```
    . . .
    (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. figure 23 summarizes the compiler options for breaking and tracing:

T	Trace if rule is satisfied. Useful for creating a running display of executed rules.
TT	Trace if rule is tested.
B	Break if rule is satisfied.
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 satisfied. 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 specification 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 specifications 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 satisfied and before the RHS is executed. The rule-specific 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 effected 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 satisfied, 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 first 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 affect 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-specific 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 filed 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 files: 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 first four packages must be included in any loadup of Loops; the second are ones we find 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 notifications 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 fills 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 first 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 definition.
`PP!`	PrettyPrint Class definition including inherited information.
`PPV!`	Same as `PP!` without seeing methods.
`PPM`	Puts up a pop-up menu of all of the methods defined in the class, and prettyprints the definition 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 defined 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 find 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 flashed, and its nameis printed in the Prompt Window. This final 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 buffer. 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 defined 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	Define 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 definition 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 first 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 specified 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 specified 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` finds 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 *obj1* selected in the browser. By buttoning PP (or PP!) in the action menu, *obj1* would 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 *obj1* might 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 *object objName*)
(← ($ Lb) MiddleShiftSelect *object objName*)

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 affect the underlying structure, just the display.

Format of the Browser Window — One can obtain a browser display with a specified title or in an existing window. If one specifies *windowOrTitle* in

(← ($ Lb) Show *browseList windowOrTitle*)

then if *windowOrTitle* is a string, it will be used as the title of a new window for the browser. If *windowOrTitle* is 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, define 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 flashTime*) [Method of LatticeBrowser]
Call `FlipNode` 2*N* times, delaying for *flashTime* milliseconds between flips. Default values: *N*=3, *flashTime*=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 buffer

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 benefits 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 specifies 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 definition 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.

(← *object* Edit)

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 identifier. When this is read back in from the string editing buffer of TTYIN, a readmacro for # converts it back into a pointer to an instance with that unique identifier. 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 definition for a method, and then finish making the specifications 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 specified. The name of the actual function is printed out as you enter the editing process. Aside from the syntactic convention of having the first 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 filler 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-specific 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-specific 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*=`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 fielded 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 different 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* as 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 filled 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 defined 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 specifies 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 affects the class, and and therefore affects only instances created after the change. Already existing instances are not changed.

(← *self* Add *type name value propertyName*) [Method of Class]

Add an instance specified 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 specified 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 exp*) [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 specified by *selector* from this class to the specified 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 *file*) [Method of Class]
Returns T if *self* is defined on the file *file*.

(← *self* PP *file*) [Method of Class]
Prettyprints the class on the file *file*.

(← *self* PP! *file*) [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 specified 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 specified 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 defined class. A specialized response to this can be tailored in a given Loops application by first reseting the global Interlisp variable DefaultObject to point to an object. This default object will field 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 definition on *file*.

(← *self* PP! *file*) [Method of Object]

PrettyPrints an instance to all levels.

(← *self* PrintOn *file*) [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 finds 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 finds 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 definedFlg*) [Function]
Given a list of classes, this function computes the list of all functions called by those

classes. If *definedFlg*=T, only returns the list of those functions which are defined.

14.6.2 Moving and Renaming Variables

It is sometimes convenient to be able to move methods and variables when reconfiguring 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 Definitions on Files

Loops has been integrated with the Interlisp file system to allow saving of class definitions on files. The file command:

`(CLASSES * classNameList)`

added to the filecoms of any file will allow one to dump out the prettyprinted version of the source you see when you edit the class definition. 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 defined.

`(INSTANCES * instanceNameList)`

added to the filecoms of any file 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 first element is a class, or ordinary Interlisp datatypes. In the first case, the first 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 first 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 fields, 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* $arg_1 \cdots arg_N$)

is expanded as a compiler MACRO into

(APPLY* (FetchMethodOrHelp *object* '*selector*) *object* $arg_1 \cdots arg_N$)

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.

INDEX

Filed on: {Indigo}<Loops>LoopsCourse>LoopsCourseSummary.bravo
Last edited: GB June 14, 1983 6:47 PM

LOOPS Course Summary

Accessing Objects and Variables

($ *name*)	evaluates to the object or class named *name*.
($! *atom*)	evaluates to the object or class whose name is the value of *atom*
(@ *accessExpr*)	evaluates to the value of the instance variable, class variable, or property of these referred to by *accessExpr* in self.
(@ *obj accessExpr*)	evaluates to the value of the instance variable, class variable, or property of these referred to by *accessExpr* in *obj*
(←@ *accessExpr newValue*)	sets the value of the variable accessed by *accessExpr* in self to *newValue*
(←@ *obj accessExpr newValue*)	sets the value of the variable accessed by *accessExpr* in *obj* to *newValue*

accessExpr is the concatenation of any combination of the following with evaluation strictly left to right

:*ivName*	instance variable *ivName*
::*cvName*	class variable *cvName*
:,*propName*	value of property *propName*
.*selector*	value returned by sending the unary message *selector*

N.B. a ! (bang) after any of the puctuation in the four lines above will cause the atom following it to be evaluated and that value to be used as the name. Within an *accessExpr* a lisp variable is prefixed with a backslash "\"
(i.e. **::fee.fie:!\foe:,fum** will get the value of CV **fee** of self and send it the message **fie**, then it will get the instance variable whose name is the value of the lisp variable **foe** from the object returned by the message fie, then it will get the value of property **fum** of that IV)

Defining and Editing Classes

(DC *className supersList*)	(← ($ Class) New *className supersList*)	create a class with name *className* and supers *supersList*
(EC *className*)	(← ($ *className*) Edit)	edit the class definition of class *className*

Defining and Editing Methods

(DM *className selector*)	creates a function with the name *className.selector* to be used by the method called by *selector* and puts you in the editor
(DM *className selector fnName*)	causes the function with the name *fnName* to be used by the method called by *selector*
(EM *className selector*)	edit the method used by *selector* in class *className*

Creating, Editing, and Inspecting Instances

(← *class* New)		creates a new instance of *class*
(← *class* New *'name*)		creates a new instance of *class* with the name *name*
(← *obj* Edit)	(EI *obj*)	edit *obj*
(← *obj* Inspect)	(INSPECT *obj*)	create an inspect window for *obj*
(←New *class selector arg1 ... argN*)		create a new instance of *class* and sends it the the message *selector* with arguments *arg1 ... argN*

Sending Messages

(← *obj selector arg1 ... argN*)	send *obj* the message *selector* with arguments *arg1 ... argN*
(←Super *obj selector arg1 ... argN*)	in method *selector* invokes super method for that *selector* with arguments *arg1 ... argN*
(←SuperFringe *obj selector arg1 ... argN*)	invokes all the immediate super methods of *obj* for that *selector* with the arguments *arg1 ... argN*
(←! *obj expr arg1 ... argN*)	send *obj* the message whose selector is the value of *expr* with the arguments *arg1 ... argN*

Active Values

#(*localState getFn putFn*)	*localState* is where the value is stored (this may be another active value) *getFn* is the function called on read access and *putFn* is called on write access the value returned by *getFn* in the value of the get operation and *putFn* has responsibility for changing the value of *localState* using the function PutLocalState

Debugging

(BreakIt *obj varName*)	break whenever the instance variable *varName* of *obj* is accessed
(UnBreakIt *obj varName*)	remove the break on variable *varName* of *obj*
(BreakMethod *className selector*)	break whenever the method *selector* is used by any instance of class *className*
(TraceMethod *className selector*)	trace whenever the method *selector* is used by any instance of class *className*
(UNBREAK *onlyMostRecentFlg*)	standard Lisp function to unbreak or untrace methods
(BreakIt *obj varName propName type breakOnGetAlsoFlg*)	break whenever the variable *varName* of *obj* is accessed
(TraceIt *obj varName propName type breakOnGetAlsoFlg*)	trace whenever the variable *varName* of *obj* is accessed
(UnBreakIt *obj varName propName type*)	remove the break on variable *varName* of *obj*

To attach a gauge and monitor a variable:

(←New ($ *gaugeType*) Attach *obj ivName selector*)	attaches a gauge of type *gaugeType* to the instance variable *ivName* of *obj*

Rules

↑F gets you into the Rule Executive
(OK gets you out of it and UE puts you in the User Executive (where OK will get you back again))
Variables are accesed by using the access expressions as defined above

accessExpr	gets value of variable (do not use @)
accessExpr← *newValue*	variable accessed gets *newValue*
\lispVarName	for referring to lisp variables use backslash
.selector	sends unary message to self (unary message is one that requires no arguments besides self)

(DefRSM *className selector*)	creates a new rule set for the class *className* invoked by *selector* and places you in the rule editor
(← *ruleSet* CopyRules *'newRuleSetName*)	copies the ruleset *ruleSet* into a new one called *newRuleSetName*
(← *ruleSet* ER) ER(*ruleSet*)	edit *ruleSet*
(ListRuleSets *className*)	generates a listing of all the rule sets defined for the class *className*

Browsers

(Browse *classList*)	creates a browser window for the class lattice structure of the classes in *classList* and their descendants left or middle button in title area of the browser window updates the lattice structure
Left Mouse Button	gets pop-up menu to print information about class structure and methods
Middle Mouse Button	gets pop-up menu to aid in generating new classes or methods

An asterisk at the end of the name of any item in the menu signifies that there are multiple options for this item

To use the default option, click the left button, for a menu of options click the middle button (i.e. EM* will get a menu with EM and EM!)

To copy from class to class use the left button to "BoxNode" of recepient class then with the middle button menu select the "Move" item with the middle button to get a menu for either copying of moving of IVs, CVs, Methods, or RuleSets

"Specialize" on the middle button menu will create a new subclass of the one selected and ask for a name in the prompt window

"DefineMethod" on the middle button menu will create a new method for that class and prompt for its selector

Saving and Restoring Files

(FILES?)	Lisp will ask you to assign a *filename* to each entity it does not already have a file name for Type yes to specify the file names. For each entity type the *filename* to save it or] to not have it saved LineFeed (LF) means the same as the previous entity
(MAKEFILE *filename*)	saves the file on the file server on the directory currently connected
(LOAD *filename*)	loads the file from the file server on the directory currently connected