Scientific DataLink

Report 82-01
Stanford -- KSL

GLISP Users' Manual.
Gordon S. Novak Jr,
Jan 1982

January 1982

Report. No. STAN-CS-82-895

Also numbered:
HPP-82-1

GLISP Users' Manual

by

Gordon S. Novak, Jr.

Department of Computer Science

Stanford University
Stanford, CA 94305

GLISP User's Manual

Gordon S. Novak Jr.
Computer Science Department
Stanford University
Stanford, California 94305

11 January 1982

GLISP is a high-level, LISP-based language which is compiled into LISP. GLISP provides a powerful abstract datatype facility, allowing description and use of both LISP objects and objects in A.I. representation languages. GLISP language features include PASCAL-like control structures, infix expressions with operators which facilitate list manipulation, and reference to objects in PASCAL-like or English-like syntax. English-like definite reference to features of objects which are in the current computational context is allowed; definite references are understood and compiled relative to a knowledge base of object descriptions. Object-centered programming is supported; GLISP can substantially improve runtime performance of object-centered programs by optimized compilation of references to objects. This manual describes the GLISP language and use of GLISP within INTERLISP.

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.

Table of Contents

1. Introduction 1
1.1. Overview of GLISP 1
1.2. Implementation 2
2. Object Descriptions 3
2.1. Declaration of Object Descriptions 3
2.2. Structure Descriptions 4
2.2.1. Syntax of Structure Descriptions 4
2.2.2. Examples of Structure Descriptions 6
3. Reference To Objects 7
3.1. Accessing Objects 7
3.2. Creation of Objects 8
3.3. Predicates on Objects 8
3.3.1. Self-Recognition Adjectives 8
4. GLISP Program Syntax 11
4.1. Function Syntax 11
4.2. Statement Syntax 12
4.2.1. Expressions 12
4.2.1.1. Self-Assignment Operators[25]
4.2.2. Compound Statements 14
4.2.2.1. IF Statement 14
4.2.2.2. FOR Statement 14
4.2.2.3. Definite Reference to a Particular Object 15
4.2.2.4. WHILE Statement 15
4.2.2.5. REPEAT Statement 16
4.2.3. Messages 16
4.2.3.1. Compilation of Messages 17
4.2.3.2. Compilation of Properties and Adjectives 18
4.2.3.3. Declarations for Message Compilation 18
4.2.3.4. Operator Overloading 19
5. Context Rules and Reference 21
5.1. Organization of Context 21
5.2. Rules for Using Definite Reference 21
5.3. Type Inference 22
6. GLISP and Knowledge Representation Languages 23
7. GLISP Hacks 25
7.1. Overloading Basic Types 25
7.2. Disjunctive Types 26
7.3. Generators 26
8. Program Examples 29
8.1. GLTST1 File 29
8.2. GLTST2 File 29

[25] This section may be skipped by the casual user of GLISP.

1. Introduction

1.1. Overview of GLISP

GLISP is a LISP-based language which provides high-level language features not found in ordinary LISP. The GLISP language is implemented by means of a compiler which accepts GLISP as input and produces ordinary LISP as output; this output can be further compiled to machine code by the LISP compiler.

The goal of GLISP is to allow structured objects to be referenced in a convenient, succinct language, and to allow the structures of objects to be changed without changing the code which references the objects. The syntax of many GLISP constructs is English-like; much of the power and brevity of GLISP derive from the compiler features necessary to support the relatively informal, English-like language constructs. The following example function illustrates how GLISP permits definite reference to structured objects.

```
(HourlySalaries (GLAMBDA ( (a DEPARTMENT) )
   (for each EMPLOYEE who is HOURLY
      (PRIN1 NAME) (SPACES 3) (PRINT SALARY) )  ))
```

The features provided by GLISP include the following:

1. GLISP maintains knowledge of the "context" of the computation as the program is executed. Features of objects which are in context may be referenced directly; the compiler will determine how to reference the objects given the current context, and will add the newly referenced objects to the context. In the above example, the function's argument, an object whose class is DEPARTMENT, establishes an initial context relative to which EMPLOYEEs can be found. In the context of an EMPLOYEE, NAME and SALARY can be found.

2. GLISP supports flexible object definition and reference with a powerful abstract datatype facility. Object classes are easily declared to the system. An object declaration includes a definition of the storage structure of the object and declarations of properties of the object; these may be declared in such a way that they compile open, resulting in efficient object code. GLISP supports object-centered programming, in which processes are invoked by means of "messages" sent to objects. Object structures may be LISP structures (for which code is automatically compiled) or Units in the user's favorite representation language (for which the user can supply compilation functions).

3. Loop constructs, such as (FOR EACH <item> WITH <property> DO ...) , are compiled into loops of the appropriate form.

4. Compilation of infix expressions is provided for the arithmetic operators and for additional operators which facilitate list manipulation. Operator overloading for user-defined objects is provided using the message facility.

5. The GLISP compiler infers the types of objects when possible, and uses this knowledge to generate efficient object code. By performing *compilation relative to a knowledge base* , GLISP is able to perform certain computations (e.g., inheritance of an attached procedure from a parent

class of an object in a knowledge base) at compile time rather than at runtime, resulting in much faster execution.

6. By separating object definitions from the code which references objects, GLISP permits radical changes to object structures with no changes to code — a goal long sought in high-level languages, but one which has largely been unrealized for structures involving pointers.

1.2. Implementation

GLISP is implemented by means of a compiler, which incrementally compiles each function the first time the function is called. [Of course, compilation can be invoked directly as well.] GLISP functions are indicated by the use of GLAMBDA instead of LAMBDA in the function definition. When the INTERLISP interpreter sees the GLAMBDA, it effects an "interrupt" to the GLISP compiler[1] , which compiles the GLISP function and returns a normal LISP EXPR; thereafter, the LISP version is used. Thus, use of GLISP entails the cost of a single compilation, but otherwise is about as efficient as normal LISP. The LISP code produced by GLISP can be further compiled to machine code by the LISP compiler.

To use GLISP, it is only necessary to load the compiler: LOAD(GLISP.LSP) . Thereafter, whenever a function which has GLAMBDA in its definition is interpreted, the compiler will be called automatically. The GLISP compiler is also called automatically when LISP compilation of a function is requested. An individual function can be compiled explicitly by invoking `GLCOMPILE(<fn>)`, where `<fn>` is the name of the function. If it is desired to explicitly compile all the GLISP functions in a file, this can be done by invoking `(GLCOMPCOMS <file>COMS)`, where `<file>COMS` is bound to the list of file package commands for the file; this will call the GLISP compiler for each function whose definition begins with GLAMBDA.

The compiled code, result type, and original code for compiled functions are stored on the property list of the function name. Properties of GLISP functions and Structure names can be examined with the function `GLED(<name>)`, which calls the INTERLISP editor on the property list of `<name>`.

---

[1] using the LAMBDATRAN feature of INTERLISP, written by Ron Kaplan.

2. Object Descriptions

2.1. Declaration of Object Descriptions

An *Object Description* in GLISP is a description of the structure of an object in terms of named substructures, together with definitions of ways of referencing the object. The latter may include *virtual fields* (i.e., fields whose values are not stored, but are computed from the values of other fields), adjectival predicates, and *messages* which the object can receive; the messages can be used to implement operator overloading and other compilation features.

Object Descriptions are obtained by GLISP in several ways:

1. The descriptions of basic datatypes (e.g., INTEGER) are automatically known to the compiler.
2. Structure descriptions (but not full object descriptions) may be used as *types* in function definitions.
3. The user may declare object descriptions to the system using the function GLDEFSTRQ.
4. Object descriptions may be included as part of a knowledge representation language, and are then furnished to GLISP by the interface package written for that representation language.

LISP data structures are declared using the function GLDEFSTRQ ("GLisp DEFine STRucture Quote"). GLDEFSTRQ takes one or more object descriptions as arguments, assuming the descriptions to be quoted. The format of each description is as follows:

```
(<object-name>  <structure-description>
          PROP  <property-descriptions>
          ADJ   <adjective-descriptions>
          ISA   <predicate-descriptions>
          MSG   <message-descriptions>    )
```

The <object-name> and <structure-description> are required; the other property/value pairs are optional, and may appear in any order. Each `<description>` specified with PROP, ADJ, ISA, or MSG has the following format:

```
(<name> <response> <prop_1> <value_1> ... <prop_n> <value_n>)
```

where `<name>` is the (atomic) name of the property, `<response>` is a function name or a list of GLISP code to be compiled in place of the property, and the `<prop><value>` pairs are optional properties which affect compilation. The compilation of all of these properties is described in the section "Compilation of Messages".

Once declared, object descriptions may be included in INTERLISP program files by including in the

<file>COMS a statement of the form:

```
(GLISPOBJECTS <object-name_1> ... <object-name_n>)
```

The following example illustrates some of the declarations which might be made to describe the object type *Vector.*

```
(GLDEFSTRQ

   (VECTOR   (CONS (X NUMBER) (Y NUMBER))

     PROP    ( (MAGNITUDE  ((SQRT X*X + Y*Y))) )

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

     MSG     ( (+          VECTORPLUS OPEN T)
               (-          VECTORDIFFERENCE) )

  ))
```

Since GLISP compilation is performed relative to the knowledge base of object descriptions, the object descriptions must be declared prior to GLISP compilation of functions using those descriptions.

2.2. Structure Descriptions

Much of the power of GLISP is derived from its use of Structure Descriptions. A Structure Description (abbreviated "<sd>") is a means of describing a LISP data structure and giving names to parts of the structure; it is similar in concept to a Record declaration in PASCAL. Structure descriptions are used by the GLISP compiler to generate code to retrieve and store parts of structures.

2.2.1. Syntax of Structure Descriptions

The syntax of structure descriptions is recursively defined in terms of basic types and composite types which are built up from basic types. The syntax of structure descriptions is as follows:[2]

1. The following basic types are known to the compiler:

   ATOM
   INTEGER
   REAL
   NUMBER (either INTEGER or REAL)
   STRING

[2] The names of the basic types and the structuring operators must appear in upper-case as shown here. In general, other GLISP keywords and user program names may be in upper-case, lower-case, or mixed-case.

BOOLEAN	(either T or NIL)
ANYTHING	(an arbitrary structure)

2. An object type which is known to the compiler, either from a GLDEFSTRQ declaration or because it is a Class of units in the user's knowledge representation language, is a valid type for use in a structure description. The <name> of such an object type may be specified directly as <name> or, for readability, as (A <name>) or (An <name>).[3]

3. Any substructure can be named by enclosing it in a list prefixed by the name: (<name> <sd>). This allows the same substructure to have multiple names. The names used in forming composite types (given below) are treated as reserved words, and may not be used as names.

4. Composite Structures: Structured data types composed of other structures are described using the following structuring operators:

   a. (CONS <sd$_1$> <sd$_2$>)
   The CONS of two structures whose descriptions are <sd$_1$> and <sd$_2$>.

   b. (LIST <sd$_1$> <sd$_2$> ... <sd$_n$>)
   A list of exactly the elements whose descriptions are <sd$_1$> <sd$_2$> ... <sd$_n$>.

   c. (LISTOF <sd>)
   A list of zero or more elements, each of which has the description <sd>.

   d. (ALIST (<name$_1$> <sd$_1$>) ... (<name$_n$> <sd$_n$>))
   An association list in which the atom <name$_i$>, if present, is associated with a structure whose description is <sd$_i$>.

   e. (ATOM (BINDING <sd>) (PROPLIST (<pname$_1$> <sd$_1$>) ... (<pname$_n$> <sd$_n$>)))
   This describes an atom with its binding and/or its property list; either the BINDING or the PROPLIST group may be omitted. Each property name <pname$_i$> is treated as a property list indicator as well as the name of the substructure. When creation of such a structure is specified, GLISP will compile code to create a GENSYM atom.

   f. (TRANSPARENT <name>)
   An object of type <name> is incorporated into the structure being defined in *transparent mode*, which means that all fields and properties of the object of type <name> can be directly referenced as if they were properties of the object being defined. The object of type <name> may also contain TRANSPARENT objects; the graph of TRANSPARENT object references must of course be acyclic.

   g. (RECORD <recordname> (<name$_1$><sd$_1$>) ... (<name$_n$><sd$_n$>))
   This description allows the use of INTERLISP RECORD or DATATYPE records with GLISP. <recordname> is the name of the record type (which must be declared separately to INTERLISP); the ordering of the (<name><sd>) pairs is independent of the actual structure of the record. GLISP will compile fetch, replace, and create for access to and creation of RECORD structures.

---

[3] Whenever the form (A ...) is allowed in GLISP, the form (An ...) is also allowed.

2.2.2. Examples of Structure Descriptions

The following examples illustrate the use of Structure Descriptions.

```
(GLDEFSTRQ

    (CAT (LIST (NAME ATOM)
               (PROPERTIES (LIST (CONS (SEX ATOM)
                                       (WEIGHT INTEGER))
                                 (AGE INTEGER)
                                 (COLOR ATOM)))
               (LIKESCATNIP BOOLEAN)))

    (GRANDMOTHER (ATOM
         (PROPLIST
              (GRANDCHILDREN (LISTOF (A PERSON)))
              (AGE INTEGER)
              (PETS (LIST (CATS (LISTOF (A CAT)))
                          (DOGS (LISTOF (A DOG))) ))
             )))
)
```

The first structure, CAT, is entirely composed of list structure. An example CAT might look like:

```
(PUFF ((MALE . 10) 5 CALICO) T)
```

Given a CAT object X, we could ask for its WEIGHT [equivalent to (CDAADR X)] or for a subrecord such as PROPERTIES [equivalent to (CADR X)]. Having set a variable Y to the PROPERTIES, we could also ask for the WEIGHT from Y [equivalent to (CDAR Y)]. In general, whenever a subrecord is accessed, the structure description of the subrecord is associated with it by the compiler, enabling further accesses to parts of the subrecord. Thus[4], the meaning of a subrecord name depends on the type of record from which the subrecord is to be retrieved. The subrecord AGE has two different meanings when applied to GRANDMOTHERs and CATs. The second structure, GRANDMOTHER, illustrates a description of an object which is a LISP atom with properties stored on its property list. Whereas no structure names appear in an actual CAT structure, the substructures of a **PROPLIST** (or **ALIST**) operator must be named, and the names appear in the actual structures. For example, if X is a GRANDMOTHER structure, retrieval of the AGE of X is equivalent to (GETPROP X (QUOTE AGE)). A subrecord of a PROPLIST record can be referenced directly; e.g., one can ask for the DOGS of a GRANDMOTHER directly, without cognizance of the fact that DOGS is part of the PETS property.

---

[4] in contrast to the CLISP record package

3. Reference To Objects

3.1. Accessing Objects

The problem of reference is the problem of determining what object, or feature of a structured object, is referred to by some part of a statement in a language. Most programming languages solve the problem of reference by unique naming: each distinct object in a program unit has a unique name, and is referenced by that name. Reference to a part of a structured object is done by giving the name of the variable denoting that object and a path specification which tells how to get to the desired part from the whole.

GLISP permits reference by unique naming and path specification, but in addition permits *definite reference relative to context.* A *definite reference* is a reference to an object which has not been explicitly named before, but which can be understood relative to the current context of computation. If, for example, an object of type VECTOR (as defined earlier) is in context, the program statement

```
(IF X IS NEGATIVE ...
```

contains a definite reference to "X", which may be interpreted as the X substructure of the VECTOR which is in context. The definition of the computational context and the way in which definite references are resolved are covered in a later section of this manual.

In the following section, which describes the syntaxes of reference to objects in GLISP, the following notation is used. "<var>" refers to a variable name in the usual LISP sense, i.e., a LAMBDA variable, PROG variable, or GLOBAL variable; the variable is assumed to point to (be bound to) an object. "<type>" refers to the type of object pointed to by a variable. "<property>" refers to a property or subrecord of an object.

Two syntaxes are available for reference to objects: an English-like syntax, and a PASCAL-like (or CLISP-like) syntax. The two are equivalent, and may be intermixed freely within a GLISP function. The allowable forms of references in the two syntaxes are shown in the table below.

"PASCAL" Syntax	"English" Syntax	Meaning
<var>	<var>	The object denoted by <var>
:<type>	The <type>	The object whose type is <type>
:<property> *or* <property>	The <property>	The <property> of some object
<var>:<property>	The <property> of <var>	The <property> of the object denoted by <var>

These forms can be extended to specify longer paths in the obvious way, as in "The AGE of the SPOUSE of the HEAD of the DEPARTMENT" or "DEPARTMENT:HEAD:SPOUSE:AGE". Note that there is no distinction between reference to substructures and reference to properties as far as the syntax of the

referencing code is concerned; this facilitates hiding the internal structures of objects.

3.2. Creation of Objects

GLISP allows the creation of structures to be specified by expressions of the form:

(A <type> *with* <property$_1$> = <value$_1$>, ..., <property$_n$> = <value$_n$>)

In this expression, the "*with*", "=", and "," are allowed for readability, but may be omitted if desired; if present, they must all be delimited on both sides by blanks. In response to such an expression, GLISP will generate code to create the specified structure. The <property> names may be specified in any order. Unspecified properties will be defaulted according to the following rules:

1. Basic types will be defaulted to 0 for INTEGER and NUMBER, 0.0 for REAL, and NIL for other types.

2. Composite structures will be created from the defaults of their components, except that missing PROPLIST and ALIST items will be omitted.

Except for missing PROPLIST and ALIST elements, as noted above, a newly created LISP structure will contain all of the fields specified in its structure description.

3.3. Predicates on Objects

Adjectives defined for structures using the `ADJ` and `ISA` specifications may be used in predicate expressions on objects in If and For statements. The syntax of basic predicate expressions is:

```
<object> is <adjective>
<object> is a <isa-adjective>
```

Basic predicate expressions may be combined using AND, OR, NOT or ~, and grouping parentheses.

The compiler has pre-defined the LISP adjectives ATOMIC, NULL, NIL, INTEGER, REAL, SMALL, ZERO, NUMERIC, NEGATIVE, and MINUS, and the ISA-adjectives ATOM, LIST, NUMBER, INTEGER, and LITATOM; user definitions have precedence over these pre-defined adjectives.

3.3.1. Self-Recognition Adjectives

If the ISA-adjective `self` is defined for an object Class, the Class name may be used as an ISA-adjective to test whether a given object is a member of that Class. Given a predicate phrase of the form " `X is a Y` ", the compiler first looks at the definition of the object class of `X` to see if `Y` is defined as an ISA-adjective for such objects. If no such ISA-adjective is found, and `Y` is the name of a Class of objects, the compiler looks to see if `self` is defined as an ISA-adjective for `Y`, and if so, compiles it.

If a `self` ISA-adjective predicate is compiled as the test of an If, While, or For statement, and the tested object is a simple variable, the variable will be known to be of that type within the scope of the test. For example, in the statement

```
(If X is a FOO then (← X Print) ...
```

the compiler will know that X is a FOO if the test succeeds, and will compile the Print message appropriate for a FOO, even if the type of X was declared as something other than FOO earlier. This feature is useful in implementing disjunctive types, as discussed in a later section.

BLANK PAGE

4. GLISP Program Syntax

4.1. Function Syntax

GLISP function syntax is essentially the same as that of LISP with the addition of type information and RESULT and GLOBAL declarations. The basic function syntax is:[5]

```
(<function-name> (GLAMBDA (<arguments>)
                          (RESULT <result-description>)
                          (GLOBAL <global-variable-descriptions>)
        (PROG (<prog-variables>)
              <code>   )))
```

The RESULT declaration is optional; in many cases, the compiler will infer the result type automatically. The main use of the RESULT declaration is to allow the compiler to determine the result type without compiling the function, which may be useful when compiling another function which calls it. The <result-description> is a standard structure description or <type>.

The GLOBAL declaration is used to inform the compiler of the <type>s of any variables used globally; this declaration is optional, but it must be used if subrecords of global variables are to be referenced.

The major difference between a GLISP function definition and a standard LISP definition is the presence of type declarations for variables, which are in PASCAL-like syntax of the following forms:

```
<variable>:<type>
<variable>:(A <type>)
<variable>,<variable>,...:<type>
<variable>,<variable>,...:(A <type>)
          :<type>
           (A <type>)
```

In addition to declared <type>s, a Structure Description may be used directly as a <type> in a variable declaration.

Type declarations are required only for variables whose subrecords will be referenced. In general, if the value of a variable is computed in such a way that the type of the value can be inferred, the variable will receive the appropriate type automatically; in such cases, no type declaration is necessary. Since GLISP maintains a *context* of the computation, it is often unnecessary to name a variable which is an argument of a function; in such cases, it is only necessary to specify the <type> of the argument, as shown in the latter two

---

[5] The PROG is not required; RESULT and GLOBAL, if present, must be in the order shown.

syntax forms above. PROG and GLOBAL declarations must always specify variable names (with optional types); the ability to directly reference features of objects reduces the number of PROG variables needed in many cases. Initial values for PROG variables may be specified, as in INTERLISP; however, the type of a variable which is given an initial value cannot be explitly specified.

4.2. Statement Syntax

4.2.1. Expressions

GLISP provides translation of infix expressions of the sort usually found in programming languages. In addition, it provides additional operators which facilitate list manipulation and other operations. Overloading of operators for user-defined types is provided by means of the *message* facility, as described in Section 4.2.3.4.

Expressions may be written directly in-line within function references, as in `(SQRT X*X + Y*Y)` , or they may be written within parentheses; parentheses may be used for grouping in the usual way. Operators may be written with or without delimiting spaces, *except for the "-" operator, which* must *be delimited by spaces.*[6] Expression parsing is done by an operator precedence parser, using the same precedence ordering as in FORTRAN.[7] The operators which are recognized are as follows:

```
Assignment      ←
Arithmetic      +   -   *   /   ↑
Comparison      =   ~=  <   <=  >   >=
Logical         AND  OR  NOT  ~
Compound        ←+  ←-  +←  -←
```

Each compound operator performs an operation involving the arguments of the operator and assigns a value to the left-hand argument. The meaning of a compound operator depends on the type of its left-hand argument, as shown in the following table:

---

[6] The "-" operator is required to be delimited by spaces since "-" is often used as a hyphen within variable names. The "-" operator will be recognized within "atom" names if the flag GLSEPMINUS is set to T.

[7] The precedence of compound operators is higher than assignment but lower than that of all other operators. The operators ↑ ← ←+ +← ←- -← are right-associative; all others are left-associative.

Operator	Mnemonic	NUMBER	LISTOF	BOOLEAN
←+	*Accumulate*	PLUS	NCONC1	OR
←-	*Remove*	DIFFERENCE	REMOVE	AND NOT
+←	*Push*	PLUS	PUSH	OR
-←	*Pop*		POP[8]	

As an aid in remembering the list operators, the arrow may be thought of as representing the list, with the head of the arrow being the front of the list and the operation (+ or -) appearing where the operation occurs on the list. Thus, for example, ←+ adds an element at the end of the list, while +← adds an element at the front of the list.

Each of the compound operators performs an assignment to its left-hand side; the above table shows an abbreviation of the operation which is performed prior to the assignment. The following examples show the effects of the operator "←+" on local variables of different types:

Type	Source Code	Compiled Code
INTEGER	`I ←+ 5`	`(SETQ I (IPLUS I 5))`
BOOLEAN	`P ←+ Q`	`(SETQ P (OR P Q))`
LISTOF	`L ←+ ITEM`	`(SETQ L (NCONC1 L ITEM))`

4.2.1.1. Self-Assignment Operators[9]

There are some cases where it would be desirable to let an object perform an assignment of its own value. For example, the user might want to define *PropertyList* as an abstract datatype, with messages such as GETPROP and PUTPROP, and use PropertyLists as substructures of other datatypes. However, a message such as PUTPROP may cause the PropertyList object to modify its own structure, perhaps even changing its structure from NIL to a non-NIL value. If the function which implements PUTPROP performs a normal assignment to its "self" variable, the assignment will affect only the local variable, and will not modify the PropertyList component of the containing structure. The purpose of the Self-Assignment Operators is to allow such modification of the value within the containing structure.

The Self-Assignment Operators are ←←, ←←+, ←+←, and ←←-, corresponding to the operators ←, ←+, +←, and ←-, respectively. The meaning of these operators is that the assignment is performed to the object on the left-hand side of the operator, *as seen from the structure containing the object.*

[8] For the Pop operator, the arguments are in the reverse of the usual order, i.e., (TOP -← STACK) will pop the top element off STACK and assign the element removed to TOP.

[9] This section may be skipped by the casual user of GLISP.

The use of these operators is highly restricted; any use of a Self-Assignment Operator must meet all of the following conditions:

1. A Self-Assignment Operator can only be used within a Message function which is compiled OPEN.

2. The left-hand side of the assignment must be a simple variable which is an argument of the function.

3. The left-hand-side variable must be given a unique (unusual) name to prevent accidental aliasing with a user variable name.

As an example, the PUTPROP message for a PropertyList datatype could be implemented as follows:

```
(PropertyList.PUTPROP (GLAMBDA (PropertyListPUTPROPself prop val)
   (PropertyListPUTPROPself ←←
              (LISTPUT PropertyListPUTPROPself prop val)) ))
```

4.2.2. Compound Statements

GLISP compiles code for certain compound statements which allow more pleasing syntax than ordinary LISP allows.

4.2.2.1. IF Statement

The format of the IF statement is as follows:

```
(IF           <condition_1> THEN <action_11> ... <action_1i>
    ELSEIF <condition_2> THEN <action_21> ... <action_2j>
    ...
    ELSE   <action_m1> ... <action_mk>)
```

Such a statement is translated to a COND of the obvious form. The "THEN" keyword is optional.

4.2.2.2. FOR Statement

The FOR statement generates a loop through a set of elements (typically a list). Two syntaxes of the FOR statement are provided:

```
(FOR EACH <set> DO <action_1> ... <action_n>)

(FOR <variable> IN <set> DO <action_1> ... <action_n>)
```

The keyword "DO" is optional. In the first form of the FOR statement, the singular form of the <set> is specified; GLISP will convert the given set name to the plural form.[10] The <set> may be qualified by an

---

[10] For names with irregular plurals, the plural form should be put on the property list of the singular form under the property name PLURAL, e.g., (PUTPROP 'MAN 'PLURAL 'MEN).

adjective or predicate phrase in the first form; the allowable syntaxes for such qualifying phrases are shown below:

```
<set> WITH <predicate>
<set> WHICH IS <adjective>
<set> WHO IS   <adjective>
<set> THAT IS  <adjective>
```

<predicate> and <adjective> phrases may be combined with AND, OR, NOT, and grouping parentheses. Within the FOR loop, the current member of the <set> which is being examined is automatically put into *context* at the highest level of priority.

As an example, suppose that the current context contains a substructure whose description is

```
(PLUMBERS (LISTOF EMPLOYEE))
```

Assuming that EMPLOYEE contains the appropriate definitions, the following FOR loop could be written:

```
(for each PLUMBER who is not a TRAINEE do SALARY ←+ 1.50)
```

To simplify the collection of features of a group of objects, the <action>s in the FOR loop may be replaced by the CLISP-like constructs

```
... COLLECT <form>)
... COLLECT <form> WHEN <predicate>)
```

4.2.2.3. Definite Reference to a Particular Object

In order to simplify reference to a particular member of a group, definite reference may be used. Such an expression is written using the word The followed by the singular form of the group and qualifying phrases (as described for the For statement). For example, a particular Slot could be selected by the statement:

```
(The Slot with SlotName = NAME)
```

If there is no object satisfying the specified condition, the value of the expression is NIL.

4.2.2.4. WHILE Statement

The format of the WHILE statement is as follows:

(WHILE <condition> DO <action$_1$> ... <action$_n$>)

The actions <action$_1$> through <action$_n$> are executed repeatedly as long as <condition> is true. The keyword DO may be omitted. The value of the expression is NIL.

4.2.2.5. REPEAT Statement

The format of the REPEAT statement is as follows:

(REPEAT <action$_1$> ... <action$_n$> UNTIL <condition>)

The actions <action$_1$> through <action$_n$> are repeated (always at least once) until <condition> is true. The value of the expression is NIL. The keyword UNTIL is required.

4.2.3. Messages

GLISP supports the *Message* metaphor, which has its roots in the languages SMALLTALK and SIMULA. These languages provide *Object-Centered Programming*, in which objects are thought of as being active entities which communicate by sending each other *Messages*. The internal structures of objects are hidden; a program which wishes to access "variables" of an object does so by sending messages to the object requesting the access desired. Each object contains[11] a list of *Selectors*, which identify the messages to which the object can respond. A *Message* specifies the destination object, the selector, and any arguments associated with the message. When a message is executed at runtime, the selector is looked up for the destination object; associated with the selector is a procedure, which is executed with the destination object and message arguments as its arguments.

GLISP treats reference to properties, adjectives, and predicates associated with an object similarly to the way it treats messages. The compiler is able to perform much of the lookup of *selectors* at compile time, resulting in efficient code while maintaining the flexibility of the message metaphor. Messages can be defined in such a way that they compile open, compile as function calls to the function which is associated with the selector, or compile as messages to be interpreted at runtime.

A *message* in GLISP has the following syntax:

(SEND <object> <selector> <arg$_1$> ... <arg$_n$>)

The keyword "SEND" may be replaced by "←". The <selector> is assumed to be quoted. Zero or more arguments may be specified; the arguments other than <selector> are evaluated. <object> is evaluated; if <object> is a non-atomic expression, it must be enclosed in at least one set of parantheses, so that the <selector> will always be the third element of the list.

---

[11] typically by inheritance from some parent in a Class hierarchy

4.2.3.1. Compilation of Messages

When GLISP encounters a message statement, it looks up the <selector> in the MSG definition of the <type> of the object to which the message is sent.[12] Each <selector> is paired with the appropriate <response> in a list, (<selector> <response>).[13] Code is compiled depending on the form of the <response> associated with the <selector>, as follows:

1. If the <response> is an atom, that atom is taken as the name of a function which is to be called in response to the message. The code which is compiled is a direct call to this function,

   ```
   (<response> <object> <arg1> ... <argn>)
   ```

2. If the <response> is a list, the contents of the list are recursively compiled in-line as GLISP code, with the name "self" artificially "bound" to the <object> to which the message was sent. Because the compilation is recursive, a message may be defined in terms of other messages, substructures, or properties, which may themselves be defined as messages.[14] The outer pair of parentheses of the <response> serves only to bound its contents; thus, if the <response> is a function call, the function call must be enclosed in an additional set of parentheses.

The following examples illustrate the various ways of defining message responses.

```
(MYMESSAGE    MYRESPONSEFN)

(SUCCESSOR    (self + 1))

(MAGNITUDE    ((SQRT X*X + Y*Y)))
```

In the first example, a message with <selector> MYMESSAGE is compiled as a direct call to the function MYRESPONSEFN. In the second example, the SUCCESSOR message is compiled as the sum of the object receiving the message (represented by "self") and the constant 1; if the object receiving the message is bound to the variable J and has a structure of type INTEGER, the code generated for the SUCCESSOR would be `(ADD1 J)`. The third example illustrates a call to a function, SQRT, with arguments containing definite references to X and Y (which presumably are defined as part of the object whose MAGNITUDE is sought). Note that since MAGNITUDE is defined by a function call, an extra pair of parentheses is required around the function call to distinguish it from in-line code.

The user can determine whether a message is to be compiled open or compiled as a function call by the way

---

[12] If the <type> of the destination object is unknown, or if the <selector> cannot be found, GLISP compiles the (SEND ...) statement as if it is a normal function call.

[13] If an appropriate representation language is provided, the <selector> and its associated <response> may be inherited from a parent class in a class hierarchy.

[14] Such recursive definitions must of course be acyclic.

in which the <response> is specified in the knowledge base. Open compilation operates like macro expansion; since the "macro" is a GLISP expression, it is easy to define messages and properties in terms of other messages and properties. The ability to use definite reference in GLISP makes the definition and use of the "macros" simple and natural.

4.2.3.2. Compilation of Properties and Adjectives

Properties, Adjectives, and ISA-adjectives are compiled in the same way as Messages. Since the syntax of use of properties and adjectives does not permit specification of any arguments, the only argument available to code or a function which implements the `<response>` for a property or adjective is the `self` argument, which denotes the object to which the property or adjective applies. A `<response>` which is written directly as GLISP code will use the name `self` directly[15], as in the SUCCESSOR example above; a function which is specified as the `<response>` will be called with the object as its single argument.

4.2.3.3. Declarations for Message Compilation

Declarations which affect compilation of Messages, Adjectives, or Properties may be specified following the <response> for a given message; such declarations are in (INTERLISP) property-list format, $<prop_1><value_1>$ ... $<prop_n><value_n>$. The following declarations may be specified:

1. RESULT `<type>`
   This declaration specifies the *type* of the result of the message or other property. Specification of result types helps the compiler to perform type inference, thus reducing the number of type declarations needed in user programs. The RESULT type for simple GLISP expressions will be inferred by the compiler; the RESULT declaration should be used if the `<response>` is a complex GLISP expression or a function name.[16]

2. OPEN T
   This declaration specifies that the function which is specified as the <response> is to be compiled OPEN at each reference. A <response> which is a list of GLISP code is always compiled OPEN; however, such a <response> can have only the "self" argument. If it is desired to compile OPEN a Message <response> which has arguments besides "self", the <response> must be coded as a function (in order to bind the arguments) and the OPEN declaration must be used. Functions which are compiled OPEN may not be recursive via any chain of OPEN-compiled functions.

---

[15] The name `self` is "declared" by the compiler, and does not have to be specified in the Structure Description.

[16] Alternatively, the result of a function may be specified by the RESULT declaration within the function itself.

4.2.3.4. Operator Overloading

GLISP provides operator overloading for user-defined objects using the Message facility. If an arithmetic operator is defined as the *selector* of a message for a user datatype, an arithmetic subexpression using that operator will be compiled as if it were a message call with two arguments. For example, the type VECTOR might have the declarations and function definitions below:

```
(VECTOR  (CONS (X INTEGER) (Y INTEGER))
   MSG  ((+  VECTORPLUS OPEN T)
         (←+ VECTORINCR OPEN T)) )


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

(VECTORINCR (GLAMBDA (U,V:VECTOR)
    (U:X ←+ V:X)
    (U:Y ←+ V:Y) ))
```

With these definitions, an expression involving the operators + or ←+ will be compiled by open compilation of the respective functions.

The compound operators (←+ +← ←- -←) are thought of as "destructive replacement" operators; thus, the expression (U ← U + V) will create a new VECTOR structure and assign the new structure to U, while the expression (U ←+ V) will smash the existing structure U, given the definitions above. The convention of letting the compound operators specify "destructive replacement" allows the user to specify both the destructive and non-destructive cases. However, if the compound operators are not overloaded but the arithmetic operators + and - are overloaded, the compound operators are compiled using the definitions of + for ←+ and +←, and - for ←- and -←. Thus, if only the + operator were overloaded for VECTOR, the expression (U ←+ V) would be compiled as if it were (U ← U + V).

BLANK PAGE

5. Context Rules and Reference

The ability to use definite reference to features of objects which are in *Context* is the key to much of GLISP's power. At the same time, definite reference introduces the possibility of ambiguity, i.e., there might be more than one object in context which has the specified feature. In this chapter, guidelines are presented for use of definite reference which allow the user to avoid harmful ambiguity.

5.1. Organization of Context

The Context maintained by the compiler is organized in levels, each of which may have multiple entries; in most cases, the sequence of levels is best thought of as a stack. Searching of the Context proceeds from the top (nearest) level of the stack to the bottom (farthest) level. The bottom level of the stack is composed of the LAMBDA variables of the function being compiled. New levels are added to the Context in the following cases:

1. When a PROG is compiled. The PROG variables are added to the new level.

2. When a For loop is compiled. The "loop index" variable (which may be either a user variable or a compiler variable) is added to the new level, so that it is in context during the loop.

3. When a While loop is compiled.

4. When a new clause of an If statement is compiled.

When a Message, Property, or Adjective is compiled, that compilation takes place in a new context consisting only of the " `self` " argument and other message arguments.

5.2. Rules for Using Definite Reference

The possibility of referential ambiguity is disturbing to many programmers; however, this problem is easily controlled in practice. First, it should be noted that the traditional methods of unique naming and complete path specification ("PASCAL style") are available, and should be used whenever there is any possibility of ambiguity. Second, there are several cases which are guaranteed to be unambiguous:

1. In compiling GLISP code which implements a Message, Property, or Adjective, only the " `self` " argument is in context initially; definite reference to any substructure or property of the object is therefore unambiguous.[17]

2. Within a For loop, the loop variable is the closest thing in context.

---

[17] Unless there are duplicated names in the object definition. However, if the same name is used as both a Property and an Adjective, for example, it is not considered a duplicate since Properties and Adjectives are specified by different source language constructs.

3. In many cases, a function will only have a single structured argument; in such cases, definite reference is unambiguous.

If "PASCAL" syntax (or the equivalent English-like form) is used for references other than the above cases, no ambiguities will occur.

5.3. Type Inference

In order to interpret definite references to features of objects, the compiler must know the *types* of the objects. However, explicit type specification can be burdensome, and makes it difficult to change types without rewriting existing type declarations. The GLISP compiler performs type inference in many cases, relieving the programmer of the burden of specifying types explicitly. The following rules enable the programmer to know when types will be inferred by the compiler.

1. Whenever a variable is set to a value whose type is known using the ← operator (or one of its variants), the type of the variable is inferred to be the type of the value to which it was set.

2. If a variable whose initial value was NIL (e.g., a PROG variable) appears on the left-hand side of the ←+ operator, its type is inferred to be `(LISTOF <type>)`, where `<type>` is the type of the right-hand side of the ←+ expression.

3. Whenever a substructure of a structured object is retrieved, the type of the substructure is retrieved also.

4. Types of infix expressions are inferred.

5. Types of Properties, Adjectives, and Messages are inferred if:

   a. The `<response>` is GLISP code whose type can be inferred.

   b. The `<response>` has a RESULT declaration associated with it.

   c. The `<response>` is a function whose definition includes a RESULT declaration, or whose property list contains a GLRESULTTYPE declaration.

6. The type of the "loop variable" in a **For** loop is inferred.

7. If an **If** statement tests the type of a variable using a " `self` " adjective, the variable is inferred to be of that type if the test is satisfied. Similar type inference is performed if the test of the type of the variable is the condition of a **While** statement.

8. When possible, GLISP infers the type of the function it is compiling and adds the type of the result to the property list of the function name under the indicator GLRESULTTYPE.

6. GLISP and Knowledge Representation Languages

GLISP provides a convenient *Access Language* which allows uniform specification of access to objects, without regard to the way in which the objects are actually stored; in addition, GLISP provides a basic *Representation Language*, in which the structures and properties of objects can be declared. The field of Artificial Intelligence has spawned a number of powerful Representation Languages, which provide power in describing large numbers of object classes by allowing hierarchies of *Class* descriptions, in which instances of Classes can inherit properties and procedures from parent Classes. The *Access Languages* provided for these Representation Languages, however, have typically been rudimentary, often being no more than variations of LISP's GETPROP and PUTPROP. In addition, by performing inheritance of procedures and data values at runtime, these Representation Languages have often been computationally costly.

A marriage between GLISP and a Representation Language is particularly felicitous because the strengths of each can overcome the weaknesses of the other. The Representation Language, by permitting hierarchical descriptions of Classes, provides more powerful facilities than GLISP for describing large numbers of object Classes. In addition, the Representation Language can provide the ability to interpret at runtime those messages, procedures, and data values which cannot be resolved at compile time. GLISP provides a convenient and uniform language for accessing both objects in the Representation Language and LISP objects. In addition, GLISP can greatly improve the efficiency of programs which access the representations by performing lookup of procedures and data in the Class hierarchy *at compile time*. Finally, a LISP structure can be specified *as the way of implementing* instances of a Class in the Representation Language, so that while the objects in such a class appear the same as other objects in the Representation Language and are accessed in the same way, they are actually implemented as LISP objects which are efficient in both time and storage.

A clean[18] interface between GLISP and a Representation Language is provided. With such an interface, each *Class* in the Representation Language is acceptable as a GLISP *type*. When the program which is being compiled specifies an access to an object which is known to be a member of some Class, the interface module for the Representation Language is called to generate code to perform the access. The interface module can perform inheritance within the Class hierarchy, and can call GLISP compiler functions to compile code for subexpressions. Properties, Adjectives, and Messages in GLISP format can be added to Class definitions, and can be inherited by subclasses at compile time. In an Object-Centered representation language or other representation language which relies heavily on procedural inheritance, substantial improvements in execution speed can be achieved by performing the inheritance lookup at compile time and compiling direct

---

[18] Cleanliness is in the eye of the beholder and, being next to Godliness, difficult to attain. However, it's *relatively* clean.

procedure calls to inherited procedures when the procedures are static and the type of the object which inherits the procedure is known at compile time.

Specifications for an interface module for GLISP are contained in a separate document[19]. To date, GLISP has been interfaced to our own GIRL representation language, and to LOOPS.[20]

---

[19] to be written.

[20] LOOPS, a LISP Object Oriented Programming System, is being developed at Xerox Palo Alto Research Center by Dan Bobrow and Mark Stefik.

7. GLISP Hacks

This chapter discusses some ways of doing things in GLISP which might not be entirely obvious at first glance.

7.1. Overloading Basic Types

GLISP provides the ability to define properties of structures described in the Structure Description language; since the elementary LISP types are structures in this language, objects whose storage representation is an elementary type can be "overloaded" by specifying properties and operators for them. The following examples illustrate how this can be done.

```
(GLDEFSTRQ


(ArithmeticOperator  (self ATOM)

   PROP ((Precedence OperatorPrecedenceFn · RESULT INTEGER)
         (PrintForm  ((GETPROP self 'PRINTFORM) or self)) )

   MSG  ((PRIN1      ((PRIN1 the PrintForm)))) )


(IntegerMod7         (self INTEGER)

   PROP ((Modulus    (7))
         (Inverse    ((If self is ZERO then 0
                             else (Modulus - self))) ))

   ADJ  ((Even       ((ZEROP (LOGAND self 1))))
         (Odd        (NOT Even)))

   ISA  ((Prime      PrimeTestFn))

   MSG  ((+          IMod7Plus  OPEN T  RESULT IntegerMod7)
         (←          IMod7Store OPEN T  RESULT IntegerMod7)) )

)
(DEFINEQ

(IMod7Store  (GLAMBDA (LHS:IntegerMod7 RHS:INTEGER)
        (LHS:self ←← (IREMAINDER RHS Modulus)) ))

(IMod7Plus   (GLAMBDA (X,Y:IntegerMod7)
        (IREMAINDER (X:self + Y:self) X:Modulus) ))
)
```

A few subtleties of the function IMod7Store are worth noting. First, the left-hand-side expression used in storing the result is LHS:self rather than simply LHS. LHS and LHS:self of course refer to the same actual structure; however, the *type* of LHS is IntegerMod7, while the type of LHS:self is INTEGER. If LHS were

used on the left-hand side, since the ← operator is overloaded for IntegerMod7, the function IMod7Store would be invoked again to perform its own function; since the function is compiled OPEN, this would be an infinite loop. A second subtlety is that the assignment to LHS:self must use the self-assignment operator, ←← , since it is desired to perform assignment as seen "outside" the function IMod7Store, i.e., in the environment in which the original assignment operation was specified.

7.2. Disjunctive Types

LISP programming often involves objects which may in fact be of different types, but which are for some purposes treated alike. For example, LISP data structures are typically constructed of CONS cells whose fields may point to other CONS cells or to ATOMs. The GLISP Structure Description language does not permit the user to specify that a certain field of a structure is a CONS cell *or* an ATOM. However, it is possible to create a GLISP datatype which encompasses both. Typically, this is done by declaring the structure of the object to be the complex structure, and testing for the simpler structure explicitly. This is illustrated for the case of the LISP tree below.

```
(LISPTREE  (CONS (CAR LISPTREE) (CDR LISPTREE))

  ADJ    ((EMPTY     (~self)))

  PROP   ((LEFTSON   ((If self is ATOMIC then NIL else CAR)))
          (RIGHTSON  ((If self is ATOMIC then NIL else CDR))) ))
```

7.3. Generators

Often, one would like to define such properties of an object as the way of enumerating its parts in some order. Such things cannot be specified directly as properties of the object because they depend on the previous state of the enumeration. However, it is possible to define an object, associated with the original datatype, which contains the state of the enumeration and responds to Messages. This is illustrated below by an object which searches a tree in Preorder.

```
(PreorderSearchRecord  (CONS (Node LISPTREE)
                             (PreviousNodes (LISTOF LISPTREE)))

   MSG  ((NEXT  ((PROG (TMP)
                   (If TMP←Node:LEFTSON
                       then (If Node:RIGHTSON
                                 then PreviousNodes+←Node)
                            Node←TMP
                       else TMP-←PreviousNodes
                            Node←TMP:RIGHTSON) ))))


(TP (GLAMBDA ((A LISPTREE))
      (PROG (PSR)
         (PSR ← (A PreorderSearchRecord
                   with Node = (the LISPTREE)))
         (While Node (If Node is ATOMIC (PRINT Node))
                     (← PSR NEXT)) )))
```

The object class PreorderSearchRecord serves two purposes: it holds the state of the enumeration, and it responds to messages to step through the enumeration. With these definitions, it is easy to write a program involving enumeration of a LISPTREE, as illustrated by the example function TP above. By being open-compiled, messages to an object can be as efficient as in-line hand coding; yet, the code for the messages only has to be written once, and can easily be changed without changing the programs which use the messages.

BLANK PAGE

8. Program Examples

In this chapter, examples of GLISP object declarations and programs are presented. Each example is discussed as a section of this chapter; the code for the examples and the code produced by the compiler are shown for each example at the end of the chapter.

8.1. GLTST1 File

The GLTST1 file illustrates the use of several types of LISP structures, and the use of fairly complex Property definitions for objects. SENIORITY of an EMPLOYEE, for example, is defined in terms of the YEAR of DATE-HIRED, which is a substructure of EMPLOYEE, and the YEAR of the function (CURRENTDATE).[21]

8.2. GLTST2 File

The GLTST2 file illustrates the use of Messages for ordinary LISP objects. By defining the arithmetic operators as Message selectors for the object VECTOR, use of vectors in arithmetic expressions is enabled; OPEN compilation is specified for these messages.

The definition of GRAPHICSOBJECT uses VECTORs as components. While the actual structure of a GRAPHICSOBJECT is simple, numerous properties are defined for user convenience. The definition of CENTER is easily stated as a VECTOR expression.

The Messages of GRAPHICSOBJECT illustrate how different responses to a message for different types of objects can be achieved, even though for GLISP compilation of messages to LISP objects the code for a message must be resolved at compile time.[22] The DRAW and ERASE messages get the function to be used from the property list of the SHAPE name of the GRAPHICSOBJECT and APPLY it to draw the desired object.

MOVINGGRAPHICSOBJECT contains a GRAPHICSOBJECT as a TRANSPARENT component, so that it inherits the properties of a GRAPHICSOBJECT; a MOVINGGRAPHICSOBJECT is a GRAPHICSOBJECT which has a VELOCITY, and will move itself by the amount of its velocity upon the

---

[21] The *type* of (CURRENTDATE) must be known to the compiler, either by compiling it first, or by including a RESULT declaration in the function definition of CURRENTDATE, or by specifying the GLRESULTTYPE property for the function name.

[22] For objects in a Representation Language, messages may be compiled directly as LISP code or as messages to be interpreted at runtime, depending on how much is known about the object to which the message is sent and the compilation declarations in effect.

message command STEP.[23] The compilation of the message `(← MGO STEP)` in the function TESTFN1 is of particular interest. This message is expanded into the sending of the message `(← self MOVE VELOCITY)` to the MOVINGGRAPHICSOBJECT. The MOVINGGRAPHICSOBJECT cannot respond to such a message; however, since it contains a GRAPHICSOBJECT as a TRANSPARENT component, its GRAPHICSOBJECT responds to the message.[24] A GRAPHICSOBJECT responds to a MOVE message by erasing itself, increasing its START point by the (vector) distance to be moved, and then redrawing itself. All of the messages are specified as being compiled open, so that the short original message actually generates a large amount of code.

A rectangle is drawn by the function DRAWRECT. Note how the use of the properties defined for a GRAPHICSOBJECT allows an easy interface to the system functions MOVETO and DRAWTO in terms of the properties LEFT, RIGHT, TOP, and BOTTOM.

---

[23] This example is adapted from the MovingPoint example written by Dan Bobrow for LOOPS.

[24] TRANSPARENT substructures thus permit procedural inheritance by LISP objects.

```
(FILECREATED "30-NOV-81 17:09:17" {DSK}GLTST1.LSP;8 2199

     changes to:  GLTST1COMS CURRENTDATE (CURRENTDATE GLRESULTTYPE) (COMPANY1 ELECTRICIANS) (COMPANY1
PRESIDENT) GIVE-RAISE

     previous date: "30-NOV-81 16:48:16" {DSK}GLTST1.LSP;6)


(PRETTYCOMPRINT GLTST1COMS)

(RPAQQ GLTST1COMS ((FNS GIVE-RAISE CURRENTDATE)
                   (GLISPOBJECTS EMPLOYEE DATE COMPANY)
                   (PROP GLRESULTTYPE CURRENTDATE)
                   (PROP ALL COMPANY1)))
(DEFINEQ

(GIVE-RAISE
  (GLAMBDA ((A COMPANY))                                          (* edited: "30-NOV-81 17:05")
           (FOR EACH ELECTRICIAN WHO IS NOT A TRAINEE
              DO (SALARY ←+(IF SENIORITY > 1
                              THEN 2.5
                            ELSE 1.5))
                 (PRINTOUT NIL (THE NAME OF THE ELECTRICIAN)
                           ,
                           (THE PRETTYFORM OF DATE-HIRED)
                           , MONTHLY-SALARY T))))

(CURRENTDATE
  (GLAMBDA NIL                                                    (* edited: "30-NOV-81 16:33")
           (A DATE WITH YEAR = 1981 , MONTH = 11 , DAY = 30)))
)

[GLDEFSTRQ

(EMPLOYEE

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

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

   ADJ    ((MALE (SEX = 'MALE))
           (FEMALE (SEX = 'FEMALE)))

   ISA    ((TRAINEE (TRAINEE))
           (GREENHORN (TRAINEE AND SENIORITY < 2)))

   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)))  )
(COMPANY

   [ATOM (PROPLIST (PRESIDENT (AN EMPLOYEE))
                   (ELECTRICIANS (LISTOF EMPLOYEE]  )
]
```

```
(PUTPROPS CURRENTDATE GLRESULTTYPE DATE)
  (PUTPROPS COMPANY1 ELECTRICIANS (("Cookie Monster" (7 21 1947)
                                                      12.5 NIL MALE)
                                   ("Betty Lou" (5 14 1980)
                                                 9.0 NIL FEMALE)
                                   ("Grover" (6 13 78)
                                              3.0 T MALE))
                    PRESIDENT ("Oscar the Grouch" (3 15 1907)
                                                   88.0 NIL MALE))
(DECLARE: DONTCOPY
  (FILEMAP (NIL (440 1003 (GIVE-RAISE 450 . 832) (CURRENTDATE 834 . 1001)))))
STOP
```

Compiled Versions:

```
←GLED(GIVE-RAISE)
edit
*PP
[EXPR (GLAMBDA ((A COMPANY))  **COMMENT**
               (FOR EACH ELECTRICIAN WHO IS NOT A TRAINEE
                  DO (SALARY ←+(IF SENIORITY > 1
                                   THEN 2.5
                                 ELSE 1.5))
                     (PRINTOUT NIL (THE NAME OF THE ELECTRICIAN)
                               ,
                               (THE PRETTYFORM OF DATE-HIRED)
                               , MONTHLY-SALARY T)))
      GLRESULTTYPE NIL GLCOMPILED
      (LAMBDA (GLVAR1)
        (MAPC (GETPROP GLVAR1 (QUOTE ELECTRICIANS))
              (FUNCTION (LAMBDA (GLVAR2)
                  (COND
                    ((NOT (CADDDR GLVAR2))
                      [RPLACA (CDDR GLVAR2)
                              (PLUS (CADDR GLVAR2)
                                    (COND
                                      ((IGREATERP (IDIFFERENCE (CADDR (CURRENTDATE))
                                                               (CADDR (CADR GLVAR2)))
                                                  1)
                                        2.5)
                                      (T 1.5]
                      (PRINTOUT
                        NIL
                        (CAR GLVAR2)
                        ,
                        [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]
                        ,
                        (TIMES (CADDR GLVAR2)
                               174)
                        T]
```

```
←GLED(CURRENTDATE]
edit
*PP
[GLRESULTTYPE DATE EXPR
              (GLAMBDA NIL   **COMMENT**
                       (A DATE WITH YEAR = 1981 , MONTH = 11 , DAY = 30))
              GLCOMPILED
              (LAMBDA NIL
                (APPEND (QUOTE (11 30 1981]
```

```
(FILECREATED "11-JAN-82 16:16:35" {DSK}GLTST2.LSP;8 4578

     changes to:  VECTORMOVE DRAWRECT TESTFN1 GRAPHICSOBJECTMOVE

     previous date: " 2-DEC-81 14:35:01" {DSK}GLTST2.LSP;4)


(PRETTYCOMPRINT GLTST2COMS)

(RPAQQ GLTST2COMS [(GLISPOBJECTS VECTOR GRAPHICSOBJECT MOVINGGRAPHICSOBJECT)
        (FNS VECTORPLUS VECTORDIFF VECTORTIMES VECTORQUOTIENT VECTORMOVE GRAPHICSOBJECTMOVE
             MGO-ACCELERATE TESTFN1 TESTFN2 DRAWRECT)
        (PROP DRAWFN RECTANGLE)
        (P (SETQ MYWINDOW
                 (CREATEW (create REGION LEFT ← 400 BOTTOM ← 400 WIDTH ← 400 HEIGHT ← 400)
                          "GraphicsWindow"))
           (SETQ DSPS (WINDOWPROP MYWINDOW (QUOTE DSP])

[GLDEFSTRQ

(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)
           (- VECTORDIFF OPEN T)
           (* VECTORTIMES OPEN T)
           (/ VECTORQUOTIENT OPEN T)
           (←+ VECTORMOVE OPEN T)
           (PRIN1 ((PRIN1 "(")
                   (PRIN1 X)
                   (PRIN1 ",")
                   (PRIN1 Y)
                   (PRIN1 ")")))
           (PRINT ((← self PRIN1)
                   (TERPRI]  )

(GRAPHICSOBJECT

   (LIST (SHAPE ATOM)
         (START VECTOR)
         (SIZE VECTOR))

   PROP   ((LEFT (START:X))
           (BOTTOM (START:Y))
           (RIGHT (LEFT+WIDTH))
           (TOP (BOTTOM+HEIGHT))
           (WIDTH (SIZE:X))
           (HEIGHT (SIZE:Y))
           (CENTER (START+SIZE/2))
           (AREA (WIDTH*HEIGHT)))

   MSG    ([DRAW ((APPLY* (GETPROP SHAPE 'DRAWFN)
                          self
                          (QUOTE PAINT]
           [ERASE ((APPLY* (GETPROP SHAPE 'DRAWFN)
                           self
                           (QUOTE ERASE]
           (MOVE GRAPHICSOBJECTMOVE OPEN T))  )

(MOVINGGRAPHICSOBJECT

   (LIST (TRANSPARENT GRAPHICSOBJECT)
         (VELOCITY VECTOR))

   MSG    [(ACCELERATE MGO-ACCELERATE OPEN T)
           (STEP ((← self MOVE VELOCITY]  )
```

```
(DEFINEQ

(VECTORPLUS
  (GLAMBDA (V1,V2:VECTOR)
           (A VECTOR WITH X = V1:X + V2:X , Y = V1:Y + V2:Y)))

(VECTORDIFF
  (GLAMBDA (V1,V2:VECTOR)
           (A VECTOR WITH X = V1:X - V2:X , Y = V1:Y - V2:Y)))

(VECTORTIMES
  (GLAMBDA (V:VECTOR N:NUMBER)
           (A VECTOR WITH X = X*N , Y = Y*N)))

(VECTORQUOTIENT
  (GLAMBDA (V:VECTOR N:NUMBER)
           (A VECTOR WITH X = X/N , Y = Y/N)))

(VECTORMOVE
  (GLAMBDA (V,DELTA:VECTOR)                                  (* edited: "11-JAN-82 12:35")
           (V:X ←+
                DELTA:X)
           (V:Y ←+
                DELTA:Y)))

(GRAPHICSOBJECTMOVE
  (GLAMBDA (self:GRAPHICSOBJECT DELTA:VECTOR)                (* edited: "11-JAN-82 16:07")
           (← self ERASE)
           (START ←+
                  DELTA)
           (← self DRAW)))

(MGO-ACCELERATE
  (GLAMBDA (SELF: MOVINGGRAPHICSOBJECT ACCELERATION: VECTOR)
           VELOCITY ←+
           ACCELERATION))

(TESTFN1
  (GLAMBDA NIL
                                                             (* edited: "11-JAN-82 15:22")
                                                             (* Make a moving graphics object and step across the
                                                             screen.)
           (PROG (MGO N)
                 (MGO ←(A MOVINGGRAPHICSOBJECT WITH SHAPE =(QUOTE RECTANGLE)
                          , SIZE =(A VECTOR WITH X = 4 , Y = 3)
                          , VELOCITY =(A VECTOR WITH X = 3 , Y = 4)))
                 (N ← 0)
                 (WHILE (N←+1)
                        <100 (← MGO STEP))
                 (← (the START of MGO)
                    PRINT))))

(TESTFN2
  (GLAMBDA ((A GRAPHICSOBJECT))                              (* edited: " 1-DEC-81 09:46")
           (LIST SHAPE                                       (* SHAPE)
                 START                                       (* START)
                 SIZE                                        (* SIZE)
                 LEFT                                        (* LEFT)
                 BOTTOM                                      (* BOTTOM)
                 RIGHT                                       (* RIGHT)
                 TOP                                         (* TOP)
                 WIDTH                                       (* WIDTH)
                 HEIGHT                                      (* HEIGHT)
                 CENTER                                      (* CENTER)
                 AREA                                        (* AREA)
                 )))

(DRAWRECT
  (GLAMBDA ((A GRAPHICSOBJECT)
            DSPOP:ATOM)                                      (* edited: "11-JAN-82 12:40")
           (PROG (OLDDS)
                 (OLDDS ←(CURRENTDISPLAYSTREAM DSPS))
                 (DSPOPERATION DSPOP)
                 (MOVETO LEFT BOTTOM)
```

```
                (DRAWTO LEFT TOP)
                (DRAWTO RIGHT TOP)
                (DRAWTO RIGHT BOTTOM)
                (DRAWTO LEFT BOTTOM)
                (CURRENTDISPLAYSTREAM OLDDS))))
)

(PUTPROPS RECTANGLE DRAWFN DRAWRECT)
(SETQ MYWINDOW (CREATEW (create REGION LEFT ← 400 BOTTOM ← 400 WIDTH ← 400 HEIGHT ← 400)
                        "GraphicsWindow"))
(SETQ DSPS (WINDOWPROP MYWINDOW (QUOTE DSP)))
(DECLARE: DONTCOPY
  (FILEMAP (NIL (1743 4357 (VECTORPLUS 1753 . 1850) (VECTORDIFF 1852 . 1949) (VECTORTIMES 1951 . 2038) (
VECTORQUOTIENT 2040 . 2130) (VECTORMOVE 2132 . 2290) (GRAPHICSOBJECTMOVE 2292 . 2474) (MGO-ACCELERATE
2476 . 2591) (TESTFN1 2593 . 3140) (TESTFN2 3142 . 3932) (DRAWRECT 3934 . 4355)))))
STOP
```

Compiled Versions:

```
←GLED(TESTFN1)
edit
*PP
[GLRESULTTYPE NIL GLCOMPILED
            [LAMBDA NIL
              (PROG (MGO N)
                    [SETQ MGO (COPY (QUOTE ((RECTANGLE (0 0)
                                                       (4 3))
                                            (3 4]
                    (SETQ N 0)
                    [PROG NIL
                      GLLABEL1
                          (COND
                            ((ILESSP (SETQ N (ADD1 N))
                                     100)
                              (PROG ((DELTA (CADR MGO)))
                                    (APPLY* (GETPROP (CAAR MGO)
                                                     (QUOTE DRAWFN))
                                            (CAR MGO)
                                            (QUOTE ERASE))
                                    [PROG ((V (CADAR MGO)))
                                          (RPLACA V (IPLUS (CAR V)
                                                           (CAR DELTA)))
                                          (RPLACA (CDR V)
                                                  (IPLUS (CADR V)
                                                         (CADR DELTA]
                                    (APPLY* (GETPROP (CAAR MGO)
                                                     (QUOTE DRAWFN))
                                            (CAR MGO)
                                            (QUOTE PAINT)))
                              (GO GLLABEL1]
                    (PROG ((self (CADAR MGO)))
                          (PRIN1 "(")
                          (PRIN1 (CAR self))
                          (PRIN1 ",")
                          (PRIN1 (CADR self))
                          (PRIN1 ")")
                          (TERPRI]
            CODE {CCODEP}#6,73404 EXPR
            (GLAMBDA NIL   **COMMENT**    **COMMENT**
                (PROG (MGO N)
                      (MGO ←(A MOVINGGRAPHICSOBJECT WITH SHAPE =(QUOTE
                                                                 RECTANGLE)
                            , SIZE =(A VECTOR WITH X = 4 , Y = 3)
                            , VELOCITY =(A VECTOR WITH X = 3 , Y = 4)))
                      (N ← 0)
                      (WHILE (N←+1)
                             <100 (← MGO STEP))
                      (← (the START of MGO)
                         PRINT]
```

```
←GLED(TESTFN2]
edit
*pp*
[GLRESULTTYPE NIL GLCOMPILED
              [LAMBDA (GLVAR1)
                (LIST (CAR GLVAR1)                   (* SHAPE)
                      (CADR GLVAR1)                  (* START)
                      (CADDR GLVAR1)                 (* SIZE)
                      (CAADR GLVAR1)                 (* LEFT)
                      (CADADR GLVAR1)                (* BOTTOM)
                      (IPLUS (CAADR GLVAR1)
                             (CAADDR GLVAR1))        (* RIGHT)
                      (IPLUS (CADADR GLVAR1)
                             (CADR (CADDR GLVAR1)))
                                                     (* TOP)
                      (CAADDR GLVAR1)                (* WIDTH)
                      (CADR (CADDR GLVAR1))          (* HEIGHT)
                      [PROG [(V1 (CADR GLVAR1))
                             (V2 (PROG ((V (CADDR GLVAR1)))
                                       (RETURN (LIST (QUOTIENT (CAR V)
                                                               2)
                                                     (QUOTIENT (CADR V)
                                                               2]
                            (RETURN (LIST (IPLUS (CAR V1)
                                                 (CAR V2))
                                          (IPLUS (CADR V1)
                                                 (CADR V2]
                                                     (* CENTER)
                      (ITIMES (CAADDR GLVAR1)
                              (CADR (CADDR GLVAR1)))
                                                     (* AREA)
                      ]
```

```
←GLED(DRAWRECT]
edit
*PP
[GLRESULTTYPE NIL GLCOMPILED [LAMBDA (GLVAR1 DSPOP)
              (PROG (OLDDS)
                    (SETQ OLDDS (CURRENTDISPLAYSTREAM DSPS))
                    (DSPOPERATION DSPOP)
                    (MOVETO (CAADR GLVAR1)
                            (CADADR GLVAR1))
                    [DRAWTO (CAADR GLVAR1)
                            (IPLUS (CADADR GLVAR1)
                                   (CADR (CADDR GLVAR1]
                    [DRAWTO (IPLUS (CAADR GLVAR1)
                                   (CAADDR GLVAR1))
                            (IPLUS (CADADR GLVAR1)
                                   (CADR (CADDR GLVAR1]
                    (DRAWTO (IPLUS (CAADR GLVAR1)
                                   (CAADDR GLVAR1))
                            (CADADR GLVAR1))
                    (DRAWTO (CAADR GLVAR1)
                            (CADADR GLVAR1))
                    (CURRENTDISPLAYSTREAM OLDDS]
        CODE {CCODEP}#6,73424 EXPR (GLAMBDA ((A GRAPHICSOBJECT)
                                            DSPOP:ATOM)  **COMMENT**
                                           (PROG (OLDDS)
                                                 (OLDDS ←(CURRENTDISPLAYSTREAM
                                                          DSPS))
                                                 (DSPOPERATION DSPOP)
                                                 (MOVETO LEFT BOTTOM)
                                                 (DRAWTO LEFT TOP)
                                                 (DRAWTO RIGHT TOP)
                                                 (DRAWTO RIGHT BOTTOM)
                                                 (DRAWTO LEFT BOTTOM)
                                                 (CURRENTDISPLAYSTREAM OLDDS]
```

Copyright © 1985 by KSL and
Comtex Scientific Corporation

FILMED FROM BEST AVAILABLE COPY