Report 82-35
Stanford -- KSL

Scientific DataLink

GLISP: A High-Level Language for AI Programming.
Gordon S. Novak Jr,
Dec 1982

Heuristic Programming Project
Report No. HPP-82-35

December 1982

GLISP: A High-Level Language for A.I. Programming

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

This paper appeared in *Proc. Second National Conference on Artificial Intelligence*, Pittsburgh, PA, August 1982

GLISP: A High-Level Language for A.I. Programming

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

ABSTRACT

GLISP is a high-level LISP-based language which is compiled into LISP using a knowledge base of object descriptions. Lisp objects and objects in A.I. representation languages are treated uniformly; this makes program code independent of the data representation used, and permits changes of representation without changing code. GLISP's object description language provides a powerful abstract datatype facility which allows the structures and properties of objects to be described. Reference to objects is permitted in an English-like syntax, including definite reference relative to the current context of the computation. Object-centered programming is supported. When interfaced to a hierarchical representation language, GLISP can perform inheritance at compile time, resulting in substantial performance improvements. In addition, a LISP structure can be specified as the way of implementing a class of objects in the representation language, making simple objects efficient in both time and storage.

1. Introduction

Progress in A.I. is limited by our ability to manage the complexity inherent in A.I. problems; in particular, such problems often involve large numbers of different types of objects, whose properties and interactions must be modelled by a program. The need to manage a large number of object types has led to the development of a number of A.I. representation languages; however, these languages have generally suffered from two problems. First, while the languages provide benefits for *storage* of data and inheritance of procedures, *access* to objects must often be performed by low-level functions; in addition, the *type* of the data retrieved must often be remembered by the programmer. These factors make programs difficult to change after their initial implementation. Second, hierarchical representation languages have achieved power in data representation at the cost of performing data access interpretively at runtime; this has a high cost in execution time.

This research was supported 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.

GLISP solves these problems, as well as being a powerful programming language in its own right. Data objects are described separately from code which references the objects, making code largely representation-independent, as well as shorter and more understandable. Type inference is performed when features of an object are accessed, and type information is propagated by the compiler during compilation. A change in representation, made in only one place, is reflected throughout the program upon recompilation. GLISP compiles efficient code for access to object properties. By performing lookup of inherited properties at compile time, expanding definitions of property access methods into open code, and using Lisp structures to implement instances of objects described in an A.I. representation language, GLISP can greatly increase the execution efficiency of programs using such a language.

The following example function illustrates some of the features of GLISP:

```
(GIVE-RAISE  (GLAMBDA ( (A COMPANY) )
   (FOR EACH ELECTRICIAN WHO IS NOT A TRAINEE
     DO (SALARY ← (IF SENIORITY > 2
                      THEN 2.50 ELSE 1.50))
        (PRINT THE NAME OF THE ELECTRICIAN)
        (PRINT THE PRETTYFORM OF DATE-HIRED)
        (PRINT MONTHLY-SALARY) ) ))
```

The GLAMBDA in the function definition causes the INTERLISP interpreter to call the GLISP compiler to incrementally compile the function the first time it is referenced. In this example, only the *type* of the function's argument, (A COMPANY), is specified; this argument provides an initial *Context* of computation. Since GLISP allows definite references to features of objects which are in Context, it is not always necessary to give variable names to objects. With a COMPANY in Context, the definite reference to the ELECTRICIANs can be understood by the compiler as the ELECTRICIANs of that COMPANY. Within the FOR loop, the current ELECTRICIAN is in Context, permitting references to SALARY, SENIORITY, etc. to be resolved.

Such a function is compiled *relative to a knowledge base* of object descriptions; the GLISP code itself is independent of the actual data representations used. Separation of object descriptions from code permits substantial changes to object structures with no changes to the code. SENIORITY, for example, might be stored directly as part of an EMPLOYEE object, or it might be computed from other properties; for example, SENIORITY could be defined as

```
(  (THE YEAR OF (CURRENTDATE))
 - (THE YEAR OF DATE-HIRED)     )
```

where (CURRENTDATE) is a function which returns the current date and DATE-HIRED is a substructure of an EMPLOYEE. Uniform treatment of stored and computed properties facilitates hiding the distinction between properties which are actually stored and those which are computed as needed. The compiled LISP code produced by GLISP is nearly as efficient as hand-coded LISP; the user must pay for compilation, but does not incur a substantial runtime penalty. Ordinary LISP is a subset of GLISP; normal LISP code is simply passed through by the GLISP compiler without change.

2. Object Descriptions

An *Object Description* contains a *Structure Description*, which describes the way an object is actually stored, *Properties*, which are computed from stored values, *Adjectives*, which may be used as predicates on objects, and *Messages* to which the object can respond. The following example illustrates an object description for a VECTOR:

```
(VECTOR

   (LIST (X INTEGER) (Y INTEGER))

   PROP  ( (MAGNITUDE  ((SQRT X↑2 + Y↑2)) ) )

   ADJ   ( (ZERO       (X IS ZERO
                            AND Y IS ZERO))
           (NORMALIZED (MAGNITUDE = 1.0)) )

   MSG   ( (+         VECTORPLUS       OPEN T)
           (-         VECTORDIFFERENCE OPEN T)
           (PRIN1     ((PRIN1 "(")
                       (PRIN1 X)
                       (PRIN1 ",")
                       (PRIN1 Y)
                       (PRIN1 ")")))
           (PRINT     ((Send self PRIN1)
                       (TERPRI))) )
)
```

The actual structure of the object is a LIST of two INTEGERs, X and Y. The language of Structure Descriptions allows the standard LISP structures to be specified. Objects may be defined in terms of other objects, and can inherit the features and behavior of component objects. The rest of the Object Description defines Properties, Adjectives, and Messages for the object. The MAGNITUDE is defined in terms of a LISP function, SQRT, whose argument is an expression involving definite references to the X and Y components of the VECTOR. Adjectives are used in predicate expressions on objects, e.g., `(IF THE VECTOR IS NOT NORMALIZED THEN ...)`. The adjective NORMALIZED is easily defined using definite reference to the property MAGNITUDE, which in turn is defined in terms of other properties.

3. Definite Reference and English-Like Programming

GLISP permits English-like statements, including definite reference to objects which are in the current computational *Context*. The Context, a compile-time construct which initially contains the arguments of a function, is propagated through the program along control flow paths by the compiler. Newly referenced objects are added to Context, so that their features may then be referenced directly. A variable may be referenced by its name ("X") or its type ("The ELECTRICIAN"). A substructure or Property of an object may be referenced in English-like forms ("The SALARY of X" or "The SALARY" or "SALARY") or in PASCAL-like syntax ("X:SALARY"). The forms "The ELECTRICIAN", "The SALARY" and "SALARY" are definite references, since the source object is unspecified; such references are resolved by finding an object in the Context which is of the specified type or has the specified property. Definite reference to the member(s) of a group which satisfy a specified condition is also allowed, as in `(The Slot with SlotName = NAME)` or `(Those Faculty-Members who are Tenured)`.

Definite reference allows programs to be shorter and more readable, allows easy definition of object properties in terms of other properties, and makes programs easier to modify. Definite reference is potentially ambiguous (i.e., there could be multiple objects in the Context with a specified property); however, in practice this has not proved to be a problem. The ordering of the Context search by the compiler and the user's skill in use of definite reference as a speaker of natural language generally prevent inadvertent references; a reference can always be made unambiguous by specifying the source object.

4. Messages

Object-Centered Programming, inspired by SIMULA [1] and SMALLTALK [2] [3], has become increasingly popular; it views objects as active entities which communicate by exchanging *Messages*. A Message specifies the destination object, a *Selector* which identifies the message type, and optional *Arguments*. When a message is received, the receiving object looks up the Selector to find a corresponding procedure to respond to the message and executes it. Typically, objects

are organized in a Class hierarchy, so that the procedure which is used to execute a message to an object is inherited from the object's Class.

GLISP permits optimized compilation of Messages. An Object Description contains the Selector and associated *Response* for each message the object can receive. The Response may be either GLISP code, which is recursively compiled in-line, or a function name, in which case a direct call to the function (or open code) is compiled. *Properties* and *Adjectives* are compiled as if they were messages without arguments. Because the Response to a message can be GLISP code which is compiled recursively in the context of the object in question, and which can use definite reference to access substructures and properties of the object, it is easy to define properties in terms of other properties. If arithmetic operators are defined as message Selectors for a class of objects, arithmetic expressions involving objects of that type will be compiled as calls to the corresponding Response functions, as illustrated in the VECTOR example above; open compilation of the Response for operators allows the resulting code to be efficient.

5. GLISP and Knowledge Representation Languages

An interface is provided to allow GLISP to be used with a knowledge representation language of the user's choice; each *Class* of objects in the representation language becomes a valid *type* for GLISP. Use of a representation language gives GLISP additional power, since properties and messages can be inherited through the language's hierarchies. In addition, GLISP can significantly improve runtime performance by doing procedural inheritance at compile time and compiling a direct call to an inherited procedure; for simple functions, open compilation avoids the function call as well. Messages which cannot be resolved at compile time are interpreted at runtime, as usual. Messages which in fact specify data access can be recognized as such and can be compiled to perform the data access directly. Representation languages often involve structural overhead which is costly in terms of storage and access time; the overhead encourages users of such a language to have a mixed representation, with complex objects represented in the language and simple objects represented as LISP structures which are manipulated directly. GLISP allows the user to have the best of both worlds. A LISP structure can be specified as the way of implementing a Class of objects. While such an object appears the same as other objects in the representation language in terms of its definition and access language, direct LISP code is compiled for all accesses to its data values. In this way, all objects appear uniform, but simple objects are represented efficiently.

Recursive compilation of object properties can be used to achieve a certain amount of automatic programming. For example, a knowledge base can define the property DENSITY as MASS/VOLUME at a high level, for all physical objects. If a function references the DENSITY of a PLANET object, this definition of density will be inherited and compiled recursively *in the context of the original object*, namely a PLANET. If PLANET has SPHERE as one of its parent classes, the definition for VOLUME can be inherited from SPHERE. The result of such recursive compilation is that different in-line code is compiled for different kinds of objects from the same high-level definition of DENSITY.

6. Discussion

GLISP provides several novel and valuable language features:

1. Object Descriptions provide a powerful abstract datatype facility, including Properties, Adjectives, Messages, and operator overloading for user datatypes. Objects may be composed from other objects, inheriting their properties.

2. Objects in Knowledge Representation Languages, as well as LISP objects, can be used in a uniform fashion.

3. English-like programming, including definite reference relative to the Context, is allowed.

4. Optimized compilation is performed for access to objects in hierarchical representation languages.

These capabilities are integrated, and reinforce each other synergistically. While other systems, e.g. CLISP [4] and Flavors [5] provide some of the features of GLISP, none (to our knowledge) do so in a way that is integrated and allows the user to choose the data representation.

7. Implementation Status

GLISP [6], including all features described in this paper, is currently running. To date, it has been interfaced to the GIRL representation language, and to the object-centered representation language LOOPS [7]. GLISP was originally implemented within INTERLISP; it is currently available for INTERLISP, MACLISP, FRANZ LISP, and UCI LISP. GLISP thus provides a high-level Lisp-based language which is transportable across the major dialects of Lisp.

8. Example

This section illustrates how the example function discussed earlier is compiled for a particular choice of data structures. The compiler optimizes the iteration to avoid explicit construction of the set of ELECTRICIANS.

```
(GIVE-RAISE (GLAMBDA ((A COMPANY))
   (FOR EACH ELECTRICIAN WHO IS NOT A TRAINEE
      DO (SALARY ←+(IF SENIORITY > 2
                       THEN 2.5  ELSE 1.5))
         (PRINT THE NAME OF THE ELECTRICIAN)
         (PRINT THE PRETTYFORM OF DATE-HIRED)
         (PRINT MONTHLY-SALARY))))
```

```
(G 'SPOBJECTS

   (COMPANY (ATOM (PROPLIST
             (PRESIDENT (AN EMPLOYEE))
             (EMPLOYEES (LISTOF EMPLOYEE))))

      PROP ((ELECTRICIANS
               ((THOSE EMPLOYEES WITH
                 JOBTITLE='ELECTRICIAN)))))

   (EMPLOYEE (LIST (NAME STRING)
                   (DATE-HIRED (A DATE))
                   (SALARY REAL)
                   (JOBTITLE ATOM)
                   (TRAINEE BOOLEAN))

      PROP  ((SENIORITY
                ((THE YEAR OF (CURRENTDATE))
                 -(THE YEAR OF DATE-HIRED)))
             (MONTHLY-SALARY (SALARY * 174)))

      ADJ   ((HIGH-PAID (MONTHLY-SALARY>2000)))

      ISA   ((TRAINEE (TRAINEE)))

      MSG   ((YOURE-FIRED (SALARY ← 0)))  )

   (DATE     (LIST (MONTH INTEGER)
                   (DAY INTEGER)
                   (YEAR INTEGER))

      PROP  ((MONTHNAME ((CAR (NTH '(January
                   February March April May
                   June July August September
                   October November December)
                            MONTH))))
             (PRETTYFORM
                 ((LIST DAY MONTHNAME YEAR)))
             (SHORTYEAR (YEAR - 1900))))
)
```

With these data structure definitions, the function GIVE-RAISE is compiled (for INTERLISP) as follows:

```
(GIVE-RAISE (LAMBDA (GLVAR1)
  (MAPC (GETPROP GLVAR1 (QUOTE EMPLOYEES))
      (FUNCTION (LAMBDA (GLVAR2)
   (AND (EQ (CADDDR GLVAR2)
            (QUOTE ELECTRICIAN))
     (COND
       ((NOT (CAR (NTH GLVAR2 5)))
        [RPLACA (CDDR GLVAR2)
          (PLUS (CADDR GLVAR2)
            (COND
              ((IGREATERP (IDIFFERENCE
                    (CADDR (CURRENTDATE))
                    (CADDR (CADR GLVAR2)))
                  2) 2.5)
              (T 1.5]
        (PRINT (CAR GLVAR2))
        [PRINT (PROG ((self (CADR GLVAR2)))
            (RETURN (LIST (CADR self)
                (CAR (NTH (QUOTE (January
                  February March April May
                  June July August September
                  October November December))
                          (CAR self)))
                (CADDR self]
        (PRINT (TIMES (CADDR GLVAR2) 174]
```

References

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

2. Ingalls, D., "The Smalltalk-76 Programming System: Design and Implementation," *5th ACM Symposium on Principles of Programming Languages*, ACM, 1978, pp. 9-16.

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

4. Teitelman, W.Xerox Palo Alto Research Center, *INTERLISP Reference Manual*, 1978.

5. Cannon, Howard I., "Flavors: A Non-Hierarchical Approach to Object-Oriented Programming," Tech. report (unnumbered), M.I.T. A.I. Lab, Oct. 1981.

6. Novak, Gordon S., "GLISP Reference Manual," Tech. report STAN-CS-895, Computer Science Dept., Stanford Univ., Jan. 1982.

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