differently in its offspring classes.

Smalltalk groups all classes into a single inheritance hierarchy (although experimental systems allowing multiple hierarchies have been implemented [3]). In GLISP, an object can be a member of multiple hierarchies, which provides more power for representing "orthogonal" properties of objects. Figure 6-1 shows how a single definition of a property, in this case the definition of DENSITY as MASS/VOLUME, can be effective for several kinds of objects. DENSITY is defined once for all PHYSICAL-OBJECTs, and this definition is inherited by subclasses of PHYSICAL-OBJECT. When DENSITY is inherited by a subclass of PHYSICAL-OBJECT, its definition as MASS/VOLUME is compiled recursively in the context of the original object. For PLANETs, the property MASS is

```
(PHYSICAL-OBJECT ANYTHING
  PROP ((DENSITY (MASS/VOLUME))))

(ORDINARY-OBJECT ANYTHING
  PROP ((MASS (WEIGHT / 9.88)))
  SUPERS (PHYSICAL-OBJECT))

(SPHERE ANYTHING
  PROP ((VOLUME ((4.0 / 3.0) * 3.1415926
                             * RADIUS ↑ 3))))

(PARALLELEPIPED ANYTHING
  PROP ((VOLUME (LENGTH*WIDTH*HEIGHT))))

(PLANET (LISTOBJECT (MASS REAL)
                    (RADIUS REAL))
  SUPERS (PHYSICAL-OBJECT SPHERE))

(BRICK (OBJECT (LENGTH REAL)
               (WIDTH REAL)
               (HEIGHT REAL)
               (WEIGHT REAL))
  SUPERS (ORDINARY-OBJECT PARALLELEPIPED))

(BOWLING-BALL (ATOMOBJECT (TYPE ATOM)
                          (WEIGHT REAL))
  PROP ((RADIUS ((IF TYPE='ADULT THEN 0.1
                                 ELSE 0.07))))
  SUPERS (ORDINARY-OBJECT SPHERE))
```

Figure 6-1: Inheritance from multiple hierarchies.

stored directly, while for ORDINARY-OBJECTs the WEIGHT of the object is stored and MASS is computed by dividing the weight by the value of gravity.[4] The VOLUME is also computed differently for each class of objects. BRICKs are defined to be PARALLELEPIPEDs and inherit the VOLUME computation from that class. Both BOWLING-BALLs and PLANETs inherit their VOLUME definition from SPHERE. RADIUS is stored for PLANETs; BOWLING-BALLs are defined so that there are two fixed RADIUS values, depending on whether the TYPE of the BOWLING-BALL is "ADULT" or "CHILD". Given this set of data-type descriptions, the DENSITY of a BOWLING-BALL B is compiled as follows:

```
(QUOTIENT
  (QUOTIENT (GETPROP B (QUOTE WEIGHT))
            9.88)
  (TIMES
    4.18879
    (EXPT (COND
            ((EQ (GETPROP B (QUOTE TYPE))
                 (QUOTE ADULT))
             .1)
            (T .07))
          3)))
```

This example illustrates how properties such as the definition of density or the volume of a sphere can be defined once at a high level and can then become effective for many classes of objects.

7. Data Abstraction in GLISP

Data abstraction delays binding between the abstract features of objects and the implementation of those features until compile time [12]. By doing so, data abstraction can make program code smaller, less prone to error, and easier to change. In this section, abstraction features provided by GLISP are discussed; these features make it possible for generic functions to be effective for a variety of instance data types.

7.1. Independence of Form of Stored Data

Compile-time property inheritance allows objects that are implemented in different ways to share the same property definitions. For example, vectors might have X and Y values of various possible types (e.g., integer or real) stored in various ways. A single abstract class VECTOR can define vector properties (e.g., how to add vectors) that can be inherited by the various kinds of vector implementations. This is illustrated in Figure 7-1. The class VECTOR defines a storage structure that is a list of two integers. The definition of "+" as a message selector causes this operator to be overloaded for objects of type VECTOR; "+" is implemented by the function VECTORPLUS, which is specified as being compiled *open*, that is, macro-expanded in-line. The class FVECTOR defines a different storage structure with elements whose types are STRING and BOOLEAN; since VECTOR is a superclass of FVECTOR, FVECTORs inherit the overloading of the "+" operator.

```
(VECTOR  (LIST (X INTEGER) (Y INTEGER))
   MSG   ((+  VECTORPLUS  OPEN T)))

(FVECTOR (CONS (Y BOOLEAN) (X STRING))
   SUPERS (VECTOR))

(VECTORPLUS (GLAMBDA (U,V:VECTOR)
   (A (TYPEOF U) WITH X = U:X + V:X ,
                      Y = U:Y + V:Y)))
```

Figure 7-1: Two kinds of vectors.

The TYPEOF operator that appears in the function VECTORPLUS returns the compile-time type of the expression that is its argument; this allows VECTORPLUS to produce a new object of the same type as its first argument. Given an expression "F+G", where F and G are VECTORs, the compiler will produce the code:

```
(LIST (IPLUS (CAR F) (CAR G))
      (IPLUS (CADR F) (CADR G)))
```

If F and G are FVECTORs, the compiler will produce the code:

```
(CONS (OR (CAR F) (CAR G))
      (CONCAT (CDR F) (CDR G)))
```

The "+" operators within VECTORPLUS are interpreted according to the types of the components of the actual vector type, so that the BOOLEAN components are ORed and the STRING components are concatenated.

A second form of data-structure independence is independence of the particular set of "equivalent" values that is stored. For example, an implementation of a circle object might equally well store either the radius or the diameter of the circle and compute the other. In GLISP, the syntax for referencing substructures and properties of objects is the same, so the distinction between stored and computed values is hidden. A circle implementation in which the diameter is stored can define RADIUS as a property and can then inherit all the properties, adjectives, and messages of the abstract type CIRCLE shown earlier. However, if code is to be independent of the set of properties that is actually stored, it must be possible not only to compute the desired properties but also to

[4] MKS measurement units are assumed.

"store into" computed properties as if they were actually stored properties. GLISP permits a computed property that is an algebraic function of a single stored value to be "assigned a value". For example, given an object C that is a CIRCLE, a program may include the code (C:AREA ↔ 100) ("The AREA of C is increased by 100"). The compiler will automatically compile code to compute the AREA from the stored RADIUS value, add 100 to it, compute the corresponding RADIUS for that AREA, and store the result back into the RADIUS datum. The resulting code for the CIRCLE object type shown earlier is:

```
(RPLACA (CDR C)
        (SQRT
          (QUOTIENT (PLUS (TIMES 3.1415926
                                 (EXPT (CADR C) 2))
                          100)
                    3.1415926)))
```

The compiler's ability to "invert" algebraic expressions that are constant functions of stored data allows a program to "store into" any member of a set of equivalent properties (e.g., RADIUS, DIAMETER, CIRCUMFERENCE, or AREA of a CIRCLE) without knowing which member of the set is actually stored. This provides a high degree of independence of object implementation.

7.2. Virtual Objects

In some cases, one would like to view an object as being an object of a different type, but without creating a separate data structure for the alternate view. For example, a name that is drawn on a display screen might be viewed as a REGION (a rectangle on the screen) for testing whether the display mouse is positioned on the name. Such a region is shown in the Figure 7-2. GLISP allows such a view to be specified as a *virtual object*, which is defined in terms of the original object. A virtual object definition for the name area illustrated above is:

```
(NAMEREGION  ((VIRTUAL REGION WITH
                   START = NAMEPOS ,
                   WIDTH = 8*(NCHARS NAME) ,
                   HEIGHT = 12)))
```

Given this definition, properties of the abstract data type REGION can be used for the virtual object NAMEREGION; in particular, the message that tests whether a region contains a given point can be inherited to test whether the name region contains the mouse position.

Virtual objects are implemented by creating a compiler-generated data type whose stored implementation is the original data type. The type of the view is made a superclass of the new type, and the features of the superclass are implemented as property definitions in the new type.

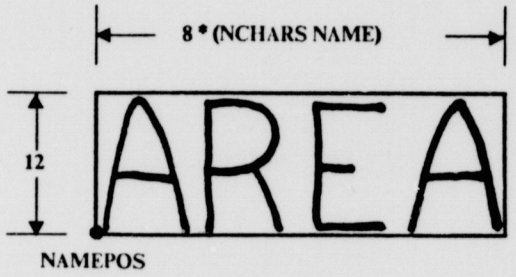

Figure 7-2: A virtual region.

7.3. Compilation of Generic Functions

GLISP can compile a generic function for a specified set of argument types, resulting in a closed Lisp function specialized for the particular argument types. For example, given a generic function for searching a binary tree and a view of a sorted array as a tree, GLISP produces a binary search of a sorted array.

Generic functions can also combine separately written algorithms into a composite algorithm. For example, a number of iterative programs can be viewed as being composed of the following components:

Iterator:	Collection → Element*
Filter:	Element → Boolean
Viewer:	Element → View
Collector:	
Initialize:	nil → Aggregate
Accumulate:	Aggregate × View → Aggregate
Report:	Aggregate → Result

The Iterator enumerates the elements of the collection in temporal order; the Filter selects the elements to be processed; the Viewer views each element in the desired way; and the Collector collects the views of the element into some aggregate. For example, finding the average monthly salary of the plumbers in a company would involve enumerating the employees of the company, selecting only the plumbers, viewing an employee record as "monthly salary", and collecting the monthly salary data for the average.

GLISP allows such an iterative program to be expressed as a single generic function; this function can then be instantiated for a given set of component functions to produce a single Lisp function that performs the desired task. Such an approach allows

programs to be constructed very quickly. The Iterator for a collection is determined by the type of the collection, and the element type is likewise determined. A library of standard Collectors (average, sum, maximum, etc.) is easily assembled; each Collector constrains the type of View that it can take as input. The only remaining items necessary to construct such an iterative program are the Filter and Viewer; these can easily be acquired by menu selection using knowledge of the element type (as is done in GEV, described below).

7.4. Symbolic Optimization and Conditional Compilation

The GLISP compiler performs symbolic optimization of the compiled code before it is returned. Operations on constants are performed at compile time; those may cause tests within conditional statements to have constant values, allowing some or all of the conditions to be eliminated. This symbolic optimization permits conditional compilation in a clean form. Certain variables may be declared to the compiler to have values that are considered to be compile-time constants; code involving these variables will then be optimized, causing unnecessary code to vanish. For large software packages, such as symbolic algebra packages, elimination of unwanted options can produce large savings in code size and execution time without changing the original source code. The language used to specify conditional compilation is the same as the language used for run-time code; tests of conditions can be made at compile time or at run time as desired.

Symbolic optimization provides additional efficiency for compiled object-centered programming. Messages to objects in ordinary object-centered languages are *referentially opaque*, that is, it is not possible to "see inside" the messages to see how they work. This opacity inhibits optimization, since most optimizations are of the form "If both operations A and B are to be performed, there is a way to do A and B together that is cheaper than doing each separately." If the insides of A and B cannot be seen, the opportunity for optimization cannot be recognized. For example, suppose it is desired to print the names of the female "A" students in a class. The most efficient way to do this may be to make one pass over the set of students, selecting those who are both female and "A" students. However, in an object-centered programming system in which the female students and "A" students were found by sending messages, it would not be possible to perform this optimization because it would not be known how the two sets were computed.

GLISP allows simple filters such as those that select females and "A" students to be written easily in a form that compiles open:

```
PROP ((WOMEN
        ((THOSE STUDENTS WITH SEX='FEMALE)))
       (A-STUDENTS
        ((THOSE STUDENTS WITH AVERAGE>90))))
```

The desired loop can be written using the "•" operator, which is interpreted as intersection for sets:

```
(FOR S IN CLASS:WOMEN • CLASS:A-STUDENTS
                        DO (PRINT S:NAME))
```

The expansion of the property code makes possible loop optimizations that result in a single loop over the students without actual construction of intermediate sets. The transformations used for this example are:

```
(subset S P) ∩ (subset S Q)
  →  (subset S P∧Q)

(for each (subset S P) do F)
  →  (for each S do (if P then F))
```

These and other similar transformations allow the compiler to produce efficient code for loops that are elegantly stated at the source code level.

7.5. Lisp Dialect Independence

The implementors of different Lisp systems have unfortunately introduced many variations in the names, syntax, and semantics of the basic system functions of Lisp. GLISP runs within a number of different Lisp systems and must therefore cope with these differences; it is also desirable that code written in GLISP be easily transportable to GLISP systems running within different Lisp systems.

The primary version of the GLISP compiler is written in Interlisp-D and runs on a Xerox 1100 "Dolphin" Lisp machine. This version is translated into the other Lisp dialects by a source-to-source Lisp translator; a few pages of compatibility functions written for each dialect then allow the compiler to run in the foreign dialects. However, the compiler must not only run in foreign dialects but also generate code for them. In order to do this, the appropriate Lisp translator is translated (by itself!) for the target dialect and included as part of the compiler. The GLISP compiler running on the target machine generates Interlisp but then immediately translates it for the machine on which it is running.

The GLISP compiler contains knowledge about the Lisp system within which it runs; this knowledge simplifies programming and aids program transportability. The compiler is able to infer the types of the results returned by commonly used Lisp system functions; this relieves the user of the burden of writing type declarations for these values. Lisp data types are described by

GLISP object descriptions, which allows features of these types (e.g., the print name of a Lisp Atom) to be referenced directly in a dialect-independent manner. Such descriptions also facilitate inspection of basic Lisp data for debugging, since alternative views of data (e.g., viewing a string as a sequence of ASCII codes) are built-in and can be directly seen using the GEV data inspector.

The data-abstraction facilities of GLISP encourage the user to write abstract-data-type packages that mediate the interaction between user programs and idiosyncratic system features. Global data types can be defined that have no storage realization but that translate property references (e.g., "MOUSE:POSITION") and messages into the appropriate calls to the operating system. The GEV data inspector is written using *window* and *menu* abstract data types that allow it to work with a variety of display media.

8. Interactive Features

GLISP provides an interactive programming environment that complements the Lisp environment and provides support for abstract data types. Interactive versions of GLISP statements are provided for creating objects, sending messages to them, and retrieving their properties and substructures. Interfaces to the Lisp editor are provided for editing GLISP functions and abstract-data-type descriptions; the GEV[5] program is provided for editing GLISP data.

GEV [11] is an interactive display-based program that allows the user to inspect data, "zoom in" on features of interest, edit objects, display computed properties, send messages to objects, and interactively write programs. GEV is initiated by giving it a pointer to an object and the type of the object. Using the data-type description of the object, GEV interprets the data and displays them within a window, as shown in Figure 8-1. Data are displayed in the window in three sections: the edit path (that shows the path by which the currently displayed object was reached from the original object), the actual data contained in the object, and computed properties that have been requested or that are specified in the object description to be displayed automatically. Often, the full value of an item cannot be displayed in the limited space available. In such cases, the SHORTVALUE property of the object is computed and displayed; a tilde (~) before the value indicates a SHORTVALUE display. The SHORTVALUE provides a meaningful "view from afar" for large data objects; for example, the SHORTVALUE of an employee record could be defined to be the employee's name.

[5] For GLISP Edit Value.

Most interaction with GEV is done with the display mouse (or short mnemonic commands on ordinary CRT terminals). If the name of a displayed item is selected, the type of that item is printed. If a value is selected, GEV "zooms in" on that value, displaying it in greater detail according to its data-type description. The command menu below the display window is used to specify additional commands to GEV. The EDIT command calls a type-specific editor, or the Lisp editor, on the current object. The PROP, ADJ, and MSG commands cause a menu of the available properties, adjectives, or messages for the type of the current object to be displayed; the property selected from this menu is computed for the current object and added to the display. A GLISP object description can specify that certain properties be displayed automatically whenever an object of that type is displayed. For example, the type RADIANS (whose stored implementation is simply a real number) can automatically display the equivalent angle in degrees.

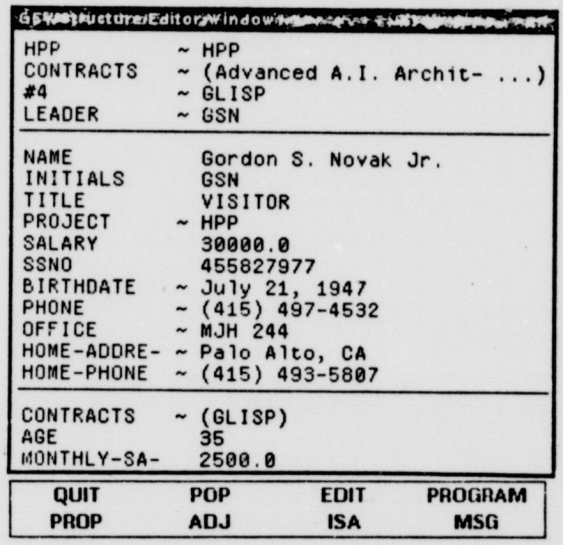

Figure 8-1: GEV data inspector display.

The PROGRAM command allows the user to create looping programs that operate on the currently displayed object; these programs are specified interactively using menu selection. This process and its result are illustrated in Figure 8-2.

After the PROGRAM command is selected, a menu is presented for selection of the operation to be performed. The next menu selects the set over which the program will operate; it contains all substructures of the current object that are represented as lists. Next, menus of all appropriate items of computed or stored data visible from the current item (initially the "loop index" item from the selected set) are presented until a terminal item type (e.g., a number) is reached. GEV constructs from the program specifications a GLISP program to perform the specified computation, compiles it, and runs it on the current object;

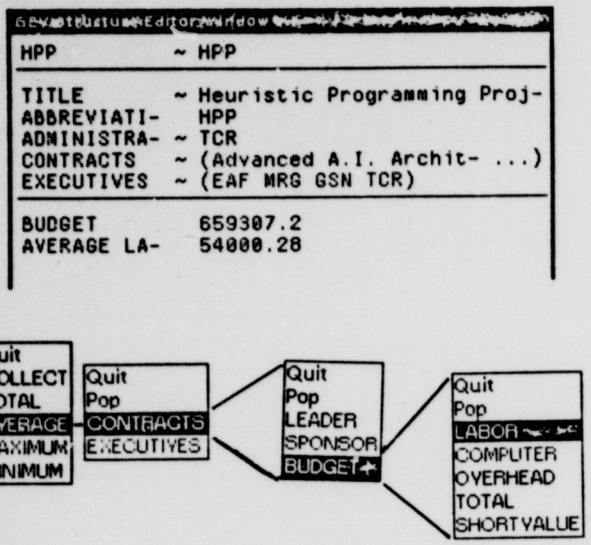

Figure 8-2: GEV menu programming.

typically, this process takes less than one second. The results of the program are printed and added to the display.

The user of GEV does not need to know the actual implementations of the objects that are being examined. This makes GEV useful as an interactive database query language that is driven by the data-type descriptions of the objects being examined. Since GLISP object descriptions are themselves GLISP objects, they can be inspected with GEV.

GEV is written in GLISP. The GEV code is compiled relative to a package of *window* and *menu* abstract data types; these data types mediate the interactions between GEV and the display that is used. The window and menu abstract data types enable the same GEV code to work with Xerox "Dolphin" Lisp machines, vector graphics displays, or ordinary CRT terminals.

9. Discussion

We have discussed the methods by which the GLISP system provides several novel programming language capabilities:

1. An extended form of abstract data type, including properties, adjectives, and messages, is provided. Data types may be organized in multiple hierarchies with inheritance of properties.

2. Properties of objects are compiled recursively relative to their actual compile-time data types.

3. Optimized compilation is performed for object-centered programming by performing inheritance at compile time; this markedly improves performance of object-centered programs.

4. Generic programs and expressions can be compiled into closed functions that are specialized for particular data types.

5. Interactive programming and display-based editing are provided for abstract data types.

10. Implementation Status

GLISP and GEV, including all features described in this paper, are running and are being used for application programs at several sites. Versions of the compiler for the major dialects of Lisp are available. The GLISP User's Manual [9] is available over the ARPANET as [SUMEX-AIM]<GLISP>GLUSER.LPT ; Chapter 8 of the manual tells how to obtain the code.[6]

11. Bibliography

1. Birtwistle, Dahl, Myhrhaug, and Nygaard. *SIMULA Begin.* Auerbach, Philadelphia, PA, 1973.

2. Bobrow, D. G., and Stefik, M. The LOOPS Manual. Tech. Rept. KB-VLSI-81-13, Xerox Palo Alto Research Center, 1981.

3. Borning, A., and Ingalls, D. A Type Declaration and Inference System for Smalltalk. Proc. 9th Conf. on Principles of Programming Languages, ACM, 1982.

4. Goldberg, A., et al. Special Issue on Smalltalk. In *BYTE Magazine*, , August 1981.

5. Ingalls, D. The Smalltalk-76 Programming System: Design and Implementation. Proc. 5th ACM Symposium on Principles of Programming Languages, 1978.

6. Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. "Abstraction Mechanisms in CLU." *CACM 20*, 8 (Aug. 1977).

7. Novak, G. S. GIRL and GLISP: An Efficient Representation Language. Tech. Rept. TR-172, Computer Science Dept., Univ. of Texas at Austin, March, 1981.

8. Novak, G. S. GLISP: An Efficient, English-Like Programming Language. Proc. 3rd Annual Conf. of Cognitive Science Society, Univ. California, Berkeley, 1981.

9. Novak, G. S. GLISP User's Manual. Tech. Rept. STAN-CS-82-895, Computer Science Dept., Stanford Univ., Jan., 1982.

10. Novak, G. S. GLISP: A High-Level Language for A.I. Programming. Proc. 2nd National Conf. on Artificial Intelligence, Carnegie-Mellon University, 1982.

11. Novak, G. S. The GEV Display Inspector/Editor. Tech. Rept. HPP-82-32, Heuristic Programming Project, Computer Science Dept., Stanford Univ., Nov., 1982.

12. Wiederhold, G. Binding in Information Processing. Tech. Rept. STAN-CS-81-851, Computer Science Dept., Stanford Univ., May, 1981.

13. Wulf, W. A., London, R., and Shaw, M. "An Introduction to the Construction and Verification of Alphard Programs." *IEEE Transactions on Software Engineering SE-2*, 4 (Dec. 1976).

[6] The login "ANONYMOUS GUEST" may be used to FTP files.