AI Magazine Volume 4 Number 3 (1983) (© AAAI)

GLISP: A Lisp-based Programming System with Data Abstraction

Gordon S. Novak Jr.

Heuristic Programming Project
Computer Science Department
Stanford University
Stanford, CA 94305

Abstract

GLISP is a high-level language that is compiled into LISP It provides a versatile abstract-data-type facility with hierarchical inheritance of properties and object-centered programming GLISP programs are shorter and more readable than equivalent LISP programs The object code produced by GLISP is optimized, making it about as efficient as handwritten LISP An integrated programming environment is provided, including automatic incremental compilation, interpretive programming features, and an intelligent display-based inspector/editor for data and data-type descriptions GLISP code is relatively portable; the compiler and the data inspector are implemented for most major dialects of LISP and are available free or at nominal cost

GLISP (NOVAK 1982, 1983A, 1983B) 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. Use of GLISP makes writing, debugging, and modifying programs significantly easier; at the same time, the code produced by the compiler is optimized so that its execution efficiency is comparable to that of handwritten LISP This article describes features of GLISP and illustrates them with examples Most of the syntax of GLISP is similar to LISP syntax or PASCAL syntax, so explicit treatment of GLISP syntax will be brief.

GLISP programs are compiled relative to a knowledge base of object descriptions, a form of abstract data types (Liskov *et al.* 1977; Wulf, London, & Shaw 1976). A primary goal of the use of abstract data types in GLISP is to *make programming easier* The implementations of objects are described in a single place; the compiler uses the object descriptions to convert GLISP code written in terms of user objects into efficient LISP code written in terms of the implementations of the objects in LISP This allows the implementations of objects to be changed without changing the code; it also 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 Figure 1 illustrates the combination of information from these three sources; the recursive use of abstract data types and generic programs in the compilation process provides multiplicative power for describing programs

Overall, GLISP program syntax is like that of LISP

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

Author's present address: Computer Science Department, University of Texas at Austin, Austin, TX: 78712

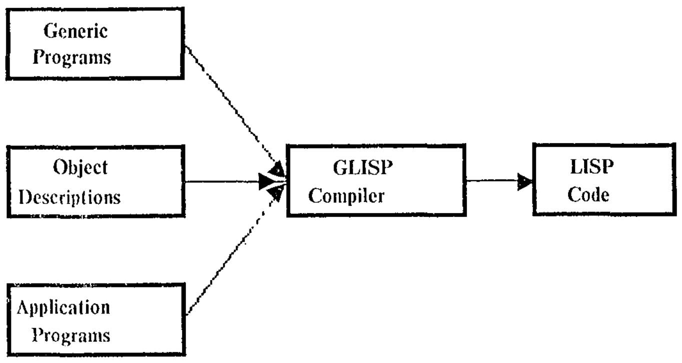


Figure 1 GLISP compilation

GLISP contains ordinary LISP as a sublanguage; LISP code can be mixed with GLISP code, so that no capabilities of the underlying LISP system are lost. GLISP provides PASCAL-like reference to substructures and properties, infix arithmetic expressions, and PASCAL-like control statements Object-centered programming is built in; optimized compilation allows object-centered programs to run efficiently.

GLISP is easily extensible for new object representations. Operator overloading for user-defined objects occurs automatically when arithmetic operators are defined as message selectors for those objects The compiler can compile optimized code for access to objects represented in user-specified representation languages. GLISP has also been extended as a hardware description language for describing VLSI designs.

Object Descriptions

In GLISP, programs are separated into a knowledge base of object descriptions and application programs that are written in terms of objects The compiler uses the object descriptions to guide the translation of GLISP programs, written in terms of objects, into LISP programs, written in terms of the implementations of objects in LISP. The use of object descriptions has several advantages:

1. Object descriptions provide multiplicative power for describing programs The code for a property of an object is stated once in the object description but can then be invoked many times in programs that reference the property, either directly or through inheritance [1]
2. The implementations of objects can be changed without changing code that references the objects.
3 Generic programs can be used for conceptually similar objects that are implemented in different ways, facilitating the accumulation of programming knowledge in the form of collections of abstract object descriptions and generic programs
4 Existing LISP data structures can be described, so that GLISP and its associated programs (such as the GEV data inspector) can be used with existing programs that are not written in GLISP
5 Object descriptions provide valuable program documentation.

[1]GLISP properties have more generality than macros for describing computed properties

The payoff from using abstract data types in GLISP increases as systems become larger. GLISP programs are typically shorter than equivalent LISP programs by a factor of two to three.

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 shown in Figure 2 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 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 The language for describing the storage structures of objects allows most of the common data structures of LISP to be described.

Figure 3 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

```
(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 A GLISP object description

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

  PROP   ((RADIUS (DIAMETER/2)))

  SUPERS(CIRCLE))
```

Figure 3   A different implementation of circle objects

(a LISP atom with data stored on its property list), 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 CIRCLEs can be inherited by DCIRCLEs simply by naming CIRCLE as one of the SUPERS (superclasses) of DCIRCLE.

Compilation of Property References

Compilation of a GLISP function occurs automatically the first time it is called; recompilation occurs automatically if the function or an object description on which it depends is modified. Thus, it appears to the user that a "GLISP interpreter" exists There are also facilities for double compilation (from GLISP to LISP to compiled LISP) using the existing LISP compiler. When a GLISP function is compiled, the original GLISP definition is saved on the function's property list, and the function is redefined as a LISP EXPR

GLISP functions are defined using a defining form similar to the one in the underlying LISP dialect; the examples in this article are shown in the form appropriate for INTERLISP (Teitelman 1978). Types of function arguments may be declared, as in PASCAL, using the syntax

"<variable> <type>".

Within program code, substructures or properties of objects may be referenced using the syntax

"<variable> <property>".

A simple function named CR, which retrieves the RADIUS of a CIRCLE, could be written as follows:

```
(DEFINEQ
 (CR (GLAMBDA (C CIRCLE)
       C RADIUS)) )
```

Such a function can be explicitly compiled by calling the function GLCP (GLISP Compile and Print), as shown below:

```
(GLCP 'CR)
   GLRESULTTYPE  REAL
   (LAMBDA (C) (CADR C))
```

GLCP prints the type of the result returned by the function (which is inferred by the compiler) and the LISP code produced.

Properties of an object are referenced in the same way as stored data. This facilitates hiding the internal structures of objects, so that programs do not depend on which property values are stored and which property values are computed. For example, a CIRCLE has RADIUS stored, while a DCIRCLE has DIAMETER stored; this distinction is transparent to programs using these objects. As an example of property reference, the area of a CIRCLE C can be referenced as:

```
C AREA
```

which is compiled as:

```
(TIMES 3 1415926
       (EXPT (CADR C) 2))
```

If the absence of a distinction in program code between data that are stored and data that are computed is to be maintained, it must be possible to "store into" computed data The compiler is able to "invert" arithmetic expressions involving constants and a single stored value For example, the area of a circle can be increased by 100 using the following code (the operator "←+" means "is increased by"):

```
(C AREA ←+ 100)
```

which compiles into:

```
(RPLACA (CDR C)
      (SQRT
         (QUOTIENT (PLUS (TIMES 3 1415926
                                (EXPT (CADR C) 2))
                               100)
                         3 1415926)))
```

As this example illustrates, the features of GLISP form an integrated whole and can be combined. In this case, the GLISP source code is easier to understand than the equivalent LISP code, and there is a high ratio of output code to input code.

GLISP Statements

GLISP provides several kinds of statements that are translated into equivalent code in LISP; each is identified by a key word at the front of a list containing the code for the statement. Many of these statements are similar to those provided by PASCAL:

```
If ..then .else
While ...Do
Repeat . .Until
Case
```

These control statements provide a compact and well-structured way of stating some commonly used control constructs and also provide a degree of LISP-dialect independence As an example, a simple square-root function can be written as follows:

```
(SQRT (GLAMBDA (X REAL)
  (PROG (S)
    (S ← X)
    (IF X < 0 THEN (ERROR)
       ELSE (WHILE (ABS S*S - X) > 0 00001
                  DO (S ← (S+X/S) * 0 5)))
    (RETURN S))))
```

Two special forms, THE and THOSE, are provided to select a single element or subset of elements that satisfy a given condition from a set of similar elements:

```
(THE COWBOY WITH HORSE='TRIGGER)
(THOSE EMPLOYEES WITH SENIORITY > 3)
```

The A function is provided to create data in a representation-independent manner. Given a set of name/value pairs, the A function creates a new data structure having the specified values:

```
(A CIRCLE WITH RADIUS = R)
```

Given the earlier object description for CIRCLE, this will compile as:

```
(LIST (APPEND '(0 0)) R)
```

The A function works interpretively as well as within compiled code

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 inferred 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. Using 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

Compilation of Messages

Object-centered programming, which treats data objects as active entities that communicate by sending messages, was introduced in SIMULA (Birtwistle *et al* 1973) and popularized by SMALLTALK (Goldberg *et al.* 1981, Ingalls 1978). Object-centered programming has recently been implemented for several LISP dialects as well (Bobrow & Stefik 1981, Cannon 1981) 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 (Borning & Ingalls 1982) 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[2] in an object description is a *response* specification that tells how to compile the corresponding message; the response consists of 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 compile-time 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 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

Property Inheritance

Object-centered languages organize objects into a hierarchy of classes and instances; 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 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 original object whose property was referenced. This allows an abstract data type to define its properties in terms of other properties that are implemented differently in its subclasses. It also allows expansion of code through multiple levels of implementation description.

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 4 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 The class VOFV defines a vector whose components are VECTORs Since FVECTOR and VOFV have VECTOR as a superclass, these classes inherit the overloading of the "+" operator defined in class VECTOR

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:

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

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

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

(VOFV (LIST (X VECTOR) (Y VECTOR))
     SUPERS (VECTOR))

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

Figure 4 Three kinds of vectors and a generic function

```
(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)))
```

Finally, if F and G are VOFVs, the compiler will produce the code:

```
(LIST [PROG ((U (CAR F))
             (V (CAR G)))
            (RETURN (LIST (IPLUS (CAR U)
                                 (CAR V))
                          (IPLUS (CADR U)
                                 (CADR V]
      [PROG ((U (CADR F))
             (V (CADR G)))
            (RETURN (LIST (IPLUS (CAR U)
                                 (CAR V))
                          (IPLUS (CADR U)
                                 (CADR V])
```

The "+" operators within VECTORPLUS are interpreted according to the types of the components of the actual vector type with respect to which VECTORPLUS is compiled, so that the BOOLEAN components are ORed, STRING components are concatenated, and VECTOR components invoke VECTORPLUS again

```
(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 5 Inheritance from multiple hierarchies

Multiple Inheritance. In GLISP, an object can be a member of multiple hierarchies; this aids representation of "orthogonal" properties of objects Figure 5 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 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 (assuming MKS measurements) 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 BOWLING-BALL's TYPE 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 'WEIGHT)
            9 88)
  (TIMES
    4 18879
    (EXPT (COND
            ((EQ (GETPROP B 'TYPE)
                 '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.

Virtual Objects In some cases, one would like to view an object as being an object of a different type, but without materializing 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 the purpose of testing whether the display mouse is positioned on the name. In this example, we assume that the position of the lower left-hand corner of the region is stored explicitly, that the region has a constant height of 12 pixels, and that the width of the region is 8 pixels times the number of characters in the name. Such a region is shown in Figure 6. GLISP allows such a view to be specified as a *virtual object*, which is defined in terms of the original object. A property definition for the name area illustrated above as a virtual object 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

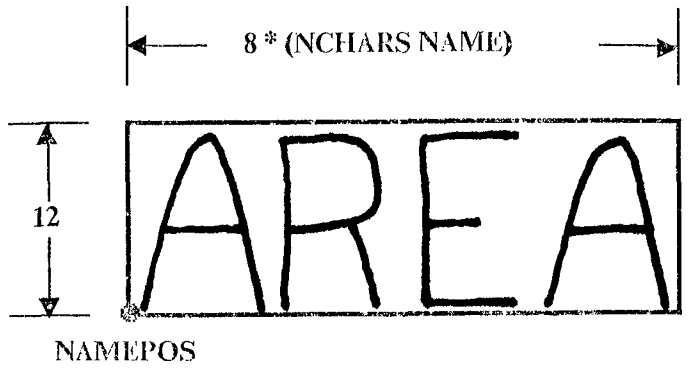


Figure 6. A Virtual Region.

implemented as property definitions in the new type.

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 those 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 made up 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 might 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 could easily be acquired by menu selection using knowledge of the element type (as is done in GEV, described below).

Symbolic Optimization and Conditional Compilation

The GLISP compiler performs symbolic optimization of the compiled code Operations on constants are performed at compile time; these may cause tests within conditional statements to have constant values, allowing some or all of the conditions to be eliminated This is illustrated in compilation of the following example:

```
(SQUASH
  (GLAMBDA NIL
    (IF 1>3 THEN 'AMAZING
      ELSEIF (SQRT 7 2) < 2
              THEN 'INCREDIBLE
      ELSEIF 2 + 2 = 4 THEN 'OKAY
      ELSE 'JEEZ)))
```

which is compiled as:

```
(LAMBDA NIL 'OKAY)
```

Symbolic optimization permits conditional compilation in a clean form Certain variables can 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 as properties in a form that compiles open:

```
(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:

$$(\text{subset } S\ P) \bigcap (\text{subset } S\ Q) \rightarrow (\text{subset } S\ P \bigwedge Q)$$

$$(\text{for each } (\text{subset } S\ P) \text{ do } F) \rightarrow (\text{for each } S \text{ do } (\text{if } P \text{ then } F))$$

These and other transformations allow the compiler to produce efficient code for loops that are elegantly stated at the source code level The above example is compiled as:

```
(MAPC (GETPROP CLASS 'STUDENTS)
    (FUNCTION (LAMBDA (S)
        (AND (EQ (GETPROP S 'SEX)
                  'FEMALE)
              (GEQ (STUDENT-AVERAGE S)
                    90)
              (PRINT (GETPROP S 'NAME))))))
```

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 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. 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. For example, the programmer need not remember which of the six comparison operators are implemented for strings in the LISP dialect in question or what the names of the functions are; GLISP translates expressions involving any of the operators into appropriate forms using the available functions. 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. Basic LISP data types are themselves described by GLISP object descriptions, allowing features of these types (e.g., the LENGTH of a STRING) 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 or viewing an integer in octal) 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 in different LISP environments.

Programming Environment

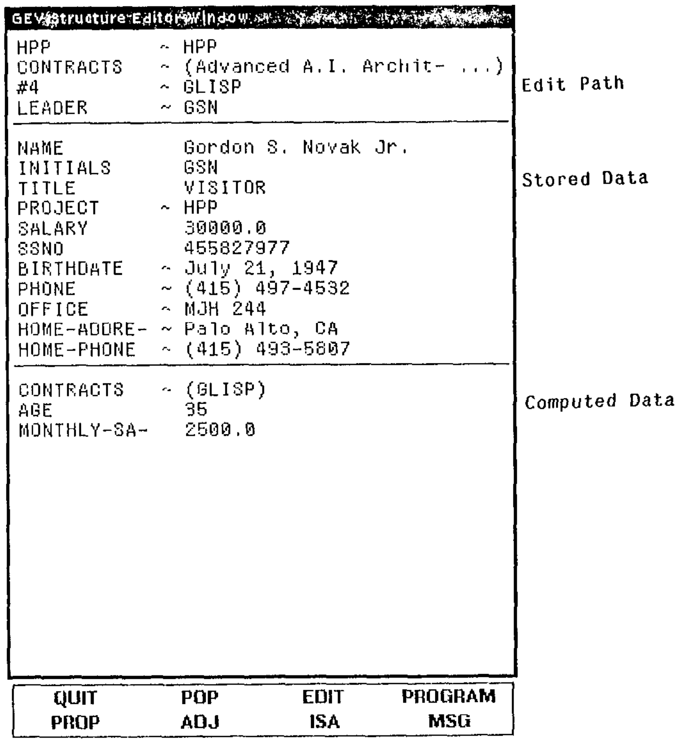
GLISP provides an interactive programming environment that complements the LISP environment and provides support for abstract data types The compiler performs error checking and provides explanatory error messages; many common errors are caught by the compiler, simplifying debugging Facilities are provided to compile files of GLISP code into files of LISP code in the underlying LISP dialect. Interactive versions of GLISP statements are provided for creating objects, sending messages to them, and retrieving their properties and substructures. The interpreted message features are available for LISP data, as well as for object-centered data, when the class of the LISP data is specified. Interpreted messages may reference properties, adjectives, and substructures of an object as well as messages When a property of an object is first referenced by an interpreted message, GLISP compiles code to perform the requested access as a LAMBDA form and caches it in the class of the object Interfaces to the LISP editor are provided for editing GLISP functions and abstract-data-type descriptions; the GEV program is provided for inspecting and editing GLISP data.

GEV[3] (Novak 1983) is an interactive display-based program that allows the user to inspect data based on its data-type description, "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 it within a window, as shown in Figure 7.

Data are displayed in the window in three sections: the edit path (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

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 the LISP editor, or a type-specific 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 should be displayed automatically whenever an object of that type

[3]For <u>G</u>LISP <u>E</u>dit <u>V</u>alue

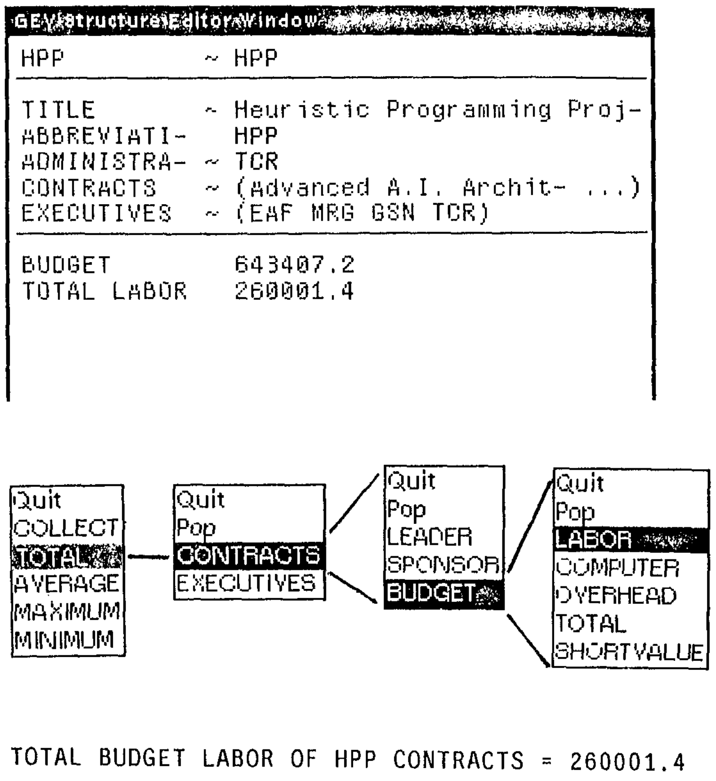


```
HPP            ~ HPP
CONTRACTS      ~ (Advanced A.I. Archit- ...)
#4             ~ GLISP
LEADER         ~ GSN

NAME             Gordon S. Novak Jr.
INITIALS         GSN
TITLE            VISITOR
PROJECT        ~ HPP
SALARY           30000.0
SSNO             455827977
BIRTHDATE      ~ July 21, 1947
PHONE          ~ (415) 497-4532
OFFICE         ~ MJH 244
HOME-ADDRE-    ~ Palo Alto, CA
HOME-PHONE     ~ (415) 493-5807

CONTRACTS      ~ (GLISP)
AGE              35
MONTHLY-SA-      2500.0
```

Figure 7. GEV data inspector display



```
HPP            ~ HPP

TITLE          ~ Heuristic Programming Proj-
ABBREVIATI-      HPP
ADMINISTRA-    ~ TCR
CONTRACTS      ~ (Advanced A.I. Archit- ...)
EXECUTIVES     ~ (EAF MRG GSN TCR)

BUDGET           643407.2
TOTAL LABOR      260001.4
```

```
TOTAL BUDGET LABOR OF HPP CONTRACTS = 260001.4
```

Figure 8 GEV menu programming

is displayed. For example, the type RADIANS (whose stored implementation is simply a real number) can automatically display the equivalent angle in DEGREES

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. 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; 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 LISP machines, vector graphics displays, or ordinary CRT terminals.

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

GLISP and GEV, including all features described in this paper, are running and are being used for application programs at several sites

How to Obtain GLISP

Versions of the compiler are available for INTERLISP (DEC 2060, Xerox LISP machines), MACLISP, FRANZ LISP, UCI LISP, ELISP, and Portable Standard LISP (DEC 2060, VAX, HP 9836); a version for ZETALISP is planned As of this writing, GEV is available for INTERLISP, MACLISP, and Portable Standard LISP; versions for the other dialects are planned.

GLISP and its documentation are available free to members of the ARPANET community; the files are contained on the host SUMEX-AIM in the directory <GLISP> [4] The GLISP User's Manual (Novak 1983) is contained in the file GLUSER LPT (line printer form) or GLUSER MSS (SCRIBE source form); printed copies may be ordered for $5.00 at the address given below Chapter 8 of the manual tells how to obtain the code for GLISP and GEV

GLISP is available to non-ARPANET sites for a nominal taping charge Address correspondence to the author at Computer Science Department, University of Texas at Austin, Austin, TX 78712 Phone (512)471-4353 The author's ARPANET address is CS NOVAK@UTEXAS

References

Birtwistle, Dahl, Myhrhaug, and Nygaard. (1973) *SIMULA Begin* Philadelphia, PA: Auerbach

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

Borning, A , and Ingalls, D. (1982) *A Type Declaration and Inference System for Smalltalk Proc 9th Conf. on Principles of Programming Languages,* Association for Computing Machinery, New York

Cannon, H. I (1981) *Flavors A Non-Hierarchical Approach to Object-Oriented Programming* Working Paper, A I Lab, Massachusetts Institute of Technology

Goldberg, A., et al (1981) Special Issue on Smalltalk BYTE Magazine.

Ingalls, D (1978) *The Smalltalk-76 Programming System Design and Implementation* Association for Computing Machinery, *5th ACM Symposium on Principles of Programming Languages,* New York

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

Novak, G S (1983) GLISP Reference Manual Tech Rept. HPP-82-1, Heuristic Programming Project, Computer Science Dept , Stanford University

Novak, G. S (1982) *GLISP A High-Level Language for A I Programming* American Association for Artificial Intelligence, *Proc*

(Continued on page 53)

[4]The login "ANONYMOUS GUEST" may be used at SUMEX to FTP files

NASA Ames Research Center
Moffett Field, CA 94035

EXPERT SYSTEMS RESEARCHER

Ames Research Center (35 miles south of San Francisco) is seeking a senior investigator for the development of generic software tools and computer architectures applicable to image-based expert systems. The research areas include knowledge representation, system control (rule interpreter), symbolic representation languages, and information extraction (pattern recognition). Poorly understood issues include the impact of the space environment on software and hardware architectures including reliability (fault-tolerance) for image-based expert systems; portability of languages between von Neumann and non-von Neumann systems and ease of user interface; and ability of the crew to interact, understand, and diagnose failures for spaceborne expert systems. Responsibilities will include (1) directing and participating in basic artificial intelligence (AI) research for the development of new hardware and software technologies for image-based expert systems within the constraints of the space environment; (2) managing pilot demonstrations of new expert system technology applications; (3) participating in and monitoring the cooperative AI research grants; and (4) establishing and maintaining peer interaction with the science and technical communities.

Specified qualifications include: (1) in-depth knowledge of expert system development, and computer system architectures (double-weighted); (2) ability to plan, conduct, direct, and report on expert system research (double-weighted); (3) knowledge of cognitive psychology and/or linguistics; and (4) ability to direct and work as a member of an applications team. U.S. citizenship and Ph.D. or equivalent in electrical/electronic engineering and/or computer science are required. Permanent position in Federal Service. Salary ranges between $41,277 and $63,115, commensurate with experience/education. For further details regarding requirements and application procedures, write 28-83 at the above address or phone (415)965-5084. Formal applications must be filed by September 30, 1983. An Equal Opportunity Employer.

gramming," and had the Athenians personal computers with LISP-controlled graphics they might well have sentenced him anyway There is great contrast between the pleasures of programming and the tedium of analysis, between the challenge of the mysterious bug and the death of a beautiful hypothesis at the hands of an ugly fact

Rational psychology also offers improved communications The frequency of reinvention of ideas in artificial intelligence is legendary. While it is unreasonable to expect (and undesirable to attempt) to make reinventions rare occurrences, artificial intelligence clearly seems extravagant It is not alone in this. There is the old joke in computer science about the result that was lost because it was only published four times But even the magnitude of the problem is unclear Not only do researchers lack deep understanding of their own proposals, but they usually cannot understand those of others either. This incomprehension is not due to stupidity, but to the vague, metaphorical terms on which the field relies in the absence of precise, formal vocabularies for presenting theories. In mathematics, physics, and many other sciences, papers, if properly written, define concepts in terms of the accepted vocabulary, state claims or discoveries, and then leave comprehension up to the intelligence and motivation of the reader. In artificial intelligence, even conscientiously written papers can be unintelligible no matter how capable and motivated the reader, for much of the accepted vocabulary is about as precise as that of poetry, and about as substantive as that of advertising copy If we had adequate mathematical concepts, if we had conventions for clear, exact statements of problems — two large ifs — then we could hope for reduced reinvention, more rapid communication, comparison, and reproduction of ideas, and a true chance to build on the work of others: things all taken for granted in other fields

Conclusion

A mathematical, analytical enterprise like rational psychology is not for everyone Indeed, rational psychology feeds on intuitions gained only through experience, so it makes no more sense for everyone to abandon the usual efforts of artificial intelligence and cognitive science than for all physicists to forsake experiment and experience in favor of rational mechanics On the other hand, rational psychology need not be purely parasitic, for its pursuit may someday advance the construction of thinking machines, much as aerodynamics has advanced the construction of flying machines But these practical benefits cannot be realized without effort At least some people must stray from the usual investigations of artificial intelligence and cognitive science, and their work must be judged by the aims and methods of rational psychology instead of by those of artificial intelligence and cognitive science. I would not bother to invent the label "rational psychology" for these aims and methods, except that they *are* somewhat different from the usual ones of artificial intelligence and cognitive science, and more easily understood and encouraged when explicitly recognized For example, questions about implementation status or experimental verification of theories are legitimate questions for artificial intelligence and cognitive science, but not for rational psychology, even though the same theories may be under discussion As with chemistry and cookery, mere recipes for constructing machines and men do not guarantee understanding the product And for rational psychology, the main question is whether the theories have been adequately understood.

References

Courant, R and Robbins, H , (1944) *What is mathematics? An elementary approach to ideas and methods*, London: Oxford University Press

Doyle, J. (1982a) The foundations of psychology, Pittsburgh: Department of Computer Science, Carnegie-Mellon University

Doyle, J (1982b) Some theories of reasoned assumptions: an essay in rational psychology, Pittsburgh: Department of Computer Science, Carnegie-Mellon University

Doyle, J (1983) What is rational psychology? Toward a modern mental philosophy, Pittsburgh: Department of Computer Science, Carnegie-Mellon University

Hartmanis, J (1981) Remarks in "Quo Vadimus: computer science in a decade," in J F Traub (ed), *Communications of the ACM* 24, 351-369

Minsky, M. (1962) Problems of formulation for artificial intelligence, *Proc Symp on Mathematical Problems in Biology*, Providence: American Mathematical Society, 35-46

Nilsson, N. J. (1980) The interplay between experimental and theoretical methods in artificial intelligence, Menlo Park: SRI International, TN 229

Truesdell, C. (1958) Recent advances in rational mechanics, *Science* 127, 729-739

(Continued from page 49)

Second National Conference on Artificial Intelligence, Pittsburg, PA., 300–331

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

Novak, G. S (1983) *Knowledge-Based Programming in GLISP* American Association for Artificial Intelligence, *Proc Third National Conference on Artificial Intelligence*, Washington, D C , in press

Teitelman, W (1978) *INTERLISP Reference Manual* Xerox Palo Alto Research Center

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