GLISP User's Manual

Technical Report TR-83-25
Gordon S. Novak Jr.
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

(512) 471-9569
novak@cs.utexas.edu

Revised:

1 Overview of GLISP

GLISP is a Lisp-based language that provides high-level language features not found in ordinary Lisp. The GLISP language is implemented by means of a compiler that accepts GLISP as input and produces ordinary Lisp as output; this output can be further compiled to machine code by the Lisp compiler. GLISP is available for several Lisp dialects, including Interlisp, Maclisp, UCI Lisp, ELISP, Franz Lisp, and Portable Standard Lisp.

The goal of GLISP is to make programming easier by allowing design decisions to be expressed in a single place and letting the compiler generate code according to those design decisions. This philosophy allows source program code to be shorter and more readable and allows design decisions to be changed without requiring that program code be rewritten.

GLISP provides both Pascal-like and English-like syntaxes; much of the power and brevity of GLISP derives from the compiler features necessary to support the relatively informal, English-like language constructs. The following example of a GLISP function illustrates the use of definite reference to structured objects.

```
(gldefun HourlySalaries ((D 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 object types and of the 'context' of the computation as the program is compiled. Features of objects that 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 example above, 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 data type 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.

3. 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).

4. Pascal-like control statements and loop constructs, such as (FOR EACH `<item>` WITH `<property>` DO . . .) , are compiled into Lisp code of the appropriate form.

5. Compilation of infix expressions is provided for the arithmetic operators and for additional operators that facilitate list manipulation. Operators are interpreted appropriately for Lisp data types as well as for numbers; operator overloading for user-defined objects is provided by the message facility.

6. 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 run time, resulting in much faster execution.

7. By separating object definitions from the code that references objects, GLISP permits radical changes to object structures with no changes to code.

2 Implementation

GLISP is implemented by means of a compiler, which produces a normal Lisp EXPR from the GLISP code. When a GLISP function is compiled, the original GLISP code is saved on the function's property list, and the compiled code is stored as the function definition. 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.

GLISP functions are defined in such a way that they are automatically compiled the first time they are referenced. In some Lisp dialects, GLISP functions are indicated by the use of GLAMBDA instead of LAMBDA in the function definition. In other dialects, a special defining function is used; this function sets up a macro definition which invokes compilation the first time the function is referenced. The user can explicitly invoke compilation of GLISP functions by calling the compiler function `GLCC` or `GLCP`.

To use GLISP, it is first necessary to load the compiler file into Lisp. Users' files containing structure descriptions and GLISP code are then loaded. Compilation of a GLISP function is requested by:

`(GLCC '<fn>)`	Compile the function `<fn>`.
`(GLCP '<fn>)`	Compile `<fn>` and 'prettyprint' the result.
`(GLP '<fn>)`	Print the compiled version of `<fn>`.

In Interlisp, all the GLISP functions (beginning with GLAMBDA) in a file can be compiled by invoking `(GLCOMPCOMS <file>COMS)`, where `<file>COMS` is the list of file package commands for the file.

Properties of compiled functions are stored on the property list of the function name:

`GLORIGINALEXPR`	Original (GLISP) version of the function.[1]
`GLCOMPILED`	GLISP-compiled version of the function.
`GLRESULTTYPE`	Type of the result of the function.
`GLARGUMENTTYPES`	Types of the arguments of the function.

Functions are provided to call the Lisp editor for examination and editing of GLISP functions and object descriptions:

`(GLED '<name>)`	Edit the property list of `<name>`.
`(GLEDF '<fn>)`	Edit the GLISP definition of `<fn>`.
`(GLEDS '<str>)`	Edit the structure `<str>`.

3 Error Messages

GLISP provides detailed error messages when compilation errors are detected; many careless errors, such as misspellings, will be caught by the compiler. When the source program contains errors, the compiled code generates run time errors upon execution of the erroneous expressions.

4 Interactive Features of GLISP

Several features of GLISP are available interactively as well as in compiled functions:

1. The `A` function, which creates structured objects from a readable property-value list, is available as an interactive function. See Section 13.
2. Messages to objects can be executed interactively. See Section 25

3. A display-oriented editor and inspector, GEV, is available for use with bitmap graphics terminals.[2] GEV interprets objects according to their GLISP structure descriptions; it allows the user to inspect objects, edit them, interactively construct programs that operate on them, display computed properties, send messages to objects, and zoom in on features of interest.

5 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 *properties* (i.e., data whose values are not stored but are computed from the values of stored data), adjectival predicates, and *messages* that 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 data-types (e.g., INTEGER) are automatically known to the compiler.
2. Structure descriptions (but not full object descriptions) may be used directly as *types* in function definitions.
3. The user may declare object descriptions to the system by the function GLISPOBJECTS; the names of the object types may then be used as types in function definitions and definitions of other structures.
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 by the function GLISPOBJECTS[3] , which takes one or more object descriptions as arguments (assuming the descriptions to be quoted). 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. The format of each description is as follows:

```
(<object name>
     <structure description>
          PROP    <property descriptions>
          ADJ     <adjective descriptions>
          ISA     <isa-adjective descriptions>
```

[2]GEV is currently implemented only for Xerox Lisp machines.

[3]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`$_n$`>)`

```
          MSG     <message descriptions>
          SUPERS  <list of superclasses>
          VALUES  <list of values>              )
```

The `<object name>` and `<structure description>` are required; the other property-value pairs are optional and may appear in any order. The following example illustrates some of the declarations that might be made to describe the object type `VECTOR`.

```
(GLISPOBJECTS

   (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)) )

      ISA    ( (UNITVECTOR (self IS NORMALIZED)) )

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


   (PLANET   (LISTOBJECT (NAME ATOM) (MASS REAL) (RADIUS REAL))

      SUPERS (PHYSICAL-OBJECT SPHERE) )
)
```

5.1 Property Descriptions

The PROP, ADJ, ISA, and MSG descriptions specify computed properties of the object; the four kinds of properties differ in the way in which they are referenced in GLISP program code. PROPerties are properties of an object which are computed rather than being stored directly; they are referenced within code in the same way as stored values. ADJ and ISA specify adjectival predicates which are used to test properties of objects within conditional statements (see Section 15). MSG properties specify messages which the object can receive (see Section 20). Messages may take arguments, and may have side-effects.

Each `<description>` specified with PROP, ADJ, ISA, or MSG has the following format:

```
(<name> <response> <prop1> <value1> ... <propn> <valuen>)
```

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 that affect compilation. All four kinds of properties are compiled in a similar fashion, as described in Section 21. The uses of each kind of property are described below.

5.1.1 PROP Descriptions

PROP descriptions specify *properties* of an object, that is, features of the object which are treated as if they were stored data but which are in fact computed from stored data. By convention, properties are assumed to be functional, that is, to have no side-effects.

5.1.2 ADJ and ISA Descriptions

ADJ and ISA descriptions specify adjectival *predicates* which may be used to test features of an object. The ISA form is used for nominal adjectives, that is, those which would normally be written with ‘a’ or ‘an’ in English. Adjectives are normally implemented by code that returns a BOOLEAN value.

5.1.3 MSG Descriptions

MSG descriptions specify *messages* which can be sent to an object. Messages can take arguments, and may have side effects.

5.2 SUPERS Description

The SUPERS list specifies a list of *superclasses*, that is, the names of other object descriptions from which the object may inherit PROP, ADJ, ISA, and MSG properties. Inheritance from superclasses can be recursive, as described in Section 21.

5.3 VALUES Description

The VALUES list is a list of pairs, `( <name> <value>)`, which is used to associate symbolic names with constant values for an object type. If VALUES are defined for the type of the *selector* of a CASE statement, the corresponding symbolic names may be used as the selection values for the clauses of the CASE statement.

6 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.

6.1 Syntax of Structure Descriptions

The syntax of structure descriptions is recursively defined in terms of basic types and composite types that are built up from basic types. The syntax of structure descriptions is as follows: [4]

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

SYMBOL	
INTEGER	
REAL	
NUMBER	(either INTEGER or REAL)
STRING	
BOOLEAN	(either T or NIL)
ANYTHING	(an arbitrary structure)

2. An object type that is known to the compiler, either from a GLISPOBJECTS 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>`). [5]

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. 'A', 'AN', and 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** $\texttt{<sd}_1\texttt{>}$ $\texttt{<sd}_2\texttt{>}$)
   The CONS of two structures whose descriptions are $\texttt{<sd}_1\texttt{>}$ and $\texttt{<sd}_2\texttt{>}$.

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

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

[4]The names of the basic types and the structuring operators must be all upper case or lower case, depending on the case which is usual for the underlying Lisp system. In general, other GLISP keywords and user program names may be in upper case, lower case, or mixed case, if mixed cases are permitted by the Lisp system.

[5]Whenever the form (**A . . .**) is allowed in GLISP, the form (**AN . . .**) is also allowed.

(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) (**PROPLIST** (`<name`$_1$`>` `<sd`$_1$`>`) . . . (`<name`$_n$`>` `<sd`$_n$`>`))
An association list in 'property-list format' (alternating names and values) in which the atom `<name`$_i$`>`, if present, is associated with a structure whose description is `<sd`$_i$`>`.

(f) (**ATOM** (**PROPLIST** (`<pname`$_1$`>` `<sd`$_1$`>`) . . . (`<pname`$_n$`>` `<sd`$_n$`>`))
(**BINDING** `<sd>`))
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.

(g) (**RECORD** `<record-name>` (`<name`$_1$`>` `<sd`$_1$`>`) ... (`<name`$_n$`>` `<sd`$_n$`>`))
RECORD specifies the use of contiguous records for data storage. `<record-name>` is the name of the record type; it is optional and is not used in some Lisp dialects.[6]

(h) (**TRANSPARENT** `<type>`)
An object of type `<type>` is incorporated into the structure being defined in *transparent mode*, which means that all fields and properties of the object of type `<type>` can be directly referenced as if they were properties of the object being defined. A substructure that is a named *type* and that is not declared to be TRANSPARENT is assumed to be opaque; that is, its internal structure cannot be seen unless an access path explicitly names the subrecord[7]. The object of type `<type>` may also contain TRANSPARENT objects; the graph of TRANSPARENT object references must, of course, be acyclic.

(i) (**OBJECT** (`<name`$_1$`>` `<sd`$_1$`>`) . . . (`<name`$_n$`>` `<sd`$_n$`>`))
(**ATOMOBJECT** (`<name`$_1$`>` `<sd`$_1$`>`) . . . (`<name`$_n$`>` `<sd`$_n$`>`))
(**LISTOBJECT** (`<name`$_1$`>` `<sd`$_1$`>`) . . . (`<name`$_n$`>` `<sd`$_n$`>`))
These declarations describe *objects*, that is, data structures which can receive messages at run time. These three types of objects are implemented as records, atoms, or lists, respectively. In each case, the system adds to the object a `CLASS` datum that points to the name of the type of the object. An object declaration may only appear as the top-level declaration of a named object type.

(j) (**TYPEOF** `<expression>`)
This declaration specifies the same type that `<expression>` would have if it were

[6]RECORDs are implemented using RECORDs in Interlisp, HUNKs in Maclisp and Franz Lisp, VECTORs in Portable Standard Lisp, and lists in UCI Lisp and ELISP. In Interlisp, appropriate RECORD declarations must be made to the system by the user in addition to the GLISP declarations.

[7]For example, a PROFESSOR record might contain some fields that are unique to professors, plus a pointer to an EMPLOYEE record. If the declaration in the PROFESSOR record were `(EMPREC (TRANSPARENT EMPLOYEE))`, then a field of the employee record, say, SALARY, could be referenced directly from a variable P that points to a PROFESSOR record as `(salary P`; if the declaration were `(EMPREC EMPLOYEE)`, it would be necessary to say `(salary (emprec P`.

evaluated. The TYPEOF declaration can only be used within functions (not within object descriptions), and it can only be used where `<expression>` is a valid expression whose type can be inferred by the compiler. TYPEOF is useful in writing *generic functions*, that is, abstract functions which are defined over a set of object types.

6.2 Examples of Structure Descriptions

The following examples illustrate the use of structure descriptions.

```
(GLISPOBJECTS

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

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

The first structure, CAT, is entirely composed of list structure. A CAT structure 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, the meaning of a subrecord name depends on the type of record from which the subrecord is retrieved. The subrecord AGE has two different meanings when applied to PERSONs and to CATs. The second structure, PERSON, illustrates a description of an object that 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 operator must be named, and the names appear in the actual structures. For example, if X is a PERSON structure, retrieval of the AGE of X is equivalent to `(GETPROP X 'AGE)`. A subrecord of a PROPLIST record can be referenced directly; for example, one can ask for the DOGS of a PERSON directly, without cognizance of the fact that DOGS is part of the PETS property. A substructure can have multiple names; for example, CATS and GATOS name the same subrecord of a PERSON.

7 Editing of Object Descriptions

An object description can be edited by calling `(GLEDS TYPE)`, where `TYPE` is the name of the object type. This will cause the Lisp editor to be called on the object description of `TYPE`.

8 Interactive Editing of Objects

An interactive structure inspector and editor, GEV, is available for the Xerox 1100-series Lisp machines. GEV allows the user to inspect and edit any structures that are described by GLISP object descriptions, to zoom in on substructures of interest, and to display the values of computed properties automatically or on demand. GEV is described in a separate document [?].

9 Global Variables

The types of free variables can be declared within the functions that reference them. Alternatively, the types of global variables can be declared to the compiler using the form:[8]

```
(GLISPGLOBALS  (<name> <type>) ... )
```

Following such a declaration, the compiler will assume that a free variable `<name>` is of the corresponding `<type>`. A GLOBAL object does not have to actually exist as a storage structure; for example, one could define a global object 'MOUSE' or 'SYSTEM' whose properties are actually implemented by calls to the operating system.

10 Compile Time Constants

The values and types of compile-time constants can be declared to the compiler using the form:[9]

[8] `(GLISPGLOBALS <name`$_1$`> ... <name`$_n$`>)` is defined as a file-package command for Interlisp.

[9] `(GLISPCONSTANTS <name`$_1$`> ... <name`$_n$`>)` is defined as a file-package command for Interlisp. The GLISPCONSTANTS expressions written to a file will contain the original `<value-expression>` entries

```
(GLISPCONSTANTS  (<name> <value-expression> <type>) ... )
```

The `<name>` and `<type>` fields are assumed to be quoted. The `<value-expression>` field is a GLISP expression that is parsed and evaluated; this allows constants to be defined by expressions involving previously defined constants.

11 Conditional Compilation

The GLISP compiler will perform many kinds of computations on constants at compile time, reducing the size of the compiled code and improving execution speed.[10] In particular, arithmetic, comparison, logical, conditional, and CASE function calls are optimized, with elimination of dead code. This permits conditional compilation in a clean form. Code can be written that tests the values of flags in the usual way; if the flag values are declared to be compile-time constants using GLISPCONSTANTS, the tests will be performed at compile time, and the unneeded code will vanish.

12 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 the object and a path specification that 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 that has not been explicitly named before but that 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 Section 26.

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, that

rather than their evaluated values.

[10]Ordinary Lisp functions are evaluated on constant arguments if the property `GLEVALWHENCONST` is set to T on the property list of the function name. This property is set by the compiler for the basic arithmetic functions.

is, a LAMBDA variable, PROG variable, or GLOBAL variable; the variable is assumed to point to (i.e., 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 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>` *or* `<property>`	`<var>`	The object denoted by `<var>`
`(<property> <var>)`	**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 SPECIALTY of the HEAD of the DEPARTMENT' or '(specialty (head (department))). 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.

13 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, '*with*', '=', and ',' are allowed for readability but may be omitted if desired;[11] if present, they must all be delimited on both sides by blanks. In response to such an expression, GLISP will generate code to create a new instance of the specified structure. The `<property>` names may be specified in any order. Unspecified properties are defaulted according to the following rules:

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

2. Composite structures are created from the defaults of their components, except that missing PROPLIST and ALIST items that would default to NIL are omitted.

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

[11] Some Lisp dialects, e.g., Maclisp, will interpret commas as 'backquote' commands and generate error messages. In such dialects, the commas must be omitted or be 'slashified', i.e., preceded by a slash. It may be desirable to avoid commas in all GLISP forms to aid transportability.

14 Interpretive Creation of Objects

The 'A' function is defined for interpretive use as well as for use within GLISP functions. For example, one could say:

```
(SETQ EARTH (A PLANET WITH MASS = 5.98E24 RADIUS = 6.37E6))
```

15 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 `~`[12], and grouping parentheses.

The compiler automatically recognizes the Lisp adjectives ATOMIC, NULL, NIL, INTEGER, REAL, ZERO, NUMERIC, NEGATIVE, MINUS, and BOUND and the ISA-adjectives ATOM, LIST, NUMBER, INTEGER, SYMBOL, STRING, ARRAY, and BIGNUM;[13] user definitions have precedence over these predefined adjectives.

15.1 Self-Recognition Adjectives

If the ISA-adjective `self` is defined for an object type, the type name may be used as an ISA-adjective to test whether a given object is a member of that type. Given a predicate phrase of the form '`X is a Y`', the compiler first looks at the definition of the object type 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 a type name, 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 (Send 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

[12]The operators 'NOT' and ' ' are equivalent.

[13]where applicable

FOO earlier. This feature is useful in implementing disjunctive types, as discussed in a later section.

15.2 Testing Object Classes

For those data types that are defined using one of the OBJECT structuring operators, the class name is automatically defined as an ISA-adjective. The ISA test is implemented by run-time examination of the CLASS datum of the object.

16 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: [14]

```
(<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 that calls it. The `<result-description>` is a standard structure description or `<type>`.

The GLOBAL declaration is used to inform the compiler of the types of free variables. The function GLISPGLOBALS can be used to declare the types of global variables, making GLOBAL declarations within individual functions unnecessary.

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>
 (<variable>  <type>)
            (A <type>)
            <type>    if <type> is a non-atomic structure description
```

[14]This diagram illustrates function syntax used within Interlisp. For other dialects, the defining syntax is similar to the defining syntax used for ordinary Lisp functions; see Section 35. The PROG, of course, is not required. In Lisp dialects other than Interlisp and Franz Lisp, LAMBDA may be used instead of GLAMBDA.

In addition to declared `<type>`s, a structure description may be used directly as a `<type>` in a variable declaration. The following examples illustrate legal forms of variable declarations:

```
X
(V VECTOR)
(x real)
(x (units real meter))
(c cat)
(A COMPANY)
(CONS (FIRST ATOM) (SECOND INTEGER))
(EXPR (CONS (FN ATOM) (ARGS (LISTOF ANYTHING))))
```

Type declarations are required only for variables whose subrecords or properties 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 that 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 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, by enclosing the variable and its initial value in a list:[15]

```
(PROG (X (N 0) (Y REAL)) ...)
```

In this example, the variable N will be initialized to the value 0 when the PROG is entered. The syntax of variable declarations does not permit the type of a variable and its initial value both to be specified.

17 Expressions

GLISP provides translation of infix expressions of the kind usually found in programming languages. It also provides additional operators that facilitate list manipulation and other operations. Overloading of operators for user-defined types is provided by means of the *message* facility.

Expressions may be written directly in-line within function calls, as in `(SQRT X*X + Y*Y)`, or they may be written as statements within parentheses, as in

```
(AREA = PI*R^2)
```

[15]This feature is available in all Lisp dialects.

Parentheses may be used for grouping in the usual way. Operators may be written with or without delimiting spaces,[16] *except for the '-' operator, which* **must** *be delimited by spaces.* [17] Expression parsing is done by an operator precedence parser, using the same precedence ordering as in FORTRAN. [18] The operators that are recognized are as follows:

Assignment	`_` *or* `:=`
Arithmetic	`+ - * / ^`
Comparison	`=` ~= `<> < <= > >=`
Logical	`AND OR NOT ~`
Compound	`\_+` _- +_ -_ *

The operator `_` is equivalent to the operator ':='; '~' is equivalent to 'NOT', and '~=' is equivalent to '<>'.

17.1 Interpretation of Operators

In addition to the usual interpretation of operators when used with numeric arguments, some of the operators are interpreted appropriately for other Lisp types.

17.1.1 Operations on Strings

For operands of type STRING, the operator '+' performs concatenation. All of the comparison operators are defined for STRINGs.

17.1.2 Operations on Lists

Several operators are defined in such a way that they perform set operations on lists of the form `(LISTOF <type>)`, where `<type>` is considered to be the element type. The following table shows the interpretations of the operators:

`<list> + <list>`	Set Union
`<list> - <list>`	Set Difference
`<list> * <list>`	Set Intersection
`<list> + <element>`	CONS
`<element> + <list>`	CONS
`<list> - <element>`	REMOVE
`<element> <= <list>`	MEMBER or MEMB
`<list> >= <element>`	MEMBER or MEMB

[16]However, numeric constants should *always* be surrounded by spaces to avoid misinterpretation by the Lisp reader.

[17]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.

[18]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.

17.1.3 Compound Operators

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

Operator	*Mnemonic*	*NUMBER*	*LISTOF*	*BOOLEAN*
`_+`	*Accumulate*	PLUS	NCONC1	OR
`_-`	*Remove*	DIFFERENCE	REMOVE	AND NOT
`+_`	*Push*	PLUS	PUSH	OR
`-_`	*Pop*		POP[19]	

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 table above shows an abbreviation of the operation that 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))`

When the compound operators are not specifically defined for a type, they are interpreted as specifying the operation ('`+`' or '`-`') on the two operands, followed by assignment of the result to the left-hand operand.

17.1.4 Assignment

Assignment of a value to the left-hand argument of an assignment operator is relatively flexible in GLISP. The following kinds of operands are allowed on the left-hand side of an assignment operator:

1. Variables.
2. Stored substructures of a structured type.
3. PROPerties of a structured type, whenever the interpretation of the PROPerty would be a legal left-hand side.
4. Algebraic expressions involving numeric types, *provided* that the expression ultimately involves only one occurrence of a variable or stored value.[20]

[20]For example, `(X2̂ = 2.0)` is acceptable but `(X*X = 2.0)` is not because the variable `X` occurs twice.

5. Standard Lisp data access functions which could be used to get an item.

For example, consider the following object description for a CIRCLE:

```
(CIRCLE (LIST (START VECTOR) (RADIUS REAL))
  PROP  ((PI           (3.1415926))
         (DIAMETER      (RADIUS*2))
         (CIRCUMFERENCE (PI*DIAMETER))
         (AREA          (PI*RADIUS^2))) )
```

Given this description, and a CIRCLE `C`, the following are legal assignments:

```
((radius c) = 5.0)
((area c) = 100.0)
((area c) _+ 100.0)
((CADR (start c)) _+ 3)
```

17.1.5 Self-assignment Operators

[21] There are some cases in which 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 data-type, with messages such as GETPROP and PUTPROP, and use `PropertyList`s as substructures of other data types. 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 that 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.*

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 that is compiled OPEN.

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

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

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)) ))
```

18 Control Statements

GLISP provides several Pascal-like control statements; these statements are bounded by parentheses, so that the keyword for each statement appears in the position of a Lisp function name. In the descriptions below, required keywords are shown in boldface, and optional keywords are shown in bold italics.

18.1 IF Statement

The syntax of the IF statement is as follows:

```
( IF <condition1>
     <trueaction>
     <falseactionm>)
```

This is consistent with the IF statement of Common Lisp. The `<condition>`s may be adjectival phrases.

18.2 CASE Statement

The CASE statement selects a set of actions based on an atomic selector value; its syntax is:

```
( CASE <selector> OF
       ( <case1> <action11> ... <action1i>)
       ( <case2> <action21> ... <action2j>)
          ...
       ELSE   <actionm1> ... <actionmk>)
```

The `<selector>` is evaluated and is compared with the given `<case>` specifications. Each `<case>` specification is either a single, atomic specification or a list of atomic specifications.

All `<case>` specifications are assumed to be quoted. The 'ELSE' clause is optional; the 'ELSE' actions are executed if `<selector>` does not match any `<case>`.

If the *type* of the `<selector>` has a VALUES specification, `<case>` specifications that match the VALUES for that type will be translated into the corresponding values.

18.3 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 <action1> ... <actionn>)

(FOR <variable> IN <set> DO <action1> ... <actionn>)
```

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. [22] The `<set>` may be qualified by an 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>
```

The `<predicate>` and `<adjective>` phrases may be combined with AND, OR, NOT, and grouping parentheses. These phrases may be followed by a qualifying phrase of the form:

```
 WHEN <expression>
```

The 'WHEN' expression is ANDed with the other qualifying expressions to determine when the loop body will be executed.

Within the FOR loop, the current member of the `<set>` that is being examined is automatically put into *context* at the highest level of priority. For 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:

[22] 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)`.

```
(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 construct:

```
( ... COLLECT <form>)
```

18.4 WHILE Statement

The format of the WHILE statement is as follows:

```
( WHILE <condition> DO <action1> ... <actionn>)
```

The actions `<action1>` through `<actionn>` are executed repeatedly as long as `<condition>` is true. The keyword '**DO**' may be omitted. The value of the expression is NIL.

18.5 REPEAT Statement

The format of the REPEAT statement is as follows:

```
( REPEAT <action1> ... <actionn> UNTIL <condition>)
```

The actions `<action1>` through `<actionn>` are repeated (always at least once) until `<condition>` is true. The value of the expression is NIL. The keyword **UNTIL** is required.

19 Definite Reference to Particular Objects

In order to simplify reference to particular member(s) 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, or **THOSE** followed by the plural form of the group, and qualifying phrases (as described for the **FOR** statement). The following examples illustrate these expressions.

```
(THE SLOT WITH SLOTNAME = NAME)
(THOSE EMPLOYEES WITH JOBTITLE = 'ELECTRICIAN)
```

The value of **THE** is a single object (or NIL if no object satisfies the specified conditions); **THOSE** produces a list of all objects satisfying the conditions.[23]

[23] In general, nested loops are optimized so that intermediate lists are not actually constructed. Therefore, the use of nested THE or THOSE statements is not inefficient.

20 Messages

GLISP supports the *message* metaphor, which has its roots in the languages SIMULA and SMALLTALK. These languages provide *object-centered programming*, in which objects are thought of as being active entities that communicate by sending each other *messages*. The internal structures of objects are hidden; a program that wishes to access 'variables' of an object does so by sending messages to the object requesting the access desired. Each object contains [24] 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 run time, 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, which results 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 that is associated with the selector, or compile as messages to be interpreted at run time.

The sending of a *message* in GLISP is specified using the following syntax:

```
(SEND <object> <selector> <arg1> ... <argn> )
```

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 parentheses, so that the `<selector>` will always be the third element of the list.

21 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 or in one of the SUPERS of the type. [25] Each `<selector>` is paired with the appropriate `<response>` to the message. Code is compiled depending on the form of the `<response>` associated with the `<selector>`, as follows: [26]

1. If the `<response>` is a symbol, that symbol is taken as the name of a function that is

[24]typically by inheritance from some parent in a class hierarchy

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

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

to be called in response to the message. The code that is compiled is a direct call to this function,

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

For example, if the message list for the type of the variable `OBJ` contained the entry:

```
(EDIT         EDITV)
```

the source code `(SEND OBJ EDIT)` would generate the object code `(EDITV OBJ)`.

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. [27] 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. For example, suppose that the type of the variable `J` contained the PROPerty entry:

   ```
   (SUCCESSOR    (self + 1))
   ```

   The source code `(successor j)` would generate the object code `(ADD1 J)`. The object description for a VECTOR, given earlier, contains the PROPerty entry:

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

   This example illustrates a call to a function, SQRT, with arguments containing definite references to X and Y, which are defined as stored fields of the VECTOR. 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. Given this definition and a VECTOR object `V`, the source code `V:MAGNITUDE` would generate the object code:

   ```
   (SQRT (IPLUS (ITIMES (CAR V) (CAR V))
                (ITIMES (CADR V) (CADR V))))
   ```

[27] Such recursive definitions must, of course, be acyclic.

The user can determine whether a message is to be compiled open, compiled as a function call, or compiled as a message that is to be executed at run time. When a GLISP expression is specified as a `<response>`, the `<response>` is always compiled open; open compilation can be requested by using the OPEN property when the `<response>` is a function name. 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 combined capabilities of open compilation, message inheritance, conditional compilation, and flexible assignment provide a great deal of power. The ability to use definite reference in GLISP makes the definition and use of the 'macros' simple and natural.

22 Compilation of Properties and Adjectives

Properties, adjectives, and ISA-adjectives are compiled in the same way as messages. Since the syntax of the use of properties and adjectives does not permit specification of any arguments, the only argument available to the code or function that 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>` that is written directly as GLISP code may use the name `self` directly, [28] as in the SUCCESSOR example above; a function that is specified as the `<response>` will be called with the `self` object as its single argument.

23 Declarations for Message Compilation

Declarations that 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, `<prop1> <value1>  ...  <propn> <valuen>`. 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. [29]

2. **OPEN T**
   This declaration specifies that the function that is specified as the `<response>` is to be compiled open at each reference. A `<response>` that is a list of GLISP code is always compiled open; however, such a `<response>` can have only the `self` argument. If it

[28] The name `self` is 'declared' by the compiler and does not have to be specified in the structure description.

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

is desired to compile open a message `<response>` that 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 that are compiled open may not be recursive via any chain of open-compiled functions.

3. **MESSAGE T**
   This declaration specifies that a run-time message should be generated for messages with this `<selector>` sent to objects of this class. Typically, such a declaration would be used in a higher-level class whose subclasses have different responses to the same message `<selector>`.

4. **ARGTYPES** (<type$_2$> ... <type$_n$>)
   This declaration specifies the argument types for a Message entry. The message entry will only be effective for arguments which match the specified types[30]. The type of the object itself is not included in the ARGTYPES list. The ARGTYPES list allows different interpretations of the same message selector depending on the argument types at compile time. For example, the interpretations of `<vector> * <vector>` and `<vector> * <number>` could be different.

5. **SPECIALIZE T**
   This declaration is used where a property is implemented by a named closed generic function. If the property is inherited by a subclass of the class in which it is defined, a version of the function that is specialized for the particular argument types involved will be created and stored with the subclass.

24 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 data-type, 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 declaration and function definitions below:

```
(glispobjects
   ((vector (list (x integer)
                  (y integer))
      MSG  ((+   vectorplus  open t argtypes (vector))
            )

(gldefun vectorplus ((v1 vector) (v2 vector))
  (a (typeof v1) with x = (x v1) + (x v2)  y = (y v1) + (y v2)))
```

[30]An argument matches a type if it is of the specified type or if its type has the specified type as a SUPER.

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

The compound operators ('_+', '+_', '_-', '-_') are conventionally 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 modify 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 nondestructive 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).

25 Run-time Interpretation of Messages

In some cases, the type of the object that will receive a given message is not known at compile time; in such cases, the message must be executed interpretively, at run time. Interpretive execution is provided for all types of GLISP messages.

An interpretive message call (i.e., a call to the function SEND) is generated by the GLISP compiler in response to a message call in a GLISP program when the specified message selector cannot be found for the declared type of the object receiving the message or when the MESSAGE flag is set for that selector. Alternatively, a call to SEND may be entered interactively by the user or may be contained in a function that has not been compiled by GLISP.

SEND messages are interpreted for those objects that are represented as one of the OBJECT types; variants of SEND are provided for use with other data types. The <selector> of the message is looked up in the MSG declarations of the CLASS; if it is not found there, the SUPERS of the CLASS are examined (depth first) until the selector is found. The <response> associated with the <selector> is then examined. If the <response> is a function name, that function is simply called with the specified arguments.[31] If the <response> is a GLISP expression, the expression is compiled as a LAMBDA form and cached for future use.

For convenience of interactive use, the interactive SEND function will accept the names of PROP, ADJ, and ISA entries or names of substructures in addition to message selectors[32]. While acceptably fast for interactive use, this feature is not recommended for use within functions. If a PROPerty or substructure name is suffixed with a colon and followed by a value, the specified value will be 'stored' into the specified property. For example, if DC is a DCIRCLE object, the message (SEND DC AREA: 100.0) may be used to set the AREA of

[31]The object to which the message is sent is always inserted as the first argument, followed by the other arguments specified in the message call.

[32]The order of searching for names is MSG, STR, PROP, ADJ, ISA.

DC to the value 100.

Several variants of the SEND function are provided to allow specification of the property type (PROP, ADJ, ISA, MSG, STR) and to allow messages to be used with data objects which are not defined as Object types:

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

(GLSEND    <object>         <selector>            <args>)

(SENDC     <object> <class> <selector>            <args>)

(SENDPROP  <object>         <selector> <proptype> <args>)

(SENDPROPC <object> <class> <selector> <proptype> <args>)

(GLSENDB   <object> <class> <selector> <proptype> <args>)
```

GLSENDB evaluates all of its arguments; the other forms all call GLSENDB. The SEND forms do not evaluate `<class>`, `<selector>`, or `<proptype>` (where `<proptype>` is `PROP`, `ADJ`, `ISA`, `MSG`, or `STR`). `<object>` and `<args>`, if any, are evaluated. `GLSEND` is the same as `SEND`; use of `GLSEND` allows the function name `SEND` to be used for another purpose.

26 Context Rules and Reference

The ability to use definite reference to features of objects that are in *context* is the key to much of GLISP's power. At the same time, definite reference introduces the possibility of ambiguity; that is, there could be more than one object in context that has a feature with a specified name. In this section, guidelines are presented for the use of definite reference to allow the user to avoid ambiguity.

27 Organization of Context

The context maintained by the compiler is organized in levels, each of which may have multiple entries; the sequence of levels is 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.

28 Rules for Using Definite Reference

The possibility of referential ambiguity 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 that are guaranteed to be unambiguous:

1. In compiling GLISP code that 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. [33]

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

3. In many cases, a function will have only 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.

29 Type Inference

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, thus relieving the programmer of the burden of specifying types explicitly. The following rules allow the programmer to know when types will be inferred by the compiler.

[33]This holds true 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.

1. Whenever a variable is set to a value whose type is known, 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 type was NIL (e.g., an untyped 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 and is added to a new level of context by the compiler.
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.
9. The types returned by many standard Lisp functions are known by the compiler.

30 GLISP and Knowledge Representation Languages

GLISP provides a convenient *access language* that 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. Artificial Intelligence has given rise to a number of powerful representation languages; these languages 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 run time, these representation languages have often been computationally costly.

Facilities are provided for interfacing GLISP with representation languages of the user's choice. When this is done, GLISP provides a convenient and uniform language both for accessing objects in the representation language and for accessing Lisp objects. In addition, GLISP can greatly improve the efficiency of programs that 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 that are efficient in both time and storage.

A method for declaring an 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 that is being compiled specifies an access to an object that 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; it 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 that relies heavily on procedural inheritance, substantial improvements in execution speed can be achieved by performing the inheritance lookup at compile time and compiling direct procedure calls to inherited procedures when the procedures are static and the type of the object that inherits the procedure is known at compile time.

Specifications for an interface module for GLISP are contained in a separate document.[34] To date, GLISP has been interfaced to the GIRL[**?**] representation language and to LOOPS[**?**].

31 Obtaining and Using GLISP

GLISP and its documentation are available free of charge over the ARPANET. The GLISP files are contained in the directory GLISP on the host computer `SUMEX-AIM`; SUMEX will accept the login-name 'ANONYMOUS GUEST' for transferring files with FTP.

32 Documentation

This user's manual, in line printer format, is contained in the file GLUSER.LPT. The Scribe source file is GLUSER.MSS. Printed copies of this manual can be ordered from the author ($5.00 prepaid); the printed version may not be as up-to-date as the on-line version.

[34]to be written

33 Getting Started

Useful functions for invoking GLISP are:

`(GLCC 'FN)`	Compile FN.
`(GLCP 'FN)`	Compile FN and prettyprint result.
`(GLP 'FN)`	Prettyprint GLISP-compiled version of FN.
`(GLED 'NAME)`	Edit the property list of NAME.
`(GLEDF 'FN)`	Edit the original (GLISP) definition of FN. (The original definition is saved under the property 'GLORIGINALEXPR' when the function is compiled, and the compiled version replaces the function definition.)
`(GLEDS 'STR)`	Edit the structure declarations of STR.

The editing functions call the 'BBN/Interlisp' structure editor.

To try out GLISP, load GLISP and the GLTEST file and use GLCP to compile the functions CURRENTDATE, GIVE-RAISE, MGO-TEST, TESTFN2, DRAWRECT, PRINTLEAVES, GROWCIRCLE, and SQUASH. To run compiled functions on test data, do:

```
(GIVE-RAISE 'COMPANY1)
(PRINTLEAVES '(((A (B (C D (E (G H (I J (K))))))))))
(GROWCIRCLE MYCIRCLE)
```

34 Reserved Words and Characters

GLISP contains ordinary Lisp as a sublanguage. However, to avoid having code that was intended as 'ordinary Lisp' interpreted as GLISP code, it is necessary to follow certain conventions when writing 'ordinary Lisp' code.

34.1 Reserved Characters

The colon and the characters that represent the arithmetic operators should not be used within atom names, since GLISP splits apart 'atoms' that contain operators. The set of characters to be avoided within atom names is:

```
+ * / ^ _ ~ =  < > : ' ,
```

The character 'minus' (`-`) is permitted within atom names;[35] correspondingly, the *operator* '`-`' must appear with a blank on each side.

[35] unless the flag `GLSEPMINUS` is set

Some GLISP constructs permit (but do not require) use of the character 'comma' (','); since the comma is used as a 'backquote' character in some Lisp dialects, the user may wish to avoid its use. In Lisp dialects that use the comma as a backquote character, all commas must be 'escaped' or 'slashified'; this makes GLISP code containing commas less portable.

34.2 Reserved Function Names

Most function, variable, and property used within GLISP names begin with 'GL' to avoid conflict with user names. Those 'function' names that are used in GLISP constructs or in interpretive functions should be avoided as names of user functions. This set includes the following names:

A	IF	THE
AN	REPEAT	THOSE
CASE	SEND	WHILE
FOR	SENDPROP	

34.3 Other Reserved Names

Words that are used within GLISP constructs should be avoided as variable names. This set of names includes:

A	ELSEIF	THAT
AN	IN	THEN
AND	IS	UNTIL
COLLECT	NOT	WHEN
DO	OF	WHICH
EACH	OR	WHO
ELSE	THE	WITH

35 Lisp Dialect Idiosyncrasies

GLISP code passes through the Lisp reader before it is seen by GLISP; some Lisp readers produce strange results for 'atom' names which contain special characters. For this reason, operators in expressions may need to be set off from operands by blanks; the operator '-' should always be surrounded by blanks, and the operator '+' should be separated from numbers by blanks. In most Lisp dialects, floating-point constants should be surrounded by blanks.

35.1 Interlisp

GLISP functions are written using GLAMBDA rather than LAMBDA to cause automatic compilation. GLISP declarations are integrated with the file package. The system flag `NORMALCOMMENTSFLG` should be `T` when the GLISP compiler is read in.

An interactive display inspector and editor for GLISP data, GEV[?], is available for use on Xerox Lisp machines.

36 GLISP Hacks

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

37 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)) )

)
```

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, that is, in the environment in which the original assignment operation was specified.

38 Disjunctive Types

Lisp programming often involves objects that may in fact be of different types but that 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 data type that encompasses both. Typically, this is done by declaring the structure of the object to be the complex structure and then 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)))))
```

39 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 data type, that contains the state of the enumeration and responds to messages. This is illustrated below by an object that 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) ))))


(PRINTLEAVES (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 function example PRINTLEAVES above. By being open-compiled, messages to an object can be as efficient as in-line hand coding, yet the code for the messages has to be written only once and can easily be changed without changing the programs that use the messages.

40 Acknowledgments

Many people have contributed help and useful comments about GLISP.

Help in adapting GLISP to particular Lisp dialects was generously provided by Larry Masinter and Ron Kaplan (Interlisp), Dick Fateman and Milt Grinberg (Franz Lisp), and Martin Griss (Portable Standard Lisp). Christopher Schmidt provided much help with the local Lisp environment.

Early users of GLISP have provided many helpful bug reports, suggestions, and comments concerning this manual. Special thanks are due to Aaron Temin, Henry Sowizral, Keith

Wescourt, Ken Anderson, Bill White, and Jay Lark. Helpful comments have also been provided by Dan Bobrow, Harold Brown, Phil Gerring, and Mark Stefik.

Dianne Kanerva proofread this manual, making many stylistic improvements.

Finally, particular thanks are due Ed Feigenbaum, Bruce Buchanan, and Tom Rindfleisch for their encouragement and support of this project. Ira Goldstein and Hewlett-Packard Co. have provided support in adapting GLISP for Portable Standard Lisp.

41 Program Examples

This section presents examples of the use of GLISP. The examples include object descriptions, functions which use the objects, creation of data objects, and interactive use of GLISP. The examples are shown in the form appropriate for Interlisp; the test file GLTEST which is provided for each Lisp dialect contains versions of these examples in the appropriate form for that dialect.

42 Company Example

The following set of object descriptions describes a company database.

```
(GLISPOBJECTS

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

)
```

The `A` function can be used either within functions or interactively to create data objects:

```
(SETQ COMPANY1 (A COMPANY WITH
   PRESIDENT = (AN EMPLOYEE WITH NAME = ‘OSCAR THE GROUCH’
                            SALARY = 88.0
                            JOBTITLE = 'PRESIDENT
                            DATE-HIRED = (A DATE WITH MONTH = 3
                                            DAY = 15 YEAR = 1907))
   EMPLOYEES = (LIST
               (AN EMPLOYEE WITH NAME = ‘COOKIE MONSTER’
                            SALARY = 12.50
                            JOBTITLE = 'ELECTRICIAN
                            DATE-HIRED = (A DATE WITH MONTH = 7
                                            DAY = 21 YEAR = 1947))
               (AN EMPLOYEE WITH NAME = ‘BETTY LOU’
                            SALARY = 9.00
                            JOBTITLE = 'ELECTRICIAN
                            DATE-HIRED = (A DATE WITH MONTH = 5
                                            DAY = 15 YEAR = 1980))
```

```
                (AN EMPLOYEE WITH NAME = ‘GROVER’
                             SALARY = 3.00
                             JOBTITLE = ’ELECTRICIAN
                             TRAINEE = T
                             DATE-HIRED = (A DATE WITH MONTH = 6
                                             DAY = 13 YEAR = 1978))
)))
```

The following program will give raises to the electricians.

```
    (GIVE-RAISE
       (GLAMBDA (:COMPANY)
   (FOR EACH ELECTRICIAN WHO IS NOT A TRAINEE
      DO (SALARY _+(IF SENIORITY > 1
       THEN 2.5
       ELSE 1.5))
  (PRINT (THE NAME OF THE ELECTRICIAN))
                 (PRINT (THE PRETTYFORM OF DATE-HIRED))
                 (PRINT MONTHLY-SALARY) )))
```

The CURRENTDATE function returns a constant date.

```
    (CURRENTDATE
       (GLAMBDA ()    (RESULT DATE)
   (A DATE WITH YEAR = 1981   MONTH = 11   DAY = 30)))
```

43 Vectors Example

This section contains a set of object descriptions and functions which describe two-element vectors. The vector functions are written as generic functions, so that they can be used for several different kinds of vectors.

First, the basic vector object is described. The actual stored structure for a vector is simple, but it is overloaded with many properties. The MAGNITUDE property contains a call to the SQRT function; the argument of SQRT is an expression containing definite references to the X and Y substructures of the vector. The DIRECTION property illustrates the use of conditional code in defining a computed property; the conditional code will be duplicated in-line each time the DIRECTION is referenced. Defining arithmetic operators as message selectors causes automatic overloading of the operators for vector operands; this allows vector arithmetic to be specified by arithmetic expressions. The use of the ARGTYPES specification for the '*' messages allows two meanings to be assigned to the '*' operator: multiplication of a vector by a scalar and vector dot product.

```
(GLISPOBJECTS

  (VECTOR

     (LIST (X INTEGER)
   (Y INTEGER))

     PROP   ((MAGNITUDE ((SQRT X^2 + Y^2)))
             (DIRECTION ((IF X IS ZERO
                             THEN (IF Y IS NEGATIVE THEN -90.0
                                                    ELSE 90.0)
                             ELSE (ATAN2D Y X)))
                          RESULT DEGREES))

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

     MSG    ((+ VECTORPLUS OPEN T)
     (- VECTORDIFF OPEN T)
     (* VECTORTIMESSCALAR ARGTYPES (NUMBER) OPEN T)
     (* VECTORDOTPRODUCT ARGTYPES (VECTOR) OPEN T)
             (/ VECTORQUOTIENTSCALAR OPEN T)
     (_+ VECTORMOVE OPEN T)
             (PRIN1 ((PRIN1 '(')
             (PRIN1 X)
                     (PRIN1 ',')
                     (PRIN1 Y)
                     (PRIN1 ')')))
             (PRINT ((SEND SELF PRIN1)
                     (TERPRI)))  ) )
)
```

The FVECTOR object differs from the VECTOR object in two ways: it has a different structure (CONS instead of LIST), and its components are a STRING and a BOOLEAN rather than INTEGERs. Because the types of objects are propagated by the GLISP compiler, the same generic function VECTORPLUS can be used for both VECTORs and FVECTORs.

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

The DEGREES and RADIANS objects illustrate how an object whose stored representation is a basic type (in this case, a REAL number) can be overloaded with properties. Such overloading is also useful for viewing objects using the GEV inspector.

```
  (DEGREES REAL
     PROP ((RADIANS (self*(3.1415926 / 180.0)) RESULT RADIANS)))

  (RADIANS REAL
     PROP ((DEGREES (self*(180.0 / 3.1415926)) RESULT DEGREES)))
```

The following functions define arithmetic operations on Vectors. These functions are generally called OPEN (macro-expanded) rather than being called directly. The use of TYPEOF as the type within the '`A`' function calls allows these functions to be *generic*; that is, they create the same type of data they are 'called' with, and therefore can be used for a variety of vector types.

```
(gldefun vectorplus ((v1 vector) (v2 vector))
  (a (typeof v1) with x = (x v1) + (x v2)  y = (y v1) + (y v2)))

(gldefun vectortimes ((v vector) (n number))
  (a (typeof v) with x = x*n  y = y*n))


(gldefun vectorquotient ((v vector) (n number))
  (a (typeof v) with x = x / n  y = y / n))

(gldefun vectordotproduct ((v1 vector) (v2 vector))
  ( (x v1) * (x v2) + (y v1) * (y v2) ) )

(gldefun vectormove ((v vector) (delta vector))
  ((x v) _+ (x delta)) ((y v) _+ (y delta))v)
```

VECTORMOVE, which defines the '_+' operator for vectors, does a destructive addition to the vector which is its first argument. Thus, the expression '`U_+V`' will destructively change U, while '`U_U+V`' will make a new vector with the value U+V and assign its value to U.

The following example functions use the vector definitions given above. The compiled versions of the functions are shown to illustrate how the different object definitions produce different output code for similar 'surface' code.

```
(gldefun tvplus ((u vector) (v vector)) u + v)


(gldefun tvmove ((u vector) (v vector)) u _+ v)


(gldefun tvtimesv ((u vector) (v vector)) (dotproduct u v))


(gldefun tvtimesn ((u vector) (v number)) u * v)


(gldefun tfvplus ((u fvector) (v fvector)) u + v)
```

44 GraphicsObject Example

The following object descriptions are used in a graphics object test program[36]. The definition of GraphicsObject builds on that of Vector. The property LEFT is defined as being equivalent to the X component of the START vector. Vector arithmetic is used in defining the property CENTER. The messages DRAW and ERASE illustrate a way to obtain different runtime behavior for objects with different SHAPEs; this could also be done using the interpretive message mechanism.

```
(glispobjects
(graphicsobject (list (shape atom)
      (start vector)
      (size vector))
prop    ((left   ((x start)))
 (bottom ((y start)))
 (right  (left + width))
 (top    (bottom + height))
 (width  ((x size)))
 (height ((y size)))
 (center (start + size / 2))
 (area   (width * height)))
msg     ((draw   ((funcall (get shape 'drawfn) self 'paint)))
 (erase  ((funcall (get shape 'drawfn) self 'erase)))
 (move   graphicsobjectmove open t)))
)
```

An object moves itself by erasing itself, changing its starting point, and then redrawing itself.

[36]This program is derived from one written by D.G. Bobrow as a LOOPS[?] example.

```
(gldefun graphicsobjectmove ((self graphicsobject) (delta vector))
  (glsend self erase)
  (start _+ delta)
  (glsend self draw))
```

A MovingGraphicsObject is a GraphicsObject which also has a VELOCITY. The GraphicsObject is declared TRANSPARENT, which makes all of its properties directly visible from a MovingGraphicsObject. The STEP message causes the object to move itself by the amount specified as its VELOCITY; the MOVE message the object sents to itself is handled by its GraphicsObject component.

```
(movinggraphicsobject (list (transparent graphicsobject)
    (velocity vector))
msg     ((accelerate mgo-accelerate open t)
 (step ((glsend self move velocity)))))     )
```

The test function MGO-TEST creates a MovingGraphicsObject and then moves it across the screen by sending it MOVE messages. [37] Everything in this example is compiled open; the STEP message involves a great deal of message inheritance, and generates a large amount of in-line Lisp code from a relatively small amount of GLISP code. The WHILE loop illustrates an easy way to construct 'Pascal-style' FOR loops.

```
(gldefun mgo-test nil
  (let (mgo n)
    (mgo = (a movinggraphicsobject with shape = 'rectangle
size = (a vector with x = 4 y = 3)
velocity = (a vector with x = 3 y = 4)))
      (n = 0)
      (while ((n _+ 1) < 100) (step mgo))
      (print (start mgo)) ))
```

The following function draws a rectangle. Computed properties of the rectangle are used within calls to the graphics functions, making the MOVETO and DRAWTO code easy to write and understand.

```
(gldefun drawrect ((g graphicsobject) (dspop atom))
  (prog (oldds)
    (oldds = (currentdisplaystream dsps))
    (dspoperation dspop)
    (moveto left bottom)
    (drawto left top)
    (drawto right top)
```

[37]The test program MGO-TEST runs on a Xerox Lisp machine, but won't run on other machines.

```
    (drawto right bottom)
    (drawto left bottom)
    (currentdisplaystream oldds)))
```

45 Circle Example

The Circle objects illustrate the definition of a number of mathematical properties of an object in terms of stored data and in terms of other properties. The property PI illustrates the definition of a property as a constant. Many of the properties of a Circle are defined using definite reference to other properties. The messages defined for a circle illustrate 'storing into' a computed property, in this case the AREA of a circle.

```
(glispobjects
(circle (listobject (center vector)
            (radius real))
prop    (      ;   (pi (3.1415926535897931))
 (diameter (radius * 2))
 (circumference (pi * diameter))
 (area (pi * radius ^ 2))
 (squareside ((sqrt area)))
 (displayprops ('(diameter circumference area)))
msg     ((grow (area _+ 100))
 (shrink (area = area / 2))
 (standard (area = 100.0)))
adj     ((big (area > 100))
 (small (area < 80))))
     )
```

The function GROWCIRCLE illustrates assignment to a computed property.

```
(gldefun growcircle ((c circle))
  ((area c) _+ 100)
  c)
```

A DCIRCLE is implemented differently from a CIRCLE: the data structure is different, and DIAMETER is stored instead of RADIUS. By defining RADIUS as a property, all of the CIRCLE properties defined in terms of RADIUS can be inherited.

```
(glispobjects
(dcircle (listobject (center   vector)
     (diameter real))
prop    ((radius (diameter / 2)))
supers  (circle)  )
  )
```

The following transcript of an interactive session illustrates the interactive use of messages and the '`A`' function with circle objects. Since DCIRCLE is an Object type, instances of DCIRCLEs can be used with interpreted messages; instances of CIRCLEs can also be used with interpreted messages by using the function SENDC and including the name of the class of the object in the SENDC call.

```
  (SETQ MYCIRCLE (A CIRCLE))
    ((0 0) 0.0)
  (GROWCIRCLE MYCIRCLE)
    ((0 0) 5.6418959)
  (SENDC MYCIRCLE CIRCLE AREA)
    100.0
  (SENDC MYCIRCLE CIRCLE AREA: 150)
    6.9098831
  (SENDC MYCIRCLE CIRCLE AREA)
    150.000002
```

```
  (SETQ DC (A DCIRCLE WITH DIAMETER = 10.0))
    (DCIRCLE (0 0) 10.0)
  (SEND DC AREA)
    78.539815
  (SEND DC AREA: 100)
    11.2837918
  (SEND DC AREA)
    100.0
  (SEND DC GROW)
    15.9576914
  (SEND DC AREA)
    200.0
```

46 Square Root Example

The following definition of a simple square root function illustrates the use of GLISP for traditional mathematical programming.

```
(gldefun sqrtb ((x number))
  (let ((s x))
    (if (x >= 0)
(progn
  (while ((abs s * s - x) > 1.0E-6) do (s = (s + x / s) * .5))
  s)) ))
```

47 Squash Example

The function SQUASH, shown below, illustrates elimination of compile-time constants. Of course, nobody would write such a function directly. However, such forms can arise when inherited properties are compiled. Conditional compilation occurs automatically when appropriate variables are defined to the GLISP compiler as compile-time constants because the optimization phase of the compiler makes the unwanted code disappear.

```
(gldefun squash nil
  (if (> 1 3)
      ‘amaazing
      (if (sqrt 7.2) < 2
  ’incredible
  (if 2 + 2 == 4 ’okay ’jeez))))
```

48 Class Example

The following object definitions describe a student records database.

```
(glispobjects
(student (atomobject (name   string)
     (sex    symbol)
     (major  symbol)
     (grades (listof integer)))
prop    ((average student-average)
 (grade-average student-grade-average)
 (shortvalue (name))
 (displayprops (t)))
adj     ((male (sex == ’male))
 (female (sex == ’female))
 (winning (average >= 95))
 (losing (average < 60)))
isa     ((winner (self is winning))))
  )

(student-group (listof student)
prop    ((n-students length)
 (average student-group-average)
 (shortvalue ((for x in self collect (shortvalue x))))))


(class (atomobject (department symbol)
   (number     integer)
   (instructor string)
   (students   student-group))
prop    ((n-students ((n-students students)))
 (men ((those students who are male)))
 (women ((those students who are female)))
 (winners ((those students who are winning)))
 (losers ((those students who are losing)))
 (class-average ((average students)))))


(gldefun student-average ((s student))
  (result real)
  (let ((sum 0.0) (n 0))
    (for g in grades do (n _+ 1) (sum _+ g))
    (sum / n) ))


(gldefun student-grade-average ((s student))
  (result symbol)
  (let ((av (average s)))
    (if av >= 90.0
'a
        (if av >= 80.0
    'b
    (if av >= 70.0
'c
(if av >= 60.0 'd 'f) )))))


(gldefun student-group-average ((sg student-group))
  (let ((sum 0.0))
    (for s in sg do (sum _+ (average s)))
    (sum / (n-students sg)) ))
```

Next, several functions which use the above definitions are given.

```
; Print name and grade average for each student
(gldefun test1 ((c class))
  (for s in (students c) (format t "~A ~A~%" (name s) (grade-average s))))
```

```
; Print name and average of the winners in the class
(gldefun test2 ((c class))
  (for s in (winners c) (format t "~A ~A~%" (name s) (average s))))

; The average of all the male students' grades
(gldefun test3 ((c class)) (average (men c)) )

; The name and average of the winning women
(gldefun test4 ((c class))
  (for s in (women c) when s is winning
             (format t "~A ~A~%" (name s) (average s))))
```

Another way to print the names and averages of the winning women is to use the * operator to intersect the sets of women and winners. The GLISP compiler optimizes the code so that these intermediate sets are not actually constructed.

```
(gldefun test4b ((c class))
  (for s in ((women c) * (winners c))
       (format t "~A ~A~%" (name s) (average s)) ) )
```

Next, the 'A' function is used to create some test data for the above functions.

```
(setq class1
      (a class with instructor = "A. PROF" department = 'cs number = 102
 students =
 (list (a student with name = "JOHN DOE" sex = 'male
  major = 'cs  grades = ' (99 98 97 93))
       (a student with name = "FRED FAILURE" sex = 'male
  major = 'cs  grades = ' (52 54 43 27))
       (a student with name = "MARY STAR" sex = 'female
  major = 'cs  grades = ' (100 100 99 98))
       (a student with name = "DORIS DUMMY" sex = 'female
  major = 'cs  grades = ' (73 52 46 28))
       (a student with name = "JANE AVERAGE" sex = 'female
  major = 'cs  grades = ' (75 82 87 78))
       (a student with name = "LOIS LANE" sex = 'female
  major = 'cs  grades = ' (98 95 97 96)))))
```

49 Density Example

The following object definitions illustrate inheritance of properties from multiple parent classes. The three 'bottom' classes Planet, Brick, and Bowling-Ball all inherit the same definition of the property Density from the object class Physical-Object, although they are represented in very different ways. For Planets, the property Mass is stored directly, while for Ordinary-Objects the Weight of the object is stored, and Mass is computed by dividing the weight by the value of gravity. (MKS measurement units are assumed.) Bowling-Balls are defined so that there are two fixed Radius values, depending on whether the Type of the Bowling-Ball is 'Adult' or 'Child'.

```
(glispobjects
(physical-object anything
prop    ((density (mass / volume))))

(ordinary-object anything
prop    ((mass (weight / 9.88)))
supers  (physical-object))

(sphere anything
  prop    ((volume ((4.0 / 3.0) * 3.141593 * radius ^ 3))))

(parallelepiped anything
prop    ((volume (length * width * height))))
)


(planet (listobject (mass   real)
    (radius real))
supers  (physical-object sphere))

(brick (object (length real)
       (width  real)
       (height real)
       (weight real))
supers  (ordinary-object parallelepiped))

(bowling-ball (atomobject (type   symbol)
  (weight real))
prop    ((radius ((if type == 'adult .1 .07))))
supers  (ordinary-object sphere))
```

The following functions demonstrate inheritance of the DENSITY property.

```
; (dplanet earth)
(gldefun dplanet ((p planet)) density)

; (dbrick brick1)
(gldefun dbrick ((b brick)) density)
```

```
; (dbb bb1)
(gldefun dbb ((b bowling-ball)) density)
```

The following code creates some objects to test the functions on.

```
(gldefun init-objects nil
 (setq earth  (a planet with mass = 5.98E24 radius = 6.37E6))
 (setq brick1 (a brick with weight = 20.0 width = .1
 height = .05 length = .2))
 (setq bb1 (a bowling-ball with type = 'adult weight = 60.0)))
```

Since the object types Planet, Brick, and Bowling-Ball are defined as Object types (i.e., they contain the Class name as part of their stored data), messages can be sent to them directly from the keyboard for interactive examination of the objects:

```
(SEND EARTH DENSITY)
  5523.24475
(SEND BRICK1 DENSITY)
  2024.2915
(SEND BRICK1 WEIGHT: 25.0)
  25.0
(SEND BRICK1 DENSITY)
  2530.36438
(SEND BRICK1 MASS: 2.0)
  19.76
(SEND BRICK1 DENSITY)
  2000.00002


(SEND BB1 DENSITY)
  1449.79204
(SEND BB1 RADIUS)
  0.1
(SEND BB1 TYPE)
  ADULT
(SEND BB1 TYPE: 'CHILD)
  CHILD
(SEND BB1 RADIUS)
  0.07
(SEND BB1 DENSITY)
  4226.7988
```