THE LOOPS MANUAL

by Daniel G. Bobrow (Xerox PARC) & Mark Ste k (Xerox PARC)

Abstract: LOOPS adds data, object, and rule oriented programming to the procedure oriented programing of Interlisp. In object oriented programming, behavior is determined by responses of instances of classes to messages sent between these objects, with no direct access to the internal structure of an object. This approach makes it convenient to de ne program interfaces in terms of message protocols. Data oriented programming is a dual of object oriented programming, where behavior can occur as a side e ect of direct access to (permanent) object state. This makes it easy to write programs which monitor the behavior of other programs. Rule oriented programming is an alternative to programming in LISP. Programs in this paradigm are organized around recursively composable sets of pattern- action rules for use in expert system design. Rules make it convenient for describing exible responses to a wide range of events. LOOPS is integrated into Interlisp, and thus provides access to the standard procedure oriented programming of Lisp, and use of the extensive environmental support of the Interlisp- D system

Our experience suggests that programs are easier to build in a language when there is an available paradigm that matches the structure of the problem. The paradigms described here o er distinct ways of partitioning the organization of a program, as well as distinct ways of viewing the signi cance of side e ects. LOOPS provides all these paradigms within a single environment. This manual is intended as the primary documentation for users of LOOPS. It describes the concepts and the programming facilities, and gives examples and scenarios for using LOOPS.

1 INTRODUCTION

Four distinct paradigms of programming available in the computer science community today are oriented around procedures, objects, data access and rules. Usually these paradigms are embedded in di erent languages. LOOPS is designed to incorporate all of them within the Interlisp programming environment, to allow users to choose the style of programming which best suits their application.

Procedure Oriented Programming: Lisp is a procedure oriented language; the procedure oriented paradigm is the dominant one provided in most programming languages today. Two separate kinds of entities are distinguished: procedures and data. Procedures are active and data are passive. The ability to compose procedures out of instructions and to invoke them is central to organizing programs using these languages. This is a major source of leverage in synthesizing programs. Side e ects happen when separate procedures share a data structure and change parts of it independently.

Object Oriented Programming: This paradigm was pioneered by Smalltalk, and has its roots in SIMULA and in the concept of data abstraction. In contrast with the procedure- oriented paradigm, programs are not primarily partitioned into procedures and separate data. Rather, a program is organized around entities called objects that have aspects of both procedures and data. Objects have local procedures (methods) and local data (variables). All of the action in these languages comes from sending messages between objects. Objects provide local interpretation of the message form.

The object- oriented paradigm is well suited to applications where the description of entities is simpli ed by the use of uniform protocols. For example in a graphics application, windows, lines and composite structures could be represented as objects that respond to a uniform set of messages (i.e., `Display`, `Move`, and `Erase`). An important feature of these languages is an inheritance network, which makes it convenient to de ne objects which are *almost like* other objects. This works together with the use of uniform protocols because specialized objects usually share the protocols of their super classes.

Data Oriented Programming: In both of the previous paradigms, the invocation of procedures (either by direct procedure call or by message sending) is convenient for creating a description of a single process. In the data- oriented programming, action is potentially triggered when data are accessed. Data oriented programming makes use of long term storage of objects with implicit links from structures to actions.

Data oriented programming is appropriate for interfacing between nearly independent processes. A good example of this is the construction of a viewer for an independent tra c simulation process. The viewer provides a visual display of the changing tra c simulation process without a ecting the code for the simulation. This independence means that the two processes can be written and understood separately. It means that the interactions between them can often be controlled without changing them.

Rule Oriented Programming: In rule oriented programming, the behavior of the system is determined by sets of condition- action pairs. These *RuleSets* play the same role as subroutines in the procedure oriented metaphor. Within a RuleSet, invocation of rules is guided largely by patterns in the data. In the typical case, rules correspond to nearly- independent patterns in the data. The rule- oriented approach is convenient for describing exible responses to a wide range of events characterized by the structure of the data.

Our experience suggests that programs are easier to build in a language when there is an available paradigm that matches the structure of the problem. A variety of programming paradigms gives breadth to a programming language. The paradigms described here o er distinct ways of partitioning the organization of a program, as well as distinct ways of viewing the signi cance of side e ects. LOOPS provides all these paradigms within the Interlisp environment [Xerox83]. In principle, the data- oriented programming can be used with either the object-oriented or the procedure-oriented paradigms. In LOOPS, we have combined it only with variables in the object-oriented metaphor.

Summary: LOOPS adds data, object, and rule oriented programming to Interlisp. In object oriented programming, behavior is determined by responses of instances of classes to messages sent between these objects, with no direct access to the internal structure of an object. This approach makes it convenient to de ne program interfaces in terms of message protocols. LOOPS provides:

inheritance of instance behavior and structure from multiple super classes

user extendible property list descriptions of classes, their variables, and their methods

composite objects - templates for related objects that are instantiated as a group.

Data oriented programming is a dual of object oriented programming, where behavior can occur as a side e ect of direct access to (permanent) object state. This makes it easy to write programs which monitor the behavior of other programs. LOOPS provides:

active values for object variables which can cause a procedure invocation on setting or fetching

integration with facilities for long term storage of objects in shared knowledge bases

support for incremental updates (layers), and the representation of multiple alternatives.

Rule oriented programming is an alternative to programming in LISP. Programs in this paradigm are organized around recursively composable sets of pattern-action rules for use in expert system design. Rules make it convenient for describing exible responses to a wide range of events. LOOPS provides:

a concise syntax for pattern matching and rule set construction

use of objects as working memory for rule sets

primitives for executing, stepping and suspending tasks based on ruleSets

compilation of ruleSets into Lisp code for e cient execution

LOOPS is integrated into Interlisp. LOOPS provides:

classes and instances as Interlisp le objects

pseudoClasses to eld messages to standard Interlisp datatypes

This manual is intended as the primary documentation for users of LOOPS. It describes the concepts and the programming facilities, and gives examples and scenarios for using LOOPS.

1.1 Intellectual Precursors

LOOPS grew out of our research in a knowledge representation language (called Lore) for use in a project to create an *expert assistant* for designers of integrated digital systems. Along the way, we discovered that we needed to experiment with alternative versions of the representation language. A core of features was identi ed that we wanted to keep constant in our experiments. This core became a data and object-oriented programming system with many features not found in other available systems. Many of the features (e.g., active values, data bases, and composite objects) were motivated by the needs of our project, but we they would be useful for many other applications. LOOPS has been su ciently useful and general that we decided to make it available outside of our group.

The design of LOOPS owes an intellectual debt to a number of other systems, including:

(1) Smalltalk ([Goldberg82], [Goldberg81], [Ingalls78]), which has pioneered many of the concepts of object-oriented programming.

(2) Flavors [Cannon82], which supports this style of programming in the MIT Lisp Machine environment and which confronted non-hierarchical inheritance.

(3) PIE [Goldstein80], which provided facilities for incremental, sharable data bases.

(4) KRL [Bobrow77], which explored many issues in the design of frame-based knowledge representation languages and which provoked much additional work in this area.

(5) UNITS [Ste k79], which provided a substantial testbed for experiments in problem solving that have guided our decisions about the importance of several language features.

(6) EMYCIN [VanMelle80] which showed the power of rule oriented programming for building expert systems.

While all of these languages provided ideas, none of them was quite right for our current needs. For example, Smalltalk supports only hierarchical inheritance and does not have a layered data base, active values, or property lists on variables. PIE and KRL are not easily supportable or extendable. Flavors does not run on the machines available to us. UNITS was the closest existing language to our needs, but we wanted to change many of its features. Since we have compared these languages and traced the intellectual history elsewhere [Bobrow82], we will not pursue that further in this document.

In designing LOOPS, we wanted a general inheritance mechanism, a way of attaching access-triggered procedures to variables, a way of instantiating composite objects recursively, and a way of creating permanent databases of objects that can be shared and updated incrementally.

In tension with the desire for extensive language features was a desire to keep LOOPS small so that it would be easy to understand and to implement. To this end we have tried to create a small repertoire of powerful features that work well together.

1.2 Acknowledgments

from the LOOPS Manual:

Thanks to Alan Bell, Harry Barrow, Harold Brown, Gordon Foyster, Phil Gerring and Gordon Novak, Chris Tong, Schlomo Weiss, Terry Winograd and the other members of the KBVLSI project (past and present) for bug reports and suggestions, and for enduring the wait for it to mature into existence while so many things have been pressing. Special thanks to Johan de Kleer for extensive discussions of design issues, and to Richard Fikes, Adele Goldberg, Danny Hillis, Dan Ingalls, and Gordon Novak for comments on earlier drafts of this manuscript. We are grateful to Larry Masinter and Bill Van Melle for help on the integration of LOOPS with Lisp, and to the Interlisp-D group for unfailing support and encouragement. Thanks also to Lynn Conway for encouraging this work and to the Xerox Corporation for providing the intellectual and computing environments in which it could be done.

from the Rules Manual:

Special thanks to Danny Berlin and Lynn Conway for many suggestions and for the patience it takes to be the rst real users of something new. Sanjay Mittal and Terry Winograd o ered helpful criticisms and advice on the documentation and concepts of the rule language. Larry Masinter and Bill van Melle have provided substantial support in the entire Loops enterprise with Interlisp-D. Thanks to the Xerox Corporation and George Pake of Xerox PARC for providing the stimulating environment and computational facilities that made this work possible.

1.3 References

[Aiello81] Aiello, N., Bock, C., Nii, H. P., White, W. C., *AGE Reference Manual*. Technical Report, Heuristic Programming Project, Computer Science Department, Stanford University, October 1981.

[Bobrow82] Bobrow, D. G., & Ste k, M. J. Introducing new programming metaphors to LISP. (submitted to *Communications of the Association for Computing Machinery*).

[Bobrow80] Bobrow, D. G., & Goldstein, I. P. Representing design alternatives. *Proceedings of the AISB Conference*, Amsterdam, 1980.

[Bobrow77a] Bobrow, D. G., & Winograd, T. An overview of KRL, a knowledge representation language, *Cognitive Science* 1:1, 1977, pp 3-46.

[Bobrow77b] Bobrow, D. G., & Winograd, T. Experience with KRL-0, one cycle of a knowledge representation language, *Proceedings of the Fifth International Joint Conference on Arti cial Intelligence*, Cambridge, Mass. August, 1977, pp 213-222.

[Cannon82] Cannon, H. I. Flavors: a non-hierarchical approach to object-oriented programming, *personal communication*, 1982.

[Consumers80] Anon, Washing Machines. *Consumer Reports*, November 1980, pp. 679-684.

[Erman81] Erman, L. D., London, P. E., Fickas, S. F. The design and an example use of Hearsay-III. *Proceedings of the Seventh International Joint Conference on Arti cial Intelligence*, August 1981, pp. 409-415.

[Fain81] Fain, J., Gorlin, D., Hayes-Roth, F., Rosenschein, S., Sowizral, H., Waterman, D. *The ROSIE Reference Manual*, Rand Note N-1647-ARPA, Rand Corporation, December 1981.

[Feigenbaum78] Feigenbaum, E. A., The art of arti cial intelligence: themes and case studies of knowledge engineering, *AFIPS Conference Proceedings* 47 National Computer Conference, 1978, pp. 227 240.

References

[Forgy81] Forgy, C. L. *OPS5 User's Manual*. Technical Report CMU-CS-81-135. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, July 1981.

[Goldberg82] Goldberg, A., Robson, D., Ingalls, D. *Smalltalk-80: The language and its implementation*. Reading, Massachusetts: Addison-Wesley (in press).

[Goldberg81] Goldberg, A. Introducing the Smalltalk-80 System, *Byte* 6:8, August 1981.

[Goldstein80] Goldstein, I. P., & Bobrow, D. G. Extending object oriented programming in Smalltalk. *Proceedings of the Lisp Conference*, Stanford University, 1980.

[Ingalls78] Ingalls, D. H. The Smalltalk-76 programming system: design and implementation. *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pp 9-16.

[Maytag] Anon. *Operating Instructions for Model A510*. Printed by the Maytag Company, Newton Iowa 50208.

[Ste k82] Ste k, M., Aikins,5 J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., Sacerdoti, E. The organization of expert systems: a tutorial. *Arti cial Intelligence*, 18:2, March 1982, pp. 135-173.

[Ste k79] Ste k, M. An examination of a frame-structured representation system. *Proceedings of the Sixth International Joint Conference on Arti cial Intelligence*, Tokyo, Japan, August 1979, pp. 845-852.

[VanMelle80] Emycin ... To be lled in

[Weinreb81] Weinreb, D., & Moon, D. *Lisp Machine Manual*, Massachusetts Institute of Technology, 1981

[Xerox83] *Interlisp Reference Manual*, Xerox Palo Alto Research Center, October, 1983.

2 OVERVIEW

2.1 Structure of Classes and Instances

Classes: A class is a description of one or more similar objects. An *instance* is an object described by a particular class. Every object within LOOPS is an instance of exactly one class. Classes themselves are instances of a class, usually the one called `Class`. Classes whose instances are classes are called *metaclasses*.

Variables: LOOPS supports two kinds of variables - class variables and instance variables. Class variables are used to contain information shared by all instances of the class. A class variable is typically used for information about a class taken as a whole. Instance variables contain the information speci c to an instance. Both kinds of variables have names, values, and other properties. A class describes the structure of its instances by specifying the names and default values of instance variables. For example, the class `Point` might specify two instance variables, `x` and `y` with default values of `0`, and a class variable, `lastSelectedPoint`, used by methods associated with all instances of class `Point`. LOOPS also allows ‘‘variable length’’ classes, which have some instance variables that are referenced by numerical index.

Methods: A class speci es the behavior of its instances in terms of their response to *messages*. The class associates *selectors* (LISP atoms) with *methods*, the Interlisp functions that respond to the messages. All instances of a class use the same selectors and methods. Any di erence in response by two instances of the same class is determined by a di erence in the values of their instance variables. For example, `PrintOn` is used as a selector for the message which knows how to print out a representation of an object on a le.

Properties: LOOPS provides user-extendible property lists for classes, their variables, and their methods. Property lists provide places for storing documentation and additional kinds of information. A property list on a variable is used to store additional information about both the variable and its value. For example, in a knowledge engineering application, a property list for an instance variable could be used to store such information as *support* (i.e., reasons for believing a value), *certainty factors* (i.e., numeric assessments of degree of belief), *constraints* on values, *dependencies* (i.e., relationships to other variables), and *histories* (i.e., previous values).

Metaclasses: Classes themselves are instances of some class. When we want to distinguish classes whose instances are classes, we call them metaclasses, after the Smalltalk usage. When a class is sent a message, its metaclass determines the response. For example, instances of a class are created by sending the class the message `New`. For most classes, this method is provided by the standard metaclass for classes: `Class`. The user can create other metaclasses to perform specialized initialization. The metaclass for `Class` itself (called `MetaClass`) contains the `New` method for making classes. Another useful metaclass provided in the system is `AbstractClass`. It is used for classes that are placeholders in the inheritance network that it would not make sense to instantiate. Its response to a `New` message is to cause an error.

```
[DEFCLASS AreaBudget
   (MetaClass Class    EditedBy (* dgb "15-Feb-82 14:32 ")
                doc
(* * This is a sample class chosen to illustrate the syntax
 of classes in LOOPS.  Commentary on the class is inserted
 in a standard property in the class.  -- e.g. Budgets are ...))
   (Supers OwnedObject Budget)
   (ClassVariables (maxBase 25000))
   (InstanceVariables
      (owner #$VLSI doc (* organizational area that owns budget) )
      (base 1000 doc (* The initial amount of money))
      (overhead 2.25 doc (* Multiplied by base to get total.))
      (employees NIL doc (* list of employees in this area))
      (manager NIL doc (* manager of this area))
      (total #(SHARED getTotal UpdateNotAllowed)
         doc (* value of total is computed using active value.))
   (Methods
      (Report AreaBudget.Report doc (* Prints out a budget report))
      (StoreBase AreaBudget.StoreBase
         doc (* store base value checking maxBase))]
```

Figure 1. Example of a class de nition in LOOPS. The class, called `AreaBudget`, inherits variables and methods from both of its super classes (`OwnedObject` and `Budget`). The form of the de nition here does not show inherited information, only the changes and additions. In this example the new class variable `maxBase` is introduced, and six instance variables (`owner`, `base`, `overhead`, `employees`, `manager`, and `total`) are de ned. The `Methods` declaration names the Interlisp functions that implement the methods. For example, `AreaBudget.Report` is the name of a function that implements the `Report` method for instances of `AreaBudget`.

2.2 Inheriting Variables and Methods

Inheritance is an important tool for organizing information in objects. It enables the easy creation of objects that are ‘‘almost like’’ other objects with a few incremental changes. Inheritance avoids the user have to specify redundant information and simpli es updating, since information that is common need be changed in only one place.

LOOPS objects exist in an *inheritance network* of classes. An object inherits its instance variable description and message responses. All descriptions in a class are inherited by a subclass unless overridden in the subclass. For methods and class variables, this is implemented by a runtime search for the information, looking rst in the class, and then at the super classes speci ed by its *supers list*. For instance variables, no search is made at run time; default values are cached in the class, and are updated if any super is changed, thus maintaining the same semantics as the search. Each class can specify inheritance of structure and behavior from any number of super classes in its supers list.

Hierarchy: In the simplest case, each class speci es only one super class. If the class `A` has the supers list `(B)`, a one element list containing `B`, then all of the instance variables speci ed local to `A` are added to those speci ed for `B`, recursively. That is, `A` gets all those instance variables described in `B` and all of `B`’s supers. In this case one obtains strict inheritance hierarchy as in Smalltalk.

Any con ict of variable names is resolved by using the description closer to `A` in traversing up the hierarchy to its root at the class `Object`. Method lookup uses the same con ict resolution. The method to respond to a message is obtained by rst searching in `B`, and then searching recursively in `B`’s supers list. An example of this is given in gure 2.

Class	Super	InstanceVariables	Methods
`Object`	`NIL`	none	`(s4 M6)`
`C`	`Object`	`(w 7)`	`(s2 M4) (s3 M5)`
`B`	`C`	`(y 4) (z 3)`	`(s1 M2) (s2 M3)`
`A`	`B`	`(x 1) (y 0)`	`(s1 M1)`

Figure 2. In the de nitions given in the above chart, an instance of `A` would be given four instance variables, `w`, `y`, `z`, and `x` in that order. The default value for `y` would be `0`, which overrides the default value of `y` inherited from `B`. The instance would also respond to the four messages with selector `s1`, `s2`, `s3`, and `s4`. The method used for responding to `s1` is `M1`, which is said to override `M2` as the implementation of the message `s1`. Similarly, `M3` overrides `M4` as the implementation of message `s2`. Notice that the root class in the system, `Object`, has no super class. All classes in the system are subclasses of `Object`, directly or indirectly.

Multiple Super Classes: Classes in LOOPS can have more than one class speci ed on their supers list. Multiple super classes admit a modular programming style where (i) methods and associated variables for implementing a particular feature are placed in a single class and (ii) objects requiring combinations of independent features inherit them from multiple supers. If `D` had the supers list `(E A)`, rst the description from `E` and its supers would be inherited, and then the description from `A` and its supers. In the simplest usage, the di erent features have unique variable names and selectors in each super. In case of a name con ict, LOOPS uses a depth- rst left to right precedence. For example, if any super of `E` had a method for `s1`, then it would be used instead of the method `M1` from `A`. In every case, inheritance from `Object` (or any other ‘‘common’’ super class) is only considered after all other classes on the recursively de ned supers list.

2.3 Data Oriented Programming Using Active Values

In data oriented programming, one needs a way of specifying for any variable of an object whether any special procedure is to be invoked on read or write access, and if so which. In LOOPS we check on every variable access whether the value is marked as an *active value*. If so, the active value speci es the procedures to be invoked when the value of a variable (or property) is read or set. This mechanism is dual to the notion of messages; messages are a way of telling objects to perform operations, which can change their variables as a side e ect; active values are a way of accessing variables, which can send messages as a side e ect. The following notation for active values illustrates its three parts:

`#(localState getFn putFn)`

This notation is converted by a read macro into an instance of the LISP data type `activeValue`. The `localState` is a place for storing data. The `getFn` and `putFn` are the names of functions that are applied with standard arguments when a program tries to get or put the value of a variable. Every active value need not specify both a `getFn` and a `putFn` If the `getFn` is `NIL`, then a get operation returns the local state. If the `putFn` is `NIL`, then a put operation replaces the local state.

Active values enable one process to monitor another one. For example, we have developed a LOOPS debugging package that uses active values to trace and trap references to particular variables. Another example is a graphics package that updates views of particular objects on a display when their variables are changed. In both cases, the monitoring process is invisible to and isolated from the monitored process. No changes to the code of the monitored object are necessary to enable monitoring.

Model/View Controller Example: gure 3 shows an application of this to a simulation model. Suppose that we want a program that simulates the ow of tra c in a city and displays selected parts of the simulation on a screen. Active values enable us to divide the programming of this example into two parts: the tra c model and the view controller. The tra c model consists of objects representing automobiles, tra c lights, emergency vehicles, and so on. These objects exchange messages to simulate tra c interactions (e.g., when a tra c light turns green, it would send `Move` messages to start cars moving). The view controller provides windows into di erent parts of the city. It contains information about how the objects are to be displayed. We want a user to be able to move these windows around to change the view.

```
(DEFINST Automobile-1 ...
   (InstanceVariables
      (position #(Pos1 NIL UpdateDisplay)
         displayObjects (DispObj1 DispObj2 DispObj3)
         doc (* position of car in traffic coordinate system))
      (speed 25))
   ...]
```

Figure 3. Instance of an automobile in a tra c simulation model. Other classes describe such things as tra c lights, city blocks, and emergency vehicles. Instances of these classes exchange messages while simulating the vehicles moving around in the model. The instance variable `position` is used to record the location of an automobile in the tra c coordinate system. In this example, an active value in `position` is used to update view objects that control pictures of the tra c patterns on an interactive display. Whenever a simulation method puts a new value into the `position` variable, the procedure `UpdateDisplay` sends update messages to each object in a list of view objects. These messages ultimately cause the graphics display to be updated.

In gure 3, there is an active value in the `position` variable of an instance of `Automobile`. This active value is the interface between the object in the simulation model and the view controller. Whenever a method in the simulation model changes the value of a `position` variable, the procedure `UpdateDisplay` in the `putFn`of the active value is invoked. `UpdateDisplay` updates the local value and sends a message to each of the view objects in the list stored as a property of `position`. These objects respond to a message by updating the view in the windows on the display screen. The important point of this example is that it shows how the view controller can be invoked as a side e ect of running the simulation. The view can be changed without e ecting any programs in the simulation model. To change the set of simulation objects being monitored, only the interface to the view controller needs to be changed by adding active values. The objects in the view controller may also be changed (e.g., to re ect changes to relative coordinates of the window and the tra c model).

2.4 Knowledge Bases

LOOPS was created to support a design environment in which there are community knowledge bases that people share, and to which they can add incremental updates. We have chosen the term *knowledge base* instead of *data base* to emphasize the intended application of LOOPS to expert systems. In expert systems, knowledge bases contain inference rules and heuristics for guiding problem solving. This is in contrast to the tabular les of facts usually associated with data bases.

Knowledge Bases: Knowledge bases in LOOPS are les that are built up as a sequence of layers, where each layer contains changes to the information in previous layers. A user can choose to get the most recent version of a knowledge base (that is, all of the layers) or any subset of layers. The second option o ers the exibility of being able to share a community knowledge base without necessarily incorporating the most recent changes. It also provides the capability of referring to or restoring any earlier version. gure 4 illustrates this with an example.

```
------------------------ Layer 1 ------------------------
Obj1 (x 4) ...
Obj2 (y 5) (w 3) ...
------------------------ Layer 2 ------------------------
Obj2 (y 7) (w 2) ...
Obj3 (z 6) ...
------------------------ Layer 3 ------------------------
Obj1 (x 8) ...
Obj4 (z 9) ...
```

Figure 4. Knowledge bases in LOOPS are les that are built-up incrementally as a sequence of layers. Each layer contains updated descriptions of objects. When a knowledge base is opened, the information in the later layers overrides the information in the earlier layers. LOOPS makes it possible to select which layers will be used when a knowledge base is opened. In this example, if the knowledge base is opened and only the rst 2 layers are used, then `Obj1` will have an `x` variable with value `4`. If all three layers were connected, then the value would be `8`.

Community Knowledge Bases: LOOPS partitions the process of updating a community knowledge base into two steps. Any user of a community knowledge base can make tentative changes to a community knowledge base in his own (isolated) environment. These changes can be saved in a layer of his personal knowledge base, and are marked as associated with the community knowledge base. In a separate step, a data base manager can later copy such layers into a community knowledge base. This separation of tasks is intended to encourage experimentation with proposed changes. It separates the responsibility for exploring possibilities from the responsibility of maintaining consistent and standardized knowledge bases for shared use by a community. The same mechanisms can be used by two individuals using personal knowledge bases to work on the same design. They can conveniently exchange and compare layers that update portions of a design.

Unique Identi ers: The ability to determine when di erent layers are referring to the same entity is critical to the ability to share data bases. To support this feature the LOOPS data base assigns unique identi ers (based on the computer's identi cation numbers, the date, and an unbounded count) to objects before they are written to a knowledge base. This facility provides a grounding for more sophisticated notions of equality that might be desired in knowledge representation languages built on LOOPS.

Environments: A user of LOOPS works in a personalized *environment*. An environment provides a lookup table that associates unique identi ers with objects in the connected knowledge bases. In an environment, user indicate dominance relationships between selected knowledge bases. When an object is referenced through its unique identi er, the dominance relationships determine the order in which knowledge bases are examined to resolve the reference. By making personal knowledge bases dominate over community knowledge bases, a user can override portions of community knowledge bases with his own knowledge bases.

Multiple Alternatives: An important use of environments is for providing speedy access to alternative versions (e.g., multiple alternatives in a design). A user can have any number of environments available at the same time. Each environment is fully isolated from the others. Operations that move information between environments are always done explicitly through knowledge bases.

3 CREATING AND USING OBJECTS

In the LOOPS implementation of object-oriented programming, there are three types of objects: Instances, Classes, and Metaclasses. Instances are used like data objects in Lisp; they are commonly created, passed around, and modi ed by procedures (although all objects can be). Classes and metaclasses are objects which ‘‘de ne’’ a group of objects that are ‘‘instances of’’ that class or metaclass. The di erence between classes and metaclasses is that the instances of a class are instances, and the instances of a metaclass are classes all comments about classes apply to metaclasses, except where otherwise stated.

Note that the word ‘‘instance’’ is used in two separate ways: the phrase ‘‘instance of’’ refers to the relation between any object and the class (or metaclass) that ‘‘de nes’’ it. The noun ‘‘instance’’ is only used to refer to those objects which are instances of classes.

A class contains information about instance variables, class variables, and methods. Instance variables are local variables stored within each instance of the class. Class variables are variables stored within the class object, accessable from each instance of the class. Methods are procedures which are used to perform operations on instances of the class.

Each Class also contains a list of other classes called ‘‘super classes’’ or ‘‘supers’’. The super class list provides a mechanism for inheriting instance variables, class variables, and methods from other classes (see page 31).

This section rst describes how to create and use objects. Next, ‘‘sending a message’’ (the standard way to invoke a method). Next, creating and using new instances. Next, de ning and editing new classes. Finally, de ning a new method for a class.

3.1 Sending a Message to an Object

Operations in LOOPS are invoked by sending messages. Sending a message to an object invokes a method (from the class that the object is an instance of) to execute the operation. Messages are sent using the function _ as follows:

`(_ object Selector arg`$_1$ `arg`$_N$`)` [NLambda NoSpread Function]

Sends the message `Selector` to the object `object` with the arguments `arg`$_1$ `arg`$_N$. `Selector` is always implicitly quoted (i.e., not evaluated); the remaining arguments are evaluated.

`object` must be an ‘‘internal pointer’’ to the object. The internal pointer to the object with the LOOPS name `FOO` can be extracted by the form `($ FOO)`.

Note: `SEND` can be used instead of _. The arrow notation, although less mnemonic, is usually used to make expressions shorter and hence easier to type and read.

If it is necessary to *compute* the selector, one can use the function `_!`, which is just like _ except that it also evaluates its `Selector` argument.

Example:

```
(_ ($ PayRoll) PrintOut file1)
```

This sends a `PrintOut` message to the class `PayRoll` (with a single argument; the value of the Intrerlisp variable `file1`).

3.2 Creating a New Instance

To create an instance of a particular class, one sends the message `New` to the class:

`(_ class New)` [Message]

Returns a new instance of the class `class`

In the usual case, initial values for instance variables are taken from the instance variable descriptions associated with the class. LOOPS provides some other ways to exercise control over the initialization of values in instances (see page 34).

3.3 Naming and Pointing to Objects

In order to manipulate a LOOPS object, it is necessary to have a pointer to it. One way to do this is to save a pointer to the object in an Interlisp variable, for example:

```
(SETQ myVariable (_ ($ Transistor) New))
```

This creates a new instance of the `Transistor` class, and stores a pointer to this instance in the Interlisp variable `myVariable`. Pointers to instances can also be saved in instance variables.

LOOPS objects may be passed around and examined by Lisp functions. The following function is useful:

`(Object? X)` [Function]

Returns `X` if it is a LOOPS objects, otherwise `NIL`.

Another way to manipulate an object is by giving it a unique ‘‘LOOPS name’’. An object can be given a LOOPS name by sending it the message `SetName`

`(_ object SetName name)` [Message]

Sets the LOOPS name `name` to refer to `object` LOOPS names are unique in a LOOPS environment; the name is assigned in the environment speci ed by the global variable `CurrentEnvironment` (see page 41 for a complete description of environments).

If an attempt is made to assign a name already in use in the environment, and the global ag `ErrorOnNameConflict` = `T`, an error is generated. If `ErrorOnNameConflict` = `NIL`, and there is already an object `oldObject` with that name, the name is unset for `oldObject` and set for `object` without generating an error.

For example, if `I1` is an Interlisp variable whose value is a pointer to some instance, the object can be given the LOOPS name `Foo` as follows:

```
(_ I1 SetName 'Foo)
```

After naming `I1` this way, the user can refer to this object as `($ Foo)`, which returns the object whose name is `Foo`.

The user can refer to an object with a *computed* LOOPS name using the form `($! EXPR )`. For example, if the value of the lisp variable `X` is the atom `Apple`, then `($! X) = ($ Apple)`.

Classes having `NamedObject` (see page 115) as a super class inherit an instance variable, `name`, that contains the name of the objects. Instances of these classes can be named, as before, with a `SetName` message, or alternatively as a side e ect of setting the `name` instance variable.

Class objects are automatically given a LOOPS name when they are created, as described below.

3.4 De ning a New Class

The way one creates a new class is to send the message `New` to a metaclass. Usually, the metaclass named `Class` is used.

`(_ metaClass New className supersList)` [Message]

Returns a new instance of the metaclass `metaClass`. `className` is the new class name and `supersList` is a list of the names of the super classes for this new class. If the list of super class names is omitted, `supersList` defaults to `(Object)`.

Example:

```
(_ ($ Class) New 'StudentEmployee '(Student Employee))
```

This de nes a new class, `StudentEmployee` as a subclass of the known classes named `Student` and `Employee`.

An abbreviated way of de ning a class is to use the function `DC`:

`(DC className supersList)` [Function]

(‘‘de ne class’’) Sends the class `Class` an appropriate `New` message:

```
(_ ($ Class) className supersList)
```

Example:

```
(DC 'StudentEmployee '(Student Employee))
```

This speci es that the class `Student` is to be used recursively, inheriting both from `Student` and all its supers, and from `Employee` and all its supers.

After de ning the class, one can modify its structure by editing the textual source for the class with `EC`:

`(EC className_ )` [Function]

(‘‘edit class’’) `EC` envokes the Interlisp editor on the textual source for the class named `className`

The editor can also be envoked by sending the `Edit` message: `(_ ($ className) Edit)`.

For example, `(EC 'StudentEmployee)` might start the editor editing the expression:

```
[DEFCLASS StudentEmployee
```

```
     (MetaClass Class Edited: (* lc: "18-Oct-82 14:26"))
     (Supers Student Employee)
     (InstanceVariables)
     (Methods]
```

One can then change this to:

```
[DEFCLASS StudentEmployee
     (MetaClass Class Edited: (* lc: "18-Oct-82 14:26"))
     (Supers Student Employee)
     (InstanceVariables
          (sponsor NIL doc (* Name of sponsor))
          (stay     3 doc (* number of months here)))
     (Methods]
```

Leaving the editor successfully at this point would install the two instance vararible descriptions in the class `StudentEmployee`. Then, in addition to those instance variables `StudentEmployee` inherited from `Student` and `Employee`, each instance would also have two new ones, `sponsor` and `stay` with default values of `NIL` and `3` respectively. A more extensive description of editing and changing classes is found in section 13.4.

3.5 De ning a Method

In order to de ne a method for a class, one can use the Interlisp function `DM`:

`(DM className selector argsOrFnName form)` [Function]

De nes a method for the class named `className` that can be called using the selector `selector`. If `form` is non-`NIL`, then `argsOrFnName` is interpreted as the list of arguments for a function, and `form` as the body of that function. If the rst element of the list `argsOrFnName` is not `self`, then `self` is added on the front. `DM` de nes a function whose name is the concatenation of `className`, a period, and `selector`. For example, `Class.List` is the function name created for the `List` selector in the class `Class`. The function de nition is created by substituting into `(LAMBDA argsOrFnName . form)`.

If `argsOrFnName` and `form` are `NIL`, `DM` creates a skeleton de nition for the function and puts the user into the Interlisp editor, editing the skeleton.

If only `form` is `NIL`, `argsOrFnName` is interpreted as the name of a function to be used for implementing the method.

Note: a method can also be de ned by sending the `DefMethod` message to the class: `(_ class DefMethod selector argsOrFnName form)`.

Example:

```
(DM ’Number ’Increment ’(self)
    ’((* incr my IV) (_@ :myValue (ADD1 (@ :myValue)))))
```

This de nes a method with selector `Increment` for the class `Number` which adds 1 to the instance variable `myValue` (the `@`-notation for accessing variables is described on page 18). This form results in the de nition of a function named `Number.Increment` as follows:

```
(DEFINEQ
   (Number.Increment
      (LAMBDA (self)      (* incr my IV)
         (_@ :myValue(ADD1 (@ :myValue)))]
```

`(EM classNameselector )` [Function]

Calls the Interlisp editor to edit the method for the class named `className` associated with the selector `selector`

Often it is more conveniently to use the LOOPS browser to edit the code for a method (see page 102).

Example:

To edit the method from the example above, one could type:

```
(EM 'Number 'Increment)
```

This will edit the method of class `Number` which responds to the selector `Increment`, whether or not it has a name of the standard form.

4 OBJECT VARIABLES AND PROPERTIES

There are two kinds of variables associated with an instance: its private *instance variables* and the *class variables* that it shares with all instances of the class. This section deals with the functions for getting and putting values, and with a compact programming notation for referring to these variables from inside functions that implement methods. In addition, there are properties which are associated with instance variables and class variables, with the methods of a class, and with classes themselves. Given an object or a class, one can fetch or set any of these properties. This section describes the functions for accessing all of these properties and values.

4.1 Access Expressions

As mentioned above, there are a number of di erent types of variables and properties that can be associated with each class. However, most of the accessing operations (getting and putting) in methods refer to the values or properties of instance variables or class variables of an instance. LOOPS provides general functions (described later) for accessing these values, allowing variable names and property names to be computed. However, most of the time the programmer knows the variable and property name to be used, and writing calls to these functions can be cumbersome.

Therefore, a simpli ed notation has been introduced for writing many common accessing operations, which is translated into calls to the appropriate functions:

`(@ object accessExpr)` [Macro]
`(@ accessExpr)` [Macro]

Returns the variable or property value of the object `object` as speci ed by `accessExpr`. Note that `accessExpr` is not evaluated; `object` is evaluated.

If only one argument is given to `@`, it is assumed that the object is bound to the variable `self`. This is very useful because by convention the rst argument to any method is named `self`.

`(_@ object accessExpr newValue)` [Macro]
`(_@ accessExpr newValue)` [Macro]

Similar to `@`, sets the value of the variable or property speci ed by `accessExpr` (unevaluated) in the object `object` to `newValue`. Returns `newValue`. Note that `accessExpr` is not evaluated; the other arguments are evaluated.

Like `@`, if `object` is ommitted, it defaults to the value of the variable `self`.

Both `@` and `_@` take the argument `accessExpr` which is an ‘‘access expression’’ which speci es exactly which variable or property value should be retrieved or set. `accessExpr` is an atom which speci es a variable name, an optional property name, and whether the variable is an instance variable or a class variable.

Some examples:

`(@ :FOO)`	Retrieve the value of instance variable `FOO` (from the object that is the value of `self`).
`(@ XX ::FOO)`	Retrieve the value of class variable `FOO` (from the object that is the value of `XX`).

`(_@ ::FOO:,BAR 5)`
Store `5` as the value of the `BAR` property of class variable `FOO` (of the object that is the value of `self`).

4.2 Getting Variable and Property Values

The functions `GetValue` and `GetClassValue` retrieve from an instance the values of variables or their properties. If the value bound to an instance variable or class variable is an *active value* with a `getFn`, then `GetValue` and `GetClassValue` of these functions trigger the `getFn` (see page 25).

`(GetValue object varName propName)` [Function]
Returns the value or property value of the instance variable `varName` in the object `object`. Each instance of a class has its own separate set of instance variables.

If `propName` is `NIL`, `GetValue` returns the value of the variable. In proper usage, `object` is an instance and the local value of the variable is returned. If no local value has been set, `GetValue` returns the default value from the class. Since this is a common case, default values inherited from super classes of the class are cached in the class itself, thus avoiding a runtime search.

If `propName` is not `NIL`, `GetValue` returns the value associated with the property named `propName` of the variable `varName`. If none is found in the instance, it returns the default property value found in the class or one of its super classes. If no property value is found in any of the super classes, the default value used is the value of the global variable `NotSetValue` (currently bound to `?`). Note: this is di erent from Interlisp, where if no value of a property is found, then `NIL` is returned.

`GetValue` fetches a value from an *instance* of a class. It is an error to try to use `GetValue` to fetch an instance variable from a class. To fetch the default value of an instance variable from a class, use `GetClassIV` (see page 22).

`(GetClassValue object varName propName)` [Function]
Returns the value (if `propName` = `NIL`) or property value of the class variable `varName` for the class of the `object` (which may be either an instance or a class).

Class variables are inherited from the super classes. If `object` is an instance, lookup begins at the class of `object` since instances do not have class variables stored locally. If the class does not have a class variable `varName`, `GetClassValue` searches through the super classes of the class until it nds `varName`. Since this is thought to be an relatively rare in code, class variables are stored only in the class in which they are de ned, and the runtime search is necessary.

Conceptually, one should think of a class variable of a class as being shared by all instances of that class, and by all instances of any of its subclasses. For example, suppose `Transistor` is a class with class variable, `TransSeqNum`, and `DepletionTransistor` is a subclass of `Transistor`. Then setting the class variable `TransSeqNum` from an instance of `DepletionTransistor` would be seen by all instances of `Transistor`.

4.3 Putting Variable Values and Property Values

`PutValue` and `PutClassValue` are functions used for storing variable or property values in an instance. They are analogous to `GetValue` and `GetClassValue`; as with these functions, if the value of the variable or property is an active value with a `putFn`, trying to store a value for that variable or property will invoke the `putFn` (see page 25).

`(PutValue object varName newValue propName)` [Function]

Stores `newValue` as the value or property value of the instance variable `varName` in the object `object`. Returns `newValue`.

If `propName` is `NIL`, `PutValue` stores `newValue` as the value of `varName` in `object`. If `propName` is non-`NIL`, then `newValue` is stored as the value of the property `propName` of the instance variable `varName`.

For example, `(PutValue pos 'X 0)`, stores `0` as the value of the instance variable `X` of the object `pos`.

`PutValue` works for storing values in an instance of a class. It is an error to try to store a default instance variable in a class with `PutValue`. To store the default value for an instance variable directly in the class, use `PutClassIV` (see page 22).

`(PutClassValue object varName newValue propName)` [Function]

Similar to `PutValue`, except it stores `newValue` as the value or property value of a class variable and property. `object` may either be an instance or a class. Returns `newValue`.

If `varName` is not local to the class, then the value will be put in the first class in the inheritance list that `varName` is found.

The following functions push a value on the front of a list already stored in a variable:

`(PushValue object varName newValue propName)` [Function]
`(PushClassValue object varName newValue propName)` [Function]

`PushValue` and `PushClassValue` add `newValue` on the front of the list that is the value of the indicated variable or property, and store the result back in the variable or property.

These functions are defined so that if the value accessed is an active value, the `getFn` will be triggered when the old value of the list is fetched, and the `putFn` when the new value is stored back (see page 25).

The following function adds a value on the end of an instance variable list:

`(AddValue object varName newValue propName)` [Function]

Similar to `PushValue`, except that `newValue` is added to the *end* of the variable list.

There is no function for adding values to the end of class variable lists.

4.4 Non-triggering Get and Put

Using active values (page 25), it is possible to associate functions with a variable (or property) that will be called whenever the variable (or property) is read or set. In some cases, it is useful to be able to access a value from an instance or class variable without triggering any active value which might be stored. This can be done using the following functions:

`(GetValueOnly object varName propName)` [Function]
`(GetClassValueOnly object varName propName)` [Function]

`GetValueOnly` and `GetClassValueOnly` retrieve the value of instance variables and class variables, respectively, without triggering any active values.

`GetValueOnly` retrieves the default value from the class if none exists in the instance.

To store a value without triggering any active values, the following functions are provided:

`(PutValueOnly object varName newValue propName)` [Function]
`(PutClassValueOnly object varName newValue propName)` [Function]

These functions store `newValue` in the instance variable or class variable, without triggering any active values, and return `newValue`.

Note that `GetClassValueOnly` and `PutClassValueOnly` can take either a class or an instance. `GetValueOnly` and `PutValueOnly` will only take instances.

4.5 Local Get Functions

Sometimes it is desirable to nd out if a value or property is set in a particular class or instance, without inheriting any information which is not local, and not activating any active values. This can be done with the following functions:

`(GetIVHere object varName propName)` [Function]

`object` must be an instance. Returns the instance variable value that is found in the instance; if none is found, then returns the value of the global variable `NotSetValue` (initially `?`).

`(GetCVHere object varName propName)` [Function]

`object` must be a class. Returns the class variable value that is found in the class; if none is found, then returns the value of `NotSetValue`.

In both `GetIVHere` and `GetCVHere`, if the value is an active value, the actual active value is returned, without being triggered.

Note that there are no need to have special local put functions, since all put functions are local to the instance or class. For local nontriggering storage functions, use `PutValueOnly` and `PutClassValueOnly` (page 21).

4.6 Accessing Class and Method Properties

Most of the get and put functions described in the preceding sections work with instances, but not with classes. Some exceptions are `GetClassValue`, `PutClassValue`, `GetClassValueOnly`, and `PutClassValueOnly`, which can take either an instance or a class, and access class variables, and `GetCVHere` which takes a class.

The following functions access the *default* value or property value of an instance variable (which is stored in the class):

`(GetClassIV class varName propName)` [Function]

Returns the *default* value or property value of the instance variable `varName` in the class `class`.

`(PutClassIV class varName newValue propName)` [Function]

Stores `newValue` as the *default* value or property value of the instance variable `varName` in the class `class`. If `varName` is not local to the class, this will cause an error. Returns `newValue`.

Note: `GetClassIV` and `PutClassIV` do not trigger active values (page 21).

LOOPS provides property list storage for classes themselves and for methods of classes. A typical use of these properties is to document a class and its methods. Like the put and get functions for variables, these functions can trigger active values. The functions for class properties are:

`(GetClass class propName)` [Function]

Returns the value of the property `propName` of `class`. If `propName` is `NIL`, `GetClass` returns the metaclass of `class`.

Class properties are inherited like class variables, so `GetClass` will search through the super classes of `class` if `propName` is not found in `class` itself.

`(PutClass class newValue propName)` [Function]

Sets the value of the property `propName` of `class` to `newValue`. If `propName` is `NIL`, `GetClass` sets the metaclass of `class` to `newValue`.

`(GetClassOnly class propName _ )` [Function]
`(PutClassOnly class newValue propName)` [Function]

These functions are analogous to `GetClass` and `PutClass`, except that they never trigger active values.

`(GetClassHere class propName)` [Function]

Returns the local value of the property `propName` of `class`. If `propName` is not found locally, `GetClassHere` returns the value of the global variable `NotSetValue` (initially `?`).

The functions for accessing method properties are:

`(GetMethod class selector propName)` [Function]

If `propName` is `NIL`, `GetMethod` returns the method (Interlisp function name) which implements the message `selector` of the class `class`. If `propName` is non-`NIL`, it returns the value of the property `propName` of the method.

Method properties are inherited; the retrieval process involves searching through super classes of `class` if the property is not found in `class` itself.

`(PutMethod class selector newValue propName)` [Function]

If `propName` is `NIL`, `PutMethod` sets the method which implements the message `selector` of the class `class` to `newValue`. If `propName` is non-`NIL`, it sets the value of the property `propName` of the method to `newValue`. Returns `newValue`.

`(GetMethodOnly class selector propName)` [Function]
`(PutMethodOnly class selector newValue propName)` [Function]

Analogous to `GetMethod` and `PutMethod` except that they never trigger active values.

`(GetMethodHere class selector propName)` [Function]

Returns the local value of the property `propName` the the method which implements the message `selector` of `class`. If `propName` is not found locally, `GetMethodHere` returns the value of the global variable `NotSetValue` (initially `?`).

All of the above functions only work directly on classes, not on instances of those classes. In addition, if a method or class variable is inherited, then the put functions change the property in the class in which the method or class variable is found in the supers list, not in the class which was the argument of the put function.

4.7 General Get and Put Functions

The following functions are generalized get and put functions which accept a type argument and invoke the more specialized functions:

`(GetIt object varOrSelector propName type)` [Function]
`(PutIt object varOrSelector newValue propName type)` [Function]
`(GetItOnly object varOrSelector propName type)` [Function]
`(PutItOnly object varOrSelector newValue propName type)` [Function]
`(GetItHere object varOrSelector propName type)` [Function]

For all of these functions, the value of the `type` argument can be one of `IV`, `CV`, `CLASS`, or `METHOD` for instance variable, class variable, class, or method, respectively. If `type` is `NIL`, `IV` is assumed. The argument `varOrSelector` is interpreted as a variable name if `type` is `IV` or `CV`, a selector name if `type` is `METHOD`, and is ignored if `type` is `CLASS`.

These functions are interpreted as follows:

```
(GetIt    'IV)  ==>  (GetValue   )
(GetIt    'CV)  ==>  (GetClassValue    )
(GetIt    'CLASS)  ==>  (GetClass    )
(GetIt    'METHOD)  ==>  (GetMethod    )
```

The other functions are similar.

Note: Actually, if `type` = `IV`, these functions will call di erent functions depending on whether the object is a class or instance.

4.8 Summary of Get and Put Functions

In the following table, * indicates that no function is available.

	Inherit/Trigger	Inherit/DontTrigger	Local/DontTrigger
from instances:			
Get/Put fns for instance variables	`GetValue` / `PutValue`	`GetValueOnly` / `PutValueOnly`	`GetIVHere`
Get/Put fns for class variables	`GetClassValue` / `PutClassValue`	`GetClassValueOnly` / `PutClassValueOnly`	*
from classes:			
Get/Put fns for instance variables	*	*	`GetClassIV` / `PutClassIV`
Get/Put fns for class variables	`GetClassValue` / `PutClassValue`	`GetClassValueOnly` / `PutClassValueOnly`	`GetCVHere`
Get/Put fns for class properties	`GetClass` / `PutClass`	`GetClassValueOnly` / `PutClassValueOnly`	`GetClassHere`
Get/Put fns for method properties	`GetMethod` / `PutMethod`	`GetMethodOnly` / `PutMethodOnly`	`GetMethodHere`

5 ACTIVE VALUES

Active values provide a way of invoking procedures when the value of a variable (or property) is read or set. This mechanism is dual to the notion of messages; messages are a way of telling objects to perform operations, which can change their variables as a side effect; active values are a way of accessing variables, which can send messages as a side effect. This section presents the notation for creating active values. Then, the concept of nested active values is introduced. The nesting property enables many of the important applications of active values by supporting composition of the access functions. Next is described how to use active values as the default values in a class, and how to share them. Finally, the standard arguments to active value access functions are described, along with LOOPS functions that can be used in user-defined access functions.

5.1 Active Values Notation

The notation for an active value illustrates its three parts:

```
#(localState getFn putFn)
```

This notation is converted by a read macro into an instance of the Interlisp data type `activeValue`. The `localState` field is used as a place for storing data. The `getFn` and `putFn` are the names of functions that are applied with standard arguments when a program tries to get or put the value of a variable whose value is an active value. Every active value need not specify both a `getFn` and a `putFn`. If the `getFn` is `NIL`, then a get operation returns the local state. If the `putFn` is `NIL`, then a put operation replaces the local state.

5.2 Nested Active Values

Often it is desirable to associate multiple access functions with a variable. For example, we may want more than one process to monitor the state of some objects (e.g., a debugging process and a display process). To preserve the isolation of these processes, it is important that they be able to work independently. LOOPS uses nested active values as a way of composing these functions.

Nested active values are arranged so that the innermost active value is stored in the `localState` of the penultimate `localState`, and the outermost active value is the immediate value of the variable. Put operations to a variable through such nested active values trigger the `putFn`s in sequence from the outermost to the innermost. For example, suppose the variable tracing facility were used to trace access of the `position` variable from the model/view controller example (page 10). The resulting active value would look like

```
#( #(Pos1 NIL UpdateDisplay) GettingTracedVar SettingTracedVar)
```

An attempt to set the position variable would cause the function `SettingTracedVar` to be called with the new value as one of its arguments. `SettingTracedVar` would operate and call the LOOPS function `PutLocalState` to set its own `localState`. This, in turn, would trigger the inner active value causing `UpdateDisplay` to be invoked.

Get operations work in the opposite order. If there are three nested active values, a request to get the value will cause the innermost `getFn` (if any) to run, followed by the middle `getFn` (if any), followed by the outermost getFn (if any) whose value is returned by the get operation. Each getFn sees only the value returned by the next nested getFn, and the innermost getFn sees the value stored in its localState.

LOOPS provides functions for embedding and removing active values from variables. This idea of functional composition for nested active values is most appropriate when the order of composition does not matter. We have resisted the development of other combinators for the functions using the same parsimony arguments that we used earlier about specializing and combining methods. Just as inheritance from multiple super classes works most simply when the super classes describe independent features, active values work most simply when they interface between independent processes using simple functional composition. Any more sophisticated control is seen as overloading the active value mechanism. The escape for more complex cases is to combine the implicit access functions using Interlisp control structures to express the interactions.

5.3 Active Values as Default Values

Suppose that I is an instance of a class with an instance variable V, whose default value is the active value A. Further suppose that the value of V in the instance I has never been set. The first time (PutValue I V exp) is invoked, a copy of A is made. This copy is inserted in the instance itself as the the value of the instance variable, with pointers to the same contents as A. Then the putFn is invoked, with the copy as the activeVal argument; this copy of A provides a place where local state can be stored private to I.

In some cases, one knows that the putFn will not actually write into the active value, and therefore the active value which is the default could be shared instead of needing to be copied. To indicate this, the localState of A should be made the atom Shared. In the example below, the user knows that no change will be made in A itself and thus uses a shared active value.

Example: SUM is a class with three instance variables, top, bottom, and sum; top and bottom start with default values of 0, and sum is to be computed when asked for. One cannot update sum independently.

```
[DEFCLASS SUM
     (MetaClass Class)
     (Supers Object)
     (InstanceVariables
          (top 0)
          (bottom 0)
          (sum #(Shared ComputeSum NoUpdatePermitted))
     (ClassVariables)
     (Methods
          (printOn PrintColumn)]
```

The method for printOn used in this example, and the getFn, ComputeSum, and the putFn, NoUpdatePermitted, are Lisp functions whose definitions are not shown here. NoUpdatePermitted is available as part of the kernel.

5.4 Standard Access Functions

LOOPS provides a convenient set of functions for some common applications. For example, NoUpdatePermitted, described in the example above, is used to stop update of the localState of an active value. FirstFetch is a standard getFn that expects the localState of its active value to be an Interlisp expression to be evaluated; on the rst fetch, the instance variable is set to the result of evaluating the expression. This is illustrated in gure 5, which shows a class `TestDatum` that describes an instance variable `sampleX`, to be computed on the rst time that it is fetched, and then cached for future references. At the time of activation of `FirstFetch`, `self` and `varName` are bound to the instance and instance variable name in which the active value was found.

```
(DEFCLASS TestDatum
     (MetaClass Class)
     (...
     (InstanceVariables (sampleX #((RAND 0. 100.) FirstFetch)))...)
```

Figure 5. Using an active value to compute and cache a value for a variable on the rst fetch.

In some applications it is important to be able to access values indirectly from other instances. For example, Steele [Steele80] has recommended this as approach for implementing equality constraints. gure 6 shows a way of achieving this by using using the standard access functions `GetIndirect` and `PutIndirect`.

```
(DEFINST JoeAsFatherPerspective ...
     (InstanceVariables
        (age #((#$JoeAsManPerspective age) GetIndirect PutIndirect))
        ...
```

Figure 6. Active values can be used to provide indirect access to values. This is useful when it is desired for a variable in one instance to re ect the value of a variable stored elsewhere. In this example, the instance `#$JoeAsFatherPerspective` has an `age` variable which always has the same value as the `age` variable of the instance `JoeAsManPerspective`.

For some uses, the user may want to compute a default value if given, but replace the active value by the value given if the user sets the value of a variable. For this the user can employ the system provided `putFn` of `ReplaceMe`, as in:

```
#(NIL ComputeGoodValue ReplaceMe)
```

If this value is made the default in a class, then when a program tries to set this value, the instance will contain the value set. However, if the user tried to fetch the value form this variable before setting it, the `getFnComputeGoodValue` would be invoked.

5.5 User-De ned Access Functions

The `getFn` and `putFn` of an active value are functions that are called with standard arguments:

```
(self varName oldOrNewValue propName activeVal type)
```

These arguments are interpreted as follows:

`self`	The object containing this active value.
`varName`	The name of the variable where this active value was stored. This is `NIL` if it is not stored in a variable.

`oldOrNewValue`	For a `getFn`, this is the `localState` of the active value. For a `putFn`, this is the new value to be stored in the active value.
`propName`	The name of a property. This is `NIL` if the active value is not associated with the value of a property (i.e., if it is associated with the value of the variable itself).
`activeVal`	The active value in which this `getFn` or `putFn` was found.
`type`	This specifies where the active value is stored; `NIL` means a instance variable, `CV` means a class variable, `CLASS` means a class property, or `METHOD` means a method property.

The value returned by the `getFn` is returned as the value of the get operation.

The `putFn` is expected to make any necessary changes to the `localState`. This can be done using function `PutLocalState` described below. In changing the `localState`, embedded active values may be triggered.

Given an active value, the following functions can be used to retrieve or store its `localState`:

`(GetLocalState activeValue self varName propName type)` [Function]
`(PutLocalState activeValue newValue self varName propName type)` [Function]

`GetLocalState` returns the `localState` of the active value `activeValue`. `PutLocalState` stores `newValue` as the `localState` of the active value `activeValue`, and returns `newValue`.

Note that it is necessary to pass these functions the values for `self`, `varName`, `propName`, and `type`, in case any imbedded active values are triggered.

If the `localState` of the active value is itself an active value, then it will be triggered to obtain the `localState` argument for the `getFn`. For a `putFn`, an embedded active value will be triggered when the `putFn` calls `PutLocalState`. The following functions can be used to access the `localState` of an active value without triggering any embedded active values:

`(GetLocalStateOnly activeValue)` [Function]
`(PutLocalStateOnly activeValue newValue)` [Function]

`GetLocalStateOnly` returns the value of the `localState` of the active value `activeValue`. `PutLocalStateOnly` stores `newValue` as the `localState` of the active value `activeValue`, and returns `newValue`. Both functions access the `localState` without triggering embedded active values.

In some cases, it is important to be able to replace the entire active value expression by some quantity, independent of the depth of nesting of active values, without destroying the outer levels of nesting:

`(ReplaceActiveValue activeVal newValue self varName propName type)` [Function]

`ReplaceActiveValue` overwrites `activeVal` whereever it is (either directly as the value or property of an instance variable, or as the local state of an embedded active value) with `newValue`.

`ReplaceActiveValue` searches the value (property) determined by its arguments until it finds `activeVal` in the nesting. If `activeVal` is not found, an error is invoked.

Example: Suppose that we have a class `RandomDatum` which describes an instance variable `sampleX`, which we want to be computed as a random number on the first time that it is fetched, and then returned as a constant on all future fetches. We could do this by defining the class as follows:

```
(DEFCLASS RandomDatum
     (MetaClass Class)
     (...
     (InstanceVariables (sampleX #(NIL SmashRandom ReplaceMe)))
     ...)
```

where the function `SmashRandom` is defined as follows:

```
(LAMBDA (self varName value propName activeValue)
     (ReplaceActiveValue activeValue (RAND 0. 100.) self varName]
```

On the first fetch of the value of `sampleX` in any instance of `RandomDatum`, the function `SmashRandom` over-writes the active value with a random number. This is a special case of the active value function `FirstFetch` described earlier.

The function `MakeActiveValue` is used to make the value of some variable or property be an active value:

`(MakeActiveValue self varOrSelector newGetFn newPutFn newLocalSt propName type)` [Function]

`self` is the object, `varName` is typically the name of a variable when the active value is being placed in an instance variable. If the active value is being placed in a method, then `varName` should be bound to the selector name. Active values can also be used for class variables, or properties of instance or class variables, or methods. The interpretation of where to create the active value is determined by the argument `type`, which must be one of `IV` (or `NIL`), `CV`, `CLASS`, or `METHOD`.

If `newLocalSt` = `EMBED`, then a new active value is always created, containing as its `localState` whatever was found by `GetItOnly` (page 23). For other values of `newLocalSt`, an active value is created only if the current value is not an active value; otherwise the old one is simply updated with `newLocalSt`, `newGetFn`, and `newPutFn`.

If an old active value is being updated, then if `newGetFn` or `newPutFn` is `NIL`, the old `getFn` or `putFn` is not overwritten. If `newGetFn` or `newPutFn` is `T`, the old `getFn` or `putFn` is reset to `NIL`.

The easiest way to define a function for use in active values is to use the function `DefAVP`:

`(DefAVP fnName putFlg)` [Function]

`DefAVP` creates a template for defining an active value function and leaves the user in the Interlisp editor. `fnName` will be the name of the function and `putFlg` is `T` if this is to be a `putFn` and `NIL` if it is to be a `getFn`.

For `getFn`s, the template is

```
[LAMBDA (self varName localSt propName activeVal type)
     (* This is a getFn for ...)
              localSt]
```

This template incorporates the standard arguments that a `getFn` receives, and the convention that they often return the value that is in their local state.

For putFns, the template is

```
[LAMBDA (self varName newValue propName activeVal type)
     (* This is a putFn for ...)
          (PutLocalState activeVal newValue self varName propName type)]
```

This template incorporates the standard arguments that a putFn receives, and the convention that they often put their resulting newValue in the localState

6 COMBINING INHERITED METHODS

In practice, most methods used to manipulate LOOPS objects are inherited. In the simplest examples of multiple inheritance, classes represent independent features and there is no con ict between inherited methods. However, when features inherited from classes interact, it is essential to be able to describe how to combine them. Howard Cannon recognized this ‘‘mixing issue’’ as central in the design of Flavors:

> ‘‘To restate the fundamental problem: there are several separate (orthogonal) *attributes* that an object wants to have; various *facets* of behavior (features) that want to be independently speci ed for an object. For example, a window has a certain behavior as a rectangular area on a bit-mapped display. It also has its behavior as a labeled thing, and as a bordered thing. Each of these three behaviors is di erent, wants to be speci ed independently for each object, and is *essentially* orthogonal to the others. It is this ’’essentially‘‘ that causes the trouble.’’
>
> ‘‘It is very easy to combine completely non-interacting behaviors. Each would have its own set of messages, its own instance variables, and would never need to know about other objects with which it would be combined. Either the multiple object or simple multiple superclass scheme could handle this perfectly. The problem arises when it is necessary to have *modular* interactions between the orthogonal issues. Though the label does not interact *strongly* with either the window or the border, it does have some minor interactions. For example it wants to get redrawn when the window gets refreshed. Handling these sorts of interactions is the Flavor system’s main goal.’’
>
> ... from [Cannon82]

This section considers cases where the inherited features interact, and describes some LOOPS facilities for combining interacting methods. First, we describe a way of combining an inherited method with local method code. Next, we describe other ways of combining methods inherited from multiple super classes. Finally, we describe some special functions one can use to ‘‘escape’’ from the normal method inheritence conventions.

6.1 Augmenting an Inherited Method

The inheritance examples shown previously considered only cases where methods are inherited in toto. In these examples, subclasses inherit a method or value unchanged, or they override it completely. No mechanism was described that would enable a subclass to track changes in a method after it had been specialized in some way.

For combining an inherited method with local code, LOOPS provides the special method invocation `_Super`.

`(_Super object selector arg`$_1$ `arg`$_N$`)` [NLambda NoSpread Function]

`object` is the object to which the method is applied (typically `self`), `selector` is the selector for the method and `arg`$_1$ `arg`$_N$ are the arguments for the method. As with `_`, `selector` is not evaluated; the remaining arguments are evaluated.

`_Super` provides a form of relative addressing; it invokes the next more general method of the same name even when the specialized method invoking `_Super` is inherited over a distance. An example of the use of `_Super` is given in gure 7.

Note: SENDSUPER can be used instead of _Super.

```
(BorderedWindow.Refresh
   [LAMBDA (self)         (* mjs: "11-JAN-82 19:28")

      (* * Method for refreshing a window that has a border)

                         (* First use the refresh method
                            inherited from Window.)
      (_Super self Refresh)
                         (* Then Re-display the border.)
      (_ (@ :border) Display)
      self])
```

Figure 7. This Interlisp procedure implements the Refresh message for the class BorderedWindow. It uses _Super to invoke the more general method in the class Window. The object for the ‘‘border’’ of the bordered window is in the instance variable border. The specialized method returns the bordered window as its value. In more complicated examples, calls to _Super and _ can be combined using Interlisp iterative and conditional statements.

6.2 Combining Multiple Inherited Methods

Using _Super, a method can invoke the *single* next general method. However, when a class has multiple super classes, sometimes it is necessary to invoke the general methods from *each* of the super classes. In this situation, one can call _SuperFringe:

(_SuperFringe object selector arg_1 ... arg_N) [NLambda NoSpread Function]

This is similar to _Super, except that _SuperFringe invokes the next more general method of the same name for *each* of the super classes on the supers list of the class of the currently- executing method.

6.3 General Method Invocation

The functions _Super and _SuperFringe have proved to be sufficient for implementing most methods. However, sometimes it is necessary to manipulate multiple inherited methods, and invoke them in some other order. The following functions provide more general ways of invoking particular methods. It is important to note that while these functions are more powerful than _Super or _SuperFringe, they are also more ‘‘dangerous’’, in that they do not conform to the conventions of method inheritence. These functions should only be used as a last resort when a method cannot be implemented in any other way.

(DoMethod object selectorExpr class arg_1 ... arg_N) [NLambda NoSpread Function]

DoMethod allows computation of the name of the selector and the class from which that method should be found; it applies that method to object

All the arguments to DoMethod are evaluated; selectorExpr should evaluate to a selector name in the class computed from class. If class is NIL, then the class of object is used. If no method for the computed selector is found in the computed class, an error is generated. The remaining arguments, arg_1 ... arg_N are the arguments for the method.

In the case where the arguments to the method have already been evaluated, then one can use `ApplyMethod` instead of `DoMethod`:

`(ApplyMethod object selector argList class)` [Function]

`argList` is a list of all the arguments to the method (except `object`) already evaluated. The function applied is the one found by searching from `class`. If `class` is `NIL`, the class of `object` is used.

`(DoFringeMethods object selectorExpr arg`$_1$ … `arg`$_N$`)` [NLambda NoSpread Function]

Like `DoMethod`, all of the arguments are evaluated. `DoFringeMethods` calls the method for `selectorExpr` in the class of `object`, if that method is defined in that class. If the method is not defined in the class of `object`, the method of the same name for *each* of the super classes on the supers list of the class of `object` is envoked.

7 INSTANCE CREATION

The standard process of creating an instance of a class is to send a `New` message to the class. In the simplest case, this causes the information in the *instance variable descriptions* of the class to be used to establish default values for variables in the newly created instance. When that process is nished, the instance can be altered in various ways by sending it messages.

LOOPS provides a variety of facilities for controlling this by using active values, standard access functions, and metaclasses. This section summarizes some of the common cases. See page 38 for an illustratation of the use of these facilities to support the important example of composite objects.

7.1 Specifying Values at Instance Creation

The `NewWithValues` message simpli es the case where it is desired to specify values and properties in an instance when it is created. The form of this message is:

`(_ class NewWithValues valDescriptionList)` [Message]

`valDescriptionList` must evaluate to a list of value descriptions, each of which is a list of a variable name, variable value, and properties; e.g.

```
((varName₁ value₁ prop₁ propVal₁ ... )
 (varName₂ value₂ ... )
   ... )
```

The method for `NewWithValues` rst creates the object with *no* other initialization (e.g. without computing values speci ed in the class, as described in sections below). It then directly installs the values and property lists speci ed in `valDescriptionList` and returns the created object. Variables which have no description in `valDescriptionList` will be given no value in the instance, and thus will inherit the default value from the class.

7.2 Sending a Message at Instance Creation

A simpli cation in form is available when one wants to send a message to an instance immediately after its creation. For example, consider:

```
(_ (_ ($ Transistor) New) Display windowCenter)
```

which creates an instance of the `Transistor` class, and then displays it at a point `windowCenter`. A more compact notation for doing this is provided:

```
(_New ($ Transistor) Display windowCenter)
```

where `_New` (‘‘send New’’) means to create a new instance and send it a message. The value returned by `_New` is the new instance. Any value returned by the method is discarded.

In order to name an object, one can send the message `SetName` to that object. As a simpli cation, if one provides an argument to the `New` message, the default interpretation of that argument is to use it as a name, sending the newly created object the `SetName` message.

7.3 Computing a Value at First Fetch

As described earlier, one can use an active value to activate arbitrary procedures when values are fetched. The built-in function `FirstFetch` can be used as a `getFn` in an active value as the default value in the class. If no value has been assigned to the variable or property before the value is fetched for the rst time, the `FirstFetch` active value is invoked.

The local state of this active value can be a list which is a form to be evaluated. During the evaluation, the variables `self`, `varName`, and `propName` are are appropriately bound. The local state of the `FirstFetch` active value can also be an atom; if so, it is treated as the name of a function to be applied to the object, `varName` and `propName`. The value of the form or function application is made the value in the instance as well as being returned as the value of the fetch.

For example, the random number example could have been done as follows:

```
(DEFCLASS TestDatum
     (MetaClass Class)
     (...
     (InstanceVariables (sampleX #((RAND 0. 100.) FirstFetch)))
     ...)
```

In this example `FirstFetch` evaluates the form `(RAND 0. 100.)` and replaces the value of the `sampleX` variable of the instance by the random number. In many cases the form may be a _ expression.

7.4 Computing a Value at Instance Creation

In the previous example, `FirstFetch` initializes the value of an instance variables at rst access. Sometimes it is important to initialize an instance variable when the instance is created. For such cases LOOPS provides a distinguished `getFn`, `AtCreation`. If a default value of an instance variable or property contains an active value with `AtCreation` as its `getFn`, then at creation time, the `localState` of this active value will be used to determine a value to be inserted in the new instance.

As with `FirstFetch`, if the `localState` is an atom, then it will be treated as the name of a function to be applied to the object, variable name, and property name. If it is a list, then that list will be evaluated in a context in which `self`, `varName`, and `propName` are appropriately bound. Functions run at initialization time are run in the order in which they appear in the class. Default values of variables are available to these functions.

If an object is created by `NewWithValues` without a value being supplied for a variable which contains an `AtCreation` default value, then at the rst fetch of that variable, the function or form will be evaluated.

Example:

Suppose we want to have an instance variable called `creationDate` which tells the date that an instance was created. This can be implemented in LOOPS as follows:

```
(DEFCLASS DatedObject
     (MetaClass Class)
     (...
     (InstanceVariables (creationDate #((DATE) AtCreation)))
     ...)
```

The function DATE in Interlisp computes a string which is the current date and time. The value of this string at instance creation time is made the intitial value of creationDate.

Another use of an AtCreation active value might be to make an index entry to a newly created object.

7.5 Special Actions at Instance Creation

For some special cases, the user may want to have more control over the creation of instances. For example, LOOPS itself uses di erent LISP data types to represent classes and instances. The New message for classes is elded by their metaclass, usually the object MetaClass. This section shows how to create a new metaclass.

Any metaclass should have Class as one of its super classes and MetaClass as its metaclass. The easiest way to create a new metaclass is to send a New message to MetaClass as follows:

```
(_ ($ MetaClass) New metaClassNamesupers)
```

This creates a new metaclass with the name metaClassNameand with the super classes named in the list supers.The default supers for metaclasses is the list containing Class. The metaclass for the the new class is MetaClass.

One then installs the specialized method for New in the new metaclass. This method provides the mechanism for creations of instances of the class which have this as a metaclass. Sending this metaclass the message New will cause the creation of a class with the appropriate property.

As a simple example we will de ne a new metaclass ListMetaClass which will augment the instance creation process by keeping a list of all instances which have been created. This list will be kept on the class property allInstances. To create this class we go through the scenario in gure 8.

```
_ (_ ($ MetaClass) New 'ListMetaClass '(Class))
#$ListMetaClass          We have now de ned a new metaclass

                         This de nes the New method for that metaclass
_ (DM 'ListMetaClass 'New '(self name)
    '((* Create an instance and add it to list in class)
      (PROG ((newObj (_Super self New name)))
                 (* newObj created by super method from class)
         (PutClass
            self
            (CONS newObj
                  (LISTP (GetClassHere self 'AllInstances)))
            'AllInstances)
                 (* LISTP returns previous list or NIL if none)
         (RETURN newObj]
ListMetaClass.New

_ (_ ($ ListMetaClass) New 'Book)
```

```
#$Book                          This creates a new class ($ Book)
                              whose metaclass is ($ ListMetaClass)
_ (_ ($ Book) New 'B1)
#$B1                            Creating #$B1  using ListMetaClass.New
_ (_ ($ Book) New 'B2)
#$B2
_ (GetClass ($ Book) 'AllInstances)
(#$B1 #$B2)                     The list of instances created so far.
```

Figure 8. In this scenario, a new metaclass `ListMetaClass` is de ned by the `New` method of `($ MetaClass)`. It has metaclass `($ MetaClass)`. We then de ne the specialized `New` method for `ListMetaClass`. This includes a call to its super (`Class`) to actually create the object; it puts the newly created object on its list of objects. We then create `($ Book)` which has `ListMetaClass` as its metaclass. When two instances of book are created, each is placed on the list `AllInstances` which is a class property.

8 COMPOSITE OBJECTS

LOOPS extends the notion of objects to make it recursive under composition, so that one can instantiate a group of related objects as an entity. This is especially useful when relative relationships between members of the group must be isomorphic (but not equal) for distinct instances of the group. The implementation of composite objects combines many of the programming features described above. In particular, it is an application of the notion of metaclass.

8.1 Basic Concepts for Composite Objects

Parameters and Constants: LOOPS supports the use of structural templates to describe composite objects having a xed set of parts. Composite objects are normal LOOPS objects, created by an instantiation process and describable in the class inheritance network. This contrasts with the idea of using for templates data structures that are merely *copied* to yield composite objects. A primary bene t of making composite objects be classes is the ability to create slightly modi ed versions of a template by making a new subclass which inherits most of the structure of its super.

Creating a Template: To describe a composite object, one creates a class whose metaclass is `Template`. One can also use a metaclass one of whose supers is `Template`. Any class whose metaclass is `Template` or one of its subclasses is called a template. In a template, the default values for instance variables can point to other templates; these will be treated as *parameters* and will be recursively instantiated when the parent template is instantiated. All non-template classes and any other default values are treated as *constants* that are simply inherited by instances.

Instantiation: Instances of a template are created by sending it a `New` message. The instantiation process is recursive through all of the parameters of a template. Every parameter is instantiated when it is rst encountered. Multiple references to the same parameter are always replaced by references to the same instantiated instance. The instantiated composite object that is created is isomorphic to the original template structure with constants inherited and with distinct instances substituted for distinct templates (parameters). Parameters in lists or active values are found and the containing structure is copied with appropriate substitutions. If a composite object needs multiple distinct instances of the same type (e.g., two inverters), then multiple templates are needed in the description.

Example: gure 9 shows an example from digital design - a composite object for `BitAmplifier` that is composed of two series-connected inverters. The input of the rst inverter is the input of the ampli er, the output of the rst inverter is connected to the input of the second inverter, and the output of the second inverter is the output of the ampli er. Di erent instantiations of `BitAmplifier` contain distinct inverters connected in the same relative way. This example also shows a possible use of active values in templates. The containing composite object is set up so that its *output* instance variable uses an active value to track the value of the output variable of the second inverter.

```
[DEFCLASS BitAmplifier
   (MetaClass Template doc
      (* * Composite object template for an amplifer
           made of two series connected inverters.))
   (Supers Amplifier)
   (ClassVariables)
   (InstanceVariables
      (inputTerminal ($ Inverter1))
      (output #( (($ Inverter2) output) GetIndirect PutIndirect)
         doc (* Data is stored and fetched from the variable
                output in the instance of Inverter2))
   (Methods)]

[DEFCLASS Inverter1
   (MetaClass Template partOf ($ BitAmplifier)
         doc (* Instance variable Input is inherited from Inverter))

   (Supers Inverter)
   (ClassVariables)
   (InstanceVariables
      (output ($ Inverter2)
         doc (* Output connected to second inverter)))
   (Methods)]

(DEFCLASS Inverter2
   (MetaClass Template partOf ($ BitAmplifier) )
   (Supers Inverter)
   (ClassVariables)
   (InstanceVariables
      (input ($ Inverter1)
         doc (* Input connected to first inverter)))
   (Methods)]
```

Figure 9. Composite object templates for a `BitAmplifier`. When instances are made, they will have distinct instances of the two inverters, with their input and output interconnected. The instantiation process must be able to reach (possibly indirectly) all of the parts starting from the class to which the `New` message is sent. In this case, `Inverter1` and `Inverter2` are both mentioned in `BitAmplifier`. The example also illustrates the use of active values to provide indirect variable access in LOOPS. In this example, the active value enables the output variable of an instance of `BitAmplifier` to track the corresponding output variable of an instance of `Inverter2` in the same composite object.

8.2 Specializing Composite Objects

Because the templates are classes, all of the power of the inheritance network is automatically available for describing and specializing composite objects. To make this convenient, one can send the message `Specialize` to any template form. For example:

```
(_ ($ BitAmplifier) Specialize)
```

This creates a new set of templates such that each template in the new set is a specialization of a template in the old set. One can then selectively edit the templates describing the new composite object. In particular, one may want to change the names of the generated classes by sending them the message `SetName`. Unchanged portions of the template structure will continue to inherit values from the parent composite object. A user can specialize a template by overriding instance variables. To add parameters, one creates references to new templates. Conversely, one can make a parameter into a constant by overriding an inherited variable value with a non-template in a subclass.

8.3 Conditional and Iterative Templates

Because the templates are xed, they are not a su cient mechanism for describing the instantiation of composite objects having conditional or repetitive parts. Consistent with our stand on control mechanisms, we have not added *conditional* or *iterative structural descriptions* to LOOPS, but use available Interlisp control structures in methods. For these cases, a user de nes a new metaclass for the composite object. (Recall that metaclasses are classes whose instances are classes.) The metaclasses for templates should be subclasses of the distinguished metaclass `Template`. The specialized metaclass should have a `New` method that performs the conditional and iterative steps in the instantiation. This approach works well in conjunction with the LOOPS mechanisms for specializing classes and methods. For example, the specialized `New` method can use `_Super` to access the standard code for the template- directed portion of the instantiation process. gure 10 shows an example of a LOOPS template for a ring oscillator. This composite object is made of a loop of serially connected inverters.

```
(MetaRingOscillator.New
   [LAMBDA (self assocList numStages)    (* mjs: "11-JAN-82 19:28")
               (* * Procedure for creating a ring oscillator.)

   (PROG (ringOscillator firstInverter lastInverter inv1)
                                (* Create the inverter chain.)
      (SETQ inv1 (SETQ firstInverter (_ ($ Inverter) New)))
      [for i to (SUB1 numStages)
        do (SETQ lastInverter (_ ($ Inverter) New))
           (_ inv1 Connect lastInverter)
           (SETQ inv1 lastInverter]
                                (* Close the loop)
      (_ lastInverter Connect firstInverter)
                                (* Make the ringOscillator object.)
      (SETQ ringOscillator (_Super self New assocList))
               (* * the assocList here is the pairing
                    of Template classes found in the
                    instantiation of a template so far)
      (@_ (ringOscillator input) firstInverter)
      (@_ (ringOscillator output) lastInverter)
      (RETURN ringOscillator) ])
```

Figure 10. Example of an itera tively speci ed com posite ob ject, a ring oscillator. The ring oscillator is com posed of a series of in verters serially-connected to form a loop. To specify the itera tion and inter con nec tion of the in verters, a `New` method is de ned for the metaclass `MetaRingOscillator`. The Interlisp func tion for this method (`MetaRingOscillator.New`) uses `_Super` to per form the template- driven part of the in stan tia tion, that is, in stan tiat ing the ring oscillator ob ject itself. In this case, the template- driven por tion of the in stan tia tion is trivial, but the example shows how it can be com bined generally with the pro cedural descrip tion. `MetaRingOscillator.New` uses itera tive state ments to make an instance of `Inverter` for each stage of the oscillator. After con nect ing the com ponents together, it returns the ring oscillator ob ject.

9 LOOPS KNOWLEDGE BASES

Loops was created to support a design environment in which there are community knowledge bases that people share, and to which they can add incremental updates. This section describes our goals for this facility, the concepts that we have employed, and scenarios for using knowledge bases in Loops.

We have chosen the term knowledge base instead of data base to emphasize two things: the kind of information being stored and constraints on the amount of information. Loops will be used mainly for expert system applications where relatively modest amounts of information are used for guiding reasoning. This information (i.e., knowledge) consists of inference rules and heuristics for guiding problem solving. This is in contrast to potentially enormous les of facts, for example, social security records for California. Re ecting this di erence of scale, we have optimized the implementation to support fast access and updating to a smaller amount of information which is expected to t in main memory for any one session. For example, we maintain an index to the object information in computer memory.

9.1 Review of Knowledge Base Concepts

Knowledge Bases: Knowledge bases in LOOPS are les that are built up as a sequence of layers, where each layer contains changes to the information in previous layers. A user can choose to get the most recent version of a knowledge base (that is, all of the layers) or any subset of layers. The second option o ers the exibility of being able to share a community knowledge base without necessarily incorporating the most recent changes. It also provides the capability of referring to or restoring any earlier version. gure 11 illustrates this with an example.

```
------------------------ Layer 1 ------------------------
Obj1 (x 4) ...
Obj2 (y 5) (w 3) ...
------------------------ Layer 2 ------------------------
Obj2 (y 7) (w 2) ...
Obj3 (z 6) ...
------------------------ Layer 3 ------------------------
Obj1 (x 8) ...
Obj4 (z 9) ...
```

Figure 11. Knowledge bases in LOOPS are les that are built-up incrementally as a sequence of layers. Each layer contains updated descriptions of objects. When a knowledge base is opened, the information in the later layers overrides the information in the earlier layers. LOOPS makes it possible to select which layers will be used when a knowledge base is opened. In this example, if the knowledge base is opened and only the rst 2 layers are used, then `Obj1` will have an `x` variable with value `4`. If all three layers were connected, then the value would be `8`.

Community Knowledge Bases: LOOPS partitions the process of updating a community knowledge base into two steps. Any user of a community knowledge base can make tentative changes to a community knowledge base in his own (isolated) environment. These changes can be saved in a layer of his personal knowledge base, and are marked as associated with the community knowledge base. In a separate step, a data base manager can later copy such layers into a community knowledge base. This separation of tasks is intended to encourage experimentation with proposed changes. It separates the responsibility for exploring possibilities from the responsibility of maintaining consistent and standardized knowledge bases for shared use by a community. The same mechanisms can be used by two individuals using personal knowledge bases to work on the same design. They can conveniently exchange and compare layers that update portions of a design.

Unique Identi ers: The ability to determine when di erent layers are referring to the same entity is critical to the ability to share data bases. To support this feature the LOOPS data base assigns unique identi ers (based on the computer's identi cation numbers, the date, and an unbounded count) to objects before they are written to a knowledge base. This facility provides a grounding for more sophisticated notions of equality that might be desired in knowledge representation languages built on LOOPS.

Environments: A user of LOOPS works in a personalized *environment*. An environment provides a lookup table that associates unique identi ers with objects in the connected knowledge bases. In an environment, user indicate dominance relationships between selected knowledge bases. When an object is referenced through its unique identi er, the dominance relationships determine the order in which knowledge bases are examined to resolve the reference. By making personal knowledge bases dominate over community knowledge bases, a user can override portions of community knowledge bases with his own knowledge bases.

Multiple Alternatives: An important use of environments is for providing speedy access to alternative versions (e.g., multiple alternatives in a design). A user can have any number of environments available at the same time. Each environment is fully isolated from the others. Operations that move information between environments are always done explicitly through knowledge bases.

9.2 Environmental Objects and Boot Layers

Knowledge bases, environments, and layers are represented in Loops by special objects called *environmental objects*. All knowledge base and environment operations are performed by sending messages to these objects. Environmental objects are accessible from any environment in Loops.

In this section, we will need to distinguish between environmental objects and the things that they represent. gure 12 summarizes some of the terminology that we will use.

Loops Object	*Represents*	*Description*
Layer	le layer	Portion of a le which contains descriptions of objects.
KB	knowledge base	A le and sequence of le layers. A knowledge is known by the name eld of its le name.
KBState	State of a knowledge base	A sequence of le layers. Used to access a xed explicit set of le layers (e.g., a version of a knowledge base that is older than the most recent version).
Environment	environment	An environment associates names and unique identi ers with objects in working memory.

Figure 12. Summary of terminology for environmental Loops objects and the entities that they represent.

Environments: An Environment provides a name space in working memory. Each Environment associates names and unique identi ers with objects. In general, Environments are designed to be independent. For convenience, Environments are usually named. An Environment is always associated with a particular knowledge base. The speci cations for creating an Environment come from some knowledge base, and changes to the Environment are stored on that knowledge base.

Layers: A le layer is a portion of a le which contains descriptions of objects. An object description consists of a unique identi er and an expression that can be read by Interlisp to create the Loops object. A di erent unique identi er is associated with each expression. In addition, a le layer contains a mapping from names (Interlisp atoms) to unique identi ers. A le layer is represented in Loops by a Layer object. A Layer indicates the le on which it is written, the starting address of the le layer, and the name of the knowledge base with which it is conceptually associated. A Layer also contains various bookkeeping information such as the name of its creator and the date of its creation.

KBs and KBStates: A knowledge base is a set of le layers. Typically, most of the layers of a knowledge base are located on a single le. A knowledge base is known by its le name. By convention, such les have the extension ''KB''. A KB is a Loops object that represents a knowledge base. A KB has a name equal to the name eld of the le name of the knowledge base that it represents. For example, the KB with name `Test` would be associated with a version of the le `Test.KB`.

A KBState is a generalization of a KB. It refers to an explicit set of le layers. KBs and KBStates indicate their Layers using a list on an instance variable named `contents`. An element of this list must be either a Layer or a KBState. When a KBState appears in the list, it is as if the Layers listed in the KBState's contents variable appeared explicitly in the list. This provides a mechanism for indirect fetching of layers from other knowledge bases.

To indicate all of the layers of the most recent version of a knowledge base, the contents of the KBState can be the special value ''`CURRENT`''. When such a KBState appears in the list, it is as if the Layers of the most recent version of the knowledge base were inserted in the list. These Layers are retrieved by retrieving the KB from the referenced knowledge base.

Boot Layers: Environmental objects are distinguished from other objects when they are accessed and when they are written out to a knowledge base. They are accessed di erently in that they are kept in a global name table accessible in all environments. This means that an Environment can be described in terms of the environmental objects before the Environment is made current.

Environmental objects are also special in that the le layer that describes them is a special le layer at the end of a knowledge base called the boot layer. In order to access the contents of a knowledge base, it is necessary to read the boot layer rst because it contains the environmental objects that describe the knowledge base. A boot layer for a knowledge base contains a single KB describing itself, a Layer describing each of its le layers, and the KBStates mentioned (directly or indirectly) in the KB.

The Global Name Table: Loops keeps environmental objects in a global name table that is accessible from any environment. This name table also includes the basic classes that are part of the Loops kernel. If Loops is used without exercising the Environments feature, then all created objects are also placed in the global table.

When another environment is opened, objects not in core are rst looked for by UID or name in the open environment. If no object is found there, then the UID or name is looked up in the Global Environment. Thus, object descriptions in a new environments override those in Global Envrionment, but old objects which have no counterparts are still available.

9.3 Starting With No Preexisting Knowledge Bases

The knowledge base facility in Loops has been designed to cover a number of situations. Because of this generality, it is not always easy for a newcomer to discover the simplest way of using the features. The following sections describe all the features of the Knowledge Base system; however each feature is introduced within a particular scenario that shows how to do some of the most common operations for which Loops was designed.

In the rst scenario, a user wants to start from scratch using no preexisting knowledge bases. The results of this Loops session are saved in a personal knowledge base.

When a user invokes Loops, the Loops name space will contain some objects from the Loops kernel. Before creating any new objects, the user should type an expression of the form:

```
(_ $KB New 'KBName 'environmentName newVersionFlg)
```

where `KBName` is an atom (e.g., use `FOO` to create a knowledge base named `FOO.KB`) and `environmentName` will be the name of the Environment. This will create both a new KB corresponding to the `KBName` and a new Environment with the name `environmentName`.

Loops checks that a knowledge base with `KBName` does not already exist. If it does exist and `newVersionFlg` is `NIL`, Loops will report an error. If `newVersionFlg` is `T`, then Loops will create a new version of the le. Because of the way the le system works, the name of a KB must be all in upper case. If the user attempts to use a `KBName` which contains lowercase letters, Loops will correct the name to all upper case and print a warning message.

Warning: Objects created before creating and opening an Environment are placed in the global name table. Hence, any objects so created will be shared by all Environments. However, Loops will not save such objects in a knowledge base later in the session unless they are explicitly moved to some environment. Alternatively, such objects can be saved using the Interlisp le package.

The next step is to open the Environment:

```
(_ $environmentName Open)
```

This makes the new Environment be the current environment. New objects that are created will be associated with the KB.

Having created an Environment, the user can then proceed to create whatever new objects he desires in the session. To dump the current state of the environment and continue afterwards, the user can type:

```
(_ $environmentName Cleanup)
```

This does not close any les, and leaves the environment as it was, except that all changed objects have been dumped to the knowledge base, and then marked as unchanged. `Cleanup` can be done any number of times in a session.

At the end of a session the user should do a `Close`:

```
(_ $environmentName Close)
```

This writes out all of the objects to a le layer, updates the environmental objects accordingly, and writes them out to a boot layer, deletes these objects from memory, and closes all les associated with the environement. The user can then exit from Interlisp. After a `Close` is done, the user must go through the following scenario to start up again.

9.4 Continuing from a Previous Session

The case where a user wants to create a new knowledge base is less common than the case where he wants to modify or add objects to a knowledge base that he has previously created. In this scenario a user wants to resume from where he was at the end of his previous session.

The rst step is to obtain the user's knowledge base, and link it to an environment. This is done by a message to the class `KB` as follows:

```
(_ $KB Old 'KBName 'environmentName)
```

This reads the boot layer of the knowledge base named `KBName` and creates an Environment named `environmentName` that is then connected to the KB. At this point the user must open the environment to make the contents of the KB available in this environment:

```
(_ $environmentName Open)
```

This causes Loops to read in each Layer contained (possibly implicitly) in the contents of the associated KB (named `KBName`). It also makes the new Environment be the current environment. Having opened an Environment, the user can then proceed to de ne whatever new objects he desires in the session. New objects that are created will be associated with the KB. When he is done, he should type as in the previous scenario:

```
(_ $environmentName Cleanup)
```

or

```
(_ $environmentName Close)
```

9.5 Starting from a Community Knowledge Base

Users will not usually start from scratch. Rather, they will often begin by using previously created community knowledge bases. This scenario starts with obtaining a single community knowledge base. The user does not own the community knowledge base, so the results of the session will have to be saved in a personal knowledge base. The personal knowledge base will contain any new objects that created as well as any objects from the community knowledge base that have changed.

As in the rst or second scenario, the rst step is to create a personal knowledge base.

```
(_ $KB New 'KBName 'environmentName newVersionFlg)
```

or if the user has a personal knowledge base already, by doing a:

```
(_ $KB Old 'KBName 'environmentName)
```

This obtains both the KB and an Environment. The next step is to add the community knowledge base to the KB as follows:

```
(_ $KBName AddToContents 'communityKBName )
```

where `communityKBName` is an atom that is the name of the community knowledge base.

This step should be repeated for each knowledge base to be added to the KB named `KBName`. The message creates a KBState describing the ‘‘current’’ state of the community knowledge base and adds that KBState to the contents of the KB for the personal knowledge base. The e ect of this action is that Loops will remember to associate the community knowledge base with the user’s knowledge base in the future. (This step need not be repeated in any future session which uses the knowledge base `KBName`.)

At this point, the user can open the Environment as before:

```
(_ $environmentName Open)
```

This causes Loops to read in each Layer contained (possibly implicitly) in the contents of the KB named `KBName`. The `Open` message also makes the Environment named `environmentName` be the current environment.

Since the KB associated with the environment contains a KBState for `communityKBName`, those Layers will also be read. They are found by reading the boot layer of the community knowledge base. The message `AddToContents` on `KBName` will work properly even after the environment is `Open`, in the sense that when it is done on a KB connected to an `Open` environment, it causes all the layers of the newly added KB to be read in.

All creation and modi cation operations will take place in this Current Environment. The user can create new objects and modify objects in the community knowledge base. When done, the results of the session can be saved using `Cleanup` (or `Close`). This will cause two le layers to be written out to the personal knowledge base (and none to the community knowledge base). First a le layer is written out to `KBName` for changes made to the community knowledge base (if any). The Layer for this le layer is marked as associated with the community knowledge base. Second, a le layer is written out for the other objects that have been created. The Layer for this is marked as associated with `KBName`. Finally, the environmental objects for the knowledge base are written out to a boot layer.

Before the boot layer is written out, the KB for the personal knowledge base named `KBName` is updated to contain the new Layers. It contains the reference to the community knowledge base that was created by the `AddToContents` message. This continues to be interpreted as a reference to the most recent version of the community knowledge base named `communityKBName`.

If `Close` was used, then the les storing the knowledge bases have been closed and all objects in the environment have been destroyed. The environment was also made not current. This clean state is recommended as a place from which the user can then exit from Interlisp.

9.6 Freezing and Thawing References to Knowledge Bases

In the previous scenarios, the user used the most recent version of the community knowledge base. Community knowledge bases can be changed over time by their owners (i.e., their human knowledge base managers). Sometimes a knowledge base manager may update the community knowledge base, but a user may want to continue using a xed older version. For example, if the new version of a community knowledge base contains extensive changes, the user may want to nish some project before converting his personal knowledge bases to re ect the changes. To do this the user must freeze references to the community knowledge base. Freezing enables a user to continue to access a xed set of layers even though the community knowledge base may be changed by the knowledge base manager. In this scenario, the user has a personal knowledge base whose contents include a named community knowledge base. She anticipates the change to the community knowledge base before it happens and freezes reference to it.

Later, we will see how a user can return to an earlier version after a change has been made.

Freezing: The rst step is to obtain access to the user's personal knowledge base. As in the previous example, this is done by sending an `Old` message to the class `KB`:

```
(_ $KB Old 'KBName 'environmentName)
```

This creates an Environment named `environmentName` with that KB as its outputKB. To freeze the reference, the user needs to change the KBState in his personal KB that describes the community knowledge base. This can be done as follows:

```
(_ $KBName FreezeKB 'communityKBName)
```

The user can then open his Environment, do his work, and then write updates as before:

```
(_ $environmentName Open)
```
... *<make changes to objects>* ...
```
(_ $environmentName Close)
```

From his point of view, the objects in the community knowledge base will be static even if the knowledge base is changed several times. After the user ends this session and starts again the next day, his knowledge base will continue to refer to xed versions of the objects in the community knowledge base, even if new versions are added later.

Thawing: Eventually, however, the changes (and improvements) to the community knowledge base may provide a compelling reason for the user to switch to the most recent version. To do this, he should type the following messages at the beginning of a session:

```
(_ $KB Old 'KBName 'environmentName)
(_ $KBName ThawKB 'communityKBName )
```

The user can then open his Environment, do his work, and then write updates as before.

9.7 Using Several Knowledge Bases in an Environment

The partitioning of knowledge into multiple knowledge bases can be a useful tool for organizing knowledge. For example, long term storage of di erent versions of a design can be kept in separate knowledge bases in Loops. (The di erent knowledge bases in these cases correspond to di erent environments.) It is also convenient to partition knowledge bases to re ect the partitioning of responsibility for setting standards and maintaining consistency. The previous scenarios have shown the use of separate knowledge bases to keep (tentative, idiosyncratic) personal knowledge separate from (open, standardized) community knowledge. This scenario shows how a user can access several knowledge bases through a personal knowledge base.

The rst step is to open the personal knowledge base as follows:

```
(_ $KB Old 'KBName 'environmentName)
```

The next step is to add all of the other knowledge bases that the user wants as follows:

```
(_ $KBName AddToContents 'otherKBName1)
(_ $KBName AddToContents 'otherKBName2)
(_ $KBName AddToContents 'otherKBName3)
...
```

This can be repeated for each knowledge base to be added.

Each `AddToContents` message changes the `contents` variable of the knowledge base named `KBName` so that it now refers indirectly to the other KBName. These references are preserved across sessions so that the next time the user opens his knowledge base with an `Old` message, he will not need to repeat the `AddToContents` messages. These references can be removed as in the previous session.

For most applications, the order in which knowledge bases are added does not matter. However, if an object reference is ambiguous in the sense that the object is contained in more than one of the knowledge bases, then the last knowledge base added will dominate. After the knowledge bases have been added, the user can optionally freeze the references to any of them as described earlier.

The next step is to open an environment:

```
(_ $environmentName Open)
```

As the user creates new objects in his environment, he could want them to be associated with particular knowledge bases that he is using. Usually, he will want them associated with his personal knowledge base (named `KBName` in the example), and this is the default association. However, bugs in a community knowledge base will often be found by a user working on an example in a personal knowledge base. If the user simply changes the buggy objects, they will continue to be associated with the community knowledge base when he saves them at the end of his session. However, if he creates new objects that he wants associated with the community knowledge base, he can rst type:

```
(_ $environmentName AssocKB 'otherKBName1)
```

This message first checks that there is a knowledge base named `otherKBName1` in the environment. It does not cause the changes to be written to the other knowledge bases. Rather, it causes a specially marked layer to be created in the user's personal knowledge base which can be accessed later by the community knowledge base manager.

The user can then create the new objects. When he is done creating these objects, he can then switch the association back to his personal knowledge base by typing:

```
(_ $environmentName AssocKB 'KBName )
```

As before, the user can type

```
(_ $environmentName Close)
```

when he is done with the session.

Occasionally, a user may accidentally associate some objects with the wrong knowledge base. See the next section for a way to change the association of an object after it has been created.

If he later resumes the session, he will have access to all of the knowledge bases that he added.

9.8 Changing the Associations of Objects

The previous scenario depends on anticipating a change in the intended association of an object before creating it. This approach using an `AssocKB` message works fine if the creation of objects can be conveniently organized into periods such that all of the objects created during a period are associated with the same knowledge base. In practice, however, a user may forget to send the message or he may later change his mind about the appropriate association for an object. The message for changing the association of an object is the `AssocKB` message as follows:

```
(_ $objectName AssocKB 'newKBName )
```

9.9 Switching Among Environments

One of the important features of Environments is that they provide a way of having independent versions of designs. A user can have several open Environments and can switch between them by making one of them the ‘‘current’’ Environment. In this scenario, we will first consider two ways that a user can create multiple open Environments. Then we will consider how to switch among them and how to copy objects between them.

Case 1. In this case, a user is just starting a session. He has a personal knowledge base named `KBName1`, and he wants to create two knowledge bases (`KBName2` and `KBName3`) to represent two versions of a design. To do this, the user can type:

```
(_ $KB New 'KBName2 'environmentName2)
```
Create 2nd knowledge base and Environment.
```
(_ $KB New 'KBName3 'environmentName3)
```
Create 3rd knowledge base and Environment.

```
(_ $KBName2 AddToContents 'KBName1)
```
Add KBName1 to the contents of 2nd KB.
```
(_ $KBName3 AddToContents 'KBName1)
```
Add KBName1 to the contents of 3rd KB.
```
(_ $environmentName2 Open)
```
Open the 2nd Environment.
```
(_ $environmentName3 Open)
```
Open the 3rd Environment, leaving it as current.

Case 2. Alternatively, the user may discover part way through a session that he wants to branch out with another Environment. In this scenario, the user is working in Environment1 and decides to create a branch point. Before doing this, the user must rst Close that environment:

```
(_ $environmentName1 Close)
```

The user can then create the Environment2 and Environment3 as in case 1.

Switching. In both cases, the last Environment opened will be the default current one. The user can make any Environment be current by:

```
(_ $environmentName2 MakeCurrent)
```

All Loops operations will then happen in this Environment. To switch to `environmentName3` use:

```
(_ $environmentName3 MakeCurrent)
```

and so on. To test whether any particular environment, `testedEnvironment` is current, one uses:

```
(_ $testedEnvironment IsCurrent)
```

To switch to the GlobalEnvironment, one sends to the current environments:

```
(_ CurrentEnvironment MakeNotCurrent)
```

The Lisp global variable `CurrentEnvironment` is bound to the environment which is current.

When done, the updates should be written out for all of the open Environments. This can be done by sending `Cleanup` or `Close` messages to each of the environment, or can be done by sending the corresponding message to the class Environment which will send the message on to each open environment (kept on a list in the Lisp global variable `openEnvironments`):

```
(_ $Environment Cleanup)
(_ $Environment Close)
```

Copying Objects between Environments. While a user is switching between environments, he may make discover an error in some information that is global to both environments. In this scenario, the user discovers an error in some objects from a community knowledge base while he is working in Environment2. He corrects the objects in Environment2, and wants to copy those corrections into Environment3. He does this using the `CopyObjects` message as follows:

```
(_ $toEnvironment CopyObjects objectsList)
```

where `toEnvironment` is the name of the environment that the objects are copied to, and `objectsList` is a list of objects to be copied.

This message causes the objects to be copied. If the objects already exist in the `toEnvironment`, then the copies overwrite the previous objects.

In our scenario, the user would perform the following steps:

```
(_ $environmentName2 MakeCurrent)
                              Make Environment2 current.
...
                              Collect the objects.
(SETQ objectsList ...)
                              Make a list of the collected objects.
(_ $environmentName3 CopyObjects objectList)
                              Copy the objects to Environment3.
```

9.10 Saving Parts of a Session

Saving part of a session. To selectively update the knowledge base with some of the changes that he made in a session, a user can send a `Cleanup` message to his Environment with KBs speci ed. For example, to save the updates associated only with the knowledge bases named `KBName1` and `KBName2`, he can send the message:

```
(_ $environmentName Cleanup '(KBName1 KBName2))
```

This message writes out le layers to the user's personal knowledge base containing the objects that from the current Environment that are associated with the knowledge base `KBName1` and `KBName2`. The user has omitted the names of associated knowledge bases for which he wants to discard the changes. This message completes by writing out the boot layer.

The `Cleanup` message without KB's speci ed writes a layer for every associated knowledge base that has been changed, followed by a `WriteBoot`. If the user does a `(_ $envName Cleanup T)`, then all the changes will be written out in a single layer associated with the connected knowledge base.

Cancelling an entire session. The previous scenarios assumed that a user wanted to save the changes that he makes in a session. Sometimes, however, a user may prefer to discard the changes that he has made in a session. He can do this and return the environment to an unopened state by typing:

```
(_ $environmentName Cancel)
```

Cancelling this session will not go back past the last time the user did a `Cleanup`. `Cancel` backs up changes made since that time and then does what a `Close` would do, destroying objects in the environment, and closing les.

9.11 Copying Layers from one Knowledge Base to Another

The ability to describe layers using a KBState makes it possible for one knowledge base to indirectly access the le layers of another one. This mechanism works ne when it is used to extend a personal knowledge base to include a community knowledge base. It enables several users to read a community knowledge base at the same time and to write their updates to their personal knowledge bases. However, the indirection mechanism breaks down if some users want to read a knowledge base while another user is writing to it. For example, such a con ict could arise if a community knowledge base used the indirection mechanism to access a le layer in some personal knowledge base. Whenever the owner of the personal knowledge base was updating it, users of the community knowledge base would be blocked by the le system. To avoid such situations, it is necessary to create community knowledge bases that physically contain all of the le layers that they reference.

In this scenario, the user is just starting a session and no knowledge bases have been opened. The user wants to copy information from a knowledge base named `fromKBName` to a knowledge base named `toKBName`. The rst step is to read the boot layers of the two knowledge bases.

```
(_ $KB Old 'fromKBName)
(_ $KB Old 'toKBName)
```

In this scenario, one need not, and in fact should not, have an envrionment open or either of the two KBs connected to an environment. All the work will go on in the Global Environemnt.

The second step is to create a description of the layers to be moved. This description can be either a Layer or a KBState. One way to create this description is to use any of the object editors available in Loops. Another way is to send a `DescribeLayers` message as follows:

```
(_ $fromKBName DescribeLayers DateOrDays associatedKB)
```

`DateOrDays` can be an Interlisp Date or an integer number of days. If it is a date, then only those Layers created on or after the given date will be described. If it is an integer, then only Layers created within that many days will be described. If it is `NIL`, then no date lter will be applied.

`associatedKB` is the name of the knowledge base with which the Layers are associated. (If `NIL`, then the layers associated with any knowledge base will be described.)

For example:

```
(SETQ layerDescription
          (_ $fromKBName DescribeLayers 14 'toKBName))
```

returns a KBState describing the Layers created in the last fourteen days in the knowledge base named `fromKBName` that are associated with the knowledge base named `toKBName`.

Given such a description, the layers can be copied by typing:

```
(_ $toKBName CopyFileLayers layerDescription)
```

9.12 Summarizing and Combining Knowledge Bases

Summarizing a Knowledge Base. As knowledge bases evolve over time, the number of layers and amount of overridden information can consume a large fraction of the le space. Economy- minded knowledge base managers may want to create ‘‘compressed’’ versions of knowledge bases that have all of the information contained in just one layer. In this scenario, the user starts a session by typing:

```
(_ $KB Summarize fromKBName toKBName assocKBNames)
```

where `fromKBName` is the knowledge base to be summarized; `toKBName` is the knowledge base to be created. It must be a di erent name than `fromKBName`; `assocKBNames` must be a list of KBNames or `NIL`. If it is list, then all, and only those objects with associated KB's on the list will be dumped to the le. One must include `fromKBName` on `assocKBNames` if changes and objects associated with it are to be dumped to the le. If `assocKBNames` = `NIL`, all objects on the le will be dumped on a single layer if `toKBName`.

This message causes Loops to read the boot layer of the old knowledge base (`fromKBName`), create a new knowledge base (`toKBName`), create an Environment associated with the new knowledge base, read in all of the objects in `fromKBName`, write them out to a single layer, and then write a boot layer for the new knowledge base.

Combining Knowledge Bases. The `Summarize` message can also be used to combine several existing knowledge bases into a single new knowledge base. In this case, the message is as follows:

```
(_ $KB Summarize fromKBNames toKBName assocKBNames)
```

where `fromKBNames` is a list of the names of the knowledge bases to be summarized; `toKBName` is the name of the new knowledge base to be created; `assocKBNames` is as described above.

This message causes Loops to read the boot layers of the old knowledge bases, creates a new knowledge base (`toKBName`), creates an Environment associated with the new knowledge base, reads in all of the objects, writes them out to a single layer, and then writes a boot layer for the new knowledge base.

The user can create a new knowledge base which contains all of the objects in any open environment. This may include objects from any number of KB's.

```
(_ environment DumpToKB toKBName assocKBNames)
```

will create a new KB named `toKBName`, and dump from the environment all objects with associated KB on the list `assocKBNames` onto `toKBName` (or all objects if `assocKBNames` = `NIL`).

9.13 Subdividing a Knowledge Base

Sometimes a user may want to subdivide a knowledge base so that a subset of the objects are moved away to create a new knowledge base. In our scenario, the user wants to move the objects from a knowledge base in `fromEnvironmentName` to a knowledge base (`toKBName`) included in `toEnvironmentName`. In the rst step of this scenario the user uses the `MapObjectNames` message:

```
(_ $environmentName MapObjectNames (FUNCTION UserFn) AssocKBs NoUIDs)
```

where

`UserFn` is a function that will be applied to every object name. If `NIL`, then a list of object names and UIDs in environment is returned as the value of the message. If it is the atom `T`, then only names which are not UIDs will be returned.

`AssocKBs` is an optional argument. If an atom, it is interpreted as the name of the associated knowledge base for the objects. If a list, will be interpreted as a list of associated knowledge bases for the object. If

`NIL`, only objects associated with the current AssocKB of the Environment will be used.

If `NoUIDs` is `T`, then `UserFn` will only be applied to real names, and not UIDs.

In our scenario, we will assume that `MyFn` will create a list of the objects (`objectList`) that the user wants to move. The user switches to the source environment, nds the objects and moves them:

```
(_ $fromEnvironmentName MakeCurrent)
                          Switch to fromEnvrionment.
(_ $fromEnvironmentName MapObjectNames (FUNCTION MyFn))
                          Make list of objects.
```

The next step is to move the objects as follows:

```
(SETQ newObjectList
      (_ $toEnvironmentName MoveObjects objectList)
```

This causes the objects to be copied to toEnvironment and deleted from fromEnvironment (or whatever Environment they came from). The objects will continue to be associated with whatever AssocKB they were before. In this scenario, however, the user wishes them be associated with the knowledge base named toKBName.

```
(_ $fromEnvironmentName MakeCurrent)
(for object in newObjectList do (_ object AssocKB 'toKBName)
```

The nal step is to write out the changes:

```
(_ $environmentName Cleanup)
```

9.14 Going Back to a Previous Boot Layer of a Knowledge Base

Since knowledge bases are represented as objects, it is possible to recon gure their contents using the standard object access functions. However if a Layer has been deleted from the contents of a KB, that layer is no longer written out to the boot layer. This can make it di cult to get back to versions modi ed in this way. The following message makes it possible restore such knowledge bases by reading in old boot layers:

```
(_ $KB ReadOldBootLayer 'KBName numberBack)
```

The value returned is a KB which has the name KBName, and the state corresponding to the boot layer speci ed. To preserve a KBState which has these contents, the user can then use:

```
(_ $KBName Copy)
```

9.15 A ecting what is Saved

The user may not wish an object, or some part of an object saved on a knowledge base. In this section, we describe a number of ways of stopping information from being written on the knowledge base, with appropriate caveats for the use of these features.

9.15.1 Temporary Objects

If the user is creating lots of objects for temporary use (as intermediate products of a computation) then none of those objects are useful after the computation is done. To create such objects, the user should use:

```
(_ class NewTemp)
```

to create them instead of the usual `(_ class New)` message. Objects created in this way will not be given a UID, and will be not be accessible by mapping through the environment. If by some chance they are referenced from some object that is being dumped to the data base, they will then be converted into permanent objects, and dumped to that same KB.

9.15.2 Not Saving some IV values

For some instances, it is useful to store in an instance variable a Lisp dataytpe (e.g. a pointer to a window, or hash array). However, most Lisp datatypes are not stored appropriately on a KB. In general, when read back in from a KB, what was formerly an instance of a datatype looks like an atom with a funny printname. The solution we have adopted is to allow the user to specify IV values or properties which should not be dumped to a knowledge base. When read back in, the IV value or property will inherit the default value from the class which can be an active value to recreate the desired Lisp object.

For example, the class `$Environment` uses a hash table as the value of its IV nameTable. The following fragment of the de nition of Environment shows how saving the value of `nameTable` is suppressed and how an active value is used to recreate it.

```
[DEFCLASS Environment ...
      (InstanceVariables ...
         (nameTable #(NIL NewNameTable) DontSave Any)
     ...]
```

Any instance of environment will have `nameTable` lled in by `NewNameTable` the rst time it is accessed. `NewNameTable` is a specialized version of `FirstFetch` which makes the local value be a hashArray. The property `DontSave` with value `Any` (which is inherited in every instance) speci es that nothing about the IV `nameTable` should be saved on a KB. For ner control, the property `DontSave` could have been given a value which is a list of property names whose values should not be saved on the KB. If the atom `Value` is included in the list, then the value of the IV itself will not be saved. The value `Any` for `DontSave` is interpreted as meaning no porperty or value should be saved.

9.15.3 Ignoring changes on an IV

Whenever an object is modi ed during the course of a session, it is marked as changed so that a new version of the object will be written out on the KB. Suppose the user may be using an IV globally known object as a place to cache some information. In this case the user does not need or even want the known object to be marked as changed if the only change made was to store the cached information. To allow this, the special active value function `StoreUnmarked` is provided which does not mark the object as changed when it updates its localState. For example, if `$WorldView` had an instance variable `lastSelected` which was updated each time a selection was made, then if `$WorldView` looked like:

```
[DEFINST WorldView ...
   (lastSelected #(obj1 NIL StoreUnmarked) ...]
```

changes to `lastSelected` would be ignored by the KB system. It is often useful to combine this feature with `DontSave` described earlier so that when the object is dumped to a KB (because of some other change) the value in this IV is not saved. Then the `activeValue` can be inherited directly from the default value in the class. Using `DontSave` by itself is not su cient to ensure that the object will not be dumped if a value is changed in the not to be saved IV.

9.15.4 Getting rid of objects explicitly

During the course of a session users may create a number of objects they discover before the end of the session are not needed. They may also decide that some old objects are no longer needed. By using:

```
(_ obj Destroy)
```

for each such object, the user will cause any new objects to be forgotten (not written to the KB) and the incore space reclaimed. For objects which were in the KB previously, there will be stored an indication that this object has been deleted, so that later reading of this KB will not contain the object.

9.16 Examining Environmental Objects

Sending the message `MapObjectNames` to an open environment allows one access to the names and UIDs of objects in that environment. From the names and UIDs one can then access the objects themselves using `GetObjectRec`. One can determine the names and UIDs of objects in a Layer by sending that layer the message `MapObjectNames`. The form is:

```
(_ $Layer1 MapObjectNames mapFn noUIDs)
```

which applies `mapFn` to each name (and to each UID unless `noUIDs= T`). If `mapFn = NIL` then this simply returns a list of the names (and UIDs). However, unless the layer has been read in to an environment, one cannot get the object associated with that name (UID) on that layer.

PrettyPrinting a KB: A special pretty printing function is available for KB's, KBStates, and Layers which tell about its history and contents. If one does:

```
(_ $KB Old 'KBName )
```

without necessarily opening an environment, then one can send:

```
(_ $KBName  PP)
```

to see what is in the KB and its containing layers.

ChangedKBs: In a particular environment, one can change objects which originate on any number of community and personal knowledge bases. To nd out the names of any KBs that have modi ed entities associated with them, one send to that environment, say `E1`:

(_ $E1 ChangedKBs)

It is this list which is used by Cleanup to determine the set of layers that will be dumped at cleanup time.

9.17 The Class KBState

KBState [Class]

IVs:

name [IV of KBState]
: Name of le associated with this KBState. NIL as value here overrides active value in named object.

contents [IV of KBState]
: Either CURRENT, meaning the current state of the KB with name or a list of layers and KBStates specifying layerset)

Methods:

(_ self AddEntities entityList) [Method of KBState]
: Add all items on contents and self to entityList. Called by functions which write out the boot layer to make sure that all layers are added to the list of items to be dumped.

(_ self AddToContents newAddition) [Method of KBState]
: Adds a new item to contents of KB.

(_ self Connect nameTable) [Method of KBState]
: Read in object le indices from all, possibly implicit, layers in order. These are being opened for input only.

(_ self CurrentState) [Method of KBState]
: Create a KB state which re ects the current state of this KB.

(_ self DescribeLayers dateOrDays assocKB) [Method of KBState]
: Return a KBState whose contents are just those layers which occur after dateOrDays and have KB assocKB, or NIL if none.

(_ self Files |leList) [Method of KBState]
: |leList is a TCONC list of les already found. Add any new ones found. Very similar in structure to KBState.Connect.

(_ self MyKB) [Method of KBState]
: Return the KB object corresponding to this KBState.

(_ self ReadBoot) [Method of KBState]
: Read the boot le for this KB.

(_ self SetContents lst) [Method of KBState]
: Make KB have new contents. Check types of elements.

9.18 The Class KB

`KB` [Class]

IVs:

`connectedEnvs` [IV of KB]
List of Envs which have read in contents of this KB.

`contents` [IV of KB]
KBs start out with an empty list of contents.

`currentWriter` [IV of KB]
Environment which is currently writing on this KB.

`fileName` [IV of KB]
Full name of le where this KB is stored. Computed the rst time it is needed. Never stored.

`owners` [IV of KB]
List of owners of this KB.

`status` [IV of KB]
One of `Disconnected`, `Connected`, or `BootNeeded`.

Methods:

`(_ self AddToContents newAddition)` [Method of KB]
Adds a new item to contents of KB.

`(_ self ConnectForOutput nameTable)` [Method of KB]
Read in object le indices from all, possibly implicit, layers in order. This is being opened for output.

`(_ self CopyFileLayer layer)` [Method of KB]
Copies the FileLayer referred to by `layer` onto `self`, and adds a new Layer describing copied leLayer onto contents of `self`

`(_ self CopyFileLayers layerDescription)` [Method of KB]
Copy all the layers in `layerDescription` which should be a KBState into `self`

`(_ self Disconnect)` [Method of KB]
Disconnect this KB and close its le if open.

`(_ self FreezeKB name)` [Method of KB]
Find a KBState with `%@name = name` and `contents = CURRENT`. Replace it by a new KBState with `contents` = currentState of myKB. Return new KBState or `NIL` if failure.

`(_ self PrintContents |le)` [Method of KB]
Fn to Print out a formatted description of the contents of a knowledge base.

`(_ self SetContents lst)` [Method of KB]

Make KB have new contents. Check types of elements.

`(_ self ThawKB name)` [Method of KB]

Find a KBState with `(GetValue self (QUOTE name))` = `name` and `contents` not equal `CURRENT`. Replace it by a new KBState with `contents` = `CURRENT`. Return new KBState or `NIL` if failure.

`(_ self WriteBoot)` [Method of KB]

Write out boot le containing KB and all layers and KBStates it contains implicitly or explicitly.

`(_ self WriteEntityFile changedEntities namedEntities assockbName)` [Method of KB]

Writes the entities (objects) out to a layer in a given kb.

`(_ self WriteFileLayer kbName nameTable)` [Method of KB]

Writes the facts on the le, appending to le. Format of layer is: - indexFilePosition (up to 7 characters) - entityCount (up to 7 characters) - nameCount (up to 7 characters) - entity records - indexRecords (UID followed by le position,) - nameRecords (name followed by UID) - initialFilePosition.

9.19 The Class Environment

`Environment` [Class]

IVs:

`status` [IV of Environment]

One of `NotOpen` or `Open`. `Open` when indexes of KBs have been read in, `NotOpen` after `ClearObjectMemory`.

`nameTable` [IV of Environment]

nameTable for looking up UIDs and names.

`outputKB` [IV of Environment]

KB to which changes will be led, and which speci es contents.

`assocKB` [IV of Environment]

Name of the KB associated with new objects created.

Methods:

`(_ self AssocKB akb)` [Method of Environment]

Make `akb` be the assocKB of this KB.

`(_ self Cancel)` [Method of Environment]

Erase an environment without cleaning up so that environment is empty, as if it were not open, but it is still connected to the same KB. Make it not current.

`(_ self ChangedKBs)` [Method of Environment]

Finds the names of all KBs that have any modi ed entities associated with them.

The Class Environment

(_ self Cleanup KBNames noBootLayerFlg) [Method of Environment]

Write FileLayers for KBs named in KBNames. If KBNames = NIL then write a layer for each changed KB. If KBNames = T then write one layer for all changes. If KBNames is a single atom, then the update is written for that single assocKB. Finish by writing new boot layer for outputKB unless noBootLayerFlg is T.

(_ self ClearObjectMemory) [Method of Environment]

Write out boot layer if needed and clear nameTable.

(_ self Close assocKBs) [Method of Environment]

Cleanup an environment so that all les are closed, and environment is empty, as if it were just created.

(_ self ConnectOutput KB) [Method of Environment]

Make KB be the le onto which changes in this Environment will be written.

(_ self CopyObjects objList) [Method of Environment]

Copies objects on objList using the object structure of the object in Environment self with same UID, if found.

(_ self DumpToKB kbName assocKBNames) [Method of Environment]

???

(_ self Files |leLst) [Method of Environment]

Get a list of all les associated with this environment. Argument to KBState.Files is a TCONC list.

(_ self IsCurrent) [Method of Environment]

Test if current.

(_ self MakeCurrent) [Method of Environment]

Set values of CurrentNameTable and CurrentEnvironment from self and make DefaultKBName be my assocKB.

(_ self MakeNotCurrent bitchIfNotCurrent) [Method of Environment]

Makes no Environment Current if this is current, elses causes Error if not Current and bitchIfNotCurrent = T.

(_ self MapObjectNames mapFn assocKBs noUIDs) [Method of Environment]

APPLY mapFn to the name of each object stored in the environment. If assocKBs given, select only those which are in the list. If noUIDs = T then apply only to names which are not UIDs. If mapFn = NIL then just list all names and UIDs; if mapFn = T then just the names.

(_ self MarkDeleted objToBeDeleted) [Method of Environment]

Mark object as deleted in KB when new layer is written out. Done by smashing localRecord eld of entity, but NOT storedIn eld. See SelectChangedEntity.

(_ self Open) [Method of Environment]

Read in the index of all the layers referred to by contents of outputKB.

(_ self WriteBoot) [Method of Environment]

Make outputKB write it's boot le.

`(_ self WriteUpdate kbName)` [Method of Environment]

Write layer for `kbName`, or all changes if `kbName` = `T`.

9.20 The Class Layer

`Layer` [Class]

IVs:

`file` [IV of Layer]

Name of the file where FileLayer is found. Compute it on firstFetch from the kbName by searching directory path. Don't save full name on file.

`kbName` [IV of Layer]

Name of kb where this layer was stored e.g. BRIDGE.

`position` [IV of Layer]

Index on file where FileLayer is found.

`assocKB` [IV of Layer]

Name of KB with which this Layer is associated conceptually.

Methods:

`(_ self AddEntities entityList)` [Method of Layer]

Add `self` to entity list for dumping on boot layer.

`(_ self Connect nameTable)` [Method of Layer]

Open layer file and read in index.

`(_ self Files fileLst)` [Method of Layer]

Add my file to list if it is not already there.

`(_ self MapObjectNames mapFn noUIDs)` [Method of Layer]

Apply `mapFn` to objectnames in layer, or make a list of them if `mapFn` = `NIL`.

9.21 The Class KBMeta

`KBMeta` [Class]

Methods:

`(_ self New kbName envName newVersionFlg)` [Method of KBMeta]

Create a new KnowledgeBase file, and an environment if `kbName` is given, and make environment current.

`(_ self Old kbName envName)` [Method of KBMeta]

Get KB for this kbName. (Causes boot layer to be read unless KB is already in the global table.) If `envName` is given, creates an Environment of that name and connects the environment to the KB.

`(_ self ReadBoot)` [Method of KBMeta]

Read in index of existing KB given kbName.

`(_ self ReadOldBootLayer kbName numBack)` [Method of KBMeta]

Read in index of already existing KB.

`(_ self Summarize fromKBName toKBName assocKBNames namedObjectsOnly)` [Method of KBMeta]

Incorporate all objects of `fromKBName` with assocKB in `assocKBNames` (or all if `assocKBNames = NIL`) into new KB `toKBName`. If `namedObjectsOnly = T`, then only copies over all those entities referred to by a name or by a named object directly or indirectly. This latter feature provides a mechanism for garbage collection.

9.22 The Class EnvironmentMeta

`EnvironmentMeta` [Class]

Methods:

`(_ self Cleanup)` [Method of EnvironmentMeta]

Write updates for all open environments.

`(_ self Close leaveKBattachedFlg)` [Method of EnvironmentMeta]

Close all the open environments.

`(_ self OpenFiles)` [Method of EnvironmentMeta]

Returns a list of the open files for all open Environments.

10 INTRODUCTION TO RULE-ORIENTED PROGRAMMING IN LOOPS

The core of decision-making expertise in many kinds of problem solving can be expressed succinctly in terms of rules. The following sections describe facilities in Loops for representing rules, and for organizing knowledge-based systems with rule-oriented programming. The Loops rule language provides an experimental framework for developing knowledge-based systems. The rule language and programming environment are integrated with the object-oriented, data-oriented, and procedure-oriented parts of Loops.

Rules in Loops are organized into production systems (called RuleSets) with speci ed control structures for selecting and executing the rules. The work space for RuleSets is an arbitrary Loops object.

Decision knowledge can be factored from control knowledge to enhance the perspicuity of rules. The rule language separates decision knowledge from meta-knowledge such as control information, rule descriptions, debugging instructions, and audit trail descriptions. An audit trail records inferential support in terms of the rules and data that were used. Such trails are important for knowledge-based systems that must be able to account for their results. They are also essential for guiding belief revision in programs that need to reason with incomplete information.

10.1 Introduction

Production rules have been used in expert systems to represent decision-making knowledge for many kinds of problem-solving. Such rules (also called *if-then* rules) specify actions to be taken when certain conditions are satis ed. Several rule languages (e.g., OPS5 [Forgy81], ROSIE [Fain81], AGE [Aiello81]) have been developed in the past few years and used for building expert systems. The following sections describe the concepts and facilities for rule-oriented programming in Loops.

Loops has the following major features for rule-oriented programming:

(1) Rules in Loops are organized into ordered sets of rules (called RuleSets) with speci ed control structures for selecting and executing the rules. Like subroutines, RuleSets are building blocks for organizing programs hierarchically.

(2) The work space for rules in Loops is an arbitrary Loops object. The names of the instance variables provide a name space for variables in the rules.

(3) Rule-oriented programming is integrated with object-oriented, data-oriented, and procedure-oriented programming in Loops.

(4) RuleSets can be invoked in several ways: In the object-oriented paradigm, they can be invoked as methods by sending messages to objects. In the data-oriented paradigm, they can be invoked as a side-e ect of fetching or storing data in active values. They can also be invoked directly from LISP programs. This integration makes it convenient to use the other paradigms to organize the interactions between RuleSets.

(5) RuleSets can also be invoked from rules either as predicates on the LHS of rules, or as actions on the RHS of rules. This provides a way for RuleSets to control the execution of other RuleSets.

(6) Rules can automatically leave an audit trail. An audit trail is a record of inferential support in terms of rules and data that were used. Such trails are important for programs that must be able to account for their results. They can also be used to guide belief revision in programs that must reason with incomplete information.

(7) Decision knowledge can be separated from control knowledge to enhance the perspicuity of rules. The rule language separates decision knowledge from meta-knowledge such as control information, rule descriptions, debugging instructions, and audit trail descriptions.

(8) The invocation of RuleSets can also be organized in terms of tasks, that can be executed, suspended, and restarted. Using task primitives it is convenient to specify many varieties of agenda-based control mechanisms.

(9) The rule language provides a concise syntax for the most common operations.

(10) There is a fast and e cient compiler for translating RuleSets into Interlisp functions.

(11) Loops provides facilities for debugging rule-oriented programs.

(12) The rule language is being extended to support concurrent processing.

The following sections are organized as follows: This section outlines the basic concepts of rule-oriented programming in Loops. It contains many examples that illustrate techniques of rule-oriented programming. The next section describes the rule syntax. The next section discusses the facilities for creating, editing, and debugging RuleSets in Loops.

10.2 Basic Concepts

Rules express the conditional execution of actions. They are important in programming because they can capture the core of decision-making for many kinds of problem-solving. Rule-oriented programming in Loops is intended for applications to expert and knowledge-based systems.

The following sections outline some of the main concepts of rule-oriented programming. Loops provides a special language for rules because of their central role, and because special facilities can be associated with rules that are impractical for procedural programming languages. For example, Loops can save specialized audit trails of rule execution. Audit trails are important in knowledge systems that need to explain their conclusions in terms of the knowledge used in solving a problem. This capability is essential in the development of large knowledge-intensive systems, where a long and sustained e ort is required to create and validate knowledge bases. Audit trails are also important for programs that do non-monotonic reasoning. Such programs must work with incomplete information, and must be able to revise their conclusions in response to new information.

10.3 Organizing a Rule-Oriented Program

In any programming paradigm, it is important to have an organizational scheme for composing large systems from smaller ones. Stated di erently, it is important to have a method for partitioning large programs into nearly-independent and manageably-sized pieces. In the procedure-oriented paradigm, programs are decomposed into procedures. In the object-oriented paradigm, programs are decomposed into objects. In the rule-oriented paradigm, programs are decomposed into *RuleSets*. A Loops program that uses more than one programming paradigm is factored across several of these dimensions.

```
RuleSet Name: CheckWashingMachine;
WorkSpace Class: WashingMachine;
Control Structure: while1 ;
While Condition: ruleApplied;

(* What a consumer should do when a washing machine fails.)

      IF .Operational THEN (STOP T ’Success ’Working);

      IF load>1.0 THEN .ReduceLoad;

      IF ~pluggedInTo THEN .PlugIn;

{1}    IF pluggedInTo:voltage=0 THEN breaker.Reset;

{1}    IF pluggedInTo:voltage<110 THEN $PGE.Call;

{1}    THEN dealer.RequestService;

{1}    THEN manufacturer.Complain;

{1}    THEN $ConsumerBoard.Complain;

{1}    THEN (STOP T ’Failed ’Unfixable);
```

Figure 13. RuleSet of consumer instructions for testing a washing machine. The work space for the RuleSet is a Loops object of the class `WashingMachine`. The control structure `While1` loops through the rules trying an escalating sequence of actions, starting again at the beginning if some rule is applied. Some rules, called one-shot rules, are executed at most once. These rules are indicated by the preceding one in braces.

There are three approaches to organizing the invocation of RuleSets in Loops:

Procedure-oriented Approach. This approach is analogous to the use of subroutines in procedure-oriented programming. Programs are decomposed into RuleSets that call each other and return values when they are nished. *SubRuleSets* can be invoked from multiple places. They are used to simplify the expression in rules of complex predicates, generators, and actions.

Object-oriented Approach. In this approach, RuleSets are installed as methods for objects. They are invoked as methods when messages are sent to the objects. The method RuleSets are viewed analogously to other procedures that implement object message protocols. The value computed by the RuleSet is returned as the value of the message sending operation.

Data-oriented Approach. In this approach, RuleSets are installed as access functions in active values. A RuleSet in an active value is invoked when a program gets or puts a value in the Loops object. As with active values with Lisp functions for the *getFn* or *putFn*, these RuleSet active values can be triggered by any Loops program, whether rule-oriented or not.

These approaches for organizing RuleSets can be combined to control the interactions between bodies of decision-making knowledge expressed in rules.

10.4 Control Structures for Selecting Rules

RuleSets in Loops consist of an ordered list of rules and a control structure. Together with the contents of the rules and the data, a RuleSet control structure determines which rules are executed. Execution is determined by the contents of rules in that the conditions of a rule must be satis ed for it to be executed. Execution is also controlled by data in that di erent values in the data allow di erent rules to be satis ed. Criteria for iteration and rule selection are speci ed by a RuleSet control structure. There are two primitive control structures for RuleSets in Loops which operate as follows:

`Do1` [RuleSet Control Structure]

The rst rule in the RuleSet whose conditions are satis ed is executed. The value of the RuleSet is the value of the rule. If no rule is executed, the RuleSet returns `NIL`.

The `Do1` control structure is useful for specifying a set of mutually exclusive actions, since at most one rule in the RuleSet will be executed for a given invocation. When a RuleSet contains rules for speci c and general situations, the speci c rules should be placed before the general rules.

`DoAll` [RuleSet Control Structure]

Starting at the beginning of the RuleSet, every rule is executed whose conditions are satis ed. The value of the RuleSet is the value of the last rule executed. If no rule is executed, the RuleSet returns `NIL`.

The `DoAll` control structure is useful when a variable number of additive actions are to be carried out, depending on which conditions are satis ed. In a single invocation of the RuleSet, all of the applicable rules are invoked.

 gure 14 illustrates the use of a `Do1` control structure to specify three mutually exclusive actions.

```
RuleSet Name: SimulateWashingMachine;
WorkSpace Class: WashingMachine;
Control Structure: Do1 ;

(* Rules for controlling the wash cycle of a washing machine.)

   IF controlSetting=’RegularFabric
   THEN .Fill .Wash .Pause .SpinAndDrain
        .SprayAndRinse .SpinAndDrain
        .Fill .DeepRinse .Pause .DampDry;
```

```
    IF controlSetting='PermanentPress
    THEN .Fill .Wash .Pause .SpinAndPartialDrain
         .FillCold .SpinAndPartialDrain
         .FillCold .Pause .SpinAndDrain
         .FillCold .DeepRinse .Pause .DampDry;

    IF controlSetting='DelicateFabric
    THEN .Fill .Soak1 .Agitate .Soak4 .Agitate
         .Soak1 .SpinAndDrain .SprayAndRinse
         .SpinAndDrain .Fill .DeepRinse .Pause .DampDry;
```

Figure 14. Rules to simulate the control of the wash cycle of a washing machine. These rules illustrate the use of the `Do1` control structure to select one of three mutually exclusive actions. These rules were abstracted from [Maytag] for the Maytag A510 washing machine.

There are two control structures in Loops that specify iteration in the execution of a RuleSet. These control structures use an explicit while-condition associated with the RuleSet. They are direct extensions of the two primitive control structures above.

`While1` [RuleSet Control Structure]

This is a cyclic version of `Do1`. If the while-condition is satis ed, the rst rule is executed whose conditions are satis ed. This is repeated as long as the while condition is satis ed or until a `Stop` statement or transfer call is executed (see page 93). The value of the RuleSet is the value of the last rule that was executed, or `NIL` if no rule was executed.

`WhileAll` [RuleSet Control Structure]

This is a cyclic version of `DoAll`. If the while-condition is satis ed, every rule is executed whose conditions are satis ed. This is repeated as long as the while condition is satis ed or until a `Stop` statement is executed. The value of the RuleSet is the value of the last rule that was executed, or `NIL` if no rule was executed.

The ‘‘while-condition’’ is speci ed in terms of the variables and constants accessible from the RuleSet. The constant `T` can be used to specify a RuleSet that iterates forever (or until a `Stop` statement or transfer is executed). The special variable `ruleApplied` is used to specify a RuleSet that continues as long as some rule was executed in the last iteration. gure 15 illustrates a simple use of the `WhileAll` control structure to specify a sensing/acting feedback loop for controlling the lling of a washing machine tub with water.

```
RuleSet Name: FillTub;
WorkSpace Class: WashingMachine;
Control Structure: WhileAll ;
Temp Vars: waterLimit;
While Cond: T;

   (* Rules for controlling the filling of a washing machine
      tub with water.)

{1!} IF loadSetting='Small THEN waterLimit_10;
{1!} IF loadSetting='Medium THEN waterLimit_13.5;
{1!} IF loadSetting='Large THEN waterLimit_17;
```

```
{1!} IF loadSetting=’ExtraLarge  THEN waterLimit_20;

   (* Respond to a change of temperature setting at any time.)

     IF temperatureSetting=’Hot
     THEN HotWaterValve.Open  ColdWaterValve.Close;

     IF temperatureSetting=’Warm
     THEN HotWaterValve.Open  ColdWaterValve.Open;

     IF temperatureSetting=’Cold
     THEN ColdWaterValve.Open  HotWaterValve.Close;

   (* Stop when the water reaches its limit.)

     IF waterLevelSensor.Test  >= waterLimit
     THEN HotWaterValve.Close  ColdWaterValve.Close
          (Stop T ’Done ’Filled);
```

Figure 15. Rules to simulate filling the tub in a washing machine with water. These rules illustrate the use of the `WhileAll` control structure to specify an infinite sense-act loop that is terminated by a `Stop` statement. These rules were abstracted from [MayTag].

10.5 One-Shot Rules

One of the design objectives of Loops is to clarify the rules by factoring out control information whenever possible. This objective is met in part by the declaration of a control structure for RuleSets.

Another important case arises in cyclic control structures which some of the rules should be executed only once. This was illustrated in the WashingMachine example in figure 13 where we wanted to prevent the RuleSet from going into an infinite loop of resetting the breaker, when there was a short circuit in the Washing Machine. Such rules are also useful for initializing data for RuleSets as in the example in figure 15.

In the absence of special syntax, it would be possible to encode the information that a rule is to be executed only once as follows:

```
Control Structure: While1
Temporary Vars:  triedRule3;
...
IF ~triedRule3 condition1 condition2 THEN triedRule3_T action1;
```

In this example, the variable `triedRule3` is used to control the rule so that it will be executed at most once in an invocation of a RuleSet. However, the prolific use of rules with such control clauses in large systems has led to the common complaint that control clauses in rule languages defeat the expressiveness and conciseness of the rules. For the case above, Loops provides a shorthand notation as follows:

```
{1}     IF condition1 condition2 THEN action1;
```

The brace notation means exactly the same thing in the example above, but it more concisely and clearly indicates that the rule executes only once. These rules are called ‘‘one shot’’ or ‘‘execute-once’’ rules.

In some cases, it is desired not only that a rule be executed at most once, but that it be tested at most once. This corresponds to the following:

```
Control Structure: While1
Temporary Vars:  triedRule3;
...
IF ~triedRule3 triedRule3_T condition1 condition2 THEN action1;
```

In this case, the rule will not be tried more than once even if some of the conditions fail the rst time that it is tested. The Loops shorthand for these rules (pronounced ‘‘one shot bang’’) is

```
{1!}     IF condition1 condition2 THEN action1;
```

These rules are called ‘‘try-once’’ rules.

The two kinds of one-shot rules are our rst examples of the use of meta-descriptions preceding the rule body in braces. See page 80 for information on using meta-descriptions for describing the creation of audit trails.

10.6 Task-Based Control for RuleSets

* * * Tasks are Not Fully Implemented Yet * * *

Flexible control of reasoning is generally recognized as critical to the success of recent problem-solving programs. Examples of exible control are:

(1) In planning and design tasks, it is important to generate multiple alternatives. These alternatives may be carried to di erent degrees of completion, depending on success, resource limitations, and information gained during a problem-solving process. In some cases, an alternative may be temporarily set aside, only to be revived later in light of new information.

(2) In analysis tasks, it is important to pursue multiple hypotheses in parallel. As evidence and conclusions accumulate, some hypotheses may be abandoned but revived later.

(3) Search and discovery tasks can be organized as opportunistic best- rst searches. At each step only the most promising avenues are pursued. As some avenues fail to work out and new information accumulates, the other avenues can be re-evaluated and sometimes raised in priority.

These examples require the ability (1) to suspend parts of a computation with the possibility of restarting them later, and (2) to reason about the control of computational resources.

Loops provides a set of language features to support these capabilities, based on the representation of the execution of a RuleSet as a *Task*. A Task is a Loops object with much the same structure as an item in an agenda (see gure 16). It represents the RuleSet being invoked, the data on which it is operating, and the status of its execution.

```
RepairTask5:

ruleNumber:   NIL doc (* Number of the next rule to be executed.
                         Used for doNext and cycleNext.)
rs:           #$RepairWashingMachine
                  doc (* RuleSet that was invoked.)
self:         #&(FixitJob "uid1")
                  doc (* work space given to the RuleSet.)
value:        #&(MotorBrushes "uid2")
                  doc (* value returned by the RuleSet)
status:       Suspended
                  doc (* Execution status.  Examples: Started,
                         Done, Aborted, Suspended.)
reason:       TooExpensive
                  doc (* Reason for the status.  Examples: Success,
                         NoSpace, Blocked)
caller:       #$(RuleSet "uid3")
                  doc (* Caller of the RuleSet.)
priority:     300
```

Figure 16. An example of a Task object. This Task could have been created for an invocation of the RuleSet in gure 17. The Task records the RuleSet, its data, and its execution status. The instance variable `ruleNumber` is used only for the control structures `DoNext` and `CycleNext` as described in the next section. The instance variable priority was created in response to the Task Vars declaration in the RuleSet.

gure 17 illustrates a RuleSet for a task that can be suspended. This RuleSet represents part of the behavior of a washing machine repair man. The repair task may be suspended after it has started on a particular `FixitJob` object if the failure is not diagnosed or is too expensive.

```
RuleSet Name: RepairWashingMachine;
WorkSpace Class: FixitJob;
Compiler Options: S ;  (* S for Task Stepping.)
Control Structure: doAll ;
Task Vars: priority;

(* Rules for washing machine repair.)

{1}   priority_300;
...
{1}   IF ~(replacementPart_motor.FindBrokenPart)
   THEN (STOP T ’Suspended ’NoDiagnosis);

   IF replacementPart.Availability=’NotInTruck  hoursLimit < 1
   THEN (STOP badPart ’Suspended ’UnavailablePart);

   IF replacementPart:cost > dollarLimit
   THEN (STOP badPart ’Suspended ’TooExpensive);
...
```

Figure 17. A suspendable Task. This RuleSet characterizes part of the behavior of a repair man of washing machines. The Stop statements specify how the RuleSet may report failure after it has been started on a particular `FixitJob`. Information in task variables (like priority) are saved in the Task record. In this example, the machine failure may not be diagnosed or may be too expensive to x.

gure 18 illustrates a RuleSet for controlling suspendable tasks. This RuleSet represents part of the behavior of the owner of a washing machine repair business. This RuleSet may restart any suspended task by the repairman RuleSet after getting more information about the customer.

```
RuleSet Name: RePlanRepairWork;
WorkSpace Class: JobSchedule;
Control Structure: cycleAll ;
RuleVars: currentTask customer substitutePart;

(* Sample Rules -- part of the behavior of a manager of a
   Washing Machine repair business.)

...
   IF currentTask:status='Success
   THEN (STOP T 'Done 'Success);

   IF currentTask:reason='UnavailablePart
      substitutePart_expert.AskForSubstitutePart
   THEN currentTask:self:replacementPart_substitutePart
         (Start currentTask);

   IF customer:category='VIP
      currentTask:reason='TooExpensive
   THEN currentTask:self:dollarLimit _ VIP:dollarLimit
         currentTask:priority _ 100
         (Start currentTask);
...
```

Figure 18. Control of Tasks. This RuleSet characterizes part of the behavior of the manager of a washing machine repair business. When a repair task fails, the manager RuleSet may change some resource limits and start the repair task going again (e.g., if the customer is a `VIP`).

Loops has facilities for creating Task objects, starting and waiting for tasks, stepping and suspending Tasks. Task variables are used for saving state information. Distinct Tasks can refer to distinct invocations of the same RuleSet in di erent states of execution. The language features supporting Tasks are described later.

10.7 Control Structures for Generators

Since Tasks represent suspended processes with local state, it is natural to use them for describing generators. For the concise speci cation of generators, two additional control structures have been provided in Loops. To use these control structures, a Task is rst created that associates a RuleSet and a work space. The Task is then invoked repeatedly. At each invocation at most one rule is activated and the Task records which rule was activated. At the next invocation, the search for the next rule to apply starts with the rule following the rule that was last executed.

`DoNext` [RuleSet Control Structure]

At each invocation of the Task, the next rule is executed whose conditions are satis ed. The value of the RuleSet is the value of the executed rule, or `NIL` if no rule was executed. After the last rule of the RuleSet has been tried, the Task will always return `NIL`.

This control structure is convenient for specifying a generator of a limited number of items. At each invocation, the remaining rules are tried until the next item is generated. The generator returns `NIL` after all of the rules have been tried.

`WhileNext` [RuleSet Control Structure]

At each invocation of the Task, the generator rst checks whether the while condition of the RuleSet is satis ed. If yes, then the next rule is executed whose conditions are satis ed. The rules can be visualized as forming a circle, so that after the last rule of the RuleSet has been tried, the generator goes back to the beginning. During a single invocation, no rule is tried more than once and the while-condition is tested only once at the beginning of the Step. The value of the RuleSet is the value of the last rule executed or `NIL` if no rule was executed.

This control structure is convenient for specifying a generator that repeats itself periodically, and which has an extra condition that is factored from all of the rules.

If a RuleSet with one of these control structures is invoked directly (instead of through a Task), its behavior is equivalent to that of a `Do1` control structure.

The variable `ruleApplied`, which can be used in the while-condition of `While1` and `WhileAll` control structures, is not meaningful with the `WhileNext` control structure since at most one rule is applied in a given invocation.

10.8 Saving an Audit Trail of Rule Invocation

A basic property of knowledge-based systems is that they use knowledge to infer new facts from older ones. (Here we use the word ‘‘facts’’ as a neutral term, meaning any information derived or given, that is used by a reasoning system.) Over the past few years, it has become evident that reasoning systems need to keep track not only of their conclusions, but also of their reasoning steps. Consequently, the design of such systems has become an active research area in AI. The audit trail facilities of Loops support experimentation with systems that can not only use rules to make inferences, but also keep records of the inferential process itself.

10.8.1 Motivations and Applications

Debugging. In most expert systems, knowledge bases are developed over time and are the major investment. This places a premium on the use of tools and methods for identifying and correcting bugs in knowledge bases. By connecting a system’s conclusions with the knowledge that it uses to derive them, audit trails can provide a substantial debugging aid. Audit trails provide a focused means of identifying potentially errorful knowledge in a problem solving context.

Explanation Facilities. Expert systems are often intended for use by people other than their creators, or by a group of people *pooling* their knowledge. An important consideration in validating expert systems is that reasoning should be *transparent*, that is, that a system should be able to give an account of its reasoning process. Facilities for doing this are sometimes called *explanation systems* and the creation of powerful explanation systems is an active research area in AI and cognitive science. The audit trail mechanism provides an essential computational prerequisite for building such systems.

Belief Revision. Another active research area is the development of systems that can ‘‘change their minds’’. This characteristic is critical for systems that must reason from incomplete or errorful information. Such systems get leverage from their ability to make assumptions, and then to recover from bad assumptions by e ciently reorganizing their beliefs as new information is obtained. Research in this area ranges from work on non-monotonic logics, to a variety of approaches to belief revision. The facilities in the rule language make it convenient to use a user-de ned calculus of belief revision, at whatever level of abstraction is appropriate for an application.

10.8.2 Overview of Audit Trail Implementation

When *audit mode* is speci ed for a RuleSet, the compilation of assignment statements on the right-hand sides of rules is altered so that audit records are created as a side-e ect of the assignment of values to instance variables. Audit records are Loops objects, whose class is speci ed in RuleSet declarations. The audit records are connected with associated instance variables through the value of the `reason` properties of the variables.

Audit descriptions can be associated with a RuleSet as a whole, or with speci c rules. Rule-speci c audit information is speci ed in a property-list format in the meta-description associated with a rule. For example, this can include *certainty factor* information, categories of inference, or categories of support. Rule-speci c information overrides RuleSet information.

During rule execution in audit mode, the audit information is evaluated after the rule’s LHS has been satis ed and before the rule’s RHS is applied. For each rule applied, a single audit record is created and then the audit information from the property list in the rule’s meta-description is put into the corresponding instance variables of the audit record. The audit record is then linked to each of the instance variables that have been set on the RHS of the rule by way of the `reason` property of the instance variable.

Additional computations can be triggered by associating active values with either the audit record class or with the instance variables. For example, active values can be speci ed in the audit record classes in order to de ne a uniform set of side-e ects for rules of the same category. In the following example, such an active value is used to carry out a ‘‘certainty factor’’ calculation.

10.8.3 An Example of Using Audit Trails

The following example illustrates one way to use the audit trail facilities. gure 19 illustrates a RuleSet which is intended to capture the decisions for evaluating the potential purchase of a washing machine. As with any purchasing situation, this one includes the di culty of incomplete information about the product. The meta-descriptions for the rules categorize them in terms of the *basis of belief* (fact or estimate) and a *certainty factor* that is supposed to measure the ‘‘implication power’’ of the rule. (Realistic belief revision systems are usually more sophisticated than this example.)

```
RuleSet Name: EvaluateWashingMachine;
WorkSpace Class: EvaluationReport;
Control Structure: doAll ;
Audit Class: CFAuditRecord ;
Compiler Options: A;

(* Rules for evaluating a potential washing machine for a purchase.)


   ...

   {(basis_'Fact cf_1)}
   IF buyer:familySize>2  machine:capacity<20
   THEN suitability_'Poor;

   {(basis_'Fact cf_.8)}
   reliability_(_ $ConsumerReports GetFacts machine);

   {(basis_'Estimate cf_.4)}
   IF ~reliability THEN reliability_.5;
   ...
```

Figure 19. RuleSet for evaluating a washing machine for purchase. Like many kinds of problems, a purchase problem requires making decisions in the absence of complete information. For example, in this RuleSet the reliability of the washing machine is estimated to be .5 in the absence of speci c information from `ConsumerReports`. The meta-description in braces in front of each rule characterizes the rule in terms of a `cf` (certainty factor) and a `basis` (basis of belief). Within the braces, the variable on the left of the assignment statement is always interpreted as meaning a variable in the audit record, and the variables on the right are always interpreted as variables accessible within the RuleSet. This makes it straightforward to experiment with user-de ned audit trails and experimental methods of belief revision.

The result of running the RuleSet is an evaluation report for each candidate machine. Since the RuleSet was run in audit mode, each entry in the evaluation report is tagged with a reason that points to an audit record. gure 20 illustrates the evaluation report for one machine and one of its audit records.

```
EvaluationReport "uid1"
expense:        510
suitability:     Poor  cc 1 reason ...
reliability:        .5 cc .6 reason "uid2"
...


AuditRec "uid2"
rule:           "uid3"
basis:           Estimate;
cf:             #(.4 NIL PutCumulativeCertainty)
   ...
```

Figure 20. Example of an audit trail. The object for the expense report was prepared by the

RuleSet in gure 19. In this example, each of the entries in the report has a `reason` and a `cc` (for cumulative certainty) property in addition to the value. The value of the `reason` properties are *audit records* created as a side e ect of running the RuleSet. The auditing process records the meta-description information of each rule in its audit record. This information can be used later for generating explanations or as a basis for belief revision. The auditing process can have side e ects. For example, the active value in the `cf` variable of the audit record performs a computation to maintain a calculated cumulative certainty in the `reliability` variable of the evaluation report.

The result of running the RuleSet is an evaluation report for each candidate machine. The meta-descriptions for `basis` and `cf` are saved directly in the audit record. The *certainty factor* calculation in this combines information from the audit description with other information already associated with the object. To do this, the `cf` description triggers an active value inherited by the audit record from its class. This active value computes a *cumulative certainty* in the evaluation report. (Other variations on this idea would include certainty information descriptive of the premises of the rule.)

10.9 Comparison with other Rule Languages

This section considers the rationale behind the design of the Loops rule language, focusing on ways that it diverges from other rule languages. In general, this divergence was driven by the following observation:

When a rule is heavy with control information, it obscures the domain knowledge that the rule is intended to convey.

Rules are harder to create, understand, and modify when they contain too much control information. This observation led us to nd ways to factor control information out of the rules.

10.9.1 The Rationale for Factoring Meta-Level Syntax

One of the most striking features of the syntax of the Loops rule language is the factored syntax for meta-descriptions, which provides information about the rules themselves. Traditional rule languages only factor rules into conditions on the left hand side (LHS) and actions on the right hand side (RHS), without general provisions for meta-descriptions.

Decision knowledge expressed in rules is most perspicuous when it is not mixed with other kinds knowledge, such as control knowledge. For example, the following rule:

```
IF ~triedRule4 pluggedInTo:voltage=0
THEN triedRule4_T breaker.Reset;
```

is more obscure than the corresponding one-shot rule from gure 13:

```
{1}   IF pluggedInTo:voltage=0  THEN breaker.Reset;
```

which factors the control information (that the rule is to be applied at most once) from the domain knowledge (about voltages and breakers). In the Loops rule language, a meta-description (MD) is speci ed in braces in front of the LHS of a rule. For another example, the following rule from gure 19:

```
{(basis_'Fact cf_.8)}
```

```
IF buyer:familySize>2   machine:capacity<20
THEN suitability_'Poor;
```

uses an MD to indicate that the rule has a particular `cf` (‘‘certainty factor’’) and `basis` category for belief support. The MD in this example factors the description of the inference category of the rule from the action knowledge in the rule.

In a large knowledge-based system, a substantial amount of control information must be speci ed in order to preclude combinatorial explosions. Since earlier rule languages fail to provide a means for factoring meta-information, they must either mix it with the domain knowledge or express it outside the rule language. In the rst option, perspecuity is degraded. In the second option, the transparency of the system is degraded because the knowledge is hidden.

10.9.2 The Rationale for RuleSet Hierarchy

Some advocates of production systems have praised the atness of traditional production systems, and have resisted the imposition of any organization to the rules. The at organization is sometimes touted as making it *easy to add rules*. The argument is that other organizations diminish the power of pattern-directed invocation and make it more complicated to add a rule.

In designing Loops, we have tended to discount these arguments. We observe that there is no inherent property of production systems that can make rules additive. Rather, *additivity* is a consequence of the independence of particular sets of rules. Such independence is seldom achieved in large *sets* of rules. When rules are dependent, rule invocation needs to be carefully ordered.

Advocates of a at organization tend to organize large programs as a single very large production system. In practice, most builders of production systems have found it essential to create groups of rules.

Grouping of rules in at systems can be achieved in part by using *context* clauses in the rules. Context clauses are clauses inserted into the rules which are used to alter the ow of control by naming the context explicitly. Rules in the same ‘‘context’’ all contain an extra clause in their conditions that compares the context of the rules with a current context. Other rules redirect control by switching the current context. Unfortunately, this approach does not conveniently lend itself to the reuse of groups of rules by di erent parts of a program. Although context clauses admit the creation of ‘‘subroutine contexts’’, they require a user to explicitly program a stack of return locations in cases where contexts are invoked from more than one place. The decision to use an implicit calling-stack for RuleSet invocation in Loops is another example of the our desire to simplify the rules by factoring out control information.

10.9.3 The Rationale for RuleSet Control Structures

Production languages are sometimes described as having a *recognize-act cycle*, which speci es how rules are selected for execution. An important part of this cycle is the *con ict resolution strategy*, which speci es how to choose a production rule when several rules have conditions that are satis ed. For example, the `OPS5` production language [Forgy81] has a con ict resolution strategy (`MEA`) which prevents rules from being invoked more than once, prioritizes rules according to the recency of a change to the data, and gives preference to production rules with the most speci c conditions.

In designing the rule language for Loops, we have favored the use of a small number of specialized control structures to the use of a single complex con ict resolution strategy. In so doing, we have drawn on some control structures in common use in familiar programming languages. For example, `Do1` is like Lisp's `COND`, `DoAll` is like Lisp's `PROG`, `WhileAll` is similar to `WHILE` statements in many programming languages.

The specialized control structures are intended for concisely representing programs with di erent control relationships among the rules. For example, the `DoAll` control structure is useful for rules whose e ects are intended to be additive and the `Do1` control structure is appropriate for specifying mutually exclusive actions. Without some kind of iterative control structure that allows rules to be executed more than once, it would be impossible to write a simulation program such as the washing machine simulation in gure 15.

We have resisted a reductionist argument for having only one control structure for all programming. For example, it could be argued that the control structure `Do1` is not strictly necessary because any RuleSet that uses `Do1` could be rewritten using `DoAll`. For example, the rules

```
Control Structure: Do1;

IF a1 b1 c1 THEN d1 e1;
IF a2 b2 c2 THEN d2 e2;
IF a3 b3 c3 THEN d3 e3;
```

could be written alternatively as

```
Control Structure: DoAll;
Task Vars: firedSomeRule;

IF a1 b1 c1 THEN firedSomeRule_T d1 e1;
IF ~firedSomeRule a2 b2 c2 THEN firedSomeRule_T d2 e2;
IF ~firedSomeRule a3 b3 c3 THEN firedSomeRule_T d3 e3;
```

However, the `Do1` control structure admits a much more concise expression of mutually exclusive actions. In the example above, the `Do1` control structure makes it possible to abbreviate the rule conditions to re ect the assumption that earlier rules in the RuleSet were not satis ed.

For some particular sets of rules the conditions are naturally mutually exclusive. Even for these rules `Do1` can yield additional conciseness. For example, the rules:

```
Control Structure: Do1;

IF  a1  b1 c1 THEN d1 e1;
IF ~a1  b1 c1 THEN d2 e2;
IF ~a1 ~b1 c1 THEN d3 e3;
```

can be written as

```
Control Structure: Do1;

IF a1 b1 c1 THEN d1 e1;
IF    b1 c1 THEN d2 e2;
IF       c1 THEN d3 e3;
```

Similarly it could be argued that the `Do1` and `DoAll` control structures are not strictly necessary because such RuleSets can always be written in terms of `While1` and `WhileAll`. Following this reductionism to its end, we can observe that every RuleSet could be re-written in terms of `WhileAll`.

10.9.4 The Rationale for an Integrated Programming Environment

RuleSets in Loops are integrated with procedure- oriented, object- oriented, and data- oriented programming paradigms. In contrast to single- paradigm rule systems, this integration has two major bene ts. It facilitates the construction of programs which don't entirely t the rule- oriented paradigm. Rule- oriented programming can be used selectively for representing just the appropriate decision- making knowledge in a large program. Integration also makes it convenient to use the other paradigms to help organize the interactions between RuleSets.

Using the object- oriented paradigm, RuleSets can be invoked as methods for Loops objects. gure 21 illustrates the installation of the RuleSet `SimulateWashingMachineRules` to carry out the `Simulate` method for instances of the class `WashingMachine`. The use of object- oriented paradigm is facilitated by special RuleSet syntax for sending messages to objects, and for manipulating the data in Loops objects. In addition, RuleSets, work spaces, and tasks are implemented as Loops objects.

```
[DEFCLASS WashingMachine
   (MetaClass Class Edited (* "mjs: 25-Nov-82 16:42")
            doc (* Home appliance for washing clothes.))
   (Supers ElectricalDevice PlumbedDevice CleaningDevice)
   (ClassVariables)
   (InstanceVariables
      (controlSetting Medium
            doc (* One of Small, Medium, Large, ExtraLarge)) ...)
   (Methods
      (Fill WashingMachine.Fill doc (* Fill the tub with water.))
      (Wash WashingMachine.Wash doc (* Perform the wash cycle.))
      (Simulate UseRuleSet RuleSet SimulateWashingMachineRules)
     ...]
```

Figure 21. Example of using a RuleSet as a method for object- oriented invocation. This de nition of the class `WashingMachine` speci es that Lisp functions are to be invoked for `Fill` and `Wash` messages. For example, the Lisp function `WashingMachine.Fill` is to be applied when a `Fill` message is received. When a `Simulate` message is received, the RuleSet `SimulateWashingMachineRules` is to be invoked with the washing machine as its work space. `Simulate` messages to invoke the RuleSet may be sent by any Loops program, including other RuleSets.

Using the data- oriented paradigm, RuleSets can be installed in active values so that they are triggered by side- e ect when Loops programs get or put data in objects. For example:

```
(DEFINST WashingMachine (StefiksMaytagWasher "uid2")
   (controlSetting RegularFabric)
   (loadSetting #(Medium NIL RSPut) RSPutFn CheckOverLoadRules)
   (waterLevelSensor "uid3")
]
```

The above code illustrates a RuleSet named `CheckOverLoadRules` which is triggered whenever a program changes the `loadSetting` variable in the `WashingMachine` instance in the gure. This data-oriented triggering can be caused by any Loops program when it changes the variable, whether or not that program is written in the rules language.

11 THE RULE LANGUAGE

11.1 Rule Forms

A rule in Loops describes actions to be taken when speci ed conditions are satis ed. A rule has three major parts called the *left hand side* (LHS) for describing the conditions, the *right hand side* (RHS) for describing the actions, and the *meta-description* (MD) for describing the rule itself. In the simplest case without a meta-description, there are two equivalent syntactic forms:

```
LHS -> RHS ;
```

```
IF LHS THEN RHS ;
```

The `If` and `Then` tokens are recognized in several combinations of upper and lower case letters. The syntax for LHSs and RHSs is given below. In addition, a rule can have no conditions (meaning always perform the actions) as follows:

```
-> RHS ;
```

```
if T then RHS ;
```

Rules can be preceded by a meta-description in braces as in:

```
{MD } LHS -> RHS ;
```

```
{MD } If LHS Then RHS ;
```

```
{MD } RHS ;
```

Examples of meta-information include rule-speci c control information, rule descriptions, audit instructions, and debugging instructions. For example, the syntax for one-shot rules shown on page 68:

{1} IF condition$_1$ condition$_2$ THEN action$_1$;

is an example of a meta-description. Another example is the use of meta-assignment statements for describing audit trails and rules. These statements are discussed on page 89.

LHS Syntax: The clauses on the LHS of a rule are evaluated in order from left to right to determine whether the LHS is satis ed. If they are all satis ed, then the rule is satis ed. For example:

```
A B C+D (Prime D) -> RHS ;
```

In this rule, there are four clauses on the LHS. If the values of some of the clauses are `NIL` during evaluation, the remaining clauses are not evaluated. For example, if `A` is non-`NIL` but `B` is `NIL`, then the LHS is not satis ed and `C+D` will not be evaluated.

RHS Syntax: The RHS of a rule consists of actions to be performed if the LHS of the rule is satis ed. These actions are evaluated in order from left to right. Actions can be the invocation of RuleSets, the sending of Loops messages, Interlisp function calls, variables, or special termination actions.

RuleSets always return a value. The value returned by a RuleSet is the value of the last rule that was executed. Rules can have multiple actions on the right hand side. Unless there is a `Stop` statement or transfer call as described later, the value of a rule is the value of the last action. When a rule has no actions on its RHS, it returns `NIL` as its value.

Comments: Comments can be inserted between rules in the RuleSet. They are enclosed in parentheses with an asterisk for the rst character as follows:

```
(* This is a comment)
```

11.2 Kinds of Variables

Loops distinguishes the following kinds of variables:

RuleSet arguments: All RuleSets have the variable `self` as their workspace. References to `self` can often be elided in the RuleSet syntax. For example, the expression `self.Print` means to send a `Print` message to `self`. This expression can be shortened to `.Print` . Other arguments can be de ned for RuleSets. These are declared in an `Args:` declaration.

Instance variables: All RuleSets use a Loops object for their workSpace. In the LHS and RHS of a rule, the rst interpretation tried for an undeclared literal is as an instance variable in the work space. Instance variables can be indicated unambiguously by preceding them with a colon, (e.g., `:varName` or `obj:varName`).

Class variables: Literals can be used to refer to class variables of Loops objects. These variables must be preceded by a double colon in the rule language, (e.g., `::classVarName` or `obj::classVarName`).

Temporary variables: Literals can also be used to refer to temporary variables allocated for a speci c invocation of a RuleSet. These variables are initialized to `NIL` when a RuleSet is invoked. Temporary variables are declared in the `Temporary Vars` declaration in a RuleSet.

Task variables: [not implemented yet.] Task variables are used for saving information state information related to particular invocations of RuleSets. Unlike temporary variables which are reset to `NIL` at the beginning of RuleSet execution, Task variables are associated with Task objects and keep their values inde nitely. Task variables are used to hold information about a computational process, such as indices for generator Tasks. Task variables are declared indirectly they are the instance variables of the class declared as the *Task Class* of the RuleSet.

Audit record variables: Literals can also be used to refer to instance variables of audit records created by rules. These literals are used only in *meta-assignment* statements in the MD part of a rule. They are used to describe the information saved in audit records, which can be created as a side-e ect of rule execution. These variables are ignored if a RuleSet is not compiled in *audit* mode. Undeclared variables appearing on the left side of assignment statements in the MD part of a rule are treated as audit record variables by default. These variables are declared indirectly they are the instance variables of the class declared as the *Audit Class* of the RuleSet.

Rule variables: [Not implemented yet.] Literals can also be used to hold descriptions of the rules themselves. These variables are used only in *meta-assignment* statements in the MD part of a rule. They describe information to be saved in the rule objects, which are created as a side-e ect of RuleSet compilation. Rule variables are declared indirectly they are the instance variables in the *Rule Class* declaration.

Interlisp variables: Literals can also be used to refer to Interlisp variables during the invocation of a

Kinds of Variables

RuleSet. These variables can be global to the Interlisp environment, or are bound in some calling function. Interlisp variables can be used when procedure-oriented and rule-oriented programs are intermixed. Interlisp variables must be preceded by a backSlash in the syntax of the rule language (e.g., `\lispVarName`).

Reserved Words: The following literals are treated as *read-only* variables with special interpretations:

`self` [Variable]
: The current work space.

`rs` [Variable]
: The current RuleSet.

`task` [Variable]
: The Task representing the current invocation of this RuleSet.

`caller` [Variable]
: The RuleSet that invoked the current RuleSet, or `NIL` if invoked otherwise.

`ruleApplied` [Variable]
: Set to `T` if some rule was applied in this cycle. (For use only in while-conditions).

The following reserved words are intended mainly for use in creating audit trails:

`ruleObject` [Variable]
: Variable bound to the object representing the rule itself.

`ruleNumber` [Variable]
: Variable bound to the sequence number of the rule in a RuleSet.

`ruleLabel` [Variable]
: Variable bound to the label of a rule or `NIL`.

`reasons` [Variable]
: Variable bound a list of audit records supporting the instance variables mentioned on the LHS of the rule. (Computed at run time.)

`auditObject` [Variable]
: Variable bound to the object to which the reason record will be attached. (Computed at run time.)

`auditVarName` [Variable]
: Variable bound to the name of the variable on which the reason will be attached as a property.

Other Literals: As described later, literals can also refer to Interlisp functions, Loops objects, and message selectors. They can also be used in strings and quoted constants.

The determination of the meaning of a literal is done at compile time using the declarations and syntax of RuleSets. The characters used in literals are limited to alphabetic characters and numbers. The rst character of a literal must be alphabetic.

The syntax of literals also includes a compact notation for sending unary messages and for accessing instance variables of Loops objects. This notation uses *compound literals*. A compound literal is a literal composed of multiple parts separated by a periods, colons, and commas.

11.3 Rule Forms

Quoted Constants: The quote sign is used to indicate constant literals:

```
a b=3 c=’open d=f  e=’(This is a quoted expression) -> ...
```

In this example, the LHS is satis ed if `a` is non-`NIL`, and the value of `b` is 3, and the value of `c` is exactly the atom `open`, the value of `d` is the same as the value of `f`, and the value of `e` is the list `(This is a quoted expression)`.

Strings: The double quote sign is used to indicate string constants:

```
IF a b=3 c=’open d=f  e=="This is a string"
THEN (WRITE "Begin configuration task") ... ;
```

In this example, the LHS is satis ed if `a` is non-`NIL`, and the value of `b` is 3, and the value of `c` is exactly the atom `open`, the value of `d` is the same as the value of `f`, and the value of `e` equal to the string `"This is a string"`.

Interlisp Constants: The literals `T` and `NIL` are interpreted as the Interlisp constants of the same name.

```
a (Foo x NIL b) -> x_T ...;
```

In this example, the function `Foo` is called with the arguments `x`, `NIL`, and `b`. Then the variable `x` is set to `T`.

11.4 In x Operators and Brackets

To enhance the readability of rules, a few in x operators are provided. The following are in x binary operators in the rule syntax:

`+` [Rule In x Operator]

Addition.

`++` [Rule In x Operator]

Addition modulo 4.

`-` [Rule In x Operator]

Subtraction.

`--` [Rule In x Operator]

Subtraction modulo 4.

`*` [Rule In x Operator]

Multiplication.

In x Operators and Brackets

`/` [Rule In x Operator]
: Division.

`>` [Rule In x Operator]
: Greater than.

`<` [Rule In x Operator]
: Less than.

`>=` [Rule In x Operator]
: Greater than or equal.

`<=` [Rule In x Operator]
: Less than or equal.

`=` [Rule In x Operator]
: `EQ` simple form of equals. Works for atoms, objects, and small integers.

`~=` [Rule In x Operator]
: `NEQ`. (Not `EQ`.)

`==` [Rule In x Operator]
: `EQUAL` long form of equals.

`<<` [Rule In x Operator]
: Member of a list. (`FMEMB`)

In addition, the rule syntax provides two unary operators as follows:

`-` [Rule Unary Operator]
: Minus.

`~` [Rule Unary Operator]
: Not.

The precedence of operators in rule syntax follows the usual convention of programming languages. For example

```
1+5*3 = 16
```

and

```
[3 < 2 + 4] = T
```

Brackets can be used to control the order of evaluation:

```
[1+5]*3 = 18
```

Ambiguity of the minus sign: Whenever there is an ambiguity about the interpretation of a minus sign as a unary or binary operator, the rule syntax interprets it as a binary minus. For example

```
a-b c d  -e  [-f]  (g -h)  (_ $Foo Move -j) -> ...
```

In this example, the rst and second minus signs are both treated as binary subtraction statements. That is, the rst three clauses are (1) a-b, (2) c and (3) d-e. Because the rule syntax allows arbitary spacing between symbols and there is no syntax to separate clauses on the LHS of a rule, the interpretation of ''d -e'' is as a single clause (with the subtraction) instead of two clauses. To force the interpretation as a unary minus operator, one must use brackets as illustrated in the next clause. In this clause, the minus sign in the clause [-f] is treated as a unary minus because of the brackets. The minus sign in the function call (g -h) is treated as unary because there is no preceding argument. Similarly, the -j in the message expression is treated as unary because there is no preceding argument.

11.5 Interlisp Functions and Message Sending

Calls to Interlisp functions are parenthesized with the function name as the rst literal after the left parenthesis. Each expression after the function name is treated as an argument to the function. For example:

```
a (Prime b) [a -b] -> c (Display b c+4 (Cursor x y) 2) ;
```

In this example, Prime, Display, and Cursor are interpreted as the names of Interlisp functions. Since the expression [a -b] is surrounded by brackets instead of parentheses, it is recognized as meaning a minus b as opposed to a call to the function a with the argument minus b. In the example above, the call to the Interlisp function Display has four arguments: b, c+4, the value of the function call (Cursor x y), and 2.

The use of Interlisp functions is usually outside the spirit of the rule language. However, it enables the use of Boolean expressions on the LHS beyond simple conjunctions. For example:

```
a (OR (NOT b) x y) z -> ... ;
```

Loops Objects and Message Sending: Loops classes and other named objects can be referenced by using the dollar notation. The sending of Loops messages is indicated by using a left arrow. For example:

```
IF cell_(_ $LowCell Occupied? 'Heavy)
THEN (_ cell Move 3 'North);
```

In the LHS, an Occupied? message is sent to the object named LowCell. In the message expression on the RHS, there is no dollar sign preceding cell. Hence, the message is sent to the object that is the value of the variable cell.

For unary messages (i.e., messages with only the selector speci ed and the implicit argument self), a more compact notation is available as described selow.

Unary Message Sending: When a period is used as the separator in a compound literal, it indicates that a unary message is to be sent to an object. (We will alternatively refer to a period as a *dot*.) For example:

```
tile.Type='BlueGreenCross  command.Type='Slide4  -> ... ;
```

In this example, the object to receive the unary message Type is referenced indirectly through the tile instance variable in the work space. The left literal is the variable tile and its value must be a Loops object at execution time. The right literal must be a method selector for that object.

The dot notation can be combined with the dollar notation to send unary messages to named Loops objects. For example,

```
$Tile.Type='BlueGreenCross  ...
```

In this example, a unary `Type` message is sent to the Loops object whose name is `Tile`.

The dot notation can also be used to send a message to the work space of the RuleSet, that is, `self`. For example, the rule

```
IF scale>7 THEN .DisplayLarge;
```

would cause a `DisplayLarge` message to be sent to `self`. This is an abbreviation for

```
IF scale>7 THEN self.DisplayLarge;
```

11.6 Variables and Properties

When a single colon is used in a literal, it indicates access to an instance variable of an object. For example:

```
tile:type='BlueGreenCross  command:type=Slide4 -> ... ;
```

In this example, access to the Loops object is indirect in that it is referenced through an instance variable of the work space. The left literal is the variable `tile`, and its value must be a Loops object when the rule is executed. The right literal `type` must be the name of an instance variable of that object. The compound literal `tile:type` refers to the value of the `type` instance variable of the object in the instance variable `tile`.

The colon notation can be combined with the dollar notation to access a variable in a named Loops object. For example,

```
$TopTile:type='BlueGreenCross  ...
```

refers to the `type` variable of the object whose Loops name is `TopTile`.

A double colon notation is provided for accessing class variables. For example

```
truck::MaxGas<45 ::ValueAdded>600 -> ... ;
```

In this example, `MaxGas` is a class variable of the object bound to `truck`. `ValueAdded` is a class variable of `self`.

A colon-comma notation is provided for accessing property values of class and instance variables. For example

```
wire:,capacitance>5   wire:voltage:,support='simulation  -> ...
```

In the rst clause, `wire` is an instance variable of the work space and `capacitance` is a property of that variable. The interpretation of the second clause is left to right as usual: (1) the object that is the value of the variable `wire` is retrieved, and (2) the `support` property of the `voltage` variable of that object is retrieved. For properties of class variables

```
::Wire:,capacitance>5   node::Voltage:,support='simulation  -> ...
```

In the rst clause, `wire` is a class variable of the work space and `capacitance` is a property of that variable. In the second clause, `node` is an instance variable bound to some object. `Voltage` is a class variable of that object, and `Support` is a property of that class variable.

The property notation is illegal for ruleVars and lispVars since those variables cannot have properties.

11.7 Perspectives

* * * Not implemented yet in the rule language * * *

In many cases it is useful to organize information in terms of multiple points of view. For example, information about a man might be organized in terms of his role as a *father*, as an *employee*, and as a *traveler*. Each point of view, called a *perspective*, contains information for a di erent purpose. The perspectives are related to each other in the sense that they collectively provide information about the same object. As described in the Loops manual, Loops supports this organizational metaphor by providing special mixin classes called `perspectives` and `nodes`.

Loops perspectives can be accessed in the rule language by using a comma notation. In the following rule, the variable `washingMachine` is bound to an object with three perspectives: `commodity`, `electrical`, and `cleaning`. The rule accesses the `voltage` variable of the object that is the `electrical` perspective.

```
IF washingMachine,electrical:voltage<100  THEN ....
```

In this syntax, the term before the comma names a variable, and the term after the comma is the name of the perspective.

11.8 Computing Selectors and Variable Names

The short notations for instance variables, properties, perspectives, and unary messages all show the selector, variable, and perspective names *as they actually appear* in the object.

```
object.selector
object:ivName
object::cvName
object:varname:,propName
object,perspName
```

```
(_ object selector arg1 arg2)
```

For example,

```
apple:flavor
```

refers to the `flavor` instance variable of the object bound to the variable `apple`. In Interlisp terminology, this implies implicit quoting of the name of the instance variable (`flavor`).

In some applications it is desired to be able to compute the names, For this, the Loops rule language provides analogous notations with an added exclamation sign. After the exclamation sign, the interpretation of the variable being evaluated starts over again. For example

```
apple:!\x
```

refers to the same thing as `apple:flavor` if the Interlisp variable `x` is bound to `flavor`. The fact that `x` is a Lisp variable is indicated by the backSlash. If `x` is an instance variable of `self` or a temporary variable, we could use the notation:

```
apple:!x
```

If `x` is a class variable of `self`, we could use the notation:

```
apple:!::x
```

All combinations are possible, including:

```
object.!selector
object.!\selector
object.!::selector
object:!ivName
object::!cvName
object:!varname:,propName
object,!perspName

(_! object selector arg1 arg2)
```

11.9 Recursive Compound Literals

Multiple colons or periods can be used in a literal, For example:

```
a:b:c
```

means to (1) get the object that is the value of `a`, (2) get the object that is the value of the `b` instance variable of `a`, and nally (3) get the value of the `c` instance variable of that object.

Similarly, the notation

```
a.b:c
```

means to get the `c` variable of the object returned after sending a `b` message to the object that is the value of the variable `a`. Again, the operations are carried out left to right: (1) the object that is the value of the variable `a` is retrieved, (2) it is sent a `b` message which must return an object, and then (3) the value of the `c` variable of that object is retrieved.

Compound literal notation can be nested arbitrarily deeply.

11.10 Assignment Statements

An assignment statement using a left arrow can be used for setting all kinds of variables. For example,

```
x_a;
```

sets the value of the variable `x` to the value of `a`. The same notation works if `x` is a task variable, rule variable, class variable, temporary variable, or work space variable. The right side of an assignment statement can be an expression as in:

```
x_a*b + 17*(LOG d);
```

The assignment statement can also be used with the colon notation to set values of instance variables of objects. For example:

```
y:b_0 ;
```

In this example, rst the object that is the value of `y`is computed, then the value of its instance variable `b` is set to `0`.

Properties and perspectives: Assignment statements can also be used to set property values as in:

```
box:x:,origin_47    fact:,reason_currentSupport;
```

or variables of perspectives as in:.

```
washingMachine,electrical:voltage_110;
```

Nesting: Assignment statements can be nested as in

```
a_b_c:d_3;
```

This statement sets the values of `a`, `b`, and the `d` instance variable of `c` to `3`. The value of an assignment statement itself is the new assigned value.

11.11 Meta-Assignment Statements

Meta-assignment statements are assignment statements used for specifying rule descriptions and audit trails. These statements appear in the MD part of rules.

Audit Trails: The default interpretation of meta-assignment statements for undeclared variables is as audit trail speci cations. Each meta-assignment statement speci es information to be saved in audit records when a rule is applied. In the following example from gure 19, the audit record must have variables named `basis` and `cf`:

```
{(basis_'Fact cf_1)}
IF buyer:familySize>2  machine:capacity<20
THEN suitability_'Poor;
```

In this example, the RHS of the rule assigns the value of the work space instance variable `suitability` to `'Poor` if the conditions of the rule are satis ed. In addition, if the RuleSet was compiled in *audit* mode, then during RuleSet execution an audit record is created as a side-e ect of the assignment. The audit record is attached to the `reason` property of the suitability variable. It has instance variables `basis` and `cf`.

In general, an audit description consists of a sequence of meta-assignment statements. The assignment variable on the left must be an instance variable of the audit record. The class of the audit record is declared in the *Audit Class* declaration of the RuleSet. The expression on the right is in terms of the variables accessible by the RuleSet. If the conditions of a rule are satis ed, an audit record is instantiated. Then the meta-assignment statements are evaluated in the execution context of the RuleSet and their values are put into the audit record. A separate audit record is created for each of the object variables that are set by the rule.

Rule Descriptions: Meta-assignment statements can also be used to set variables in the objects that represent individual rules. This interpretation of meta-assignment statements is indicated when the assignment variable of the meta-assignment statement has been declared to be a rule variable. For example, if the variable `cf` in the previous example was declared to be a rule variable, then the meta-assignment statement would set the `cf` instance variable of the rule object to `.5` at compilation time, instead of saving a `cf` in every audit record for every rule application at execution time. The value on the right hand side of the meta-assignment statement for a rule variable must be known at compile time.

11.12 Push and Pop Statements

A compact notation is provided for pushing and popping values from lists. To push a new value onto a list, the notation `_+` is used:

```
myList_+newItem;
```

```
focus:goals_+newGoal;
```

To pop an item from a list, the `_-` notation is used:

```
item_-myList;
```

```
nextGoal_-focus:goals;
```

As with the assignment operator, the push and pop notation works for all kinds of variables and properties. They can be used in conjunction with in x operator `<<` for membership testing.

11.13 Invoking RuleSets

One of the ways to cause RuleSets to be executed is to invoke them from rules. This is used on the LHS of rules to express predicates in terms of RuleSets, and on the RHS of rules to express actions in terms of RuleSets. A short double-dot syntax for this is provided that invokes a RuleSet on a work space:

```
Rs1..ws1
```

In this example, the RuleSet bound to the variable `Rs1` is invoked with the value of the variable `ws1` as its work space. The value of the invocation expression is the value returned by the RuleSet. The double-dot syntax can be combined with the dollar notation to invoke a RuleSet by its Loops name, as in

```
$MyRules..ws1
```

which invokes the RuleSet object that has the Loops name `MyRules`.

This form of RuleSet invocation is like subroutine calling, in that it creates an implicit stack of arguments and return addresses. This feature can be used as a mechanism for *meta-control* of RuleSets as in:

```
IF breaker:status='Open
THEN source_$OverLoadRules..washingMachine;

IF source='NotFound
THEN $ShortCircuitRules..washingMachine;
```

In this example, two ‘‘meta-rules’’ are used to control the invocation of specialized RuleSets for diagnosing overloads or short circuits.

11.14 Transfer Calls

An important optimization in many recursive programs is the elimination of tail recursion. For example, suppose that the RuleSet A calls B, B calls C, and C calls A recursively. If the rst invocation of A must do some more work after returning from B, then it is useful to save the intermediate states of each of the procedures in frames on the calling stack. For such programs, the space allocation for the stack must be enough to accommodate the maximum depth of the calls.

There is a common and special case, however, in which it is unnecessary to save more than one frame on the stack. In this case each RuleSet has no more work to do after invoking the other RuleSets, and the value of each RuleSet is the value returned by the RuleSet that it invokes. RuleSet invocation in this case amounts to the evaluation of arguments followed by a direct transfer of control. We call such invocations transfer calls.

The Loops rule language extends the syntax for RuleSet invocation and message sending to provide this as follows:

```
RS..*ws
```

The RuleSet `RS` is invoked on the work space `ws`. With transfer calls, RuleSet invocations can be arbitrarily deep without using proportional stack space.

11.15 Task Operations

Tasks in the Loops rule language represent the invocation of RuleSets. They provide a mechanism for specifying and controlling processes in terms of tasks that can be created, started, suspended, and restarted. They also provide a handle for specifying concurrent processing.

A Task records the work space of a RuleSet (`ws`), the value returned (`value`), and two special variables called the `status` and `reason`. A Task can also have RuleSet- speci c instance variables called task variables for saving process information.

Creating Tasks: A Task is represented as a Loops object and can be created and associated with a work space as follows:

```
Task6_(_ $Task New RuleSetworkSpace)
```

The `workSpace` argument is optional. Specialized versions of `Task` will eventually be available, such as `RemoteTask`, Information about a Task is stored in its instance variables, and can be accessed like other Loops variables:

```
Task6:status
Task6:reason
Task6:ws
Task6:value
```

Starting Tasks: The primary operations on Tasks are starting them and waiting for them to nish execution. These operations have been designed to work when Loops is extended for concurrent processing. The operations for starting tasks are as follows:

`(Start1 taskList)` [Function]
`(StartAll taskList)` [Function]
`(StartAll taskList)` [Function]

Each of the start operations takes an argument `taskList` which is either a Task object, or a list of Task objects. A Task cannot be started if it is already running, as indicated by its `status` variable. `Start1` iterates through its `taskList` and starts the rst Task that is not already running. The value of `Start1` is the Task that was started. `StartAll` starts all of the tasks, and does not return control until all of the tasks have been started. `StartTogether` is like `StartAll` except that none of the tasks are started until all of them are ready. The synchronization aspect of `StartTogether` is important for avoiding Task deadlock situations in programs that share Tasks as resources. (It avoids the di culties associated with partial allocation of Tasks when a complete set of Tasks is needed.)

Waiting for Tasks: The following operations are provided for waiting for Tasks:

`(Wait1 taskList)` [Function]
`(WaitAll taskList)` [Function]

`Wait1` iterates through its `taskList` and returns as its value the rst Task that is not running. `WaitAll` returns when all of its Tasks have nished running The value returned by the RuleSet that ran in a Task can be obtained from the Task object, as in:

```
task6:value.
```

Running Tasks: In many cases, the speci cation of Task control can be simpli ed by using a *run* operation that combines the start and wait operations. The run operations are as follows:

`(Run1 taskList)` [Function]
`(RunAll taskList)` [Function]
`(RunTogether taskList)` [Function]

`Run1` goes through its arguments left to right and selects the rst Task that is not running. It starts that Task and then waits for it to complete. The value of `Run1` is the Task that was executed. `RunAll` starts all of the Tasks running and then waits for them all to complete. `RunTogether` waits for all of the Tasks to become available, runs them all, and then waits for them all to complete.

11.16 Stop Statements

At invocation, the `status` in the Task is set to `Running`. If a RuleSet ends normally, the `status` in the Task is set to `Done` and the `reason` saved in the RuleStep is `Success`. Other terminations can be speciﬁed in a Stop statement as follows:

`(Stop value status reason)` [RuleSet Statement]

`value` is the value to be returned by the RuleSet, `status` characterizes the termination of the Task, and `reason` is a symbolic reason for the status. Typical examples of the use of Stop are:

```
(Stop value 'Aborted reason)
(Stop value 'Suspended reason)
```

where `Aborted` means that the RuleSet has failed, and `Suspended` means that the RuleSet has stopped but may be re-invoked. Particular applications will probably develop standardized notations for status and reason. Values for these can be Interlisp atoms or Loops objects. The arguments `status` and `reason` are optional in a `Stop` statement.

12 USING RULES IN LOOPS

The Loops rules language is supported by an integrated programming environment for creating, editing, compiling, and debugging RuleSets. This section describes how to use that environment.

12.1 Creating RuleSets

RuleSets are named Loops objects and are created by sending the class `RuleSet` a `New` message as follows:

```
(_ $RuleSet New)
```

After entering this form, the user will be prompted for a Loops name as

```
RuleSet name: RuleSetName
```

Afterwards, the RuleSet can be referenced using Loops dollar sign notation as usual. It is also possible to include the RuleSet name in the `New` message as follows:

```
(_ $RuleSet New NIL RuleSetName)
```

12.2 Editing RuleSets

A RuleSet is created empty of rules. The RuleSet editor is used to enter and modify rules. The editor can be invoked with an `EditRules` message (or `ER` shorthand message) as follows:

```
(_ RuleSet EditRules)
(_ RuleSet ER)
```

If a RuleSet is installed as a method of a class, it can be edited conveniently by selecting the `EM` option from a browser containing the class. Alternatively, the `EM` function or `EditMethod` message can be used:

`(_ ClassName EditMethod selector)` [Message]

`(EM ClassName selector)` [Function]

Both approaches to editing retrieve the source of the RuleSet and put the user into the TTYIN editor, treating the rule source as text.

Initially, the source is a template for RuleSets as follows:

```
        RuleSet Name:          RuleSetName;
        WorkSpace Class:       ClassName;
        Control Structure:       doAll;
        While Condition: ;
        Audit Class:          StandardAuditRecord;
        Rule Class:              Rule;
```

```
Task Class:         ;
Meta Assignments:      ;
Temporary Vars:;
Lisp Vars:         ;
Debug Vars:      ;
Compiler Options:   ;


   (* Rules for whatever.  Comment goes here.)
```

Figure 22. Initial template for a RuleSet. The rules are entered after the comment at the bottom. The declarations at the beginning are lled in as needed and super uous declarations can be discarded.

The user can then edit this template to enter rules and set the declarations at the beginning. In the current version of the rule editor, most of these declarations are left out. If the user chooses the `EditAllDecls` option in the RuleSet editor menu, the declarations and default values will be printed in full.

The template is only a guide. Declarations that are not needed can be deleted. For example, if there are no temporary variables for this RuleSet, the `Temporary Vars` declaration can be deleted. If the control structure is not one of the while control structures, then the `While Condition` declaration can be deleted. If the compiler option `A` is not chosen, then the `Audit Class` declaration can be deleted.

When the user leaves the editor, the RuleSet is compiled automatically into a LISP function.

If a syntax error is detected during compilation, an error message is printed and the user is given another opportunity to edit the RuleSet.

12.3 Copying RuleSets

Sometimes it is convenient to create new RuleSets by editing a copy of an existing RuleSet. For this purpose, the method `CopyRules` is provided as follows:

`(_ oldRuleSet CopyRules newRuleSetName)` [Message]

This creates a new RuleSet by some of the information from the pespectives of the old RuleSet. It also updates the source text of the new RuleSet to contain the new name.

12.4 Saving RuleSets on LISP Files

RuleSets can be saved on LISP les just like other Loops objects. In addition, it is usually useful to save the LISP functions that result from RuleSet compilation. In the current implementation, these functions have the same names as the RuleSets themselves. To save RuleSets on a le, it is necessary to add two statements to the le commands for the le as follows:

```
(FNS * MyRuleSetNames)
(INSTANCES * MyRuleSetNames)
```

where `MyRuleSetNames` is a LISP variable whose value is a list of the names of the RuleSets to be saved.

12.5 Printing RuleSets

To print a RuleSet without editing it, one can send a `PPRules` or `PPR` message as follows:

```
(_ RuleSet PPRules)                                        [Message]
(_ RuleSet PPR)                                            [Message]
```

A convenient way to make hardcopy listings of RuleSets is to use the function `ListRuleSets`. The les will be printed on the `DEFAULTPRINTINGHOST` as is standard in Interlisp-D. `ListRuleSets` can be given three kinds of arguments as follows:

```
(ListRuleSets RuleSetName)
(ListRuleSets ListOfRuleSetNames)
(ListRuleSets ClassName)
(ListRuleSets FileName)
```

In the `ClassName` case, all of the RuleSets that have been installed as methods of the class will be printed. In the last case, all of the RuleSets stored in the le will be printed.

12.6 Running RuleSets from Loops

RuleSets can be invoked from Loops using any of the usual protocols.

Procedure-oriented Protocol: The way to invoke a RuleSet from Loops is to use the `RunRS` function:

(RunRS RuleSet workSpace arg$_2$ arg$_N$) [Function]

`workSpace` is the Loops object to be used as the work space. This is ‘‘procedural’’ in the sense that the RuleSet is invoked by its name. `RuleSet` can be either a RuleSet object or its name.

Object-oriented Protocol: When RuleSets are installed as methods in Loops classes, they can be invoked in the usual way by sending a message to an instance of the class. For example, if `WashingMachine` is a class with a RuleSet installed for its `Simulate` method, the RuleSet is invoked as follows:

```
(_ washingMachineInstance Simulate)
```

Data-oriented Protocol: When RuleSets are installed in active values, they are invoked by side-e ect as a result of accessing the variable on which they are installed.

12.7 Installing RuleSets as Methods

RuleSets can also be used as methods for classes. This is done by installing automatically-generated invocation functions that invoke the RuleSets. For example:

```
[DEFCLASS WashingMachine
   (MetaClass Class doc (* comment) ...)
```

```
    ...
    (InstanceVariables (owner ...))
    (Methods
             (Simulate RunSimulateWMRules)
             (Check RunCheckWMRules
                          doc (* Rules to Check a washing machine.))
...]
```

When an instance of the class WashingMachine receives a Simulate message, the RuleSet SimulateWMRules will be invoked with the instance as its work space.

To simplify the definition of RuleSets intended to be used as Methods, the function DefRSM (for ''Define Rule Set as a Method'') is provided:

(DefRSM ClassName Selector RuleSetName) [Function]

> If the optional argument RuleSetName is given, DefRSM installs that RuleSet as a method using the ClassName and Selector. It does this by automatically generating an installation function as a method to invoke the RuleSet. DefRSM automatically documents the installation function and the method.
>
> If the argument RuleSetName is NIL, then DefRSM creates the RuleSet object, puts the user into an Editor to enter the rules, compiles the rules into a LISP function, and installs the RuleSet as before.

12.8 Installing RuleSets in ActiveValues

RuleSets can also be used in data-oriented programming so that they are invoked when data is accessed. To use a RuleSet as a getFn, the function RSGetFn is used with the property RSGet as follows:

```
...
(InstanceVariables
     (myVar #(myVal RSGetFn NIL) RSGet RuleSetName))
...
```

RSGetFn is a Loops system function that can be used in an active value to invoke a RuleSet in response to a Loops get operation (e.g., GetValue) is performed. It requires that the name of the RuleSet be found on the RSGet property of the item. RSGetFn activates the RuleSet using the local state as the work space. The value returned by the RuleSet is returned as the value of the get operation.

To use a RuleSet as a putFn, the function RSPutFn is used with the property RSPut as follows:

```
...
(InstanceVariables
     (myVar #(myVal NIL RSPutFn) RSPut RuleSetName))
...
```

RSPutFn is a function that can be used in an active value to invoke a RuleSet in response to a Loops put operation (e.g., PutValue). It requires that the name of the RuleSet be found on the RSPut property of the item. RSGetFn activates the RuleSet using the newValue from the put operation as the work space. The value returned by the RuleSet is put into the local state of the active value.

12.9 Tracing and Breaking RuleSets

Loops provides breaking and tracing facilities to aid in debugging RuleSets. These can be used in conjunction with the auditing facilities and the rule executive for debugging RuleSets. gure 23 summarizes the compiler options for breaking and tracing:

`T`	Trace if rule is satis ed. Useful for creating a running display of executed rules.
`TT`	Trace if rule is tested.
`B`	Break if rule is satis ed.
`BT`	Break if rule is tested. Useful for stepping through the execution of a RuleSet.

Figure 23. Compiler options for Breaking and Tracing the execution of RuleSets.

Specifying the declaration `Compiler Options: T;` in a RuleSet indicates that tracing information should be displayed when a rule is satis ed. To specify the tracing of just an individual rule in the RuleSet, the `T` meta-descriptions should be used as follows:

```
{T}   IF cond THEN action;
```

This tracing speci cation causes Loops to print a message whenever the LHS of the rule is tested, or the RHS of the rule is executed. It is also possible to specify that the values of some variables (and compound literals) are to be printed when a rule is traced. This is done by listing the variables in the `Debug Vars` declaration in the RuleSet:

```
Debug Vars: a a:b a:b.c;
```

This will print the values of `a`, `a:b`, and `a:b.c` when any rule is traced or broken.

Analogous speci cations are provided for breaking rules. For example, the declaration `Compiler Options: B;` indicates that Loops is to enter the rule executive (see next section) after the LHS is satis ed and before the RHS is executed. The rule-speci c form:

```
{B}   IF cond THEN action;
```

indicates that Loops is to break before the execution of a particular rule.

Sometimes it is convenient in debugging to display the source code of a rule when it is traced or broken. This can be e ected by using the `PR` compiler option as in

```
Compiler Options: T PR;
```

which prints out the source of a rule when the LHS of the rule is tested and

```
Compiler Options: B PR;
```

which prints out the source of a rule when the LHS of a rule is satis ed, and before entering the break.

12.10 The Rule Exec

A Read-Compile-Evaluate-Print loop, called the rule executive, is provided for the rule language. The rule executive can be entered during a break by invoking the LISP function `RE`. During RuleSet execution, the rule executive can be entered by typing `^f` (<control>-f) on the keyboard.

On the rst invocation, `RE` prompts the user for a window. It then displays a stack of RuleSet invocations in a menu to the left of this window in a manner similar to the Interlisp-D Break Package. Using the left mouse button in this window creates an Inspector window for the work space for the RuleSet. Using the middle mouse button pretty prints the RuleSet in the default prettyprint window.

In the main rule executive window, `RE` prompts the user with ‘‘`re:`’’. Anything in the rule language (other than declarations) that is typed to this executive will be compiled and executed immediately and its value printed out. For example, a user may type rules to see whether they execute or variable names to determine their values. For example:

```
re: trafficLight:color
Red
re:
```

this example shows how to get the value of the `color` variable of the `trafficLight` object. If the value of a variable was set by a RuleSet running with auditing, then a `why` question can be typed to the rule executive as follows:

```
re: why trafficLight:color

IF highLight:color = ’Green  farmRoadSensor:cars  timer.TL
THEN highLight:color _ ’Yellow  timer.Start;

Rule 3 of RuleSet LightRules
Edited: Conway "13-Oct-82"

re:
```

The rule executive may be exited by typing `OK`.

12.11 Auditing RuleSets

Two declarations at the beginning of a RuleSet a ect the auditing. Auditing is turned on by the compiler option `A`. The simplest form of this is

```
Compiler Options: A;
```

The `Audit Class` declaration indicates the class of the audit record to be used with this RuleSet if it is compiled in *audit* mode.

```
Audit Class: StandardAuditRecord;
```

A `Meta Assignments` declaration can be used to indicate the audit description to be used for the rules unless overridden by a rule-speci c meta-assignment statement in braces.

```
Meta Assignments: ( cf_.5 support_'GroundWff);
```

13 USING THE LOOPS SYSTEM

Loops is integrated with Interlisp-D, and makes use of many of its advanced features. In order to run Loops one must have the appropriate version of the Interlisp-D system and the corresponding versions of a set of LispUsers packages. The instructions for building the system as of February 1, 1983 are contained in a document of export instructions, currently led on: `{MAXC}<LOOPS>EXPORTINSTRUCTIONS.TXT` .

13.1 Starting up the System

At PARC, we maintain two version of Loops most of the time, a current system which is a released version, an another which is the system under development. There are two command les: `loops.cm` and `newLoops.cm` which start up a Lisp and fetch the appropriate sysout from a server.

In the version of the system as loaded at PARC, we include the following Lispusers packages: `TTY`, `TMENU`, `GRAPHER`, `HISTMENU`, `SINGLEFILEINDEX`, `PATCHUP`

The rst four packages must be included in any loadup of Loops; the second are ones we nd useful. Documentation of these facilities are to be found on `<LISPUSERS>` directories on various servers.

13.2 The Loops Screen Setup

The screen as one sees it set up contains the following windows(top to bottom, left to right):

Prompt Window Small black window in upper left. Prompts for what will happen in various mouse interactions appear here. Also various noti cations of directory attachment changes. Labelled with the date of the Lisp system loadup and of the Loops system loadup.

Top Level Window Normal interaction window. Labelled with the currently connected directory.

User Exec PPDefault Window Below the EditCommands menu is a title icon of the UserExec window. When this is expanded it lls the bottom half of the screen. It can be used for TTY interactions. It can be made the primary window for such interactions by calling the function `UE`. Typing `OK` when in that window returns you to the previous `TTYDIPLAYSTREAM`. This window is also used as the default place to prettyprint class and instance descriptions.

There are three icons on the right half of the screen.

Loops Icon This circular icon is active and if buttoned gives the user the option of setting up the screen again (useful if it has been cluttered with many windows), and of producing a graph browser of the current classes in the system.

History Icon This icon will expand to give a History menu list. See the write up on `<LISPUSERS>HISTMENU.TTY` .

Edit Work Area This window is shown only by a title icon in the upper right. It expands when necessary, and takes up the entire right half of the screen. It shrinks automatically when `DoneEdit` is selected from the EditCommand menu. It can be expanded to allow you to look at the last expression being edited.

13.3 Using the Browser

Two special classes in the system are used to build browsers based on the grapher package. The general class is called `LatticeBrowser`, and the particular subClass that is used by the system is called `ClassBrowser`. We will rst describe how to use the class browser which appears when requested by buttoning in the Loops icon. We then describe how to build your own browser.

13.3.1 Using the Class Browser

The items in the class browser can be buttoned with either the left or middle button. When buttoned a pop up menu will appear, and the user can make a selection of one of these.

If a browser menu selection is followed by an asterisk (i.e., `Print*`), this means that it has a number of sub-commands. Selecting such a selection with the middle mouse button will present another pop-up menu of sub-commands. Selecting a ‘‘starred’’ selection with the left mouse button will execute the ‘‘default’’ sub-command. The left and middle mouse buttons act the same when selecting an un-starred selection.

The left button menu selections are:

`Print*` Prints a summary of information about the selected class in the ‘‘User Exec PPDefault Window’’. If selected with the middle mouse button, another pop-up menu gives a choice of what to print:

`PP`	PrettyPrint Class de nition.
`PP!`	PrettyPrint Class de nition including inherited information.
`PPV!`	Same as `PP!` without seeing methods.
`PPM`	Puts up a pop-up menu of all of the methods de ned in the class, and prettyprints the de nition of the selected one.
`PrintSummary`	Prints a summary of all of the information (instance variables, class variables, and methods) for the selected class

If `Print*` is selected with the left button, `PrintSummary` is the default sub-command that is executed.

`Doc*` Prints documentation for Classes, IVs, CVs, or Methods. If selected with the middle mouse button, another pop-up menu gives a choice of what to print:

`ClassDoc`	Prints Class doc information for selected class.
`MethodDoc`	Puts up a pop-up menu of all of the methods de ned in the class, and prints the doc information of the selected one. This pop-up menu is redisplayed until the user buttons outside the menu, so that the user can see the doc information from multiple methods.
`IVDoc`	Same as `MethodDoc`, except that it prints the doc information for

instance variables of the class.

- `CVDoc` Same as `MethodDoc`, except that it prints the doc information for class variables of the class.

If `Doc*` is selected with the left button, `ClassDoc` is the default sub-command that is executed.

`WhereIs` This command is used to nd out which super class of the selected class a particular IV, CV, or Method was inherited from. When selected with the left or middle mouse button, a pop-up menu is displayed with the elements `IVS`, `CVS`, `Methods`. Whichever element is selected, a pop-up menu of the class' instance variables (or class variables or methods) is displayed. When one of these is selected, the super class from which that IV, CV or Method was inherited is ashed, and its nameis printed in the Prompt Window. This nal pop-up menu is redisplayed until the user buttons outside the menu, so that the user select multiple IVs (or CVs or methods).

`Unread` Unreads `$className` into the typein bu er. This is useful when typing messages to particular classes.

The middle button menu selections are:

`EM*` Edit a method in the selected class. If selected with the middle mouse button, puts up another pop-up menu:

- `EM` Puts up a pop-up menu of all of the methods de ned in the class, and envokes the editor on the selected method.
- `EM!` Same as `EM`, except that includes all inherited methods in the list.

If `EM*` is selected with the left button, `EM` is the default sub-command that is executed.

`Add*` Add a new method, a specialized class, an IV, or a CV to the selected class, or make a new instance. If selected with the middle mouse button, puts up another pop-up menu:

- `Specialize` Creates a new subclass of the selected class, giving it a name typed by the user.
- `DefMethod` De ne a new method to the selected class. Asks the user (in the prompt window) to type the name of a selector, and envokes the editor on a dummy de nition for that new method.
- `DefRSM` Installs a RuleSet as a method in a class. Asks the user (in the prompt window) to type the name of a selector, and invokes the RuleSet editor. When the user exits the RuleSet editor, the RuleSet is compiled and installed as the method in the class.
- `AddIV` Asks the user to type an instance variable name, and adds it to the selected class.
- `AddCV` Asks the user to type a class variable name, and adds it to the

selected class.

`New`	Sets the Interlisp variable `IT` to a new instance of the selected class.

If `Add*` is selected with the left button, `DefMethod` is the default sub-command that is executed.

`Delete` Delete a method, IV, or CV from the selected, or the whole selected class. Puts up a pop-up menu with elements `IVs`, `CVs`, `Methods`, and `Class`. If one of the rst three is selected, a menu of the selected class' instance variables, class variables, or methods is given, and the selected one is deleted from the class. If `Class` is selected, the whole class is deleted.

`Move*` Move or copy an IV, CV, method, or super from the selected class to another class. The destination class is speci ed by using the `BoxNode` command, described below. If selected with the middle mouse button, puts up another pop-up menu:

`MoveTo`	Puts up a pop-up menu with elements `IVS`, `CVS`, `Methods`, and `Supers`. Selecting one of these will put up still another menu, listing the items of that type. Selecting one of these items will cause it to be moved to the destination class speci ed with `BoxNode`.
`CopyTo`	The same as `MoveTo`, except that the selected item is copied to the destination class.

If `Move*` is selected with the left button, `MoveTo` is the default sub-command that is executed.

`BoxNode` Draws a box around the selected class node. If the selected class is already boxed, the box is removed. If any other class node has been boxed, that box is removed. This command is used in conjunction with the `Move*` command to specify a ‘‘destination class’’, as described above.

`Rename*` Renames some part of the selected class. Puts up a pop-up menu with elements `IVS`, `CVS`, `Methods`, and `Class`. Selecting one of these will put up still another menu, listing the items of that type. Selecting one of these items will cause it to be renamed to a name typed in by the user.

`Edit*` Edit some part of the selected class. If selected with the middle mouse button, puts up another pop-up menu:

`EditObject`	Calls the editor to edit the selected class.
`EditIVs`	Calls the editor to edit the instance variables of the selected class.
`EditCVs`	Calls the editor to edit the class variables of the selected class.
`Inspect`	Call the Interlisp inspector to inspect the selected class.

If `Edit*` is selected with the left button, `EditObject` is the default sub-command that is executed.

Pressing either the left or middle mouse button in the title region at the top of the class browser brings up another pop-menu, containing commands which deal with the entire browser. The commands are:

`Recompute`	Recompute class lattice from the ‘‘starting list’’ of objects (described below).
`AddRoot`	Add named item to starting list for browser.
`DeleteRoot`	Delete named item from starting list for browser.
`SaveInIT`	Store this browser object in the Interlisp variable `IT`.

To create a Class Browser for a small set of classes, send the message `Show` to the class `ClassBrowser`:

```
(_New ($ ClassBrowser) Show browseList window)
```

This displays the class inheritance lattice starting with the ‘‘starting list’’ of objects `browseList`. `browseList` can be a single className or class, or a list of these. A new browse window will be created which contains nodes for each class mentioned, and (recursively) all subclasses of those classes in the current environment which have been accessed. If `window` is given, then it will be used as the display window.

13.3.2 Building Your Own Browser

* * * The following information is incorrect. If you want to build your own browser, try poking around the class `LatticeBrowser`. Good Luck. * * *

The general class which supports browsing is `LatticeBrowser`. The specialization `ClassBrowser` is used to generate the Class Inheritance Lattice Browser that we all use. `ClassBrowser` provides an example of how to specialize `LatticeBrowser` for your own use. The following is a brief description of the `LatticeBrowser` messages.

If `($ Lb)` is an instance of (any subclass of) `($ LatticeBrowser)` then:

```
(_ ($ Lb) Show browseList)
```

will create a graph of elements starting with those in `browseList`. `browseList` should be a list of objectNames or objects. If `browseList` is single item, it will be treated as list of that item. The browser will show a lattice of elements determined by a sub relation implemented by the `LatticeBrowser` message `GetSub`. For each object, `(_ ($ Lb) GetSubs object)` should produce a list of objects which are the ‘‘subs’’ of `object`, and `(_ ($ Lb) GetLabel object)` should produce a string to be used in the graph as a label. The `GetSubs` method in `LatticeBrowser` just obtains the value of the instance variable `sub`, if it exists in that object (no error otherwise). The `GetLabel` method in `LatticeBrowser` nds the name of the object.

Each node in the browser graph has actions associated with the left and middle mouse buttons. When either button is clicked over a node, a menu of actions is brought up. The items on the action menu are determined by the class variables `LeftButtonItems` and `MiddleButtonItems`.

The value obtained by selecting the menu item will be used as a message selector for an action. The message will be sent either to the browser or to the object itself. Selectors on the class variable `LocalCommands`, or those not understood by the object will be sent in a message to the browser, with arguments of the object and objectName. Otherwise, the object will be sent that selector as a unary message (no arguments).

For example, assume that the value of LeftButtonItems was (PP PP! EditObject) and the value of LocalCommands was NIL, and EditObject is not understood by obj1selected in the browser. By buttoning PP (or PP!) in the action menu, obj1would be sent the message PP (or PP!). Selecting EditObject would result in sending the message (_ ($ Lb) EditObject obj1(GetName obj1)).

A LatticeBrowser responds to EditObject by sending the object the message Edit *in a TTY process*. The latter is necessary to allow the mouse to continue to work in the process world. If obj1might have understood the message EditObject, then that atom should appear on the list LocalCommands to ensure that the browser is sent the message rather than obj1

As usual with menus, items need not be atoms. If an item is a list, EVAL of the second element is returned. Thus one might have the element ("Edit With EE" 'EEObject) on a menu item list, so the string "Edit With EE" will be displayed in the Menu, and the message EEObject sent when that item is selected.

If the result of selecting an item returns a list, the CAR of the list is treated as the selector, and CDR is an extra argument to send. For example, in the class browser MiddleButtonItems contains an item (EditIVs '(EditObject -2 EE)). Selecting EditIVs in the menu causes the following message to be sent: (_ ($ Lb EditObject object(-2 EE))

Shifted Selections If one selects a node with the LEFT or MIDDLE mouse button while holding down the left shift key, then a message is sent to the browser:

```
(_ ($ Lb) LeftShiftSelect objectobjName)
(_ ($ Lb) MiddleShiftSelect objectobjName)
```

The default behavior for LeftShiftSelect is to send PP! to the object, and for MiddleShiftSelect to send EEObject to the browser.

Moving Nodes Holding the CTRL key down when selecting allows one to move the selected node in the browser window. This does not a ect the underlying structure, just the display.

Format of the Browser Window One can obtain a browser display with a speci ed title or in an existing window. If one speci es windowOrTitlein

```
(_ ($ Lb) Show browseListwindowOrTitle)
```

then if windowOrTitleis a string, it will be used as the title of a new window for the browser. If windowOrTitleis a window, then that window will be used as is. If windowOrTitle= NIL, then the title is obtained from the instance variable title, and a new window is created and stored in the instance variable window. If the instance variable topAlign= T (the default) then GRAPHER will align the graph to the top of the window. The font used for labels is found in the instance variable browseFont. At any time, the last object selected is found in lastSelectedObject.

SUMMARY: To specialize a browser, de ne the method for GetSubs. If the browser is not using object names for its labels, specialize GetLabel. Set up the class variables LeftButtonItems, MiddleButtonItems and LocalCommands. Specialize LeftShiftSelect and MiddleShiftSelect if desired.

LatticeBrowser [Class]

IV's:

`boxedNode` [IV of LatticeBrowser]
The last object boxed, if any.

`browseFont` [IV of LatticeBrowser]
The font used for labels.

`lastSelectedObject` [IV of LatticeBrowser]
Last object selected.

`startingList` [IV of LatticeBrowser]
List of objects used to compute this browser.

`title` [IV of LatticeBrowser]
Title passed to GRAPHER package.

`topAlign` [IV of LatticeBrowser]
Flag used to indicate whether graph should be aligned with the top or bottom of the window. If `topAlign` = `T` (the default) then GRAPHER will align the graph to the top of the window.

`window` [IV of LatticeBrowser]
Window for browsing.

CVs:

`LeftButtonItems` [CV of LatticeBrowser]
Items for left button menu. Value sent as message to object or browser.

`LocalCommands` [CV of LatticeBrowser]
List of messages that should be sent to browser when item is selected in menu, even if object does understand them.

`MiddleButtonItems` [CV of LatticeBrowser]
Items for middle button menu. Value sent as message to object or browser.

`TitleItems` [CV of LatticeBrowser]
Items for menu in title of window.

Methods:

`(_ browser BoxNode object)` [Method of LatticeBrowser]
Draws a box around the node in the graph representing the object.

`(_ browser DoSelectedCommand command obj objName)` [Method of LatticeBrowser]
Does the selected command or forwards it to the object.

`(_ browser EEObject object objName)` [Method of LatticeBrowser]
Edit `object`, using the TTYIN editor (in a `TTYPROCESS`).

`(_ browser EditObject object objName args)` [Method of LatticeBrowser]
Edit `object` using Lisp editor (in a `TTYPROCESS`), passing the commands `args`.

`(_ browser FlashNode node N }ashTime)` [Method of LatticeBrowser]
: Call `FlipNode` 2`N` times, delaying for `}ashTime` milliseconds between ips. Default values: `N`=3, `}ashTime`=300.

`(_ browser FlashNode object)` [Method of LatticeBrowser]
: Inverts the video around the node in the graph representing `object`

`(_ browser GetLabel object)` [Method of LatticeBrowser]
: Returns the label for `object` displayed in the browser.

`(_ browser GetNodeList browseList goodList)` [Method of LatticeBrowser]
: Returns the node data structures of the tree starting at `browseList` If `goodList` is given, only include elements of it. If `goodList`=`T`, this is the same as `goodList`=`browseList`

`(_ browser GetSubs object)` [Method of LatticeBrowser]
: Returns a list of the subs from `object`

`(_ browser LeftShiftSelect object objname)` [Method of LatticeBrowser]
: Called when `object` is selected with the `LEFT` mouse button while the shift key is down.

`(_ browser MiddleShiftSelect object objname)` [Method of LatticeBrowser]
: Called when `object` is selected with the `MIDDLE` mouse button while the shift key is down.

`(_ browser ObjNamePair objOrName)` [Method of LatticeBrowser]
: `objOrName` may be either an object or a name used to label an object in the browser. Returns the pair `(object . objName)`.

`(_ browser Recompute)` [Method of LatticeBrowser]
: Recompute the browser display using same window and `browseList`

`(_ browser Show browseList windowOrTitle goodList)` [Method of LatticeBrowser]
: Show the items and their subs on a browse window.

`(_ browser Unread object objName)` [Method of LatticeBrowser]
: Put `$objName` into the tty bu er

13.4 Editing in Loops

This section is about editing in Loops. It describes the Loops interface to the standard Interlisp editors. In addition to the usual teletype oriented editor, Interlisp-D, provides a variety of other editing programs that make available the bene ts of a bitmap display and a mouse. We will describe some of the interfaces to these editors, but leave the instruction on editing to the appropriate other documents

13.4.1 Editing a Class

The editor for classes is invoked by sending the message `Edit` to the class to be edited. The message `Edit` allows an optional argument, a list of editing commands, as do all the usual Lisp editing functions.

Example: To edit StudentEmployee:

```
(_ ($ StudentEmployee) Edit)
```

An alternative way to edit a class is provided by the LISP function EC (for ‘‘edit class’’). EC takes the class name as its argument. For this example, the form is:

```
(EC ($ StudentEmployee))
```

At this point, if you prettyprint the expression you will see:

```
[DEFCLASS StudentEmployee
   (MetaClass Class)
   (Supers Student Employee)
   (InstanceVariables)
   (ClassVariables)
   (Methods)]
```

Suppose now you edit this structure to the one shown below:

```
[DEFCLASS StudentEmployee
   (MetaClass Class)
   (Supers Student Employee)
   (InstanceVariables   (name)
                       (project "KBE"))
   (ClassVariables   (numberEmployees 0))
   (Methods          (Work StudentEmployee.Work))
```

This speci es that each instance will have two instance variables, name and project, with default values of NIL and "KBE", respectively. The class has a class variable numberEmployees, initialized to 0. If we have an instance of this class bound to the Lisp variable worker, the following expression causes this instance to respond to the message Work:

```
(_ worker Work 3)
```

The result of evaluating this expression is to call the Lisp function StudentEmployee.Work with arguments (the value of) worker and 3. This is described in more detail in the section on methods.

The normal way to terminate editing is with OK. This causes the revised de nition to be installed. If you exit from this editing session with STOP or ^D, all the changes of this session will be lost, since the list structure is not saved; it is only used to build the new class structure. If you have made any syntax errors in editing, warning messages will be printed when you type OK, and you will be returned to the editor.

13.4.2 Editing an Instance

To edit an instance, send it the message Edit.

```
(_ objectEdit)
```

This will put you in the Interlisp editor editing a source for the instance. When you end with OK, the new values will be inserted in the instance.

An equivalent way to edit an instance is

```
(EI object)
```

where object is an instance. (If one has an Interlisp variable, say X1, bound to an instance then to edit one should type (EI X1).

When instances refer to other instances, they are printed out in the form #"UI&DII", that is as a hash mark (#) followed by a string which is a unique identi er. When this is read back in from the string editing bu er of TTYIN, a readmacro for # converts it back into a pointer to an instance with that unique identi er. When a class is printed out for TTYIN it prints as #$ClassName, and the # readmacro converts it bvack into a pointer to the class.

13.4.3 Editing a Method

Often it is convenient to type to enter only a skeletal de nition for a method, and then nish making the speci cations by using an editor. To edit the function for a particular method:

```
(EM className selector)
```

This puts you in the Lisp editor, editing whatever function is associated with the selector speci ed. The name of the actual function is printed out as you enter the editing process. Aside from the syntactic convention of having the rst argument to a function implementing a method be self, these methods are perfectly normal Lisp functions. However, special compilations can be done on these using the GLISP compiler for Loops. This is documented in the section on Lisp interactions.

13.5 Inspecting in Loops

Loops is integrated into the Lisp system so that one can invoke the Inspector on Loops objects. This uses the Loops inspect package, which allows a specialized way of viewing the objects in Loops terms as described in the two sections below.

13.5.1 Inspecting Classes

To inspect a class, send the message Inspect:

```
(_ ($ className) Inspect)
```

13.5.2 Inspecting Instances

An alternative way to modify an instance is to inspect it:

```
(_ object Inspect)
```

and then you can set any values and properties, and add or delete any IVs.

13.6 Errors in Loops

Most errors in Loops which are not errors in Lisp call the function HELPCHECK, which prints out a message, and goes into a Lisp break. The appropriate response to some errors is described below.

13.6.1 When the Object is Not Recognized

When the value of object in the form

(_ object selector arg_1 arg_N)

is not a Loops object, Loops activates the NoObjectForMsg method in the kernel class Object.

The response to this condition can be changed as described below.

This condition can arise if the ller refers to an object that is not in the current environment. For example,

(_ ($ FOO) selector arg_1 arg_N)

will cause the condition if there is no class named FOO in the current environment. In the default case, this causes an error. A user can return from the error by typing

```
RETURN MyValue
```

to let the process continue, returning MyValue as the value that should have been returned had the method been applied successfully.

Alternatively it is possible to create user-speci c responses to this condition by creating a class with a NoObjectForMsg method and setting the global LISP variable DefaultObject to that class. The arguments to the NoObjectForMsg method are object and Selector. This method should carry out whatever response is appropriate, apply the method that was intended, and return the value of that application.

13.6.2 When the Selector is Not Recognized

If the object is recognized but the selector is not, then the object is sent a MessageNotUnderstood message as follows:

```
(_ object MessageNotUnderstood selector)
```

In most cases, this invokes the default method on the kernel class Object which attempts to perform spelling correction. If the correction fails, then a break is caused. If the user then types

```
RETURN selector
```

to the Lisp Break Package, the selector so named will be used.

Alternatively it is possible to create user-speci c responses to this condition by providing a MessageNotUnderstood method in some super of the object. This method should return a Lisp atom other than `NIL`, which is then used as the selector as the `SEND` is tried again.

13.7 Breaking and Tracing Methods

`(BreakMethod className selector)` [Function]

This function will break the method called by `selector` in the specified class. It will find the function name and break it, even if the selector is only found in a superclass. All calls to that function will be broken, even ones that do not come from className.

`(TraceMethod className selector)` [Function]

Similar to `BreakMethod`, except that it traces the appropriate method.

The Lisp function `UNBREAK` will unbreak the function which was broken.

13.8 Monitoring Variable Access

`(BreakIt self varName propName type breakOnGetAlsoFlg)` [Function]

This function is used for causing an Interlisp break when the value of a variable or property is set or fetched. The `type` argument is one of `IV`, `CV`, `METHOD`, or `CLASS` for instance variables, class variables, method properties, or class properties respectively. If it is `NIL`, then `IV` is assumed. If `propName` is `NIL`, then type must be `IV` or `CV` and `BreakIt` refers to the value of a variable.

If `breakOnGetAlsoFLg` is `NIL` then the break is only entered when an attempt is made to store into the value. If `breakOnGetAlsoFLg` is `T`, then breaks will also occur on attempts to fetch the value.

`(TraceIt self varName propName type traceOnGetAlsoFlg)` [Function]

Similar to `BreakIt`, except that it will trace the value of a variable or property, printing the old and new values when the variable or property is accessed.

`(UnBreakIt self varName propName type)` [Function]

This function is used to remove monitoring (breaking or tracing) for the specified variable or property. If `self` is `NIL`, then all known breaks and traces are removed.

14 THE LOOPS KERNEL

14.1 The Golden Braid (Object, Class, MetaClass)

All objects are directly or indirectly a subclass of the object called `Object`. `Object` holds all the methods for the defualt behavior of objects. Heuristics for using these classes. This is the only object with no super classes.

`Class` is the class which holds the default behavior for all classes as objects. `Class` is the default MetaClass for all classes. If `Class` is not the MetaClass for a class, it must be on the supers of that metaClass. There are messages elded by `Class` that know how to create and initialize instances.

`MetaClass` is the class which holds the default behavior for classes which create classes. `MetaClass` is the metaclass for `Class`, and is the only class which is its own metaClass. In accordance with the paragraph above `Class` is a super of `MetaClass`.

14.2 Perspectives and Nodes

In many cases it is useful to organize information in terms of multiple points of view. For example, information about a man might be organized in terms of his role as a father, as an employee, and as a traveler. Each point of view, called a perspective, contains information for a di erent purpose. The perspecitives are related to each other in the sense that they collectively provide information about the same object. Loops supports this organizational metaphor by providing special mixin classes called `Perspective` and `Node`.

`Perspective` [Class]

IVs:

`perspectiveNode` [IV of Perspective]
Indirect pointer to onode containing all perspectives of this object.

Methods:

`(_ self AddPersp viewName view)` [Method of Perspective]
Adds a perspective to my node.

`(_ self DeleteMeAsPersp)` [Method of Perspective]
Delete this object as a perspective of node.

`(_ self DeletePersp viewName view dontCauseError)` [Method of Perspective]
Deletes a perspective from node.

`(_ self Destroy)` [Method of Perspective]
Destroy self but leave other perspectives on Node.

`(_ self Destroy!)` [Method of Perspective]
Destroy self, Node and all other perspectives on Node.

(_ self GetPersp perspName causeError) [Method of Perspective]

Returns the perspective of this instance with viewName perspName.

(_ self MakePersp viewName nodeType) [Method of Perspective]

If no current perspectiveNode exists, then a node will be created of class nodeType (or Node if nodeType = NIL). nodeType should be a subclass of Node. self will be made the value of the property viewName on IV perspectives of node. If self already has a node, then it is used.

Node [Class]

IVs:

perspectives [IV of Node]

Associated objects are stored on the property list of perspectives under their perspective names. The value of this IV is irrelevant.

Methods:

(_ self AddPersp viewName view dontCauseError) [Method of Node]

Adds a perspective to a node on the IV perspectives as value of property viewName.

(_ self DeletePersp viewName view dontCauseError) [Method of Node]

Deletes a perspective of a node on the IV perspectives on property viewName. Checks for consistency. Removes from IV pespectiveNode of view, self is value, and viewName from property myViewName. If view is not that perspective, then causes an error, unless surpressed.

(_ self Destroy) [Method of Node]

Destroy the node after detaching all its perspectives.

(_ self Destroy!) [Method of Node]

Destroy the node and all its perspectives.

(_ self GetPersp perspName causeError) [Method of Node]

Returns the perspective of this node with viewName of perspName

14.3 Useful Mixins

NamedObject and GlobalNamedObject contain only one instance variable, name which holds the name of this object. Any Loops object can be named, but NamedObject and GlobalNamedObject both have their names as part of their structure, and if the structure is changed they update their name. As indicated by its name, instances of GlobalNamedObject are named in the global name table and will be known independent of the environment they are in. Instances of NamedObject may only be known in a single environment, and the name may be reused in another environment.

`NamedObject` [Class]

`GlobalNamedObject` [Class]

`DatedObject` [Class]

`DatedObject` has appropriate initial active values on its two instance variables so that they are lled in at creation with the right values.

IVs:

`created` [IV of DatedObject]

Date and time of creation of object.

`creator` [IV of DatedObject]

USERNAME of creator of object.

`Varlength` [Class]

`VarLength` is a mixin class which allows a class to have indexed instance variables, from 1 to `(_ obj Length)`. These have not yet been extensively used.

IVs:

`indexedVars` [IV of Varlength]

Place where indexed variables are stored for `VarLength` classes.

Methods:

`(_ self Length)` [Method of Varlength]

Returns number of indexed variables allocated in this instance.

14.4 The MetaClass Named ‘‘Class’’

This sections describes the methods de ned in the metaClass `Class`. Any of these methods can be augmented or superceeded in a particular class. The complete list of methods associated with a class can be determined by using the browser.

The `Add`, `Delete`, `List` and `List!` methods have an argument `type` which speci es the type of element to be added, deleted, or listed. For specifying single items, `type` should be one of `IV`, `CV`, `IVProp`, `CVProp`, `Method`, `Super`, or `Meta`. For specifying sets of items, `type` should be `IVs`, `CVs`, `IVProps`, `CVProps`, `Methods`, `Supers`, `Selectors`, or `Functions`.

In the following methods, adding or deleting instance variables and instance variable properties a ects the class, and and therefore a ects only instances created after the change. Already existing instances are not changed.

`(_ self Add type name value propertyName)` [Method of Class]

Add an instance speci ed by `type` to the class. E.g. if `type` = `IV` then add an instance variable with the given name using the given value as default. If `propertyName` is given, use `value` instead as the property value on `type` created or found. The type must be one of the item types speci ed above: `IV`, `CV`, `IVProp`, `CVProp`, `Method`, `Super`, or `Meta`.

(_ self CommentMethods) [Method of Class]

For each method in the class, obtain its argument list, and insert this in the class definition under the method property args. If the source code of a method is in core, extract the comment which should be the fourth item in the source code, and insert in the class definition under the method property doc. If no comment is found in the source code, put the user into the editor looking at that function. When editing is finished, retrieve the comment from the method.

(_ self CopyMethod mySelector newClass newSelector) [Method of Class]

Copy the method associated with the selector mySelector from self to newClass (under the new selector newSelector). newSelector defaults to mySelector

(_ self DefMethod selector args expr) [Method of Class]

Adds a method for selector to class. If args and expr are NIL, puts the user into the editor)

(_ self Delete type name prop) [Method of Class]

Deletes the specified element from class. type must be one of IV, CV, IVProp, CVProp, Method, Super, or Meta.

(_ self Destroy) [Method of Class]

Destroys (deletes) a class.

(_ self Destroy!) [Method of Class]

Recursive version of Destroy. Destroys class and its subclasses.

(_ self Edit commands) [Method of Class]

Calls the Interlisp Editor on the source for class.

(_ self EditMethod selector commands) [Method of Class]

Finds the function associated with selector in class, and calls the Interlisp Editor on it.

(_ self FetchMethod selector) [Method of Class]

Returns the name of the function which implements this method in this class.

(_ self HasCV CVName prop) [Method of Class]

Tests if class has the specified class variable/property.

(_ self HasIV IVName prop) [Method of Class]

Tests if class has the specified instance variable/property.

(_ self List componentType componentName propName) [Method of Class]

List the immediate components of a class. componentType is one of the item or set specifiers described above. If componentType is one of the item specifiers, then componentName should be specified; List will show that item. If componentType is IVProps or CVProps, then List will show just the property names of the named item. Otherwise, for all set descriptors, it will list all relevant items. propName must be specified only if component is IVProps or CVProps. Selectors and Methods are synonyms, returning the list of selectors for the class; Functions returns the list of names of functions called for methods in this class.

(_ self List! type name verboseFlg) [Method of Class]
: Recursive version of List. Omits things inherited from Object and Class unless verboseFlg=T.

(_ self MethodDoc selector) [Method of Class]
: Print documentation for the method associated with selector in TTY window.

(_ self MoveMethod newClass selector) [Method of Class]
: Moves the method speci ed by selector from this class to the speci ed class, changing the name of the function if it is of form className.selector

(_ self New name supers) [Method of Class]
: New method for MetaClass. Since MetaClass is its own metaClass, this needs to work correctly whether self is Class or MetaClass or a subClass of MetaClass. Work is done by DefineClass in LOOPS.

(_ self NewTemp selector superFlg) [Method of Class]
: Make a new temporary instance of this class which will not get saved on a database unless referred to by another saved object.

(_ self OnFile |le) [Method of Class]
: Returns T if self is de ned on the le |le

(_ self PP |le) [Method of Class]
: Prettyprints the class on the le |le

(_ self PP! |le) [Method of Class]
: PrettyPrints the class at all levels.

(_ self PPM selector) [Method of Class]
: Prettyprints the function which implements selector in this class.

(_ self PPMethod selector) [Method of Class]
: Prettyprints the function which implements selector in this class.

(_ self Put type name value prop) [Method of Class]
: type must be one of IV, CV, IVProp, CVProp, Method, Super, or Meta. Adds the speci ed type to the class.

(_ self Rename newName environment) [Method of Class]
: Give a class a new name, renaming those methods of the form className.selector

(_ self ReplaceSupers supers) [Method of Class]
: Replace the entire supers list for this class.

(_ self SetName newName environment) [Method of Class]
: Change the name of the class, forgetting old name. Change the names of all methods which are of the form className.selector. Same as Rename.

(_ self SubClasses) [Method of Class]
: Returns a list of immediate subclasses currently known for this class.

14.5 The Class Named ‘‘Object’’

All classes have `Object` as one of their supers, directly or indirectly. Therefore, all instances know how to respond to the messages defined in `Object`. These can of course be overridden in any class, but `Object` provides a set of default behaviors, and generally available subroutines.

`(_ self AddIV name value prop)` [Method of Object]
Adds an IV to instance. If it is not in regular set, puts it in assoc List on otherIVs.

`(_ self AssocKB newKBName )` [Method of Object]
Change assocKB of this object to `newKBName` .

`(_ self At varName prop index)` [Method of Object]
Returns the value of an ‘‘instance variable’’ for an object. For an instance object, instance variables hold local state. For an object that is a class, we use ‘‘instance variable’’ to refer to the variables that are private to instances of the class. If the value is an active value, `GetValue` activates its `getFn`.

`(_ self BreakIt varName propName type brkOnGetAlsoFlg)` [Method of Object]
Creates an active value which will cause a break when this value is changed. If `brkOnGetAlsoFlg=T`, this will also break when the value is fetched.

`(_ self Class)` [Method of Object]
Returns the class of this object.

`(_ self ClassName)` [Method of Object]
Returns the className of the class of the object.

`(_ self CopyDeep KBC )` [Method of Object]
Copies the unit, sharing the iName list, copying instances, activeValues and lists.

`(_ self CopyShallow)` [Method of Object]
Makes a new instance (a copy of this instance, not copying the values of the instance variables), with the same contents as `self`

`(_ self DeleteIV varName propName)` [Method of Object]
Removes an IV from an instance. No longer shares IVName List with class. Some programs which depend on IV may not work.

`(_ self DeleteIVProp ivName ivProp)` [Method of Object]
Deletes a property of an instance variable.

`(_ self Destroy)` [Method of Object]
Destroy an object in an environment. Removes all IVs, class pointers, etc. For garbage collection by user.

(_ self DoMethod selector class arg_1 arg_2 arg_3 arg_4 arg_5 arg_6 arg_7 arg_8 arg_9 arg_{10}) [Method of Object]
Message form of the function `DoMethod`.

`(_ self Edit commands)` [Method of Object]
Calls the Interlisp editor on the source of the object.

(_ self HasIV ivName prop) [Method of Object]

Returns T if self contains the speci ed IV.

(_ self Inspect ASTYPE) [Method of Object]

Calls the Interlisp inspector to examine self (as an object of type ASTYPE).

(_ self InstOf className) [Method of Object]

Returns T if self is an immediate instance of the class with name className

(_ self InstOf! className) [Method of Object]

Returns T if self is an instance of the class with name className either directly or through the supers chain of its class.

(_ self IVMissing varName) [Method of Object]

Called from macro FetchIVDescr when there is no IV varName. If varName is an IV of the class, then it adds IV to the instance and returns the IVDescr as requested. Will also do this if user returns with OK from HELPCHECK.

(_ self List typeName) [Method of Object]

List IV properties, IVS of object, or other properties inherited from class.

(_ self List! type name verboseFlg) [Method of Object]

Recursive form of List for objects. Omits things inherited from Object unless verboseFlg is T.

(_ self MessageNotUnderstood selector superFlg) [Method of Object]

Invoked when a selector is not found for an object during a message sending operation. Attempts to do spelling correction on the selector. Causes an error if this fails.

(_ self NoObjectForMsg selector) [Method of Object]

Called from FetchMethodOrHelp when self is not a Loops object with a de ned class. A specialized response to this can be tailored in a given Loops application by rst reseting the global Interlisp variable DefaultObject to point to an object. This default object will eld NoObjectForMsg messages from FetchMethodOrHelp. The method for NoObjectForMsg on DefaultObject should return a default value, usually dependent on the selector.

This version of NoObjectForMsg just causes an error break. A user can return from the error by typing RETURN value, where value is the value that should have been returned as the result of sending selector to self

(_ self PP) [Method of Object]

PrettyPrints an instance de nition on |le

(_ self PP! |le) [Method of Object]

PrettyPrints an instance to all levels.

(_ self PrintOn |le) [Method of Object]

This is the default printing function for Object. It distinguishes between temporary objects, named objects, and others.

(_ self Put varName newValue propName index) [Method of Object]

Puts newValue in an instance variable (see GetValue, page 19). If the value/property of the variable contains an active value, the putFn is activated.

(_ self Rename newName environment) [Method of Object]

Removes an old name, and gives it new name.

(_ self SetName name environment noBitchFlg) [Method of Object]

Associates a name with an object in an environment. This works for instances and classes. An object can have more than one name.

(_ self TraceIt varName propName type traceGetAlsoFlg) [Method of Object]

Creates an active value which will cause tracing when this variable is changed. Will also trace on fetches if traceGetAlsoFlg = T.

(_ self UnSetName name environment) [Method of Object]

If name actually names self in environment, then delete the association between self and name.

(_ self Understands selector) [Method of Object]

Tests if self will respond to selector.

(_ self WhereIs name type propName) [Method of Object]

Searches the supers hierarchy until it nds the class from which type is inherited. type = NIL defaults to METHODS.

14.6 Functions for changing Loops Structure

14.6.1 Moving and Renaming Methods

There are a number of Interlisp functions available to help in the process of reorganizing class structures. It is often the case in the development of a set of classes for some job that one nds some common super class of a set of classes, and wants to move a method up to the super, or copy it down from the super. Also renaming both the selector and the function of a method is sometimes useful.

(RenameMethod className oldSelector newSelector) [Function]

Changes the selector oldSelector to newSelector in className and if the function name is className.oldSelector does a RENAME to className.newSelector.

(RenameMethodFunction class oldName newName) [Function]

Renames a function used as a method in class. Does not change the selector. Complains if oldName is not found.

(MoveMethod oldClassname newClassName selector) [Function]

Moves the method from oldClassname to newClassName, and renames the function if it is of the form oldClassname.selector to newClassName.selector.

(CalledFns classes de|nedFlg) [Function]

Given a list of classes, this function computes the list of all functions called by those classes. If `de|nedFlg` = `T`, only returns the list of those functions which are de ned.

14.6.2 Moving and Renaming Variables

It is sometimes convenient to be able to move methods and variables when recon guring classes in an inheritance lattice. The following functions are provided for this.:

`(RenameVariable className oldVarName newVarName classFlg)` [Function]

Changes the name of the variable from `oldVarName` to `newVarName`. Changes any references to these variables in methods of the class.

`(MoveVariable oldClassName newClassname variableName)` [Function]

Moves the entire description of an instance variable into the new class.

`(MoveClassVariable oldClassName newClassname variableName)` [Function]

Moves the entire description of a class variable into the new class.

15 LOOPS AND THE INTERLISP SYSTEM

15.1 Saving Class and Instance De nitions on Files

Loops has been integrated with the Interlisp le system to allow saving of class de nitions on les. The le command:

```
(CLASSES * classNameList)
```

added to the lecoms of any le will allow one to dump out the prettyprinted version of the source you see when you edit the class de nition. These class names can be listed in any order in a single list, provided that all super classes of a class on the list are on the list as well, or will be previously de ned.

```
(INSTANCES * instanceNameList)
```

added to the lecoms of any le will allow one to dump out the prettyprinted versions of named instances, as well as any unnamed instances that they point to.

Functions used to implement methods are ordinary Interlisp functions. Those that are named automatically by Loops as `className.selector` start with the same characters; they will be found alphabetically together on any function list which is created. The function `CalledFns` (page 120) can be used get a list of all functions used by a list of classes.

15.2 Classes for Lisp Datatypes

One can use the message sending protocol with records (lists) whose rst element is a class, or ordinary Interlisp datatypes. In the rst case, the rst element is used as the class to look up the method to be used. In the second case, the class is found using the function `(GetLispClass obj)`, which looks it up in the hash table `LispClassTable`, based on the type name of the datatype.

We call datatypes with associated classes and records with rst element a class *pseudoclasses*, and instances of them *pseudoinstances*. If `GetValue` or `PutValue` are called with `self` bound to a pseudoinstance, then the method associated with the selector `GetValue` in the pseudoclass (call it `PC`) is called as follows:

```
(APPLY* (GetMethod PC 'GetValue) instance varName propName)
```

or

```
(APPLY* (GetMethod PC 'PutValue) instance varName newValue propName)
```

If the associated class `PC` has a `GetValue` (`PutValue`) method, then values of the variables can be found. This allows a mixture of compiled access to datatype elds, and interpreted access within Loops.

15.3 Some Details of the Loops implementation

Methods are implemented by Lisp functions. The message sending expression:

(_ object selector arg_1 arg_N)

is expanded as a compiler MACRO into

(APPLY* (FetchMethodOrHelp object 'selector) object arg_1 arg_N)

GetMethod returns the name of the Interlisp function associated with selector anywhere in the class of object, or in the superClass chain of that class. Notice that the object is implicitly included as the first argument of the function, as well as being the argument for GetMethod. By syntactic convention the first argument (bound to the object) in any function which is being used as a method is called self. The expression for the object is evaluated only once.

Objects in Loops are represented in memory as Interlisp datatypes. The datatypes for classes have property lists for methods, class variables, instance variables, and their properties. Datatypes for instances have property lists for instance variables and their properties. In general, the selector names and variable names are stored in the class objects. When instances are read in from a data base, they have their local name tables aligned with the class standards. Special provisions are provided for handling instances whose variable names do not correspond to current class definitions. Instances act as if they have local tables for lookup of variables and properties, but they usually share the class name table and no storage is actually allocated for local tables unless it is needed.

Default values for instance variables and properties are not copied to an instance. No space for instance variables or properties is allocated until that variable or property has been set individually for the instance. This means that the default values are not just initial values. In particular, if a class is altered to change the default value of an instance variable, then all of the instances that do not have individualized values will reflect the new default value. Also, there is no storage overhead in instances for unchanged properties (e.g., for documentation) defined in classes. Since individualized values of variables are stored in the instances, there is no need to search the class hierarachy after a variable or property has been set in the instance. In contrast, since class variables are shared among instances it is always necessary to go to the class (or a super class) to get a value.

Although many of the ideas of the Loops database were inspired by PIE, the implementation differs along several dimensions. PIE was intended primarily for use with a browser (i.e., an interactive viewing and editing program), and efficiency was not a primary concern. Since Loops was intended for use by programs with potentially extensive computational processes, a need for efficient access was perceived and this led to some different tradeoffs in the choice of implementation.

One difference between PIE and Loops is the grainsize of the changes written in layers. PIE performs separate bookkeeping on changes to values of every variable in objects. Loops avoids the storage penalty of this by keeping track only of which objects have been changed. This means that file layers in PIE contain partial objects (e.g., a change to a single variable) while layers in Loops contain complete objects. In effect, Loops economizes on space (and time) in memory instead of space in the databases.

Another difference is that the Loops implementation tries to reduce the cost of references to values by snapping links to references. However, link snapping is fundamentally in conflict with a lookup process that takes an environment as an argument. Link snapping precludes the sharing of objects between environments in those cases where the interpretation of the references in the shared objects is sensitive to the environment. Loops preserves a complete isolation of environments, with exchange of information permitted only as a knowledge base transaction. In general, realigning an environment to incorporate changes from another environment requires writing out the changes, clearing the memory in the environments, and re-opening the associated knowledge bases. In contrast, PIE always shared information between contexts, but it paid the overhead of reinterpreting the symbolic addresses repeatedly at every reference.