GIRL and GLISP:
An Efficient Representation Language

Gordon S. Novak Jr.

TR-172

Computer Science Department
University of Texas at Austin
Austin, Texas 78712

March, 1981

(Submitted to the Seventh International Joint Conference on Artificial Intelligence, August, 1981.)

This research was supported by NSF Award No. SED-7912803 in the Joint National Institute of Education - National Science Foundation Program of Research on Cognitive Processes and the Structure of Knowledge in Science and Mathematics.

1.Ø INTRODUCTION

This paper describes GIRL, a hierarchical, "frame-like" representation language, and GLISP, a LISP-based language for accessing data represented in GIRL (or another representation language) or in ordinary LISP structures. The combination of GIRL and a compiler which compiles GLISP programs (into LISP) relative to a GIRL knowledge base provides powerful language features with high efficiency.

2.Ø GIRL REPRESENTATION LANGUAGE

GIRL is a hierarchical, "frame-like" representation language; of the existing representation languages, it is probably most like UNITS [1]. A node in the hierarchy which represents a class of units has associated with it a Prototype unit which specifies the structure of units in the class. The designer of the knowledge base can specify for each subtree whether properties are to be copied from a class into its descendants, or whether inheritance of properties is to be done at runtime; thus, the time required for run-time inheritance can be traded off against space for explicit copies as desired for each class of units.

A unit contains Slots, each of which is analogous to an expanded property list entry. A slot contains Aspects, which may include the slot's Value, If-Needed and If-Added functions, an Inheritance Mode (used primarily to guide prompting of the user when new units are being created), and a Structure Description,

which describes the expected structure of the slot's value. Normally, every slot will have a Structure Description; for Instance units, the Structure Description is typically inherited from the prototype for the unit's class rather than being copied in the instance. The use of Structure Descriptions by the GLISP compiler gives the language much of its power.

3.0 STRUCTURE DESCRIPTIONS

A Structure Description (SD) is analogous to a record structure declaration in a language such as PASCAL. The lowest level of a SD is a basic LISP datatype (e.g., ATOM, INTEGER, REAL, BOOLEAN, NIL, ANYTHING), a name which has been declared as the name of a particular SD, or (A <class>), where <class> is a class of units in the GIRL knowledge base. A part of a SD can be named by enclosing it in a list with its name, (<name> SD); thus, a substructure can have multiple names if desired. Compound structures can be specified using grouping operators such as (CONS SD1 SD2), (LIST SD1...SDn) [a list of exactly the elements SD1...SDn], (LISTOF SD) [a list of zero or more elements, each of which is of the form SD], and others.

4.0 GLISP ACCESS LANGUAGE

GLISP is designed to allow creation of and access to structures (either LISP structures or GIRL units) in a transparent and somewhat informal fashion, and to allow structures to be changed without changing the programs which reference the structures. Two syntaxes are defined for accessing

parts of structures: an English-like syntax, e.g. (THE FIELD OF RECORD), and a syntax like that of CLISP [2], e.g. RECORD:FIELD. Such a path specification may of course be extended through multiple records. Definite references to structures or fields are allowed if they are defined relative to the Context of the program at the point of reference, as in (THE COLOR OF THE CAT) or simply (THE COLOR) if a CAT is in context.

Both the interpretive version of GLISP and the GLISP compiler maintain a Context which records the structures which have been referenced within a function. Initially, the context is defined by the arguments of the function, its PROG variables, and any declared GLOBAL variables. Conceptually, whenever a new structure is accessed (whether by reference to part of a structure, application of a known LISP function to a structure, or calling a function whose result type is known), the resulting value and its structure description are added to the Context. The GLISP interpreter maintains the Context as an actual structure at runtime, while the GLISP compiler computes it at compile time using flow analysis [3]. The following example illustrates the use of context within a function:

```
(CATCOUNT  (GLAMBDA  ( (A GRANDMOTHER) )
    (PROG  ((COUNT Ø))
        (FOR EACH CAT WITH COLOR = 'CALICO DO COUNT ←+ 1)
        (RETURN COUNT)    )))
```

The first time this function is interpreted by INTERLISP [2], the

nonstandard GLAMBDA will cause the GLISP compiler to be called with the GLAMBDA expression as its argument; the compiler will compile the function and return a normal LISP EXPR, which will be used thereafter (and which could be compiled further by the LISP compiler). In this example, only the type of the function's argument is declared; since the argument will be in context, it is only necessary to give it a variable name if it must be referred to explicitly (as when, for example, there are two objects of the same type in context). The PROG variable COUNT is initialized to zero, and as a side effect receives the type INTEGER. Since a GRANDMOTHER is in context, the compiler can determine how to get her set of CATs and how to generate a loop to search through them. Within the loop, the current CAT will be in context, and its COLOR can be retrieved. The SD of the COLOR of a CAT is also retrieved, and if necessary, the constant value 'CALICO is coerced by the compiler to the appropriate value. For example, if the SD of COLOR were (A CAT-COLOR) and CALICO were specified as an alias of the instance CALICO-CAT-COLOR, the compiled code would test against the constant CALICO-CAT-COLOR. This feature allows informal specification of ambiguous terms whose meaning is determined relative to context. Operators can be "overloaded" or automatically coerced as well; the operator ←+ , whose meaning is Append when applied to lists, is interpreted as addition when applied to numbers.

We have eschewed the currently fashionable notion of strong typing in favor of the notion that the compiler should infer types whenever possible. This makes possible significant changes

in data structures with no changes in the programs which access them. A GRANDMOTHER, for example, could be a LISP structure which includes a list of CATs, or an atom with CATs on its property list, or a GIRL unit which has CATs as a substructure of the contents of the PETS slot.

Initial versions of GIRL and the GLISP compiler and interpreter are currently running.

5.0 REFERENCES

1. Stefik, M., "An Examination of a Frame-Structured Representation System", Proc. 6th IJCAI, Tokyo, 1979, pp. 845-852.

2. Teitelman, W., "INTERLISP Reference Manual", Xerox Palo Alto Research Center, 1978.

3. Novak, G., "Compilation of Definite References Using Flow Analysis of Context and Structure Description", forthcoming.