Venue

LOOPS Library Modules Manual

November, 1991
Medley Release

Address comments to:
Venue
User Documentation
1549 Industrial Road
San Carlos, CA 94070
415-508-9672

LOOPS LIBRARY MODULES MANUAL

November, 1991

Text was written and produced with Venue text formatting tools; Xerox printers were used to produce text masters. The typeface is Classic.

TABLE OF CONTENTS

[This page intentionally left blank]

PREFACE

Overview of the Manual

The *LOOPS Library Modules Manual* describes the Library Modules for Venue's Lisp Object-Oriented Programming System, LOOPS. These Library Modules, which can be loaded into Medley, provide additional functionality to LOOPS.

This manual describes the current release of the LOOPS Library Modules, which run under Medley.

Organization of the Manual and How to Use It

This manual is divided into chapters, with each chapter focussing on a particular Library Module. A Table of Contents is included to help you find specific material.

Conventions

This manual uses the following conventions:

- Case is significant in LOOPS and Medley. All selectors, methods, arguments, etc., must be typed as shown. Typically, this means that method names are capitalized and variables are not.
- You need to use an Interlisp Exec to enter all exec expressions.
- Arguments appear in italic type.
- Selectors, methods, functions, objects, classes, and instances appear in bold type.

  For example, a method appears as follows:

  (_ *self* **Selector** *Arg1 Arg2*)
- Examples are shown in the Interlisp Exec and appear in the following typeface:

  ```
  89←(←LOGIN)
  ```
- All examples are typed into an Interlisp Exec. This is the recommended Exec for all LOOPS expressions.
- Methods with an exclamation mark (!) suffix usually perform operations deeply into class structure instead of only on a given object.
- Methods with a question mark (?) suffix usually are predicates; that is, truth functions.
- Methods often appear in the form **ClassName.SelectorName**.

- Cautions describe possible dangers to hardware or software.

- Notes describe related text.

This manual describes the LOOPS items (functions, methods, etc.) by using the following template:

Purpose: Gives a short statement of what the item does.

Behavior: Provides the details of how the item operates.

Arguments: Describes each argument in the following format:

argument Description

Returns: States what the item returns, and does not appear if the item does not return a value. The phrase "Used as a side effect only." means that the purpose of the item is to perform a computation or action that is independent of any returned value, not to return a particular value.

Categories: A way to group related methods. For example, all the methods related to Masterscope on the class **FileBrowser** have the category Masterscope, not **FileBrowser**. This item appears only for methods.

Specializes: The next higher class in the class hierarchy that contains a method with the same selector. For example, **RectangularWindow.Open** can specialize **Window.Open**. This appears only for methods.

Specializations: The next lower class in the class hierarchy that contains a method with the same selector. For example, **Window.Open** is a specialization of **RectangularWindow.Open**. This appears only for methods.

Example: An example is often included to show how to use the item and what result it produces. Some examples may appear differently on your system, depending on the settings of various print flags. See the *LOOPS Reference Manual* for details.

References

The following books and manuals augment this manual.

LOOPS Reference Manual

LOOPS Release Notes

LOOPS Users' Modules Manual

Interlisp-D Reference Manual

Common Lisp: the Language by Guy Steele

Common Lisp Implementation Notes

Lisp Release Notess

Lisp Library Modules Manual

Description/Introduction

Gauges are an important part of the LOOPS user interface for both developers and end users. Gauges assist in understanding the dynamic nature of the programs. This is in contrast to the more typical case of debugging programs using static means. In the creation of user-friendly interfaces, you can use gauges to display, in analog or digital form, various data that may be changing. Also, by employing active gauges, you can provide a convenient way to interact with a system.

One of the features of gauges is the ease with which you can use them in a system. In more traditional languages, if you want to understand how a variable is changing over the course of a computation, you must make modifications in your program wherever you want to begin or end the examination of a variable. Given the capabilities of active values used by gauges, you need only attach or detach a gauge to the data that you are interested in monitoring.

The following types of gauges are available:

- Meter; a circular instrument that wraps around any number of times.
- Dial; a bounded dial, like an automobile speedometer.
- LCD; a gauge that uses the entire window to display a value.
- Scale; a horizontal or vertical display of a gauge.
- ActiveScale; a scale that allows you to change the gauge value.

Gauges are an example of the combination of programming capabilities within LOOPS. The different types of gauges are defined within the context of an inheritance lattice. This allows the more general functionality and variables to be allocated to more general gauge classes, with specific functionality placed in more restricted classes. You can also see the use of mixins to add a small amount of functionality to several different classes of gauges.

Note: Mixins are classes that are used only in conjunction with another class to create a subclass.

The methods within gauges are built upon both function calling and message sending. Gauges are "attached" to objects through the mechanism of active values. Since gauges are built upon the mechanism of active values, gauges can only be attached to data within objects. It is not possible to use gauges to monitor any arbitrary Lisp variable.

Prerequisites

The default font for gauges is Modern 10.

Installation/Loading Instructions

Gauges are divided among several different files to allow you to load only those objects and functions that you need. The table below lists the files to load for each type of gauge. The filecoms for each file will try to load any other required gauge files from **LOOPSLIBRARYDIRECTORY**. The file GAUGES.DFASL and either GAUGEINSTRUMENTS.DFASL or GAUGEALPHANUMERICS.DFASL will always be loaded; other files may also be loaded.

Gauge	File to load
LCD	GAUGEALPHANUMERICS.DFASL
METER	GAUGEMETERS.DFASL
DIAL	GAUGEDIALS.DFASL
SCALE	GAUGESCALES.DFASL
ACTIVE SCALE	GAUGEACTIVE.DFASL

Additionally, the file GAUGESELFSCALEMIXIN.DFASL can be loaded to add the class **SelfScaleMixin**, and GAUGEALARMS.DFASL can be loaded to add the class **AlarmMixin**.

To load the required files, first set the value of **LOOPSDIRECTORY** to include the directory where the gauges files are stored, then type the following expression in the Executive:

(LOAD '*FILENAME*)

To load all of the gauges, load the file GAUGELOADER and then enter `(LOADGAUGES)`. GAUGELOADER also sets the variables: **GAUGEFILES** and **GaugeClasses**.

(**LOADGAUGES** *LDFLG SOURCES?FLG*) [Function]

Purpose: Loads all the gauges.

Behavior: Assumes that all of the gauge files are on the **LOOPSDIRECTORY** search path.

All the gauge files will be loaded based upon the settings of *LDFLG* and *SOURCES?FLG*. A **FILESLOAD** expression is built up and evaluated.

Arguments: *LDFLG* Can be NIL, PROP, or SYSLOAD. See the **LDFLG** discussion under loading in the *Interlisp-D Reference Manual*.

SOURCES?FLG
Can be NIL or T. If NIL, this attempts to load the compiled files before trying to load the sources. If T, only the sources are loaded.

Returns: Used for side effect only.

GAUGEFILES [Variable]

Behavior: Initialized to (GAUGEACTIVE GAUGEALARMS GAUGEALPHANUMERICS GAUGEBOUNDEDMIXIN GAUGEDIALS GAUGEDIGIMETER GAUGEDIGISCALE GAUGEINSTRUMENTS GAUGEMETERS GAUGES GAUGESCALES GAUGESELFSCALEMIXIN)

GaugeClasses [Variable]

Behavior: Initialized to (GaugeAV ActiveGaugeMixin Gauge AlarmMixin BoundedMixin SelfScaleMixin)

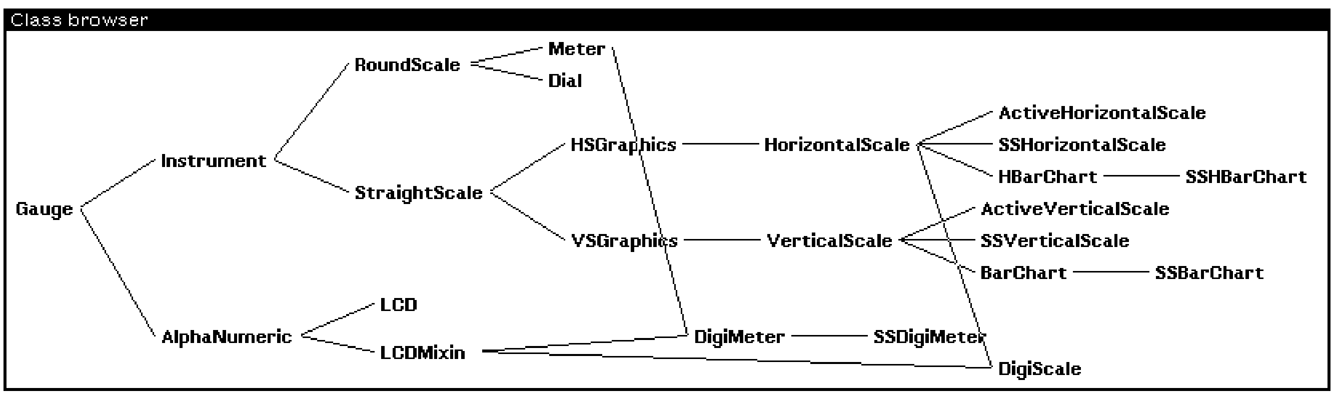
Call (Browse **GaugeClasses**) to open a browser of all of the gauge classes.

Application/Module Functionality

This section describes the gauge classes and methods.

Gauge Classes

This section describes the available gauges shown in the following browser.



Note: The browser does not include the optional mixin classes.

Within the class description of each class, the instance variables and class variables that are specializations only because they have different default values are not listed.

Name	Type	Description
ActiveGaugeMixin	AbstractClass	A gauge class that allows you to set the value of the variable being monitored with the cursor, via a SET menu.
ActiveHorizontalScale	Class	An active gauge that displays the value on a horizontal scale.
ActiveVerticalScale	Class	An active gauge that displays its value on a vertical scale.
AlarmMixin	AbstractClass	A mixin that adds alarm functionality to any gauge.
AlphaNumeric	AbstractClass	A gauge that gives an alphanumeric display of a value.

BarChart	Class	A gauge that displays more than one **VerticalScale** side-by side.
BoundedMixin	AbstractClass	Creates a bounded scale for **displayVal**; to be used as a mixin for instruments.
Dial	Class	A bounded dial, like an automobile speedometer.
DigiMeter	Class	A gauge that displays both an **LCD** and a meter.
DigiScale	Class	A gauge that displays both an **LCD** and a horizontal scale.
Gauge	AbstractClass	A class for objects that present a dynamic graphical image of a LOOPS value.
GaugeAV	Class	An active value associated with a gauge.
HBarChart	Class	A gauge that displays more than one **HorizontalScale** side-by side.
HorizontalScale	Class	A labeled, bounded scale with a bar that fills to the right.
HSGraphics	AbstractClass	Gauge that is displayed in the form of a singe horizontal scale or bar.
Instrument	AbstractClass	A numeric gauge that is externally scaled by **inputLower** and **inputRange** and scaled internally by **lower** and **range**.
LCD	Class	Differs from **AlphaNumeric** in that the entire gauge window is the printing region.
LCDMixin	AbstractClass	Computes print region differently from **LCD**.
Meter	Class	A circular instrument that wraps around any number of times.
RoundScale	AbstractClass	Abstract Class for instruments with circular (arc) scales.
SelfScaleMixin	AbstractClass	Provides for the gauge to rescale according to the reading.
SSBarChart	Class	A self-scaling version of **BarChart.**
SSDigiMeter	Class	A self-scaling version of **DigiMeter**.
SSHBarChart	Class	A self-scaling version of **HBarChart.**
SSHorizontalScale	Class	Gauge that is displayed in the form of a single scale or bar which rescales itself accordingly.
SSVerticalScale	Class	Gauge that is displayed in the form of a single vertical scale or bar which rescales itself accordingly.
StraightScale	AbstractClass	Abstract Class for instruments with straight scales.
VSGraphics	AbstractClass	Gauge that is displayed in the form of a single vertical scale or bar.
VerticalScale	Class	Gauge that is displayed in the form of a single vertical scale or bar.

ActiveGaugeMixin [Class]

Description: A gauge class that allows you to set the value of the variable being monitored with the cursor, via a **SET** menu.

MetaClass: AbstractClass

Supers: Object

Class Variables: None.

Instance Variables: **cursor** The cursor to use when changing the scale; the default is NIL.

ActiveHorizontalScale [Class]

Description: An active gauge that displays the value on a horizontal scale. This gauge shows the value of the data it is connected with and allows you to change that data with the gauge.

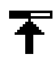
MetaClass: Class

Supers: ActiveGaugeMixin, HorizontalScale

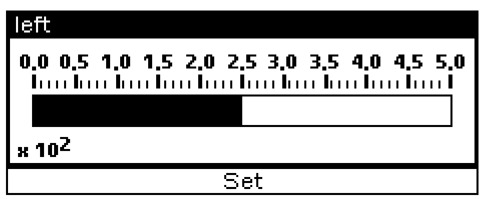
Class Variables: None.

Instance Variables: **cursor** Cursor to use when changing the scale; its property **:initform** is set to **HorizontalAGCursor**.

Example: These gauges have an attached menu at the bottom of the gauge. When you position the cursor over this menu and press a mouse button, the cursor changes to the following shape:



While the left button is held down, the system tracks movements of the cursor and changes the value that the gauge is monitoring.



ActiveVerticalScale [Class]

Description: Similar to **ActiveHorizontalScale**, except that a vertical scale is used.

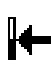
MetaClass: Class

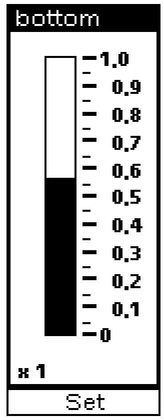
Supers: ActiveGaugeMixin, VerticalScale

Class Variables: None.

Instance Variables: **cursor** Cursor to use when changing the scale; its **:initform** property is set to **VerticalAGCursor**.

Example: Similar to **ActiveHorizontalScale**. When setting, the cursor changes to the following shape:





AlarmMixin [Class]

Description: A mixin that adds alarm functionality to any gauge. An alarm is defined as warning object that is set off when the value being monitored falls outside of the specified range. The gauge flashes and stays inverted when the alarm is tripped.

CAUTION

When a new class of gauges is created that will use the properties of **AlarmMixin**, **AlarmMixin** should be the first class on the Supers list of the new class. This guarantees that the **AlarmMixin.Set** method is invoked.

MetaClass: AbstractClass

Supers: Object

Class Variables: MiddleButtonItems

Instance Variables: **lowTripPoint**
Alarm is triggered when reading goes below this point.

hiTripPoint Alarm is triggered when reading goes above this point.

flashNumber
Number of times alarm will flash when it is tripped.

flashInverval
Interval in milliseconds between flashes.

AlphaNumeric [Class]

Description: This class contains some of the methods and data for the LCD classes. These gauges can display any type of character, letters, or numbers.

MetaClass: AbstractClass

Supers: Gauge

Class Variables: None.

Instance Variables: **precision** Number of characters displayed in the reading. The default value is 5.

BarChart [Class]

Description: A gauge that can display more than one **VerticalScale** at once, side-by side.

MetaClass: Class

Supers: VerticalScale

Class Variables: None.

Instance Variables: **maxLabelWidth**

Maximum width of labels on each bar. Default value is 0 which means no limit.

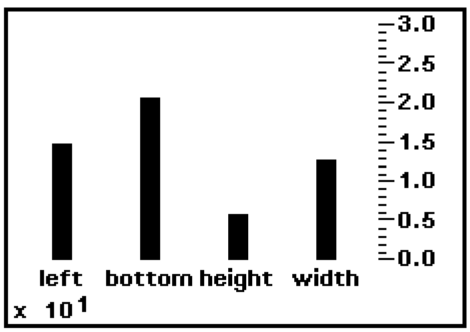
scaleLeft

Offset within the gauge window from the left for the leftmost bar. Default value is 3.

scaleBottom

Offset within the gauge window from the bottom for all the bars. Default value is 30.

Example: Here is a **BarChart** showing the size and shape of a window. It is displaying the values 15, 21, 13, and 6.



BoundedMixin [Class]

Description: This mixin is a super of the scale classes and **Dial**. If a gauge that has **BoundedMixin** as a super class tries to display a new setting that is outside of the range of the gauge, the gauge will display the minimum or maximum value as appropriate and place a "??" in the window.

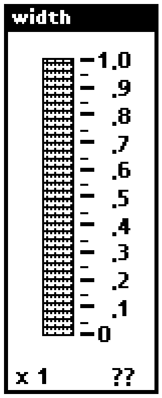
MetaClass: AbstractClass

Supers: Object

Class Variables: None.

Instance Variables: None.

Example: Here is a vertical scale that displays a reading greater than its maximum.



Dial [Class]

Description: A bounded dial, like an automobile speedometer.

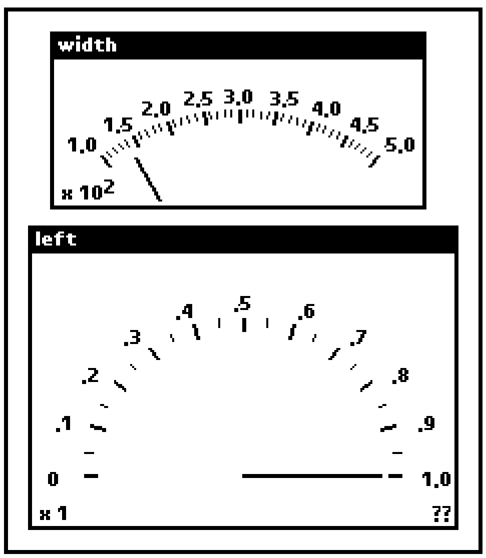
MetaClass: Class

Supers: BoundedMixin, RoundScale

Class Variables: None.

Instance Variables: This class specializes the same instance variables as **RoundScale**.

Example: The angle of the arc changes with the shape of the window.



DigiMeter [Class]

Description: A gauge that combines both a meter and an LCD.

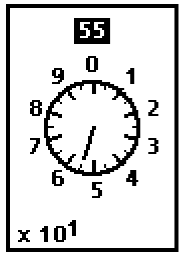
MetaClass: Class

Supers: Meter, LCDMixin

Class Variables: None.

Instance Variables: **spaceForLCD**
Vertical space required by LCD within the gauge. Defaults to 30.

Example: This **DigiMeter** is displaying 55.



DigiScale [Class]

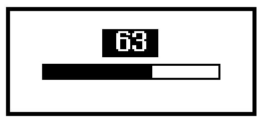
Description: A gauge that combines both a horizontal scale having no ticks and an LCD.

MetaClass: Class

Supers: HorizontalScale, LCDMixin

Class Variables: None.

Example: This **DigiScale** is displaying 63 with its scale set from 0 to 100.



Gauge [Class]

Description: A class for objects that present a dynamic graphical image of a LOOPS value. This class provides most of the methods for using gauges.

MetaClass: AbstractClass

Supers: Window

Class Variables: **LeftButtonItems**
Menu options associated with the left mouse button.

MiddleButtonItems
Menu options associated with the middle mouse button.

Instance Variables: **reading** External value of reading. The default value is 0.

containedInAV
Active value that connects the gauge to the data it is monitoring. It should be an instance of the class **GaugeAV**.

font Font that is used by a gauge; default value is (Modern 10).

width Width of a gauge; has property **min,** which specifies the minimum width for a gauge.

height Height of a gauge; has property **min**, which specifies the minimum height for a gauge.

GaugeAV [Class]

Description: An active value that is associated with a gauge.

MetaClass: Class

Supers: LocalStateActiveValue

Class Variables: None.

Instance Variables:

gauge The gauge connected to this active value.

object The object containing the variable associated with the active value.

propName The property name of the associated variable.

type Data type of the associated variable.

varName Name of the associated variable.

HBarChart [Class]

Description: A gauge that can display more than one **HorizontalScale** at once, side-by side.

MetaClass: Class

Supers: HorizontalScale

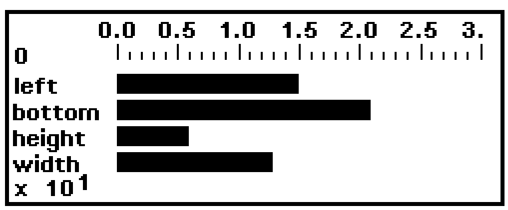
Class Variables: None.

Instance Variables:

maxLabelWidth
Maximum width of labels on each bar. Default value is 0 which means no limit.

scaleLeft
Offset within the gauge window from the left for the leftmost bar. Default value is 3.

Example: Here is an **HBarChart** showing the size and shape of a window. It is displaying the values 15, 21, 13, and 6.



HorizontalScale [Class]

Description: A labeled, bounded scale with a bar that fills to the right.

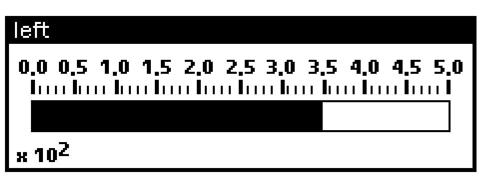
MetaClass: Class

Supers: HSGraphics

Class Variables: None.

Instance Variables: None.

Example: This **HorizontalScale** is reading 350 on a scale from 0 to 500.



HSGraphics [Class]

Description: This class provides some of the methods for displaying the graphics of a horizontal scale.

MetaClass: AbstractClass

Supers: StraightScale

Class Variables: None.

Instance Variables:

scaleBottom Bottom edge of scale in pixels. The default value is 10.

scaleLeft Left edge of scale in pixels. The default value is 12.

scaleWidth Width of inside of scale in pixels. The default value is 120.

scaleHeight Height of scale in pixels. The default value is 15.

Instrument [Class]

Description: A class that provides additional methods and data for gauges that display only numerical data. This data is externally scaled by **inputLower** and **inputRange**, and scaled internally by **lower** and **range**.

MetaClass: AbstractClass

Supers: Gauge

Class Variables: None.

Instance Variables:

ticks Scale marks on the instrument; value is a number or NIL; **smallTicks** property indicates the number of smaller ticks between each large tick.

displayVal Internal value relative to instrument.

range Range for internal **displayVal** .

inputRange Range for external reading.

lower Lower bound for internal **displayVal**.

inputLower Lower bound for external reading.

brushWidth Scale factor for width of ticks, rays, and circles in pixels.

labels The labels that will be displayed on the gauge.

labelScale A dotted pair representing the sign and exponent of a reading.

spaceForLabelScale Extra vertical space to display scale label.

LCD [Class]

Description: Differs from **LCDMixin** in that the entire gauge window is the printing region.

MetaClass: Class

Supers: AlphaNumeric

Class Variables: None.

Instance Variables: None.

Example: This **LCD** is displaying the string "Mumble", and has been **Shape**dto 120 x 60.

LCDMixin [Class]

Description: Computes printing region differently from LCD so that an LCD may be added into another window.

MetaClass: AbstractClass

Supers: AlphaNumeric

Class Variables: None.

Instance Variables: **precision** Number of characters displayed in the reading; the default value is 3. Its property is **readingRegion**; the default value is NIL.

readingY Y position of bottom of reading. The default value is 7.

Meter [Class]

Description: A circular instrument that wraps around any number of times. It displays a sign and exponent in the lower left corner of its window.

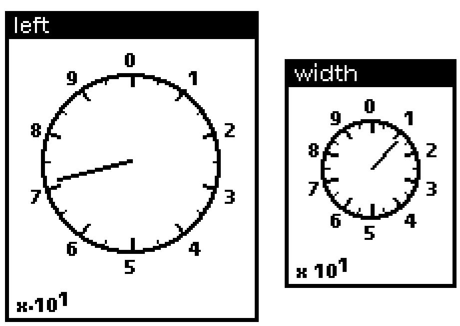
MetaClass: Class

Supers: RoundScale

Class Variables: None.

Instance Variables: This class specializes the same instance variables as **RoundScale**.

Example: The **Meter** on the left is displaying a negative value.



RoundScale [Class]

Description: Abstract Class for instruments with circular (arc) scales.

MetaClass: AbstractClass

Supers: Instrument

Class Variables: None.

Instance Variables: **needleLength**
Radius of needle in pixels. The default value is 15.

radius	Radius of arc in pixels. The default value is 10.
xc	x-coordinate window coordinate of center of arc. (See **DRAWARC** in the *Lisp Release Notes*.)
yc	y-coordinate window coordinate of center of arc. (See **DRAWARC** in the *Lisp Release Notes*.)

SelfScaleMixin [Class]

Description: Provides for the gauge to rescale according to the reading.

MetaClass: AbstractClass

Supers: Object

Class Variables: None.

Instance Variables: **lowScaleFactor**
Rescales if reading shrinks so that it will fit more than **lowScaleFactor** times in **inputRange**. The default value is 5.

SSBarChart [Class]

Description: A self-scaling version of **BarChart.**

MetaClass: Class

Supers: BarChart

Class Variables: None.

Instance Variables: None.

SSDigiMeter [Class]

Description: A self-scaling version of **DigiMeter**.

MetaClass: Class

Supers: DigiMeter

Class Variables: None.

Instance Variables: None.

SSHBarChart [Class]

Description: A self-scaling version of **HBarChart.**

MetaClass: Class

Supers: HBarChart

Class Variables: None.

Instance Variables: None.

SSHorizontalScale [Class]

Description: Gauge that is displayed in the form of a single horizontal scale or bar which rescales itself accordingly.

MetaClass: Class

Supers: VerticalScale

Class Variables: None.

Instance Variables: None.

SSVerticalScale [Class]

Description: Gauge that is displayed in the form of a single vertical scale or bar which rescales itself accordingly.

MetaClass: Class

Supers: HorizontalScale

Class Variables: None.

Instance Variables: None.

StraightScale [Class]

Description: Abstract class for instruments with straight scales.

MetaClass: AbstractClass

Supers: BoundedMixin, Instrument

Class Variables: None.

Instance Variables: **shade** Shade of bar; numeric value from 0 to 65535. The default value is 65535, which is BLACKSHADE.

VerticalScale [Class]

Description: Gauge that is displayed in the form of a singe vertical scale or bar.

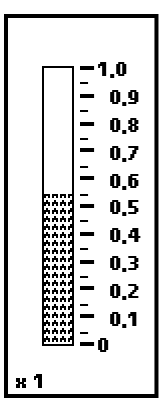
MetaClass: Class

Supers: VSGraphics

Class Variables: None.

Instance Variables: None.

Example: This **VerticalScale** is displaying the value .55 and has its **Shade** set to 1258.



VSGraphics [Class]

Description:	Similar to **HSGraphics** but for vertical scales.
MetaClass:	AbstractClass
Supers:	StraightScale
Class Variables:	None.
Instance Variables:	**scaleBottom** Bottom edge of scale in pixels. The default value is 12.
	scaleLeft Left edge of scale in pixels. The default value is 15.
	scaleWidth Width of inside of scale in pixels. The default value is 15.
	scaleHeight Height of scale in pixels. The default value is120.

Gauge Methods

This section describes the available methods and functions which are used to manipulate gauges. In many cases, a particular gauge class specializes a method defined in the class **Gauge**. In this case, the specialized method definition is not explicitly defined; instead, this is noted in the Specializes/Specializations field of the description.

Name	Type	Description
Attach	Method	Connects a gauge to an object.
Attached?	Method	Determines what the gauge is attached to.
ChangeFont	Method	Sets the gauge's instance variable **font** and updates the gauge.
Close	Method	Detaches the gauge and closes the window.
Destroy	Method	Destroys the gauge, detaching it first.
Detach	Method	Detaches the gauge from the variable it is attached to.
Reset	Method	Resets the gauge's instance variable **reading** .
SetScale	Method	Sets the scale for the gauge.
Shape	Method	Sweeps a new region.
ShapeToHold	Method	Shapes the gauge window to its smallest possible size.
Update	Method	Reinitializes the gauge and its display window to reflect the current state.

(← *self* **Attach** *obj varName propName type xOrPos y*) [Method of Gauge]

Purpose:	Connects a gauge to an object.
Behavior:	Displays the gauge on the screen and associates that gauge with the variable *varName* of *obj*. If *propName* is specified, the gauge will montior the variable's property. If *xOrPos* and *y* are not specified, a small box will appear which must be positioned to place the gauge.
Arguments:	*obj* A pointer to the object to which the gauge is to be attached.

varName The name of the instance variable, class variable, or method to which the gauge is to be attached.

propName If non-NIL, the gauge will be attached to this property.

type One of IV, CV, or METHOD, within the object being connected to the gauge. If NIL, it defaults to IV.

xOrPos A numerical value to specify where, in screen coordinates, the gauge will be placed on the display. If NIL, you are asked to place the gauge on the screen. This can be a number to specify the x coordinate or a position. If it is a number, also specify *y* .

y If *xOrPos* is not a position, this specifies the y coordinate in screen coordinates for the gauge.

Returns: *self*

Specializations: **StraightScale.Attach** has an additional *shade* argument so that the shade of the scale may be specified at the time the gauge is attached. The following shows the argument list for this method:

(← ($ *instance OfHorizontalScale*) **Attach** *obj varName shade propName type xOrPos y*)

The **Attach** methods for **BarChart**, **HBarChart**, and their subclasses take an additional *label* argument. If no *label* argument is given, the bar is labeled with *varName*. The *label* argument comes last, as follows:

(← ($ *instance OfBarChart*) **Attach** *obj varName propName propName type xOrPos y label*)

(← *self* **Attached?** *don'tPrintFlg*) [Method of Gauge]

Purpose: Determines what a gauge is attached to.

Behavior: If *don'tPrintFlg* is non-NIL this returns the value of the gauge instance variable **containedInAV**. If *dontPrintFlg* is NIL, the **object** and the **varName** the gauge is attached to will be printed in an attached window.

Arguments: *don'tPrintFlg* Suppresses displaying what the gauge is attached to.

Returns: NIL

(← *self* **ChangeFont** *newFont*) [Method of Gauge]

Purpose/Behavior: Sets the gauge's instance variable **font** to *newFont* and updates the gauge. If the gauge is too small for *newFont*, it is reshaped.

Arguments: *newFont* A font in which to display the gauge's text.

Returns: Previous value of **font**.

(← *self* **Close**) [Method of Gauge]

Purpose/Behavior: Detaches the gauge and closes the window.

Returns: CLOSED

(← *self* **Destroy**) [Method of Gauge]

Purpose/Behavior: Destroys the gauge, detaching it first before closing the window.

Returns: NIL

(← *self* **Detach**) [Method of Gauge]

Purpose/Behavior: Detaches the gauge from the variable to which it is attached. This prints in an attached window that the gauge is being detached, and deletes all of the links connecting the gauge, active value, and object being monitored. Does not close the window.

Returns: NIL

(← *self* **Reset** *newReading*) [Method of Gauge]

Purpose/Behavior: Sets the gauge's instance variable **reading** to *newReading* and updates the gauge. If the gauge is too small for *newReading* and it is SelfScaling, it is reshaped.

Arguments: *newReading*
Sets the instance variable **reading** to *newReading*, and updates the gauge without going through any intermediate steps.

Returns: NIL if gauge is **AlphaNumeric** or **RoundScale**; otherwise *self*.

Specializations: **Alphanumeric.Reset**, **RoundScale.Reset**

Example: The following example causes the LCD to be redisplayed with the *newReading*:

```
13←(← ($ lcd1)  Reset   "New Title")
```

(← *self* **SetScale** *min max*) [Method of Gauge]

Purpose/Behavior: Sets the scale for the gauge; computes the new scale values and redisplays if necessary.

Arguments: *min* Lowest value on scale.

max Highest value on scale.

Returns: *self*

(← *self* **Shape** *newRegion noUpdateFlg*) [Method of Gauge]

Purpose/Behavior: If *newRegion* is NIL, you are prompted to sweep out a region which has a minimum sized based upon a **min** property of **IV width** and **height:,min**. If *newRegion* is non-NIL, it is first checked to guarantee that it is at least as large as **width:,min** by **height:,min**.

Arguments: *newregion* List specifying the external coordinates of the window in which the gauge is displayed; list is of the form (left, bottom, width, height).

noUpdateFlg
If NIL, reshapes the gauge.

Returns: NIL

Specializes: Window

Specializations: **LCD**, **Meter**, **DigiMeter**. **Meter.Shape** has an extra argument *ExtraSpaceFlg*. If T, this will allow you to shape a fairly arbitrary region for the gauge; if NIL, the meter is constrained to be close to a square. This latter behavior is what the user sees when trying to shape the meter from the window menu.

BarChart, HBarChart , and their subclasses can only be freely **Shape**d in the direction their bars run (i.e., **BarChart**s can be **Shape**d vertically and **HBarChart**s can be **Shape**d horizontally). Their size along the other dimension is fixed by the number of values attached to the chart .

Example: This example reshapes the gauge to a location where the lower left corner is at (10,100) a width of 50 and a height of 150.

```
14←(← ($ lcd1) Shape '(10 100 50 150))
```

(← *self* **ShapeToHold**) [Method of Gauge]

Purpose/Behavior: Shapes the gauge window to its smallest possible size based on **width:,min** and **height:,min** and redisplays the gauge.

Returns: NIL

Specializations: **LCD.Shape**

(← *self* **Update**) [Method of Gauge]

Purpose/Behavior: Reinitializes the gauge and its display window to reflect the current state.

Returns: *self*

Categories: Window

Examples

The typical use pattern for a gauge is to first create it, set the scale to the appropriate value, and attach it to the desired data.

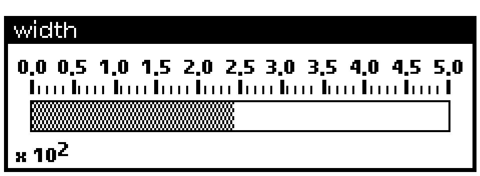
To attach a horizontal scale to a LOOPS window, **w1**, first enter

```
15←(← ($ Window) New 'w1)
#,($& HorizontalScale (|OZW0.1Y:.;h.Qm:| . 495))


16←(← ($ HorizontalScale) New 'hs1)
#,($& HorizontalScale (|OZW0.1Y:.;h.Qm:| . 496))


17←(← ($ hs1) SetScale 0 500)
NIL
```

Now make the connection.

```
18←(← ($ hs1) Attach ($ w1) 'width GRAYSHADE)
#,($& HorizontalScale (|OZW0.1Y:.;h.Qm:| . 496))
```

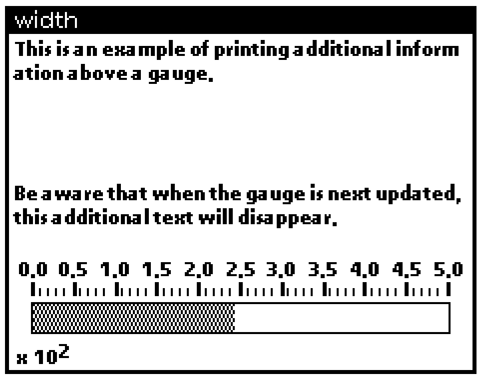
The following gauge appears and you are prompted to place it .



The title of the gauge shows the instance variable being monitored.

Gauges can be shaped larger. The graphics used to display scales do not change; extra white space is added to the top or right. You can use this space to print additional information, as follows:

```
19←(MOVETOUPPERLEFT (@ ($ hs1) window))
{WINDOW}#372,7104

20←(PRIN1 "This is an example of printing additional
information above a gauge.



Be aware that when the gauge is next updated, this
additional text will disappear." (@ ($ hs1) window))
"This is an example of printing additional information
above a gauge.



Be aware that when the gauge is next updated, this
additional text will disappear."
```



Limitations

When a font is changed, a gauge occasionally needs to be updated to be correctly displayed.

Instruments can have only floating point numbers for labels, and cannot have integers.

[This page intentionally left blank]

Description/Introduction

Masterscope has been modified to provide for analysis of files created under the Koto or Lyric/Medley release of LOOPS. A full explanation of Masterscope can be found in the *Lisp Library Modules Manual.* In addition to the relations explained there, LOOPS defines the relations described in this chapter.

Note: Masterscope data base files created under Buttress Loops will not function properly in this release. Those data base files will have to be recreated.

Installation/Loading Instructions

- Load MASTERSCOPE from your Lyric/Medley library floppies according to its loading instructions. This should load the compiled files MASTERSCOPE, MSANALYZE, and MSPARSE.
- Load LOOPSMS.DFASL from wherever you installed the LOOPS Library Modules. This should load versions of MASTERSCOPE and MSPARSE that extend Masterscope to handle LOOPS constructs.

Relations

LOOPS defines the following relations:

Name	Type	Description
SEND	Relation	Collects all places where the method is sent.
SEND SELF	Relation	Collects all places where the method is sent to *self*.
SEND NOTSELF	Relation	Collects all places where the method is sent to an object other than *self*.
GET	Relation	Locates all places where the value of an instance variable is retrieved.
GET CV	Relation	Locates all places where the value of a class variable is retrieved.
PUT	Relation	Locates all places where the value of an instance variable is set.
PUT CV	Relation	Locates all places where the value of a class variable is set.
IMPLEMENT	Relation	Locates all methods that specialize the given selector.
SPECIALIZE	Relation	Locates all methods that specialize the given selector and use ←**Super** in the body of the method.
OVERRIDE	Relation	Locates all methods that specialize the given selector and do not use ←**Super** in the body of the method.

USE IV	Relation	Used with an instance variable name to locate all places where the instance variable is used in a **GET** or **PUT.**
USE CV	Relation	Used with a class variable name to locate all places where the class variable is used in a **GET** or **PUT.**
USE OBJECT	Relation	Used with an object name to locate all places where the object is used**.**

SEND [Relation]

Purpose/Behavior: Used between method names and selectors to collect all places where the method is sent. For example, the form

```
. WHO IS SENT BY 'Helicopter.Move
```

works, but

```
. WHO IS SENT BY Move
```

does not work.

Example: The following command allows you to edit all code that sends the message **New**.

```
. EDIT ALL WHO SEND New
```

SEND SELF [Relation]

Purpose/Behavior: Used between method names and selectors to collect all places where the method is sent to *self*. Places where

(← *self methodName*)

is found are collected, while places where

(← *otherInstance methodName*)

is found are not.

Example: The following command allows you to edit all code that sends the message **Clear** to self.

```
. WHO SENDS SELF Clear
```

SEND NOTSELF [Relation]

Purpose: Same as **SEND SELF**, except the only places where the message is sent to an object other than *self*.

Example: The following allows you to edit all code that sends the message **Clear** to any instance other than *self*.

```
. SHOW ALL WHO SEND NOTSELF Clear
```

GET [Relation]

Purpose: Used with an instance variable name to locate all places where the value of the instance variable is retrieved. This relation can be used along with the **SELF** and **NOTSELF** modifiers.

Example: This command allows you to edit all code that gets the value of the instance variable **width** from an instance other than self and the value of the instance variable **height** from *self*.

```
. SHOW ALL WHO GET NOTSELF width AND GET SELF height
```

GET CV [Relation]

Purpose: Same as **GET**, except that **GET CV** locates places where the value of the class variable is retrieved. This relation can be used with the **SELF** and **NOTSELF** modifiers.

Example: This command allows you to edit all code that accesses the value of the class variable **height** of *self*.

```
. SHOW ALL WHO GET CVSELF height
```

PUT [Relation]

Purpose: Used with an instance variable name to locate all places where the value of the instance variable is set. This relation can be used along with the **SELF** and **NOTSELF** modifiers.

Example: This command allows you to edit all code that sets the value of the instance variable **width.**

```
. EDIT ANY WHO PUT width
```

PUT CV [Relation]

Purpose: Same as **PUT**, except locates places where a specified class variable is set. This relation can be used along with the **SELF** and **NOTSELF** modifiers.

Example: This command list all the sections of code that set the value of the class variable **width** for an instance other than *self*.

```
. WHO PUTS CV NOTSELF width
```

IMPLEMENT [Relation]

Purpose: Used with a method name to locate all methods that specialize the given selector.

Example: This returns a list of classes where the method **Clear** is defined.

```
. WHO IMPLEMENTS Clear
```

SPECIALIZE [Relation]

Purpose: Used with a method name to locate all methods that specialize the given selector and use ←**Super** in the body of the method.

Example: This command allows you to edit all the methods that are specializations of **Clear** and use the ←**Super** form.

```
. EDIT ANY WHO SPECIALIZE Clear
```

OVERRIDE [Relation]

Purpose: Like **SPECIALIZE** above, except it locates all methods that specialize the given selector and ←**Super** is not used in the body of the method.

Example: This command allows you to edit all the specializations of **Clear** that do not make use of the ←**Super** form.

```
. EDIT ALL WHO OVERRIDE Clear
```

USE IV [Relation]

Purpose: Used with an instance variable name to locate all places where the instance variable is used in a **Get** or **Put**. It is equivalent to using the relation form of **GET IVName** or **PUT IVName**.

Example: This command allows you to edit all code that either sets or accesses the instance variable **width**.

```
. EDIT ANY WHO USE THE IV width.
```

USE CV [Relation]

Purpose: Used with a class variable name to locate all places where the class variable is used in a **Get** or **Put**. It is equivalent to using the relation form: **GET CV CVName OR PUT CV CVName**.

Example: This command allows you to edit all code where the class variable **commonWindow** is either set or accessed.

```
. EDIT ANY WHO USE THE CV commonWindow
```

USE OBJECT [Relation]

Purpose Uses an object name to locate all places where the object is used.

Example This command returns a list of all code where the object **Window** is used.

```
. WHO USES THE OBJECT Window??
```

Limitations

Masterscope has several limitations:

- Names of methods must be quoted when used with Masterscope; for example, the method name Helicopter.Move must be entered as 'Helicopter.Move.
- The following expression will not find a call to **GetValue** when in a method:

  ```
  . WHO CALLS GetValue
  ```

  Masterscope does not record calls to **GetValue** and **PutValue** explicitly; it records them under the Get- relation along with calls of the form

  ```
  (← foo Get 'bar)
  ```

  Similarly, the following functions are recorded under relations instead of their names:

GetClassValue	Get CV
PutClassValue	Put CV
GetClassIV	Get IV
PutClassIV	Put IV

If you want to find the explicit calls to Get/PutValue, use

```
. WHO GETS ANY AND NOT SENDS Get
```

- Masterscope currently assumes calls to **GetValue** and similar accessors are accessing instance variables; i.e.,

  ```
  (GetValue foo 'bar)
  ```

  records an access to the instance variable **bar**. This is not necessarily the case; **bar** could also be a class variable.

- The methods and functions that create class and instance variables populate the appropriate **PUT NOTSELF** relations. For example, a function that does

  ```
  (←($ foo) AddCV 'bar)
  ```

  will be found by the query

  ```
  . WHO PUTS CV NOTSELF 'bar
  ```

  An exception occurs with the generalized **Add** and **Delete** method. For example,

  ```
  ($ foo) Add 'IV 'bar)
  ```

  will not be noticed as accessing the instance variable **bar**.

  Also, the templates for methods and functions that accept property lists generally only notice the first property. For example,

  ```
  ((←($ foo) NewWithValues '((bar baz chain link
  sausage)))
  ```

  notices **baz** as a property, not a link.

[This page intentionally left blank]

Description/Introduction

In many knowledge-based systems, it is useful to represent knowledge as interconnected sets of instances. A virtual copy mechanism allows a network of instances to be viewed as a prototype which can be copied. The copy of the prototype is virtual in that the contents of each instance is not completely copied at creation time. Instead, it inherits default values from the prototype (also called the original), thus continuing to share the parts not modified in the copy. The copied network is virtual also in the sense that only those instances needed in the processing are copied.

A virtual copy of an object in the prototype network has the following properties:

- It responds to at least the same set of messages as the prototype object and in the same way; that is, a copy has the same procedural behavior that is defined for the prototype.
- A copy inherits variables and their values from the prototype, and continues to do so until an explicit change is made in the copy. At that point, the new value is stored in the copy and it stops tracking the prototype for that variable. A fetch operation on a value that is not stored locally either finds or creates a virtual copy of the value obtained from the prototype.

Installation/Loading Instructions

The implementation of virtual copies is contained in the file LOOPSVCOPY.LCOM. No other files are necessary.

Application /Module Functionality

A network of instances is tied together through the values of instance variables within each of the instances. Assume an object A has an instance variable x, the value of which is the object B. A virtual copy of A will also have an instance variable named x. The value of x in the copy will point to B if B is a shared object, or x may point to a copy of B if it is to be virtual. Changing the value of x in the copy will not change the value in the original.

Overview of Operation

By default, virtual copies share instance variables. This means that changing the value of an instance variable in the original will be tracked by the copy.

Virtual copies are implemented with two additional classes:

- **VirtualCopyMixin**

  The class **VirtualCopyMixin** is a subclass of Tofu which contains two instance variables:

  - **% copyMap%**
  - **% copyOf%**

  (These unusual names are used to avoid conflicts with any other instance variable names users may create.) This class contains several methods, most of which are required to implement virtual copies and are not used by a programmer.

  Printing a virtual copy instance is a specialization of how regular instances are printed. All instances print as #,($& <class-name> UID). The class of a virtual copy is a dynamic mixin of the class **VirtualCopyMixin** and the class of the original object (see the *LOOPS Reference Manual* for more information on mixins). The virtual copy print function adds the name or unique identifier (UID) of the original object. For example,

  #,($& (VirtualCopyMixin Container1) (JFW0.0X:.aF4.R>8 . 3) c1)

  is a copy of the object named **c1**.

- **VirtualCopyContext**

  The class **VirtualCopyContext** has no methods and only one instance variable, **copyMap**. Instances are used as an argument for calls to **MakeVirtualMixin**.

Since copies can be made of copies, you often need to determine the original object of a chain of copies with the **UltimateOriginal** function.

Operands

This section describes the functions, methods, class variables, and instance variables that operate on virtual copies.

VirtualIVs [Class Variable]

Purpose/Behavior: Helps specify a class whose instances may be made into virtual copies. The value of this class variable should be either the symbol ALL, or a list of instance variables contained within instances of the class. If the value is ALL, all objects pointed to by any of the instance variables will be copied. If the value is a list of instance variables, only the instance variables on this list will have their values copied. Other instance variable values will be shared between the copy and the original.

(**MakeVirtualMixin** *x copyContextObj*) [Function]

Purpose: Creates a virtual copy of an object.

Behavior: *C*reates a dynamic mixin class combining the classes **VirtualCopyMixin** and the class of *x*. An instance of this resulting class is created and it is returned.

Arguments: *x* An object to be copied; must have the class variable **VirtualIVs** as described above.

copyContextObj
Usually NIL; used internally by **MakeVirtualMixin** when it calls itself. It can be an instance of **VirtualCopyContext** if you are creating an instance that is intended to be part of a currently existing network of copies starting from another entry point. See description in **Limitations** below for a further explanation of this point.

Returns: An object that is a copy of *x*.

Example: Refer to the section, "Example."

% copyMap% [Instance Variable of VirtualCopyMixin]

Purpose/Behavior: A mapping of original nodes (which are objects) in a network to the copied nodes. This map is stored in an instance of the class **VirtualCopyContext**.

% copyOf% [Instance Variable of VirtualCopyMixin]

Purpose/Behavior: Within an instance that is a copy, the value of this instance variable is a pointer to the object that was copied.

(← self **VirtualCopy?**) [Method of VirtualCopyMixin]

Purpose: Determines if an object is a virtual copy.

Returns: *self*

Categories: Object, VirtualCopyMixin

copyMap [Instance Variable of VirtualCopyContext]

Purpose/Behavior: The value of this instance variable is a list of dotted pairs. The CAR of each pair is the original; the CDR, the copy.

(**UltimateOriginal** *self*) [Function]

Purpose: Determines what an object is ultimately copying.

Behavior: If *self* is not a virtual copy, *self* is returned.

If *self* is a virtual copy, this recurses through the value of the instance variable **% copyOf%** until it finds the original and returns it.

Arguments: *self* A LOOPS object.

Returns: *self* or what is at the top of *self*'s copy chain.

Example

Create a class called **test** and edit it as shown.

```
44←(←($ Class) New 'test)
#,($C test)

45←(ED 'test)
```

```
SEdit #,($C test) Package: INTERLISP
((MetaClass Class Edited%:                ; Edited 11-Dec-87
                                          ; 16:50 by
  )
 (Supers Object)
 (ClassVariables (VirtualIVs (atomCopy listCopy objCopy)))
 (InstanceVariables (atom NIL)
        (atomCopy NIL)
        (list NIL)
        (listCopy NIL)
        (obj NIL)
        (objCopy NIL))
 (MethodFns))
```

Create an instance called **t0** of this class and inspect it.

```
46←(←($ test) New 't0)
#,($& test (N↑W0.1Y%:.;h.Lh9 . 556))


47←(← ($ test)
      NewWithValues
      (BQUOTE ((atom 1)
                  (atomCopy 2)
                  (list (a b c))
                  (listCopy (A B (\, (← ($ test) New (QUOTE t1)))))
                  (obj (\, (← ($ test) New (QUOTE t2))))
                  (objCopy (\, (← ($ test) New (QUOTE t3)))))))
#,($& test (N↑W0.1Y%:.;h.Lh9 . 560))

48←(← IT SetName 't0)
#,($& test (N↑W0.1Y%:.;h.Lh9 . 560))

49←(INSPECT IT]
{WINDOW}#52,51234
```

```
All Values of test ($ t0).
atom      1
atomCopy  2
list      (a b c)
listCopy  (A B #,($ t1))
obj       #,($ t2)
objCopy   #,($ t3)
```

Make a copy called **t0copy** and inspect it.

```
57←(← (MakeVirtualMixin ($ t0))
   SetName
   (QUOTE t0copy))
#,($& (VirtualCopyMixin test) N↑W0.1Y%:.;h.Lh9 . 562)
```

```
58←(INSPECT IT)
{WINDOW}#53,10150
```

```
All Values of (VirtualCopyMixin test) ($ t0copy).

atom         NIL
atomCopy     NIL
list         NIL
listCopy     NIL
obj          NIL
objCopy      NIL
| copyOf |   #,($ t0)
| copyMap |  #,($& VirtualCopyContext (N↑W0.1Y%:.;h.Lh9 . 563))
```

Make the following changes to **t0** and then reinspect **t0copy**.

```
60←(for iv in '(atom atomCopy list listCopy obj objCopy)
as val in (LIST 11 22 '(a b c d) '(A B C) ($ t3) ($ t1))
do (PutValue ($ t0) iv val]
NIL

61← (INSPECT IT)
{WINDOW}#53,10152
```

```
All Values of (VirtualCopyMixin test) ($ t0copy).
atom         11
atomCopy     22
list         (a b c d)
listCopy     (A B C)
obj          #,($ t3)
objCopy      #.($& (VirtualCopyMixin test) (N↑W0.1Y%:.;h.Lh9 . 565) t1)
| copyOf |   #,($ t0)
| copyMap |  #,($& VirtualCopyContext (N↑W0.1Y%:.;h.Lh9 . 566))
```

The copied instance variables have not changed since they do not track changes in the original object.

Limitations

Some subtle issues are involved in building and using prototype structures so that the structure is preserved in the copied network. These involve how the network is typically traversed.

A general constraint is that all the links to any shared node in the prototype either all be marked as virtual variables, or none of them are. If they are all marked, then a single copy will be made and used. If none are, then the original object from the prototype will be used. Sharing with the prototype can be useful if this object is a repository for standard information that is independent of context. However, if this constraint is violated, the topology of the virtual copy will be different from that of the prototype.

In the simplest situation the network has a single entry node. In this case, a copy-map (see the section "Operands") can be created when the entry node object is first copied. After that all values are copied using this copy-map. The mechanism works well in this situation, even if there is sharing and there are cycles within the network.

At the other extreme, networks can have arbitrary connectivity, including multiple entries from outside the network, for example, from other networks or non-objects. In this case, the following constraints are necessary to ensure correctness of the virtual copy mechanism.

The first constraint states that all access to the network must start through a copy of one of the nodes in the prototype. This condition is necessary because the criteria for copying are contained in the links from one object to another, not in the objects themselves, and a shared node could not specify a link to a node to be copied. This constraint ensures that all accesses from the outside will be copied if and only if that object would have been copied because of an internal link. Otherwise, an analogous situation would occur in which you could either reach a copy or the original node of the prototype itself depending upon which path you follow when the paths lead to the same node in the prototype.

The final constraint requires that all entries to the network should be passed the same copy-map if they are to share structure. The underlying concern in imposing these constraints is that a network be always copied the same way to maintain its topology regardless of where you start.

Suppose you want to make a virtual copy of a virtual copy, that is, to use a virtual copy of a network as a prototype itself. This is very useful if you are using a network to hold the state of a partial design and you want to try two alternative continuations of the design. Some hidden costs are associated with such multiple-level virtual copies.

Suppose further that a network N1 is used as a prototype and you make a virtual copy, N1-VC. Furthermore, N1-VC-VC is defined to be a copy of N1-VC. Values missing from N1-VC-VC are found in the corresponding object of N1-VC. If the value is missing there, the process recurs, and N1 is examined. If the value is to be a virtual copy, then this process will add a virtual copy in N1-VC, and then a second level copy in N1-VC-VC. This is necessary to preserve the semantics presented, but implies that many levels of virtual copy cannot easily do inexpensive incremental searches of a network.

References

Mittal, S. , Bobrow, D. G., and Kahn, K. *Virtual Copies, Between Classes and Instances*. ACM OOPLSA-86 Conference Proceedings, Portland, Oregon, 1986.

INDEX

[This page inte
ntionally left blank]

XEROX LOOPS
LIBRARY MODULES MANUAL

XEROX

Lyric/Medley Release
July 1988

XEROX LOOPS LIBRARY MODULES MANUAL

Lyric /Medley Release

July 1988

TABLE OF CONTENTS

[This page intentionally left blank]

PREFACE

Overview of the Manual

The *Xerox LOOPS Library Modules Manual* describes the Library Modules for Xerox's Lisp Object-Oriented Programming System, Xerox LOOPS (TM). These Library Modules, which can be loaded into your Xerox Artificial Intelligence Environment, provide additional functionality to Xerox LOOPS.

This manual describes the Lyric/Medley Release of the Xerox LOOPS Library Modules, which run under the Lyric and Medley Releases of Xerox Lisp.

Organization of the Manual and How to Use It

This manual is divided into chapters, with each chapter focussing on a particular Library Module. A Table of Contents is included to help you find specific material.

Conventions

This manual uses the following conventions:

- Case is significant in Xerox LOOPS and Lisp. All selectors, methods, arguments, etc., must be typed as shown. Typically, this means that method names are capitalized and variables are not.

- You need to use an Interlisp Exec to enter all exec expressions.

- Arguments appear in italic type.

- Selectors, methods, functions, objects, classes, and instances appear in bold type.

  For example, a method appears as follows:

  (_ *self* **Selector** *Arg1 Arg2*)

- Examples are shown in the Interlisp Exec and appear in the following typeface:

  ```
  89_(_LOGIN)
  ```

- All examples are typed into an Interlisp Exec. This is the recommended Exec for all Xerox LOOPS expressions.

- Methods with an exclamation mark (!) suffix usually perform operations deeply into class structure instead of only on a given object.

- Methods with a question mark (?) suffix usually are predicates; that is, truth functions.

- Methods often appear in the form **ClassName.SelectorName**.

- Cautions describe possible dangers to hardware or software.
- Notes describe related text.

This manual describes the Xerox LOOPS items (functions, methods, etc.) by using the following template:

Purpose: Gives a short statement of what the item does.

Behavior: Provides the details of how the item operates.

Arguments: Describes each argument in the following format:

argument Description

Returns: States what the item returns, and does not appear if the item does not return a value. The phrase "Used as a side effect only." means that the purpose of the item is to perform a computation or action that is independent of any returned value, not to return a particular value.

Categories: A way to group related methods. For example, all the methods related to Masterscope on the class **FileBrowser** have the category Masterscope, not **FileBrowser**. This item appears only for methods.

Specializes: The next higher class in the class hierarchy that contains a method with the same selector. For example, **RectangularWindow.Open** can specialize **Window.Open**. This appears only for methods.

Specializations: The next lower class in the class hierarchy that contains a method with the same selector. For example, **Window.Open** is a specialization of **RectangularWindow.Open**. This appears only for methods.

Example: An example is often included to show how to use the item and what result it produces. Some examples may appear differently on your system, depending on the settings of various print flags. See the *Xerox LOOPS Reference Manual* for details.

References

The following books and manuals augment this manual.

Xerox LOOPS Reference Manual

Xerox LOOPS Release Notes

Xerox LOOPS Users' Modules Manual

Interlisp-D Reference Manual

Common Lisp: the Language by Guy Steele

Xerox Common Lisp Implementation Notes, Lyric Release

Xerox Lisp Release Notes, Lyric and Medley Releases

Xerox Lisp Library Modules Manual, Lyric and Medley Releases

Writer’s Notes -- Conventions

This file includes notes on conventions for *Xerox LOOPS Library Modules Manual*, Lyric Beta Release. This manual is packaged with the *Xerox LOOPS Release Notes* and *Xerox LOOPS Reference Manual* to form one binder.

Writer: Raven Kontur Brewster

Printing Date: >>DA<< >>MO<< 1988

Directories and Files

The directory {ERIS}<Doc>Loops>Lyric>Beta>LibMods> contains the files for the manual. This directory has the following subdirectories:

- {ERIS}<Doc>Loops>Lyric>Beta>LibMods>Z-ReleaseInfo> contains this file on writing conventions and a file on production details.

Filenames describe the contents of the file. For example, the filename

{ERIS}<Doc>Loops>Lyric>Beta>LibMods>Gauges

contains the chapter on gauges.

Assemble the files in the following order for the manual:

{ERIS}<Doc>Loops>Lyric>Beta>LibMods>A1-TitlePage.tedit
{ERIS}<Doc>Loops>Lyric>Beta>LibMods>A2-TOC.tedit
{ERIS}<Doc>Loops>Lyric>Beta>LibMods>A3-Preface.tedit
{ERIS}<Doc>Loops>Lyric>Beta>LibMods>Gauges.tedit
{ERIS}<Doc>Loops>Lyric>Beta>LibMods>Masterscope.tedit
{ERIS}<Doc>Loops>Lyric>Beta>LibMods>VC.tedit

Conventions

This manual uses the following conventions:

- Case is significant in Xerox LOOPS and Lisp. All selectors, methods, arguments, etc., must be typed as shown. Typically, this means that method names are capitalized and variables are not.

- Arguments appear in italic type.

- Selectors, methods, functions, objects, classes, and instances appear in bold type.

  For example, a method appears as follows:

  (_ *self* **Selector** *Arg1 Arg2*)

- Examples appear in the following typeface:

  ```
  89_(_LOGIN)
  ```

- All examples are typed into an Interlisp Exec. This is the recommended Exec for all Xerox LOOPS expressions.

- Methods with an exclamation mark (!) suffix usually perform operations deeply into class structure instead of only on a given object.
- Methods with a question mark (?) suffix usually are predicates; that is, truth functions.
- Methods often appear in the form **ClassName.SelectorName**.
- Cautions describe possible dangers to hardware or software.
- Notes describe related text.

This manual describes the Xerox LOOPS items (functions, methods, etc.) by using the following template:

Purpose: Gives a short statement of what the item does.

Behavior: Provides the details of how the item operates.

Arguments: Describes each argument in the following format:

argument Description

Returns: States what the item returns, and does not appear if the item does not return a value. The phrase "Used as a side effect only." means that the purpose of the item is to perform a computation or action that is independent of any returned value, not to return a particular value.

Categories: A way to group related methods. For example, all the methods releated to Masterscope on the class **FileBrowser** have the category Masterscope, not **FileBrowser**. This item appears only for methods.

Specializes: The next higher class in the class hierarchy that contains a method with the same selector. For example, **RectangularWindow.Open** can specialize **Window.Open**. This appears only for methods.

Specializations: The next lower class in the class hierarchy that contains a method with the same selector. For example, **Window.Open** is a specialization of **RectangularWindow.Open**. This appears only for methods.

Example: An example is often included to show how to use the item and what result it produces. Some examples may appear differently on your system, depending on the settings of various print flags.

Style Sheet Addenda

Here are some guidelines I used when writing the LOOPS manuals. Items appear in rather random order.

- Avoid contractions.
- Avoid subscripts. Use WORD1 rather than WORD to avoid inconsistent line leading.
- Avoid wording that starts "Note that..." or "Notice that...". Either make it a note with correct format or eliminate the "Note that".
- Use semicolons rather than m-dashes.
- Each item in the template starts with an initial capital letter; e.g., "Describes..."
- The arguments are identical in the calland in the argument description.

- Parenthesies appear around expressions and square brackets appear around the name of the functionality.

- The arrow in the expression is the NS character ←, not _. These characters appear similarly when printed, but differently on the screen. See the section, "Special Notes and Cautions," for details

- A period appears after the word None, after argument descriptions, and Returns: item.

- Items are set to or return T (instead of true).

- Menus contain options, not items or selections.

- You drag (not roll) the mouse to the right of a menu option to see its submenu.

- Use "above" and "below" when referrering to things in the same section, section numbers and names when referrering to things in the same chapter, and chapter numbers and names when referrering to things in another chapter.

- Please study the following style sheet carefully before you start to edit. The various appearances of active value and annotated values are especially crazy making.

  These things appear in **bold**:
  class variables
  functions
  instance variables
  messages
  methods
  variables

  ActiveValue - specific class/instance
  active value - general information
  activeValue - previous implementation of **ActiveValue**

  annotatedValue - data type
  AnnotatedValue - specific class
  annotated values - general information

  bitmap

  data type

  file package
  filecoms

  inspector

  Lisp Library package
  localState - instance variable

  non-NIL

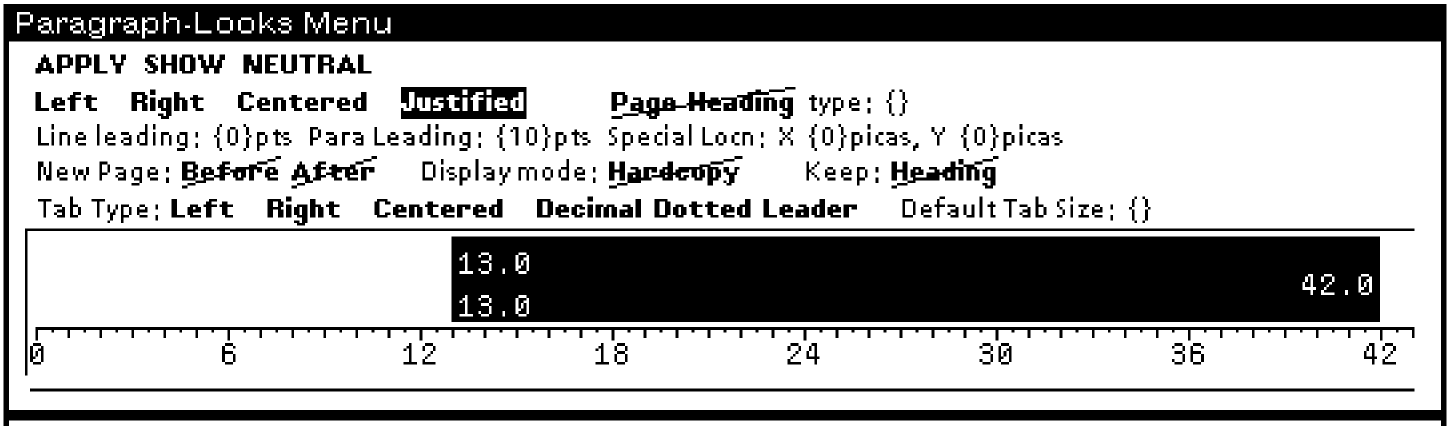
  prettyprints

  supers list

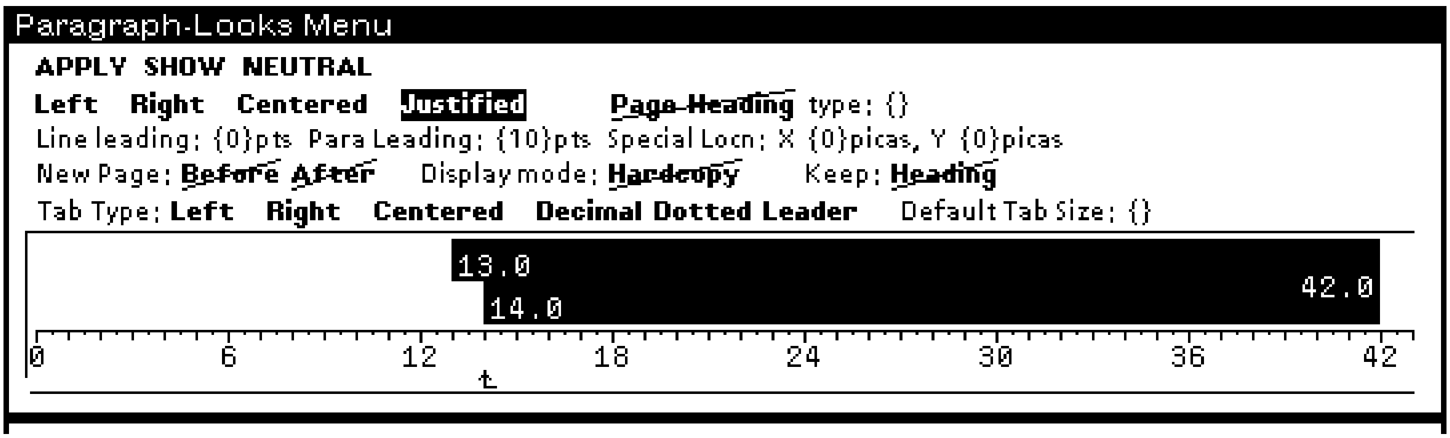
- Figures

Paragraph Formatting

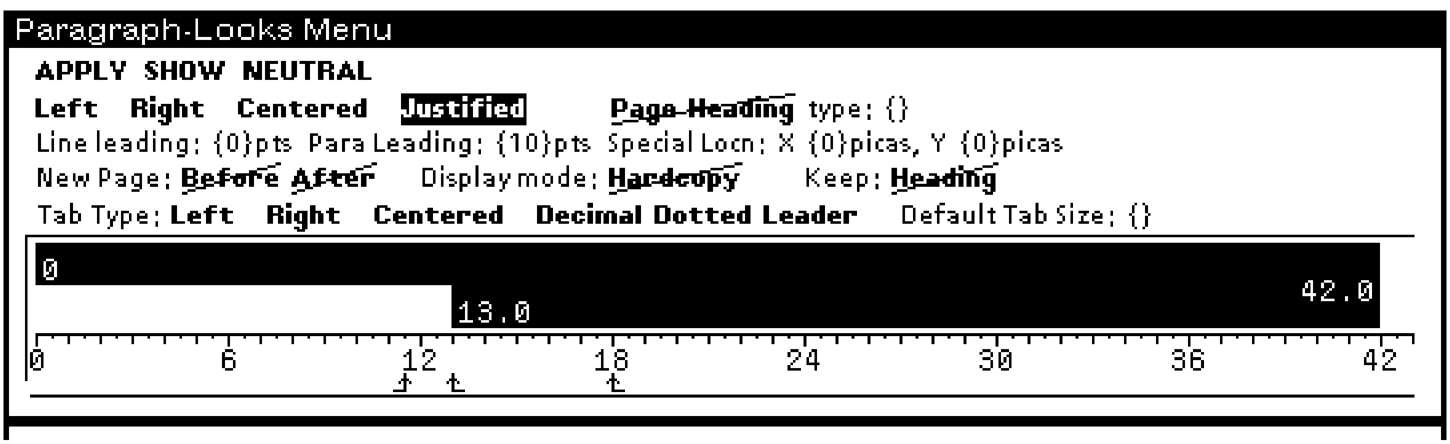
The text has the following format:



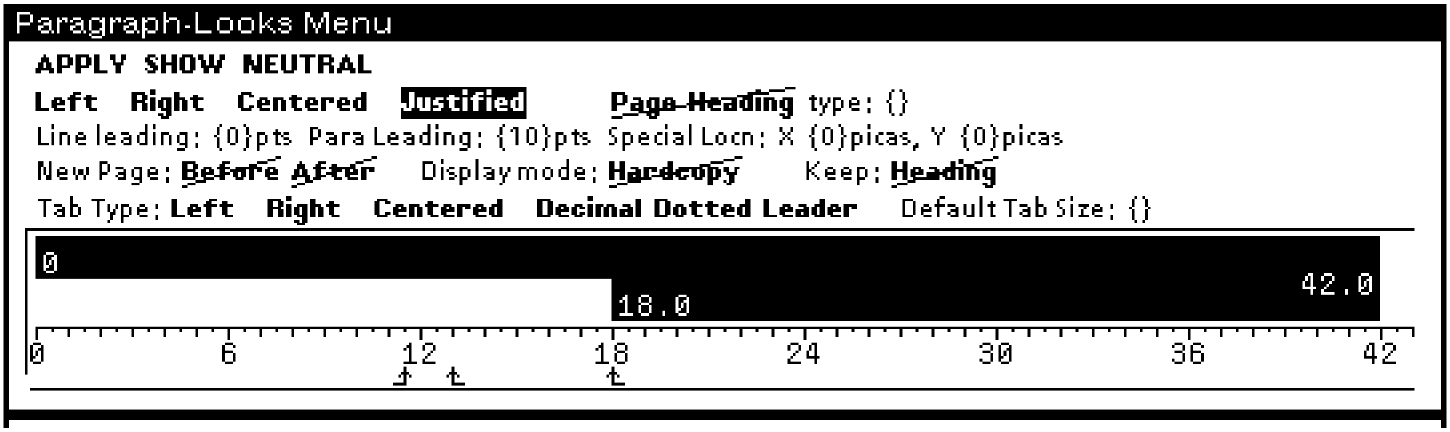
Bulleted lists have the following format:



The template has the following format:

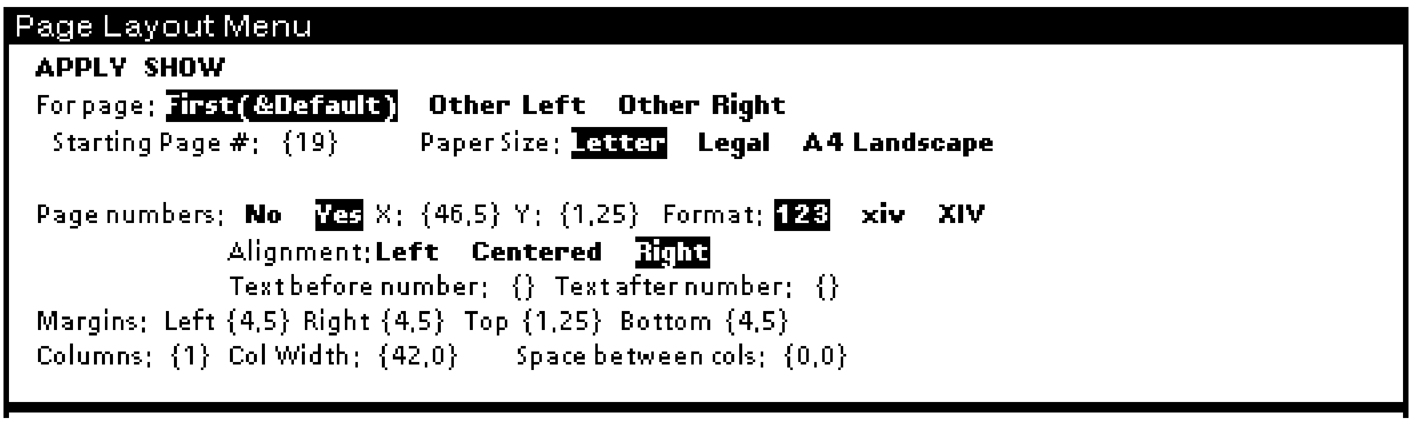


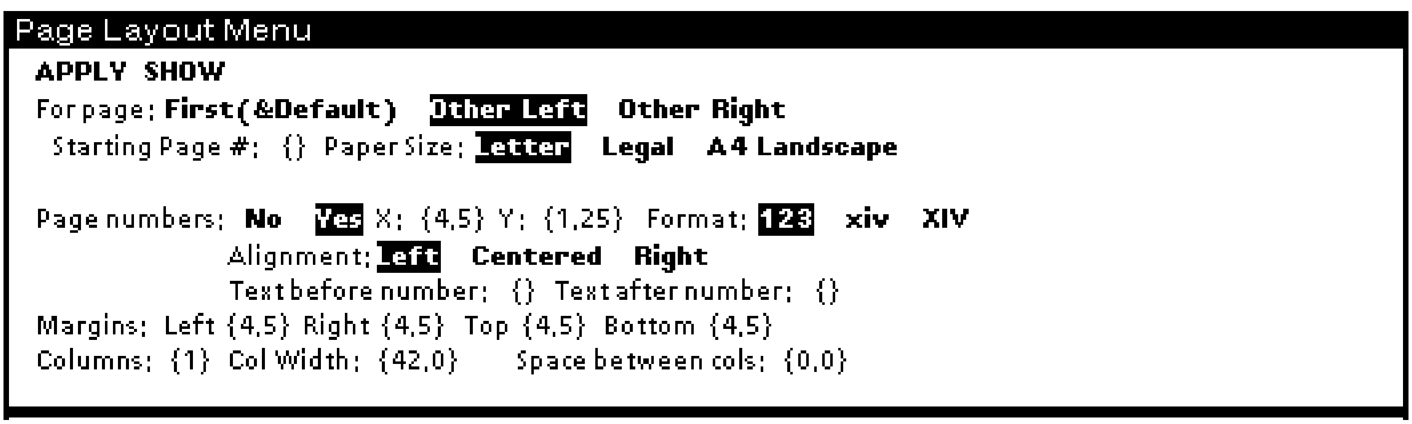
The Arguments section of the template has the second line start at 18 instead of 13.

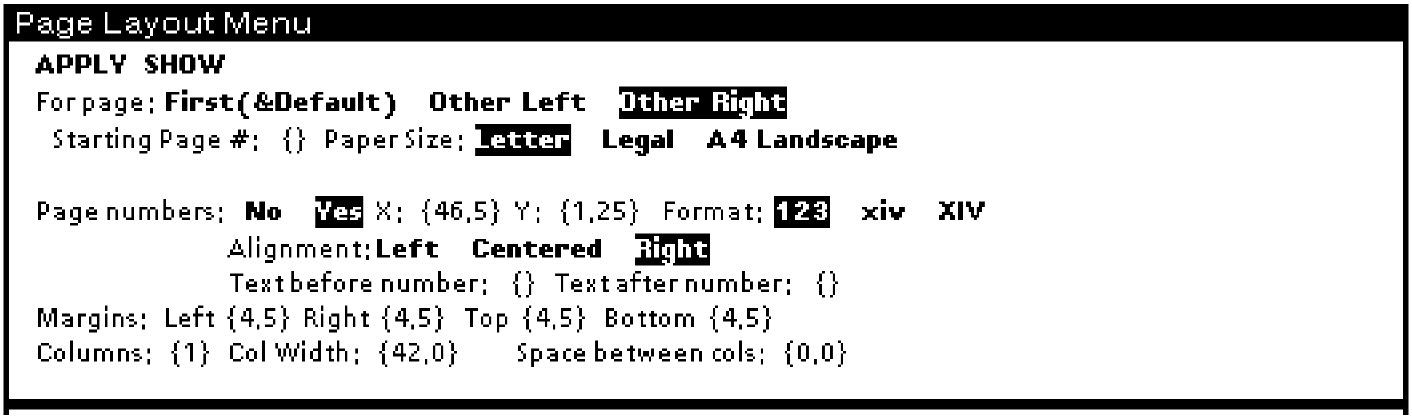


Page Layout

Page numbering varies with the chapter.







Bitmaps, Graphs, and Sketches

To do SEdit and Inspector examples for the manual, you need to reset your FONTPROFILE and scale the resulting windows to 0.8.

-- In your Interlisp Executive, enter (DV FONTPROFILE)

-- Edit the FONTPROFILE to be as follows. (some of this is probably overkill, but it does eliminate any suprizes)

```
SEdit FONTPROFILE  Package: INTERLISP
((DEFAULTFONT 1 (GACHA 12 BRR)
                (GACHA 10)
                (TERMINAL 10))
 (ITALICFONT 1 (HELVETICA 12 MIR)
               (GACHA 10 MIR)
               (MODERN 10 MIR))
 (BOLDFONT 2 (HELVETICA 12 BRR)
             (HELVETICA 10 BRR)
             (MODERN 10 BRR))
 (LITTLEFONT 3 (HELVETICA 10)
               (HELVETICA 8 MIR)
               (MODERN 10 MIR))
 (TINYFONT 6 (GACHA 10)
             (GACHA 8)
             (TERMINAL 8))
 (BIGFONT 4 (HELVETICA 14 BRR) NIL (MODERN 12 BRR))
 (MENUFONT 5 (HELVETICA 12))
 (COMMENTFONT 6 (HELVETICA 12)
                (HELVETICA 10)
                (MODERN 10))
 (TEXTFONT 7 (TIMESROMAN 12) NIL (CLASSIC 12)))
```

-- In your Interlisp Executive, enter
(FONTPROFILE FONTPROFILE)
(SEDIT.RESET)

-- Make bitmaps of the resulting windows, and scale these bitmaps to 0.8

To get the pop-up menus (and their drag-through submenus) into a bitmap for using as an illustration:

--Move your type-in point to the exec window.

--Bring up your pop-up menu.

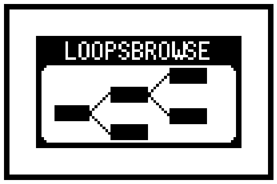
--control-G

--When the menu comes up, select Mouse *run. This will cause a break and spawn a new mouse process, so that the mouse continues to work.

--Move your type-in point to the tedit window.

--Shift-snap the menu image into the tedit window.

--Move the type-in point to the break window and type ^.



--To have your browser menus be a different font, type in your Interlisp Exec

(SETQ MENUFONT (FONTCREATE 'TERMNIAL 12 'BOLD))

--The global variable MENUFONT which is currently set to (HELVETICA 10 MRR).

--The menus that have already been created will still display this old font. Either recreate the browsers to force the menus to be recreated, or send a message to the browser

(_ browser ClearMenuCache)

to force that browser to recreate its menus.

Special Notes and Cautions

Make sure you have changed the underscore to be a left arrow before loading and printing any files. To do this,

- Enter the following commands into your Executive:

```
(GETCHARBITMAP (CHARCODE _) '(MODERN 10 MRR))
(EDITBM IT)
```

- When the bitmap editor apears, delete the underscore and insert the following left arrow:

```
..........
..........
..........
..........
..........
....X.....
...XX.....
..XXXXXX..
...XX.....
....X.....
..........
..........
..........
..........
```

- Finally, enter the following commands into your Executive to store the pattern:

```
(PUTCHARBITMAP (CHARCODE _) '(MODERN 10 MRR) IT)
(PUTCHARBITMAP (CHARCODE _) '(MODERN 10 BRR) IT)
(PUTCHARBITMAP (CHARCODE _) '(TERMINAL 10 MRR) IT)
(PUTCHARBITMAP (CHARCODE _) '(TERMINAL 10 BRR) IT)
(PUTCHARBITMAP (CHARCODE _) '(TERMINAL 12 BRR) IT)
```

INDEX

[This page intentionally left blank]

Description/Introduction

Gauges are an important part of the LOOPS user interface for both developers and end users. Gauges assist in understanding the dynamic nature of the programs. This is in contrast to the more typical case of debugging programs using static means. In the creation of user-friendly interfaces, you can use gauges to display, in analog or digital form, various data that may be changing. Also, by employing active gauges, you can provide a convenient way to interact with a system.

One of the features of gauges is the ease with which you can use them in a system. In more traditional languages, if you want to understand how a variable is changing over the course of a computation, you must make modifications in your program wherever you want to begin or end the examination of a variable. Given the capabilities of active values used by gauges, you need only attach or detach a gauge to the data that you are interested in monitoring.

The following types of gauges are available:

- Meter; a circular instrument that wraps around any number of times.
- Dial; a bounded dial, like an automobile speedometer.
- LCD; a gauge that uses the entire window to display a value.
- Scale; a horizontal or vertical display of a gauge.
- ActiveScale; a scale that allows you to change the gauge value.

Gauges are an example of the combination of programming capabilities within LOOPS. The different types of gauges are defined within the context of an inheritance lattice. This allows the more general functionality and variables to be allocated to more general gauge classes, with specific functionality placed in more restricted classes. You can also see the use of mixins to add a small amount of functionality to several different classes of gauges.

Note: Mixins are classes that are used only in conjunction with another class to create a subclass.

The methods within gauges are built upon both function calling and message sending. Gauges are "attached" to objects through the mechanism of active values. Since gauges are built upon the mechanism of active values, gauges can only be attached to data within objects. It is not possible to use gauges to monitor any arbitrary Lisp variable.

Prerequisites

The default font for gauges is Modern 10.

Installation/Loading Instructions

Gauges are divided among several different files to allow you to load only those objects and functions that you need. The table below lists the files to load for each type of gauge. The filecoms for each file will try to load any other required gauge files from **LOOPSLIBRARYDIRECTORY**. The file GAUGES.DFASL and either GAUGEINSTRUMENTS.DFASL or GAUGEALPHANUMERICS.DFASL will always be loaded; other files may also be loaded.

Gauge	File to load
LCD	GAUGEALPHANUMERICS.DFASL
METER	GAUGEMETERS.DFASL
DIAL	GAUGEDIALS.DFASL
SCALE	GAUGESCALES.DFASL
ACTIVE SCALE	GAUGEACTIVE.DFASL

Additionally, the file GAUGESELFSCALEMIXIN.DFASL can be loaded to add the class **SelfScaleMixin**, and GAUGEALARMS.DFASL can be loaded to add the class **AlarmMixin**.

To load the required files, first set the value of **LOOPSDIRECTORY** to include the directory where the gauges files are stored, then type the following expression in the Executive:

(LOAD '*FILENAME*)

To load all of the gauges, load the file GAUGELOADER and then enter `(LOADGAUGES)`. GAUGELOADER also sets the variables: **GAUGEFILES** and **GaugeClasses**.

(**LOADGAUGES** *LDFLG SOURCES?FLG*) [Function]

Purpose: Loads all the gauges.

Behavior: Assumes that all of the gauge files are on the **LOOPSDIRECTORY** search path.

All the gauge files will be loaded based upon the settings of *LDFLG* and *SOURCES?FLG*. A **FILESLOAD** expression is built up and evaluated.

Arguments: *LDFLG* Can be NIL, PROP, or SYSLOAD. See the **LDFLG** discussion under loading in the *Interlisp-D Reference Manual*.

SOURCES?FLG Can be NIL or T. If NIL, this attempts to load the compiled files before trying to load the sources. If T, only the sources are loaded.

Returns: Used for side effect only.

GAUGEFILES [Variable]

Behavior: Initialized to (GAUGEACTIVE GAUGEALARMS GAUGEALPHANUMERICS GAUGEBOUNDEDMIXIN GAUGEDIALS GAUGEDIGIMETER GAUGEDIGISCALE GAUGEINSTRUMENTS GAUGEMETERS GAUGES GAUGESCALES GAUGESELFSCALEMIXIN)

GaugeClasses [Variable]

Behavior: Initialized to (GaugeAV ActiveGaugeMixin Gauge AlarmMixin BoundedMixin SelfScaleMixin)

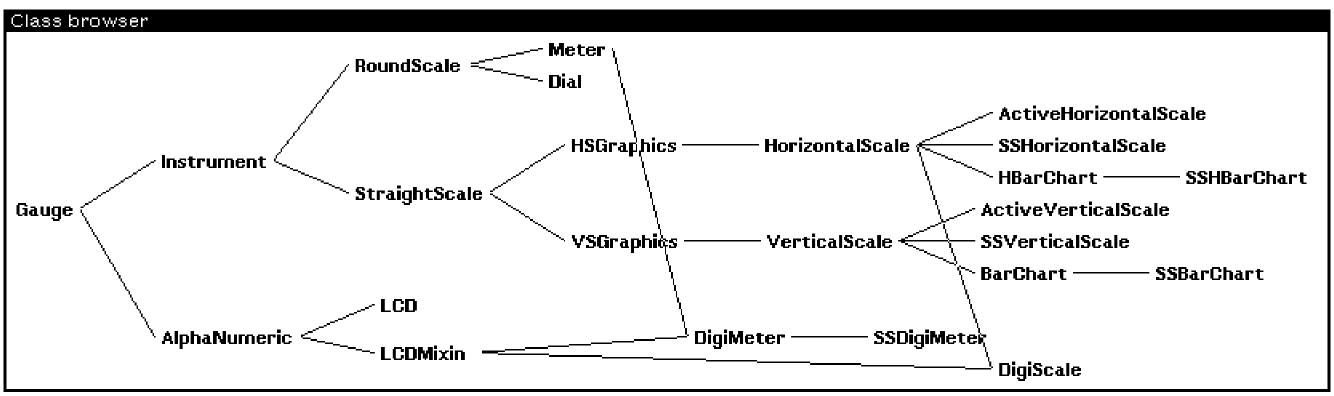
Call (Browse **GaugeClasses**) to open a browser of all of the gauge classes.

Application/Module Functionality

This section describes the gauge classes and methods.

Gauge Classes

This section describes the available gauges shown in the following browser.



Note: The browser does not include the optional mixin classes.

Within the class description of each class, the instance variables and class variables that are specializations only because they have different default values are not listed.

Name	Type	Description
ActiveGaugeMixin	AbstractClass	A gauge class that allows you to set the value of the variable being monitored with the cursor, via a SET menu.
ActiveHorizontalScale	Class	An active gauge that displays the value on a horizontal scale.
ActiveVerticalScale	Class	An active gauge that displays its value on a vertical scale.
AlarmMixin	AbstractClass	A mixin that adds alarm functionality to any gauge.
AlphaNumeric	AbstractClass	A gauge that gives an alphanumeric display of a value.

BarChart	Class	A gauge that displays more than one **VerticalScale** side-by side.
BoundedMixin	AbstractClass	Creates a bounded scale for **displayVal**; to be used as a mixin for instruments.
Dial	Class	A bounded dial, like an automobile speedometer.
DigiMeter	Class	A gauge that displays both an **LCD** and a meter.
DigiScale	Class	A gauge that displays both an **LCD** and a horizontal scale.
Gauge	AbstractClass	A class for objects that present a dynamic graphical image of a LOOPS value.
GaugeAV	Class	An active value associated with a gauge.
HBarChart	Class	A gauge that displays more than one **HorizontalScale** side-by side.
HorizontalScale	Class	A labeled, bounded scale with a bar that fills to the right.
HSGraphics	AbstractClass	Gauge that is displayed in the form of a singe horizontal scale or bar.
Instrument	AbstractClass	A numeric gauge that is externally scaled by **inputLower** and **inputRange** and scaled internally by **lower** and **range**.
LCD	Class	Differs from **AlphaNumeric** in that the entire gauge window is the printing region.
LCDMixin	AbstractClass	Computes print region differently from **LCD**.
Meter	Class	A circular instrument that wraps around any number of times.
RoundScale	AbstractClass	Abstract Class for instruments with circular (arc) scales.
SelfScaleMixin	AbstractClass	Provides for the gauge to rescale according to the reading.
SSBarChart	Class	A self-scaling version of **BarChart.**
SSDigiMeter	Class	A self-scaling version of **DigiMeter**.
SSHBarChart	Class	A self-scaling version of **HBarChart.**
SSHorizontalScale	Class	Gauge that is displayed in the form of a single scale or bar which rescales itself accordingly.
SSVerticalScale	Class	Gauge that is displayed in the form of a single vertical scale or bar which rescales itself accordingly.
StraightScale	AbstractClass	Abstract Class for instruments with straight scales.
VSGraphics	AbstractClass	Gauge that is displayed in the form of a single vertical scale or bar.
VerticalScale	Class	Gauge that is displayed in the form of a single vertical scale or bar.

ActiveGaugeMixin [Class]

Description: A gauge class that allows you to set the value of the variable being monitored with the cursor, via a **SET** menu.

MetaClass: AbstractClass

Supers: Object

Class Variables: None.

Instance Variables: **cursor** The cursor to use when changing the scale; the default is NIL.

ActiveHorizontalScale [Class]

Description: An active gauge that displays the value on a horizontal scale. This gauge shows the value of the data it is connected with and allows you to change that data with the gauge.

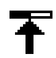
MetaClass: Class

Supers: ActiveGaugeMixin, HorizontalScale

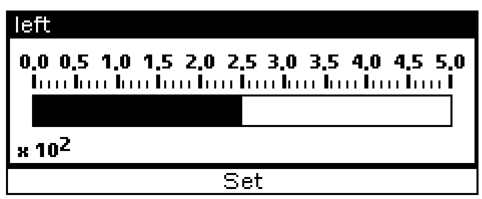
Class Variables: None.

Instance Variables: **cursor** Cursor to use when changing the scale; its property **:initform** is set to **HorizontalAGCursor**.

Example: These gauges have an attached menu at the bottom of the gauge. When you position the cursor over this menu and press a mouse button, the cursor changes to the following shape:



While the left button is held down, the system tracks movements of the cursor and changes the value that the gauge is monitoring.



ActiveVerticalScale [Class]

Description: Similar to **ActiveHorizontalScale**, except that a vertical scale is used.

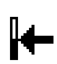
MetaClass: Class

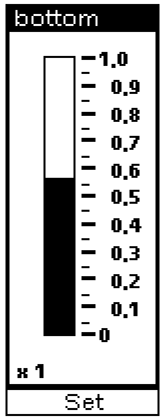
Supers: ActiveGaugeMixin, VerticalScale

Class Variables: None.

Instance Variables: **cursor** Cursor to use when changing the scale; its **:initform** property is set to **VerticalAGCursor**.

Example: Similar to **ActiveHorizontalScale**. When setting, the cursor changes to the following shape:





AlarmMixin [Class]

Description: A mixin that adds alarm functionality to any gauge. An alarm is defined as warning object that is set off when the value being monitored falls outside of the specified range. The gauge flashes and stays inverted when the alarm is tripped.

CAUTION

When a new class of gauges is created that will use the properties of **AlarmMixin**, **AlarmMixin** should be the first class on the Supers list of the new class. This guarantees that the **AlarmMixin.Set** method is invoked.

MetaClass: AbstractClass

Supers: Object

Class Variables: MiddleButtonItems

Instance Variables: **lowTripPoint**
Alarm is triggered when reading goes below this point.

hiTripPoint Alarm is triggered when reading goes above this point.

flashNumber
Number of times alarm will flash when it is tripped.

flashInverval
Interval in milliseconds between flashes.

AlphaNumeric [Class]

Description: This class contains some of the methods and data for the LCD classes. These gauges can display any type of character, letters, or numbers.

MetaClass: AbstractClass

Supers: Gauge

Class Variables: None.

Instance Variables: **precision** Number of characters displayed in the reading. The default value is 5.

BarChart [Class]

Description: A gauge that can display more than one **VerticalScale** at once, side-by side.

MetaClass: Class

Supers: VerticalScale

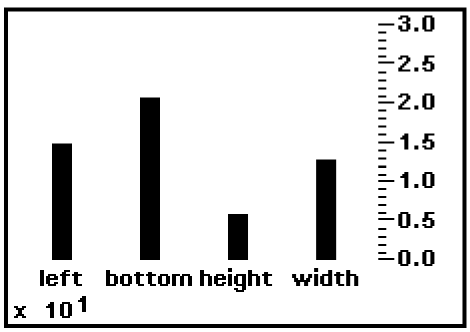
Class Variables: None.

Instance Variables: **maxLabelWidth**
Maximum width of labels on each bar. Default value is 0 which means no limit.

scaleLeft
Offset within the gauge window from the left for the leftmost bar. Default value is 3.

scaleBottom
Offset within the gauge window from the bottom for all the bars. Default value is 30.

Example: Here is a **BarChart** showing the size and shape of a window. It is displaying the values 15, 21, 13, and 6.



BoundedMixin [Class]

Description: This mixin is a super of the scale classes and **Dial**. If a gauge that has **BoundedMixin** as a super class tries to display a new setting that is outside of the range of the gauge, the gauge will display the minimum or maximum value as appropriate and place a "??" in the window.

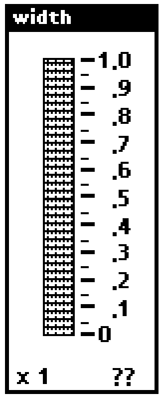
MetaClass: AbstractClass

Supers: Object

Class Variables: None.

Instance Variables: None.

Example: Here is a vertical scale that displays a reading greater than its maximum.



Dial [Class]

Description: A bounded dial, like an automobile speedometer.

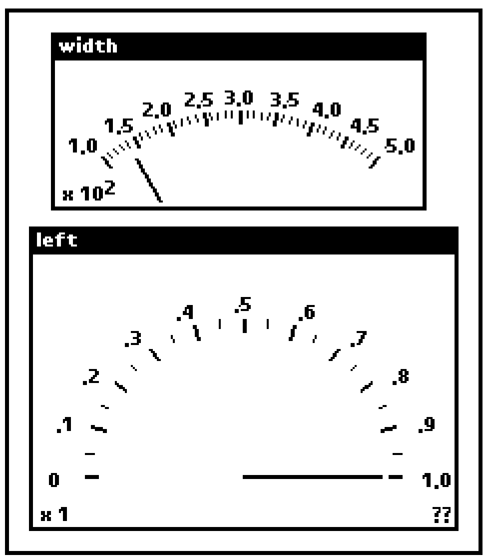
MetaClass: Class

Supers: BoundedMixin, RoundScale

Class Variables: None.

Instance Variables: This class specializes the same instance variables as **RoundScale**.

Example: The angle of the arc changes with the shape of the window.



DigiMeter [Class]

Description: A gauge that combines both a meter and an LCD.

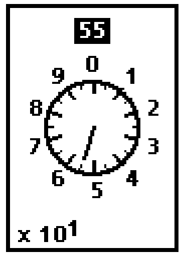
MetaClass: Class

Supers: Meter, LCDMixin

Class Variables: None.

Instance Variables: **spaceForLCD**
Vertical space required by LCD within the gauge. Defaults to 30.

Example: This **DigiMeter** is displaying 55.



DigiScale [Class]

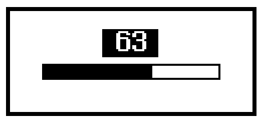
Description: A gauge that combines both a horizontal scale having no ticks and an LCD.

MetaClass: Class

Supers: HorizontalScale, LCDMixin

Class Variables: None.

Example: This **DigiScale** is displaying 63 with its scale set from 0 to 100.



Gauge [Class]

Description: A class for objects that present a dynamic graphical image of a Xerox LOOPS value. This class provides most of the methods for using gauges.

MetaClass: AbstractClass

Supers: Window

Class Variables: **LeftButtonItems**
Menu options associated with the left mouse button.

MiddleButtonItems
Menu options associated with the middle mouse button.

Instance Variables: **reading** External value of reading. The default value is 0.

containedInAV
Active value that connects the gauge to the data it is monitoring. It should be an instance of the class **GaugeAV**.

font Font that is used by a gauge; default value is (Modern 10).

width Width of a gauge; has property **min,** which specifies the minimum width for a gauge.

height Height of a gauge; has property **min**, which specifies the minimum height for a gauge.

GaugeAV [Class]

Description: An active value that is associated with a gauge.

MetaClass: Class

Supers: LocalStateActiveValue

Class Variables: None.

Instance Variables:

gauge The gauge connected to this active value.

object The object containing the variable associated with the active value.

propName The property name of the associated variable.

type Data type of the associated variable.

varName Name of the associated variable.

HBarChart [Class]

Description: A gauge that can display more than one **HorizontalScale** at once, side-by side.

MetaClass: Class

Supers: HorizontalScale

Class Variables: None.

Instance Variables:

maxLabelWidth
Maximum width of labels on each bar. Default value is 0 which means no limit.

scaleLeft
Offset within the gauge window from the left for the leftmost bar. Default value is 3.

Example: Here is an **HBarChart** showing the size and shape of a window. It is displaying the values 15, 21, 13, and 6.

HorizontalScale [Class]

Description: A labeled, bounded scale with a bar that fills to the right.

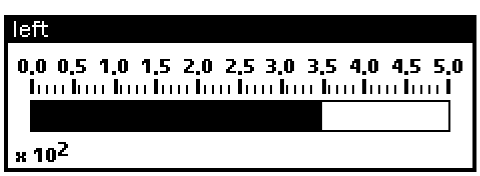
MetaClass: Class

Supers: HSGraphics

Class Variables: None.

Instance Variables: None.

Example: This **HorizontalScale** is reading 350 on a scale from 0 to 500.



HSGraphics [Class]

Description: This class provides some of the methods for displaying the graphics of a horizontal scale.

MetaClass: AbstractClass

Supers: StraightScale

Class Variables: None.

Instance Variables:

scaleBottom Bottom edge of scale in pixels. The default value is 10.

scaleLeft Left edge of scale in pixels. The default value is 12.

scaleWidth Width of inside of scale in pixels. The default value is 120.

scaleHeight Height of scale in pixels. The default value is 15.

Instrument [Class]

Description: A class that provides additional methods and data for gauges that display only numerical data. This data is externally scaled by **inputLower** and **inputRange**, and scaled internally by **lower** and **range**.

MetaClass: AbstractClass

Supers: Gauge

Class Variables: None.

Instance Variables:

ticks Scale marks on the instrument; value is a number or NIL; **smallTicks** property indicates the number of smaller ticks between each large tick.

displayVal Internal value relative to instrument.

range Range for internal **displayVal** .

inputRange Range for external reading.

lower Lower bound for internal **displayVal**.

inputLower Lower bound for external reading.

brushWidth Scale factor for width of ticks, rays, and circles in pixels.

labels The labels that will be displayed on the gauge.

labelScale A dotted pair representing the sign and exponent of a reading.

spaceForLabelScale Extra vertical space to display scale label.

LCD [Class]

Description: Differs from **LCDMixin** in that the entire gauge window is the printing region.

MetaClass: Class

Supers: AlphaNumeric

Class Variables: None.

Instance Variables: None.

Example: This **LCD** is displaying the string "Mumble", and has been **Shaped**to 120 x 60.

LCDMixin [Class]

Description: Computes printing region differently from LCD so that an LCD may be added into another window.

MetaClass: AbstractClass

Supers: AlphaNumeric

Class Variables: None.

Instance Variables: **precision** Number of characters displayed in the reading; the default value is 3. Its property is **readingRegion**; the default value is NIL.

readingY Y position of bottom of reading. The default value is 7.

Meter [Class]

Description: A circular instrument that wraps around any number of times. It displays a sign and exponent in the lower left corner of its window.

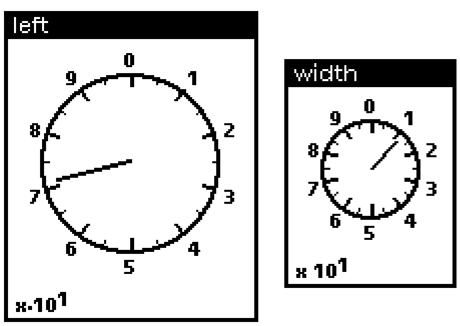
MetaClass: Class

Supers: RoundScale

Class Variables: None.

Instance Variables: This class specializes the same instance variables as **RoundScale**.

Example: The **Meter** on the left is displaying a negative value.



RoundScale [Class]

Description: Abstract Class for instruments with circular (arc) scales.

MetaClass: AbstractClass

Supers: Instrument

Class Variables: None.

Instance Variables: **needleLength**
Radius of needle in pixels. The default value is 15.

radius Radius of arc in pixels. The default value is 10.

xc x-coordinate window coordinate of center of arc. (See **DRAWARC** in the *Xerox Lisp Release Notes, Lyric Release*.)

yc y-coordinate window coordinate of center of arc. (See **DRAWARC** in the *Xerox Lisp Release Notes, Lyric Release*.)

SelfScaleMixin [Class]

Description: Provides for the gauge to rescale according to the reading.

MetaClass: AbstractClass

Supers: Object

Class Variables: None.

Instance Variables: **lowScaleFactor**
Rescales if reading shrinks so that it will fit more than **lowScaleFactor** times in **inputRange**. The default value is 5.

SSBarChart [Class]

Description: A self-scaling version of **BarChart.**

MetaClass: Class

Supers: BarChart

Class Variables: None.

Instance Variables: None.

SSDigiMeter [Class]

Description: A self-scaling version of **DigiMeter**.

MetaClass: Class

Supers: DigiMeter

Class Variables: None.

Instance Variables: None.

SSHBarChart [Class]

Description: A self-scaling version of **HBarChart.**

MetaClass: Class

Supers: HBarChart

Class Variables: None.

Instance Variables: None.

SSHorizontalScale [Class]

Description: Gauge that is displayed in the form of a single horizontal scale or bar which rescales itself accordingly.

MetaClass: Class

Supers: VerticalScale

Class Variables: None.

Instance Variables: None.

SSVerticalScale [Class]

Description: Gauge that is displayed in the form of a single vertical scale or bar which rescales itself accordingly.

MetaClass: Class

Supers: HorizontalScale

Class Variables: None.

Instance Variables: None.

StraightScale [Class]

Description: Abstract class for instruments with straight scales.

MetaClass: AbstractClass

Supers: BoundedMixin, Instrument

Class Variables: None.

Instance Variables: **shade** Shade of bar; numeric value from 0 to 65535. The default value is 65535, which is BLACKSHADE.

VerticalScale [Class]

Description: Gauge that is displayed in the form of a singe vertical scale or bar.

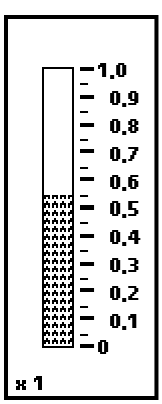
MetaClass: Class

Supers: VSGraphics

Class Variables: None.

Instance Variables: None.

Example: This **VerticalScale** is displaying the value .55 and has its **Shade** set to 1258.



VSGraphics [Class]

Description: Similar to **HSGraphics** but for vertical scales.

MetaClass: AbstractClass

Supers: StraightScale

Class Variables: None.

Instance Variables:

scaleBottom Bottom edge of scale in pixels. The default value is 12.

scaleLeft Left edge of scale in pixels. The default value is 15.

scaleWidth Width of inside of scale in pixels. The default value is 15.

scaleHeight Height of scale in pixels. The default value is120.

Gauge Methods

This section describes the available methods and functions which are used to manipulate gauges. In many cases, a particular gauge class specializes a method defined in the class **Gauge**. In this case, the specialized method definition is not explicitly defined; instead, this is noted in the Specializes/Specializations field of the description.

Name	Type	Description
Attach	Method	Connects a gauge to an object**.**
Attached?	Method	Determines what the gauge is attached to.
ChangeFont	Method	Sets the gauge's instance variable **font** and updates the gauge.
Close	Method	Detaches the gauge and closes the window.
Destroy	Method	Destroys the gauge, detaching it first.
Detach	Method	Detaches the gauge from the variable it is attached to.
Reset	Method	Resets the gauge's instance variable **reading** .
SetScale	Method	Sets the scale for the gauge.
Shape	Method	Sweeps a new region.
ShapeToHold	Method	Shapes the gauge window to its smallest possible size.
Update	Method	Reinitializes the gauge and its display window to reflect the current state.

(← *self* **Attach** *obj varName propName type xOrPos y*) [Method of Gauge]

Purpose: Connects a gauge to an object.

Behavior: Displays the gauge on the screen and associates that gauge with the variable *varName* of *obj*. If *propName* is specified, the gauge will montior the variable's property. If *xOrPos* and *y* are not specified, a small box will appear which must be positioned to place the gauge.

Arguments: *obj* A pointer to the object to which the gauge is to be attached.

varName The name of the instance variable, class variable, or method to which the gauge is to be attached.

propName If non-NIL, the gauge will be attached to this property.

type One of IV, CV, or METHOD, within the object being connected to the gauge. If NIL, it defaults to IV.

xOrPos A numerical value to specify where, in screen coordinates, the gauge will be placed on the display. If NIL, you are asked to place the gauge on the screen. This can be a number to specify the x coordinate or a position. If it is a number, also specify *y* .

y If *xOrPos* is not a position, this specifies the y coordinate in screen coordinates for the gauge.

Returns: *self*

Specializations: **StraightScale.Attach** has an additional *shade* argument so that the shade of the scale may be specified at the time the gauge is attached. The following shows the argument list for this method:

(_ ($ *instance OfHorizontalScale*) **Attach** *obj varName shade propName type xOrPos y*)

The **Attach** methods for **BarChart**, **HBarChart**, and their subclasses take an additional *label* argument. If no *label* argument is given, the bar is labeled with *varName*. The *label* argument comes last, as follows:

(_ ($ *instance OfBarChart*) **Attach** *obj varName propName propName type xOrPos y label*)

(← *self* **Attached?** *don'tPrintFlg*) [Method of Gauge]

Purpose: Determines what a gauge is attached to.

Behavior: If *don'tPrintFlg* is non-NIL this returns the value of the gauge instance variable **containedInAV**. If *dontPrintFlg* is NIL, the **object** and the **varName** the gauge is attached to will be printed in an attached window.

Arguments: *don'tPrintFlg* Suppresses displaying what the gauge is attached to.

Returns: NIL

(← *self* **ChangeFont** *newFont*) [Method of Gauge]

Purpose/Behavior: Sets the gauge's instance variable **font** to *newFont* and updates the gauge. If the gauge is too small for *newFont*, it is reshaped.

Arguments: *newFont* A font in which to display the gauge's text.

Returns: Previous value of **font**.

(← *self* **Close**) [Method of Gauge]

Purpose/Behavior: Detaches the gauge and closes the window.

Returns: CLOSED

(← *self* **Destroy**) [Method of Gauge]

Purpose/Behavior: Destroys the gauge, detaching it first before closing the window.

Returns: NIL

(← *self* **Detach**) [Method of Gauge]

Purpose/Behavior: Detaches the gauge from the variable to which it is attached. This prints in an attached window that the gauge is being detached, and deletes all of the links connecting the gauge, active value, and object being monitored. Does not close the window.

Returns: NIL

(← *self* **Reset** *newReading*) [Method of Gauge]

Purpose/Behavior: Sets the gauge's instance variable **reading** to *newReading* and updates the gauge. If the gauge is too small for *newReading* and it is SelfScaling, it is reshaped.

Arguments: *newReading*
Sets the instance variable **reading** to *newReading*, and updates the gauge without going through any intermediate steps.

Returns: NIL if gauge is **AlphaNumeric** or **RoundScale**; otherwise *self*.

Specializations: **Alphanumeric.Reset**, **RoundScale.Reset**

Example: The following example causes the LCD to be redisplayed with the *newReading*:

```
13_(_ ($ lcd1)  Reset   "New Title")
```

(← *self* **SetScale** *min max*) [Method of Gauge]

Purpose/Behavior: Sets the scale for the gauge; computes the new scale values and redisplays if necessary.

Arguments: *min* Lowest value on scale.

max Highest value on scale.

Returns: *self*

(← *self* **Shape** *newRegion noUpdateFlg*) [Method of Gauge]

Purpose/Behavior: If *newRegion* is NIL, you are prompted to sweep out a region which has a minimum sized based upon a **min** property of **IV width** and **height:,min**. If *newRegion* is non-NIL, it is first checked to guarantee that it is at least as large as **width:,min** by **height:,min**.

Arguments: *newregion* List specifying the external coordinates of the window in which the gauge is displayed; list is of the form (left, bottom, width, height).

noUpdateFlg
If NIL, reshapes the gauge.

Returns: NIL

Specializes: Window

Specializations: **LCD**, **Meter**, **DigiMeter**. **Meter.Shape** has an extra argument *ExtraSpaceFlg*. If T, this will allow you to shape a fairly arbitrary region for the gauge; if NIL, the meter is constrained to be close to a square. This latter behavior is what the user sees when trying to shape the meter from the window menu.

BarChart, HBarChart , and their subclasses can only be freely **Shape**d in the direction their bars run (i.e., **BarChart**s can be **Shape**d vertically and **HBarChart**s can be **Shape**d horizontally). Their size along the other dimension is fixed by the number of values attached to the chart .

Example: This example reshapes the gauge to a location where the lower left corner is at (10,100) a width of 50 and a height of 150.

```
14_(_ ($ lcd1) Shape '(10 100 50 150))
```

(← *self* **ShapeToHold**) [Method of Gauge]

Purpose/Behavior: Shapes the gauge window to its smallest possible size based on **width:,min** and **height:,min** and redisplays the gauge.

Returns: NIL

Specializations: **LCD.Shape**

(← *self* **Update**) [Method of Gauge]

Purpose/Behavior: Reinitializes the gauge and its display window to reflect the current state.

Returns: *self*

Categories: Window

Examples

The typical use pattern for a gauge is to first create it, set the scale to the appropriate value, and attach it to the desired data.

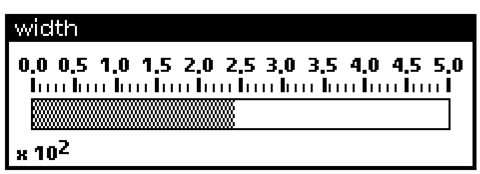
To attach a horizontal scale to a Xerox LOOPS window, **w1**, first enter

```
15_(_ ($ Window) New 'w1)
#,($& HorizontalScale (|OZW0.1Y:.;h.Qm:| . 495))


16_(_ ($ HorizontalScale) New 'hs1)
#,($& HorizontalScale (|OZW0.1Y:.;h.Qm:| . 496))


17_(_ ($ hs1) SetScale 0 500)
NIL
```

Now make the connection.

```
18_(_ ($ hs1) Attach ($ w1) 'width GRAYSHADE)
#,($& HorizontalScale (|OZW0.1Y:.;h.Qm:| . 496))
```

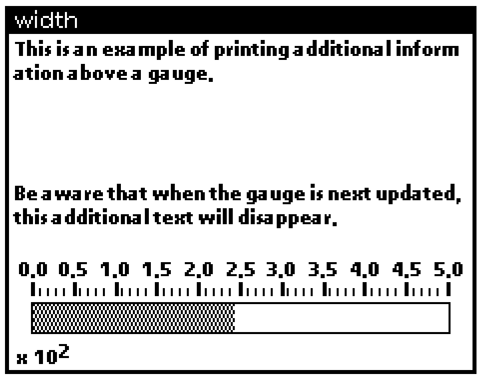
The following gauge appears and you are prompted to place it .



The title of the gauge shows the instance variable being monitored.

Gauges can be shaped larger. The graphics used to display scales do not change; extra white space is added to the top or right. You can use this space to print additional information, as follows:

```
19_(MOVETOUPPERLEFT (@ ($ hs1) window))
{WINDOW}#372,7104

20_(PRIN1 "This is an example of printing additional information above a gauge.



Be aware that when the gauge is next updated, this additional text will disappear." (@ ($ hs1) window))
"This is an example of printing additional information above a gauge.



Be aware that when the gauge is next updated, this additional text will disappear."
```



Limitations

When a font is changed, a gauge occasionally needs to be updated to be correctly displayed.

Instruments can have only floating point numbers for labels, and cannot have integers.

[This page intentionally left blank]

Description/Introduction

Masterscope has been modified to provide for analysis of files created under the Koto or Lyric/Medley release of Xerox LOOPS. A full explanation of Masterscope can be found in the *Xerox Lisp Library Modules Manual, Lyric and Medley Releases.* In addition to the relations explained there, Xerox LOOPS defines the relations described in this chapter.

Note: Masterscope data base files created under Buttress Loops will not function properly in this release. Those data base files will have to be recreated.

Installation/Loading Instructions

- Load MASTERSCOPE from your Lyric/Medley library floppies according to its loading instructions. This should load the compiled files MASTERSCOPE, MSANALYZE, and MSPARSE.
- Load LOOPSMS.DFASL from wherever you installed the Xerox LOOPS Library Modules. This should load versions of MASTERSCOPE and MSPARSE that extend Masterscope to handle Xerox LOOPS constructs.

Relations

Xerox LOOPS defines the following relations:

Name	Type	Description
SEND	Relation	Collects all places where the method is sent.
SEND SELF	Relation	Collects all places where the method is sent to *self*.
SEND NOTSELF	Relation	Collects all places where the method is sent to an object other than *self*.
GET	Relation	Locates all places where the value of an instance variable is retrieved.
GET CV	Relation	Locates all places where the value of a class variable is retrieved.
PUT	Relation	Locates all places where the value of an instance variable is set.
PUT CV	Relation	Locates all places where the value of a class variable is set.
IMPLEMENT	Relation	Locates all methods that specialize the given selector.
SPECIALIZE	Relation	Locates all methods that specialize the given selector and use **_Super** in the body of the method.

OVERRIDE	Relation	Locates all methods that specialize the given selector and do not use _**Super** in the body of the method.
USE IV	Relation	Used with an instance variable name to locate all places where the instance variable is used in a **GET** or **PUT.**
USE CV	Relation	Used with a class variable name to locate all places where the class variable is used in a **GET** or **PUT.**
USE OBJECT	Relation	Used with an object name to locate all places where the object is used**.**

SEND [Relation]

Purpose/Behavior: Used between method names and selectors to collect all places where the method is sent. For example, the form

```
. WHO IS SENT BY 'Helicopter.Move
```

works, but

```
. WHO IS SENT BY Move
```

does not work.

Example: The following command allows you to edit all code that sends the message **New**.

```
. EDIT ALL WHO SEND New
```

SEND SELF [Relation]

Purpose/Behavior: Used between method names and selectors to collect all places where the method is sent to *self*. Places where

(← *self methodName*)

is found are collected, while places where

(← *otherInstance methodName*)

is found are not.

Example: The following command allows you to edit all code that sends the message **Clear** to self.

```
. WHO SENDS SELF Clear
```

SEND NOTSELF [Relation]

Purpose: Same as **SEND SELF**, except the only places where the message is sent to an object other than *self*.

Example: The following allows you to edit all code that sends the message **Clear** to any instance other than *self*.

```
. SHOW ALL WHO SEND NOTSELF Clear
```

GET [Relation]

Purpose: Used with an instance variable name to locate all places where the value of the instance variable is retrieved. This relation can be used along with the **SELF** and **NOTSELF** modifiers.

Example: This command allows you to edit all code that gets the value of the instance variable **width** from an instance other than self and the value of the instance variable **height** from *self*.

```
. SHOW ALL WHO GET NOTSELF width AND GET SELF height
```

GET CV [Relation]

Purpose: Same as **GET**, except that **GET CV** locates places where the value of the class variable is retrieved. This relation can be used with the **SELF** and **NOTSELF** modifiers.

Example: This command allows you to edit all code that accesses the value of the class variable **height** of *self*.

```
. SHOW ALL WHO GET CVSELF height
```

PUT [Relation]

Purpose: Used with an instance variable name to locate all places where the value of the instance variable is set. This relation can be used along with the **SELF** and **NOTSELF** modifiers.

Example: This command allows you to edit all code that sets the value of the instance variable **width.**

```
. EDIT ANY WHO PUT width
```

PUT CV [Relation]

Purpose: Same as **PUT**, except locates places where a specified class variable is set. This relation can be used along with the **SELF** and **NOTSELF** modifiers.

Example: This command list all the sections of code that set the value of the class variable **width** for an instance other than *self*.

```
. WHO PUTS CV NOTSELF width
```

IMPLEMENT [Relation]

Purpose: Used with a method name to locate all methods that specialize the given selector.

Example: This returns a list of classes where the method **Clear** is defined.

```
. WHO IMPLEMENTS Clear
```

SPECIALIZE [Relation]

Purpose: Used with a method name to locate all methods that specialize the given selector and use **_Super** in the body of the method.

Example: This command allows you to edit all the methods that are specializations of **Clear** and use the **_Super** form.

```
. EDIT ANY WHO SPECIALIZE Clear
```

OVERRIDE [Relation]

Purpose: Like **SPECIALIZE** above, except it locates all methods that specialize the given selector and **_Super** is not used in the body of the method.

Example: This command allows you to edit all the specializations of **Clear** that do not make use of the **_Super** form.

```
. EDIT ALL WHO OVERRIDE Clear
```

USE IV [Relation]

Purpose: Used with an instance variable name to locate all places where the instance variable is used in a **Get** or **Put**. It is equivalent to using the relation form of **GET IVName** or **PUT IVName**.

Example: This command allows you to edit all code that either sets or accesses the instance variable **width**.

```
. EDIT ANY WHO USE THE IV width.
```

USE CV [Relation]

Purpose: Used with a class variable name to locate all places where the class variable is used in a **Get** or **Put**. It is equivalent to using the relation form: **GET CV CVName OR PUT CV CVName**.

Example: This command allows you to edit all code where the class variable **commonWindow** is either set or accessed.

```
. EDIT ANY WHO USE THE CV commonWindow
```

USE OBJECT [Relation]

Purpose Uses an object name to locate all places where the object is used.

Example This command returns a list of all code where the object **Window** is used.

```
. WHO USES THE OBJECT Window??
```

Limitations

Masterscope has several limitations:

- Names of methods must be quoted when used with Masterscope; for example, the method name Helicopter.Move must be entered as 'Helicopter.Move.
- The following expression will not find a call to **GetValue** when in a method:

```
. WHO CALLS GetValue
```

Masterscope does not record calls to **GetValue** and **PutValue** explicitly; it records them under the Get- relation along with calls of the form

```
(_ foo Get 'bar)
```

Similarly, the following functions are recorded under relations instead of their names:

GetClassValue	Get CV
PutClassValue	Put CV
GetClassIV	Get IV
PutClassIV	Put IV

If you want to find the explicit calls to Get/PutValue, use

```
. WHO GETS ANY AND NOT SENDS Get
```

• [redacted] similar accessors are accessing instance variables; i.e.,

```
(GetValue foo 'bar)
```

records an access to the instance variable **bar**. This is not necessarily the case; **bar** could also be a class variable.

• The methods and functions that create class and instance variables populate the appropriate **PUT NOTSELF** relations. For example, a function that does

```
(_($ foo) AddCV 'bar)
```

will be found by the query

```
. WHO PUTS CV NOTSELF 'bar
```

An exception occurs with the generalized **Add** and **Delete** method. For example,

```
($ foo) Add 'IV 'bar)
```

will not be noticed as accessing the instance variable **bar**.

Also, the templates for methods and functions that accept property lists generally only notice the first property. For example,

```
((_($ foo) NewWithValues '((bar baz chain link sausage)))
```

notices **baz** as a property, not a link.

[This page intentionally left blank]

Writer's Notes -- Production Details

This file includes notes on the production of *Xerox LOOPS Library Modules Manual*, Lyric Beta Release. This manual is packaged with the *Xerox LOOPS Release Notes* and *Xerox LOOPS Reference Manual* to form one binder.

Writer: Raven Kontur Brewster

Printing Date: >>DA<< >>MO<< 1988

Files Needed

To edit or print the manual, make sure you have the following files loaded:

IMTOOLS
SKETCH
GRAPHER

Fonts Used

{ERIS}<LISP>FONTS>

Modern font
- 18-point bold
- 14-point bold
- 12-point bold
- 10-point regular
- 10-point italic
- 10-point bold

Terminal font
- 10-point regular

Printing Information

The manual was printed under a Lyric sysout on the Tsunami printer.

Artwork

- The cover page for the binder is in the file {ERIS}<Doc>Loops>Lyric>Beta>BinderCover.tedit.

Special Notes and Cautions

If you bring the file into a TEdit window to print it, you must first make sure your underscore character is redefined as a left arrow. See the file on conventions for details. This restriction does not apply if you use the Hardcopy option from the File Browser.

Description/Introduction

In many knowledge-based systems, it is useful to represent knowledge as interconnected sets of instances. A virtual copy mechanism allows a network of instances to be viewed as a prototype which can be copied. The copy of the prototype is virtual in that the contents of each instance is not completely copied at creation time. Instead, it inherits default values from the prototype (also called the original), thus continuing to share the parts not modified in the copy. The copied network is virtual also in the sense that only those instances needed in the processing are copied.

A virtual copy of an object in the prototype network has the following properties:

- It responds to at least the same set of messages as the prototype object and in the same way; that is, a copy has the same procedural behavior that is defined for the prototype.
- A copy inherits variables and their values from the prototype, and continues to do so until an explicit change is made in the copy. At that point, the new value is stored in the copy and it stops tracking the prototype for that variable. A fetch operation on a value that is not stored locally either finds or creates a virtual copy of the value obtained from the prototype.

Installation/Loading Instructions

The implementation of virtual copies is contained in the file LOOPSVCOPY.LCOM. No other files are necessary.

Application /Module Functionality

A network of instances is tied together through the values of instance variables within each of the instances. Assume an object A has an instance variable x, the value of which is the object B. A virtual copy of A will also have an instance variable named x. The value of x in the copy will point to B if B is a shared object, or x may point to a copy of B if it is to be virtual. Changing the value of x in the copy will not change the value in the original.

Overview of Operation

By default, virtual copies share instance variables. This means that changing the value of an instance variable in the original will be tracked by the copy.

Virtual copies are implemented with two additional classes:

- **VirtualCopyMixin**

  The class **VirtualCopyMixin** is a subclass of Tofu which contains two instance variables:

- **% copyMap%**
- **% copyOf%**

(These unusual names are used to avoid conflicts with any other instance variable names users may create.) This class contains several methods, most of which are required to implement virtual copies and are not used by a programmer.

Printing a virtual copy instance is a specialization of how regular instances are printed. All instances print as #,($& <class-name> UID). The class of a virtual copy is a dynamic mixin of the class **VirtualCopyMixin** and the class of the original object (see the *Xerox LOOPS Reference Manual* for more information on mixins). The virtual copy print function adds the name or unique identifier (UID) of the original object. For example,

#,($& (VirtualCopyMixin Container1) (JFW0.0X:.aF4.R>8 . 3) c1)

is a copy of the object named **c1**.

- **VirtualCopyContext**

The class **VirtualCopyContext** has no methods and only one instance variable, **copyMap**. Instances are used as an argument for calls to **MakeVirtualMixin**.

Since copies can be made of copies, you often need to determine the original object of a chain of copies with the **UltimateOriginal** function.

Operands

This section describes the functions, methods, class variables, and instance variables that operate on virtual copies.

VirtualIVs [Class Variable]

Purpose/Behavior: Helps specify a class whose instances may be made into virtual copies. The value of this class variable should be either the symbol ALL, or a list of instance variables contained within instances of the class. If the value is ALL, all objects pointed to by any of the instance variables will be copied. If the value is a list of instance variables, only the instance variables on this list will have their values copied. Other instance variable values will be shared between the copy and the original.

(**MakeVirtualMixin** *x copyContextObj*) [Function]

Purpose: Creates a virtual copy of an object.

Behavior: Creates a dynamic mixin class combining the classes **VirtualCopyMixin** and the class of *x*. An instance of this resulting class is created and it is returned.

Arguments: *x* — An object to be copied; must have the class variable **VirtualIVs** as described above.

copyContextObj — Usually NIL; used internally by **MakeVirtualMixin** when it calls itself. It can be an instance of **VirtualCopyContext** if you are creating an instance that is intended to be part of a currently existing network of copies starting from another entry point. See description in **Limitations** below for a further explanation of this point.

Returns: An object that is a copy of *x*.

Example: Refer to the section, "Example."

% copyMap% [Instance Variable of VirtualCopyMixin]

Purpose/Behavior: A mapping of original nodes (which are objects) in a network to the copied nodes. This map is stored in an instance of the class **VirtualCopyContext.**

% copyOf% [Instance Variable of VirtualCopyMixin]

Purpose/Behavior: Within an instance that is a copy, the value of this instance variable is a pointer to the object that was copied.

(← self **VirtualCopy?**) [Method of VirtualCopyMixin]

Purpose: Determines if an object is a virtual copy.

Returns: *self*

Categories: Object, VirtualCopyMixin

copyMap [Instance Variable of VirtualCopyContext]

Purpose/Behavior: The value of this instance variable is a list of dotted pairs. The CAR of each pair is the original; the CDR, the copy.

(**UltimateOriginal** *self*) [Function]

Purpose: Determines what an object is ultimately copying.

Behavior: If *self* is not a virtual copy, *self* is returned.

If *self* is a virtual copy, this recurses through the value of the instance variable **% copyOf%** until it finds the original and returns it.

Arguments: *self* A Xerox LOOPS object.

Returns: *self* or what is at the top of *self*'s copy chain.

Example

Create a class called **test** and edit it as shown.

```
44_(_($ Class) New 'test)
#,($C test)

45_(ED 'test)
```

```
SEdit #,($C test) Package: INTERLISP
((MetaClass Class Edited%:                    ; Edited 11-Dec-87
                                              ; 16:50 by
  )
 (Supers Object)
 (ClassVariables (VirtualIVs (atomCopy listCopy objCopy)))
 (InstanceVariables (atom NIL)
        (atomCopy NIL)
        (list NIL)
        (listCopy NIL)
        (obj NIL)
        (objCopy NIL))
 (MethodFns))
```

Create an instance called **t0** of this class and inspect it.

```
46_(_($ test) New 't0)
#,($& test (N↑W0.1Y%:.;h.Lh9 . 556))


47_(_ ($ test)
      NewWithValues
      (BQUOTE ((atom 1)
                  (atomCopy 2)
                  (list (a b c))
                  (listCopy (A B (\, (_ ($ test) New (QUOTE t1)))))
                  (obj (\, (_ ($ test) New (QUOTE t2))))
                  (objCopy (\, (_ ($ test) New (QUOTE t3)))))))
#,($& test (N↑W0.1Y%:.;h.Lh9 . 560))

48_(_ IT SetName 't0)
#,($& test (N↑W0.1Y%:.;h.Lh9 . 560))

49_(INSPECT IT]
{WINDOW}#52,51234
```

```
All Values of test ($ t0).
atom      1
atomCopy  2
list      (a b c)
listCopy  (A B #,($ t1))
obj       #,($ t2)
objCopy   #,($ t3)
```

Make a copy called **t0copy** and inspect it.

```
57_(_ (MakeVirtualMixin ($ t0))
   SetName
   (QUOTE t0copy))
#,($& (VirtualCopyMixin test) N↑W0.1Y%:.;h.Lh9 . 562)
```

```
58_(INSPECT IT)
{WINDOW}#53,10150
```

```
All Values of (VirtualCopyMixin test) ($ t0copy).

atom          NIL
atomCopy      NIL
list          NIL
listCopy      NIL
obj           NIL
objCopy       NIL
| copyOf |    #,($ t0)
| copyMap |   #,($& VirtualCopyContext (N↑W0.1Y%:.;h.Lh9 . 563))
```

Make the following changes to **t0** and then reinspect **t0copy**.

```
60_(for iv in '(atom atomCopy list listCopy obj objCopy)
as val in (LIST 11 22 '(a b c d) '(A B C) ($ t3) ($ t1))
do (PutValue ($ t0) iv val]
NIL

61_ (INSPECT IT)
{WINDOW}#53,10152
```

```
All Values of (VirtualCopyMixin test) ($ t0copy).
atom          11
atomCopy      22
list          (a b c d)
listCopy      (A B C)
obj           #,($ t3)
objCopy       #.($& (VirtualCopyMixin test) (N↑W0.1Y%:.;h.Lh9 . 565) t1)
| copyOf |    #,($ t0)
| copyMap |   #,($& VirtualCopyContext (N↑W0.1Y%:.;h.Lh9 . 566))
```

The copied instance variables have not changed since they do not track changes in the original object.

Limitations

Some subtle issues are involved in building and using prototype structures so that the structure is preserved in the copied network. These involve how the network is typically traversed.

A general constraint is that all the links to any shared node in the prototype either all be marked as virtual variables, or none of them are. If they are all marked, then a single copy will be made and used. If none are, then the original object from the prototype will be used. Sharing with the prototype can be useful if this object is a repository for standard information that is independent of context. However, if this constraint is violated, the topology of the virtual copy will be different from that of the prototype.

In the simplest situation the network has a single entry node. In this case, a copy-map (see the section "Operands") can be created when the entry node object is first copied. After that all values are copied using this copy-map. The mechanism works well in this situation, even if there is sharing and there are cycles within the network.

At the other extreme, networks can have arbitrary connectivity, including multiple entries from outside the network, for example, from other networks or non-objects. In this case, the following constraints are necessary to ensure correctness of the virtual copy mechanism.

The first constraint states that all access to the network must start through a copy of one of the nodes in the prototype. This condition is necessary because the criteria for copying are contained in the links from one object to another, not in the objects themselves, and a shared node could not specify a link to a node to be copied. This constraint ensures that all accesses from the outside will be copied if and only if that object would have been copied because of an internal link. Otherwise, an analogous situation would occur in which you could either reach a copy or the original node of the prototype itself depending upon which path you follow when the paths lead to the same node in the prototype.

The final constraint requires that all entries to the network should be passed the same copy-map if they are to share structure. The underlying concern in imposing these constraints is that a network be always copied the same way to maintain its topology regardless of where you start.

Suppose you want to make a virtual copy of a virtual copy, that is, to use a virtual copy of a network as a prototype itself. This is very useful if you are using a network to hold the state of a partial design and you want to try two alternative continuations of the design. Some hidden costs are associated with such multiple-level virtual copies.

Suppose further that a network N1 is used as a prototype and you make a virtual copy, N1-VC. Furthermore, N1-VC-VC is defined to be a copy of N1-VC. Values missing from N1-VC-VC are found in the corresponding object of N1-VC. If the value is missing there, the process recurs, and N1 is examined. If the value is to be a virtual copy, then this process will add a virtual copy in N1-VC, and then a second level copy in N1-VC-VC. This is necessary to preserve the semantics presented, but implies that many levels of virtual copy cannot easily do inexpensive incremental searches of a network.

References

Mittal, S. , Bobrow, D. G., and Kahn, K. *Virtual Copies, Between Classes and Instances*. ACM OOPLSA-86 Conference Proceedings, Portland, Oregon, 1986.

This directory contains all the documentation for Xerox Loops in all its various incarnations.
Directories are as follows:

{ERIS}<Doc>LOOPS> - most general information
{ERIS}<Doc>loops>ProductionSpecs> - specs for all versions of Koto LOOPS

{ERIS}<Doc>Loops>Koto>Final - Product Release for Koto LOOPS (Oct 87)
{ERIS}<Doc>Loops>Lyric>Alpha - Alpha Lyric LOOPS (Jan 88)

Most directories have the following subdirectories:
Ref> - Reference Manual
RelNote> - Release Notes
LibMod> - Library Modules Manual (Lyric Release)
LibPkg> - Library Packages Manual (Koto Release)
UserMod> - Users' Modules Manual (Lyric Release)
UserPkg> - Users' Packages Manual (Koto Release)

And even further in the diectory maze,
X-Index> - has IMPTR files and resulting index
Z-ReleaseInfo> - holds more details on conventions and production details.

Enjoy,
Raven Brewster

APPENDIX A. SUN INSTALLATION PROCEDURE

This appendix describes how to install the LOOPS System files, Library Modules files, and Users' Modules files on the Sun Workstation.

Overview of the Distribution Kit

The distribution kit for LOOPS on the Sun consists of a single ¼-inch tape cartridge. It contains the complete release in "tar" format and creates appropriate directories when its contents are extracted.

Preparation

Preparing to install LOOPS requires that the Medley release of Lisp is already installed and that adequate file space is available.

Before installing LOOPS, remember that

- the Medley 1.0, 1.1 or 1.2 release of Lisp must already be installed on your Sun Workstation;
- the complete LOOPS distribution requires about 1.2 MBytes of file space.

Installation

The software installation procedure shows the steps required for installing the Medley LOOPS software on a Sun Workstation with Medley already installed. Examples are given where appropriate. Only those users who are system administrators and have **root** privileges can install the LOOPS, Medley release.

Before starting software installation, remember that the LOOPS software requires about 1.2 MBytes of file space.

1. Log in under your username.

   `login` **`yourname`**

   `prompt%`

   where **`yourname`** is replaced by your username.

2. Check for available space with the **df** command:

   prompt% **df**

Filesystem	kbytes	used	avail	capacity	Mounted on
/dev/xy0a	7437	5470	1223	82%	/
/dev/xy0h	148455	4900	128709	96%	/usr/misc

3. Determine if you need to run **su** to make a directory for the distribution. If so, type in **su**:

   prompt% **su**

4. Make a directory for the distribution. This directory should be named /usr/local/lde/loops. If you have enough space on the file system containing /usr/local/lde, then

   prompt# **mkdir /usr/local/lde/loops/**

   If you don't have enough space on /usr/local/lde, go to step 6.

5. Make yourself owner of this directory:

   prompt# **/etc/chown yourname /usr/local/lde/loops/**

   where **yourname** is your username.

6. If you don't have space on the file system which contains /usr/local/lde, but do have space somewhere else, for instance on /usr1, then make the directory there and link /usr/local/lde/loops to it:

   prompt# **mkdir /usr1/loops**

   prompt# **/etc/chown yourname /usr/usr1/loops**

   prompt# **ln -s /usr1/loops /usr/local/lde/loops**

7. If you ran **su**, leave the privileged shell by typing:

   prompt% **exit**

8. Insert the ¼-inch cartridge tape, containing the LOOPS software, in the drive.

9. Connect to /usr/local/lde/loops:

   prompt# **cd /usr/local/lde/loops**

10. Load the Medley software from tape. Indicate the appropriate device abbreviation for your tape by replacing **xx** in the example below with

    ar for the Archive drive,

    st for a SCSI tape drive.

    This example shows the command entry sequence:

    prompt# **tar xvpf /dev/rxx0**

    As the software is loaded (a process that takes some time) the system prints a series of lines in the following form:

    x ./system/LOOPS., 28552 bytes, 56 tape blocks

The x at the beginning of the line indicates that the file is being extracted from the tape.

This creates directories named:

/usr/local/lde/loops/system/

/usr/local/lde/loops/library/

/usr/local/lde/loops/users/

This is a good time to set the protection of the extracted directories and files so that the work group using LOOPS has at least read access to them.

11. Boot Medley Lisp.

12. Open an Interlisp Executive Window.

13. Make certain the time is set correctly.

14. Set the **DIRECTORIES** and **DISPLAYFONTDIRECTORIES** variables appropriately so the sysout can find your Lyric/Medley library and font files.

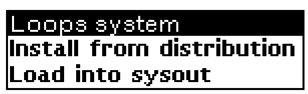
15. Make the LOOPS System directory your connected directory:

    ```
    CONN {DSK}/usr/local/lde/loops/system/
    ```

16. Enter the following into your Exec:

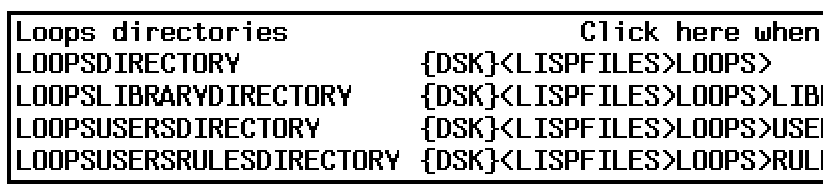
    ```
    LOAD(LOOPS)
    ```

    A menu appears that looks like this:



17. Select the menu option **Install from distribution**.

    The following menu appears:



This menu shows the current (or default, if unset) values of the variables LOOPS examines when it loads things.

If you have installed LOOPS under /usr/local/lde/loops/ click the mouse on the menu items to set these directories to point where the tape was unloaded:

LOOPSDIRECTORY {dsk}/usr/local/lde/loops/system/
LOOPSLIBRARYDIRECTORY {dsk}/usr/local/lde/loops/library/
LOOPSUSERSDIRECTORY {dsk}/usr/local/lde/loops/users/
LOOPSUSERSRULESDIRECTORY {dsk}/usr/local/lde/loops/users/

As the last installation step, the installation tool automatically modifies the file LOOPSSITE, writes it out to the vaule of the variable **LOOPSDIRECTORY**, and compiles it.

When this step is finished, the first menu reappears:

```
Loops system
Install from distribution
Load into sysout
```

18. Select the menu option **Load into sysout** to load LOOPS into your system. The following menu appears:

```
Load Which?
Loops
Loops Masterscope
Gauges
LoopsBackwards
VirtualCopy
```

19. Select **LOOPS** from the menu to load LOOPS from the location where you installed it.

    Once LOOPS is loaded the LOOPS System menu reappears. To load one of the other LOOPS library or Users' modules, select the appropriate name in the Load Which? menu.

20. Position your mouse cursor anywhere on the screen except for the Load Which? menu, then press the left mouse button to exit the installation procedure.

Medley LOOPS is now installed on your Sun Workstation.

Loading After Installation

This section describes how to reload LOOPS into a newly started Lisp sysout after LOOPS has been previously installed.

1. Start up Medley on your Sun Workstation.
2. Open an INTERLISP Exec window.
3. Make sure **DIRECTORIES** points to a directory containing GRAPHER.LCOM, and **DISPLAYFONTDIRECTORIES** points to a directory containing the Helvetica display font files from your Lisp distribution kit.
4. Connect to the directory containing the LOOPS system files:

    ```
    (CNDIR '{DSK}/USR/LOCAL/LDE/LOOPS/SYSTEM/)
    ```

5. Load LOOPS loader program:

    ```
    (FILESLOAD LOADLOOPS)
    ```

6. Run the LOOPS loader program:

    ```
    (LOADLOOPS)
    ```

This procedure loads only the LOOPS system files. Please see the manuals describing the LOOPS Library and Users' Modules for their loading procedures.

CAUTION

LOOPS uses the new compiler and its macrolet facilities. When LOOPS is loaded, it sets your ***DEFAULT-CLEANUP-COMPILER*** to **'CL:COMPILE-FILE**. More information on this cleanup flag and the new compiler is available in the *Lisp Release Notes*, in your Medley Lisp kit.

[This page intentionally left blank]

DOCUMENT UPDATE SHEET

Document Name: ***LOOPS Manual***

Document Number: ***310000***

DOC. VERSION	RELEASE DATE	REPLACE PAGES	INSERT PAGES	INSTRUCTIONS/ NOTES
Lyric/Medley	Oct., 1988	NA	NA	Please read the Errata Sheet, accompanying this release material, for last minute release notes.
Lyric/Medley	Oct., 1988	NA	NA	The Lyric/Medley LOOPS documentation contains numerous references to Xerox LOOPS. Xerox LOOPS is now known as Envos LOOPS.
Medley	Oct., 1988	NA	A-1-A-4	Add *Appendix A, Sun Installation Procedure,* to your *LOOPS Release Notes.*

BACK COVER

FONT

Xerox Corporation
AIS Administration
250 North Halstead St.
5910-432
800-824-6449 (Calif.)
800-228-5325 (U.S.)
Pasadena, California 91107

E

REGULAR or
TRIUMVIRAT
BLACK
ITALIC

COVER TEXT:

NOTE: USE APPROPIRATE POINT SIZES TO ACHIEVE EFFECT
SHOWN BELOW. USE XEROX STANDARDS FOR <<8-1/2x11>> BINDER COVERS

FONT

ARTIFICIAL INTELLIGENCE SYSTEMS

XEROX LOOPS

HELVETICA BOLD ITALIC

RELEASE NOTES
REFERENCE MANUAL
LIBRARY PACKAGES MANUAL

REGULAR or TRIUMVIRATE BLACK ITALIC

RELEASE 1.0

SPINE (3 INCH):

ARTIFICIAL INTELLIGENCE SYSTEMS

XEROX LOOPS

HELVETICA BOLD ITALIC

RELEASE NOTES
REFERENCE MANUAL
LIBRARY PACKAGES MANUAL

REGULAR or TRIUMVIRATE BLACK ITALIC

RELEASE 1.0

Subject: Software Configuration: Xerox LOOPS Documentation, Release 1.0
To: Fournier.pasa

I. This is a request to configure for production the artwork for the following Documentation Kit.

Kit: Xerox LOOPS, Release 1.0

For AICCB use only. Xerox Part Number: >>12R number<<
Internal Part Number: >>our number<<

DOCUMENTATION SET

# of Books in Set: 4

Volume Names: *Xerox LOOPS Reference Manual*
Xerox LOOPS Release Notes
Xerox LOOPS Library Packages Manual
Xerox LOOPS Users' Packages Manual

II. This is a request for artwork for the Documentation Kit.

BINDING

Volume Names: *Xerox LOOPS Release Notes*
Xerox LOOPS Reference Manual
Xerox LOOPS Library Packages Manual

Book Size: 8-1/2 X 11

Binder Size: 3 inch

Binding Type: vinyl

Book Cover Material:

Other Binding Specifications: plastic sleeves for slipin on front, spine, and back

Binder Rings:

Sheet Lifters: yes

Inside Pocket:

Binder Colors: PMS Gray 422-C

Special Instructions: Bind all manuals in one binder

BINDING

Volume Names: *Xerox LOOPS Users' Packages Manual*

Book Size: 8-1/2 X 11

Binder Size: 1 inch

Binding Type: vinyl

Book Cover Material:

Other Binding Specifications: plastic sleeves for slipin on front, spine, and back

Binder Rings:

Sheet Lifters:

Inside Pocket:

Binder Colors: PMS Gray 422-C

Special Instructions:

Please see the following files for actual cover artwork specifications:

{ERINYES}<doc>LoopsRef>ProductionSpecs>BackCover.Sketch;2
{ERINYES}<doc>LOOPSREF>PRODUCTIONSPECS>ComboCoverSpine.sketch;1
{ERINYES}<doc>LOOPSREF>PRODUCTIONSPECS>UsersCoverSpine.sketch;2

TABS

Tab Set Part Number: >>For AICCB use: Part Number<<

Please see the following files for the tab specifications:

{ERINYES}<doc>LOOPSREF>PRODUCTIONSPECS>TabSample1a.Sketch;3
{ERINYES}<doc>LOOPSREF>PRODUCTIONSPECS>TabSample1B.Sketch;6
{ERINYES}<doc>LOOPSREF>PRODUCTIONSPECS>TABSAMPLE1C.Sketch;2
{ERINYES}<doc>LOOPSREF>PRODUCTIONSPECS>TabSample2A.sketch;3
{ERINYES}<doc>LOOPSREF>PRODUCTIONSPECS>TabSample2B.sketch;3

COVER TEXT:

FONT

NOTE: USE APPROPRIATE POINT SIZES TO ACHIEVE EFFECT
SHOWN BELOW. USE XEROX STANDARDS FOR <<8-1/2x11>> BINDER COVERS

ARTIFICIAL INTELLIGENCE SYSTEMS

XEROX LOOPS

HELVETICA
BOLD ITALIC

XEROX LOOPS RELEASE NOTES
XEROX LOOPS REFERENCE MANUAL
XEROX LOOPS LIBRARY PACKAGES MANUAL

KOTO RELEASE

REGULAR or
TRIUMVIRATE
BLACK
ITALIC

SPINE (3 INCH):

ARTIFICIAL INTELLIGENCE SYSTEMS

HELVETICA
BOLD ITALIC

XEROX LOOPS RELEASE NOTES
XEROX LOOPS REFERENCE MANUAL
XEROX LOOPS LIBRARY PACKAGES MANUAL

XEROX LOOPS

KOTO RELEASE

TAB TEXT

POINT SIZE:8

FONT: OPTIMA

TABS:

XEROX LOOPS RELEASE NOTES

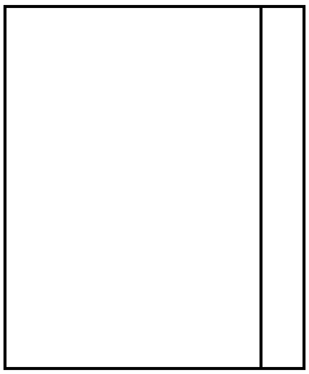
XEROX LOOPS REFERENCE MANUAL

XEROX LOOPS LIBRARY PACKAGES MANUAL

TABS FOR LOOPS (3" BINDER)

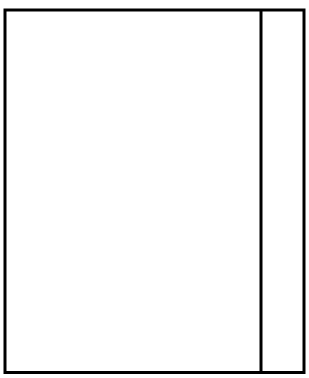
TAB COLOR: PMS GRAY 422-C

BANK 1



XEROX LOOPS RELEASE NOTES

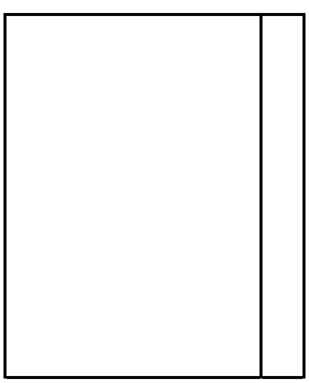
BANK 2



XEROX LOOPS REFERENCE MANUAL

SUBTABS: See following page

BANK 3



XEROX LOOPS LIBRARY PACKAGES MANUAL

PACKAGING

DOCUMENTATION SET

# of Books in Set: 4

Volume Names: *Xerox LOOPS Reference Manual*
Xerox LOOPS Release Notes
Xerox LOOPS Library Packages Manual
Xerox LOOPS Users' Packages Manual

BINDING

Volume Names: *Xerox LOOPS Release Notes*
Xerox LOOPS Reference Manual
Xerox LOOPS Library Packages Manual

Book Size: 8-1/2 X 11

Binder Size: 3 inch

Binding Type:

Book Cover Material:

Other Binding Specifications:

Binder Rings:

Sheet Lifters:

Inside Pocket:

Binder Colors: PMS Gray 422-C

Special Instructions: Bind all manuals in one binder

BINDING

Volume Names: *Xerox LOOPS Users' Packages Manual*

Book Size: 8-1/2 X 11

Binder Size: 1 inch

Binding Type:

Book Cover Material:

Other Binding Specifications:

Binder Rings:

Sheet Lifters:

Inside Pocket:

Binder Colors: PMS Gray 422-C

Special Instructions:

FLOPPY DISKS

Floppy Disk Packaging:

Label Specifications:

Special instructions:

PRINTING

Printing Method:

Paper Weight:

Paper Size:

Exceptions (e.g., oversize diagrams):

Special Instructions:

LOOPS PART NUMBERS

3102466 LOOPS Manual (Reference, Release Notes, Installation Guide)

3102477 LOOPS Users’ Packages

SUBTABS FOR LOOPS (3" BINDER)

COLOR OF TABS: WHITE

BANK 1

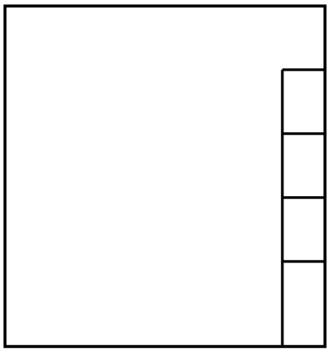
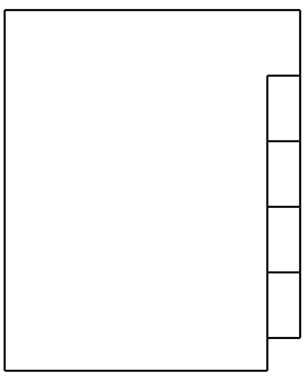


Table of Contents

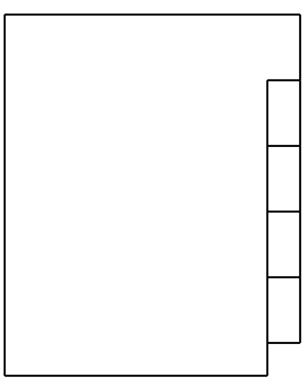
1. Introduction
2. Instances
3. Classes
4. Metaclasses

BANK 2



5. Active Values
6. Methods
7. Message Sending Forms
8. Iterative Statements
9. Miscellaneous

BANK 3



10. Browsers
11. Errors and Breaks
12. Breaking and Tracing
13. Editing
14. File Package

Continued on next page

SUBTAB TEXT

POINT SIZE:

FONT:

SUBTABS:

Table of Contents Metaclasses	1. Introduction	2. Instances	3. Classes	4.
5. Active Values Miscellaneous	6. Methods	7. Message Sending Forms	8. Iterative Statements	9.
10. Browsers File Package	11. Errors and Breaks	12. Breaking and Tracing	13. Editing	14.
15. Masterscope User Input/ Output Packages	16. Performance Issues	17. Processes	18. Reading and Printing	19.
20. LOOPS Windows	A. Previous Active Values	Glossary	Index	

COVER TEXT:

NOTE: USE APPROPRIATE POINT SIZES TO ACHIEVE EFFECT
SHOWN BELOW. USE XEROX STANDARDS FOR <<8-1/2x11>> BINDER COVERS

FONT

ARTIFICIAL INTELLIGENCE SYSTEMS

XEROX LOOPS

HELVETICA
BOLD ITALIC

USERS' PACKAGES MANUAL

KOTO RELEASE

REGULAR or
TRIUMVIRATE
BLACK
ITALIC

SPINE (1 INCH):

ARTIFICIAL INTELLIGENCE SYSTEMS

XEROX LOOPS ***USERS' PACKAGES MANUAL***

KOTO RELEASE

HELVETICA
BOLD ITALIC

REGULAR or
TRIUMVIRATE
BLACK
ITALIC

The two big things about Xerox LOOPS are:

- You have to change your underscore to a left arrow.

- You have to change your FONTPROFILE if you want to any examples of menus.

See the conventions file for details.

{ERIS}<Doc>LOOPS>*>Conventions.tedit

5

Annotated Values and Active Values

In the previous chapters, IVs, CVs, and their properties have been treated as passive entities without structure. ***Annotated values*** are a way of associating behavior and annotations with variables. In keeping with the object oriented programming style of LOOPS, these annotations are objects. Annotation objects are called ***active values***. When a variable containing an annotated value is accessed, a message is sent to the active value. 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 chapter first describes the structure and implementation of annotated values. Functions for explicitly dealing with annotated values are documented. Then the class `ActiveValue` is introduced and the standard protocol for active values is described. Next, the standard subclasses of `ActiveValue` are explained.

5.1. Annotated Values

LOOPS defines a new INTERLISP data type called `annotatedValue`. Each `annotatedValue` contains a single field. This field contains an object, the annotated value's active value. The standard variable access functions described in previous chapters (`GetValue`, `PutValue`, `GetClassValue`, `PutClassValue`) treat values that are annotated values specially. `GetValue` and `GetClassValue` do not return the annotated value. Instead, they send the contained active value a message, and return the result of that message. Similarly, if the current value of a variable is an annotated value, `PutValue` and `PutClassValue` operate by sending the contained active value a message.

`type? annotatedValue` *value* [*Macro*]

Returns true if *value* is an annotated value, false otherwise. This is the standard way to test to see if a value is an annotated value.

`create annotatedValue annotatedValue ←` *object* [*Macro*]

Creates a new annotated value with active value *object*. No checking of *object* is performed.

`fetch annotatedValue of` *value* [*Macro*]

Returns the active value contained in the annotated value *value*. If *value* is not an annotated value, generates an error.

`replace annotatedValue of` *value* `with` *object* [*Macro*]

Replaces the active value contained in the annotated value *value* with *object*. If *value* is not an annotated value, generates an error. No checking of *object* is performed.

`←AV` *av selector* `.` *args* [*Macro*]

`←AV` is a message sending form that can be used with annotated values. `(←AV` *av selector* `.` *args*`)` → `(← (fetch annotatedValue of` *av*`)` *selector* `.` *args*`)`.

`AnnotatedValue` [*Class*]

Sometimes people forget to extract the active value from an annotated value, and they end up trying to use an annotated value as an object. Using the `LispDataType` feature, LOOPS takes care of this for you. Annotated values are considered to belong to the LOOPS class `AnnotatedValue`. If you send a message to an annotated value, the behavior is found in the class `AnnotatedValue`. There, the method for `MessageNotUnderstood` forwards the message off to the contained active value. Similarly, if you attempt to get an IV from an annotated value, the get ends up happening to the wrapped active value.

5.2. The Abstract Class ActiveValue

Active values follow a standard protocol that allow them to be used inside of annotated values.

In the description of methods for active values, the arguments *containingObj*, *varName*, *propName*, and *type* are used to describe the variable containing the active value. *type* is one of `IV`, `CV`, or `NIL` : a *type* of `IV` or `NIL` indicates that the variable is an instance variable or an instance variable property of *containingObj*; a *type* of `CV` indicates a class variable or class variable property of *containingObj*. If *propName* is `NIL`, the variable is either an IV or a CV, otherwise it is an IV or CV property with name *propName*. *containingObj* is the instance or class that contains the variable.

`ActiveValue` [*Abstract class*]

The class `ActiveValue` captures the protocol followed by all active value objects. `ActiveValue` is an abstract class, so you cannot make instances of `ActiveValue`. Specializations of `ActiveValue` need to specialize the `GetWrappedValueOnly` and `PutWrappedValueOnly` methods. Methods that you want to specialize include `AVPrintSource`, `GetWrappedValue`, `PutWrappedValue`, `WrappingPrecedence`, and `CopyActiveValue`.

5.2.1 Displaying Annotated Values

← *self* AVPrintSource [*ActiveValue method*]

An annotated value determines how it will print out by sending the AVPrintSource message to the its active value. This message returns a form suitable for use by the INTERLISP function DEFPRINT. The result should be a pair of the form (*item1* . *item2*). *item1* will be printed using PRIN1, and then *item2* will be printed by PRIN2 (see the IRM description of DEFPRINT for more details).

The default method in ActiveValue returns the list

("#." $AV *className avNames* (*ivName value propName value* ...) (*ivName* ...)...)

which will cause the annotated value to print out as

#.($AV *className avNames* (*ivName value propName value* ...) (*ivName* ...)...).

className is the name of the class of the active value. *avNames* is a list of names of *self*; the last element of *avNames* is the uid of *self*. The lists (*ivName value propName value* ...) describe the state of the IVs of the active value. Note that the uid of the active value is included in the printed form, so the identity of the active value object can be recovered. In this way, different annotated values can share the same active value, and have this sharing maintained across a dump/load-up.

$AV *className avNames* . *ivForms* [*Special Form*]

$AV is used to reconstruct a dumped annotated value. It returns a new annotated value whose active value is reconstructed from the *avNames* and *ivForms*.

5.2.2 Fetching and Replacing Wrapped Values

← *self* GetWrappedValue *containingObj varName propName type* [*ActiveValue method*]

The GetWrappedValue message provides a way to perform arbitrary actions when a variable is read. When GetValue (or GetClassValue) finds an annotated value in an instance, it does not return the annotated value. Instead, it sends the contained active value the GetWrappedValue message and returns the result of this message.

The default method in ActiveValue sends the message GetWrappedValueOnly to *self*. If this value is an annotated value, it is triggered by sending it the GetWrappedValue message, and the result is returned; otherwise the value is returned with no further processing.

← *self* GetWrappedValueOnly [*ActiveValue method*]

Returns the immediate "local state" of the variable that is wrapped by the active value *self*. If this local state is a nested active value, it is not triggered. The default implementation causes an error by calling SubclassResponsibility.

← *self* `PutWrappedValue` *containingObj varName*
newValue propName type [*ActiveValue method*]

The `PutWrappedValue` message provides a way to perform arbitrary actions when a variable is set. When `PutValue` (or `PutClassValue`) attempts to replace an annotated value, it instead sends the contained active value the `PutWrappedValue` message.

The default method in `ActiveValue` checks to see if the current value is a nested active value by sending the GetWrappedValueOnly message to *self*. If the result is an annotated value, `PutWrappedValue` forwards the message on the the nested active value; otherwise it sends the message `PutWrappedValueOnly` to *self* and returns the result.

← *self* `PutWrappedValueOnly` *newValue* [*ActiveValue method*]

Replaces the immediate "local state" of the variable that is wrapped by the active value *self*. The current local state is replaced. If the current value is a nested active value, it is not triggered. The default implementation causes an error by calling `SubclassResponsibility`.

5.2.3 Inheriting Active Values

Typical implementations of `PutWrappedValue` store the new value in the active value. However, if the active value is shared among different instances all these instances would see this change. In particular, if the active value is inherited from the class of the instance, all other instances of the class would see this change. This behavior is usually not desired. The `GetWrappedValue` method of active values is also free to alter the internal state of the active value, causing the same problem. To get around this problem, the annotated value is first copied, and this copy is stored in the instance. The `CopyActiveValue` method implements this copying. When `GetValue` or `PutValue` finds no local value, it first checks to see if the current value is an annotated value inherited from the class. If it is, it sends `CopyActiveValue` to the active value, and stores the result in the instance. The put or get then proceeds.

← *self* `CopyActiveValue` *annotatedValue* [*ActiveValue method*]

annotatedValue is an annotated value that surrounds *self*. `CopyActiveValue` should return a copy of *annotatedValue*, containing a copy of *self*. It is possible, and in some cases desirable, for an implementation of `CopyActiveValue` to return *annotatedValue*.

The default behavior returns a new annotated value wrapped around a copy of *self*. IV values of self are not copied, the values are shared with the copy, except that IVs of self that contain annotated values are copied using the `CopyActiveValue` message.

5.2.4 Adding and Deleting Annotations

← *self* `AddActiveValue` *containingObj varName*
propName type annotatedValue [*ActiveValue method*]

Adds the annotated value *annotatedValue* to the variable specified by *containingObj*, *varName*, *propName*, and *type*. If *annotatedValue* is not specified or is NIL, *annotatedValue* defaults to a newly created annotated value containing the active value *self*. If the variable is already an annotated value, the `AddActiveValue` method uses the `WrappingPrecedence` message (below) to determine if *annotatedValue* should be nested in the current annotated value or wrapped around it. The method returns *annotatedValue*.

← *self* `WrappingPrecedence` [*ActiveValue method*]

Specifies where an annotated value containing *self* should be added to an existing annotated value. `T` means that this active value must go on the outside of any other annotated values. `NIL` means it must go on the inside. A number specifies a precedence: active values with larger `WrappingPrecedence` values go outside ones with smaller `WrappingPrecedence` values. If two active values have the same (numeric) `WrappingPrecedence`, the order is not determined. The default implementation of `WrappingPrecedence` returns `100`.

← *self* `DeleteActiveValue` *containingObj varName propName type* [*ActiveValue method*]

Finds the first annotated value on the variable specified by *containingObj*, *varName*, *propName*, and *type* that has *self* as its active value and deletes it from that variable. Returns that annotated value if one was found, `NIL` otherwise.

← *self* `ReplaceActiveValue` *newVal containingObj*
varName propName type [*ActiveValue method*]

It is sometimes desirable to replace an annotated value in a variable with some new value. (← *self* `ReplaceActiveValue` *newVal containingObj varName propName type*) replaces the annotated value containing *self* in the variable described by *containingObj*, *varName*, *propName*, and *type* with the new value *newVal*.

5.2.4 Manipulating Active Values

Some programs need to explicitly test and trigger active values. The following functions can be used to access IVs and CVs without triggering active values.

`GetValueOnly` *object varName propName* [*Function*]

`GetValueOnly` is the same as `GetValue`, except that `GetValueOnly` does not trigger any active values. `GetValueOnly` returns the immediate value of the variable. If this is not an annotated value, `GetValueOnly` returns the same value as `GetValue`. If there is no local value, the inherited value is returned. See also the function `GetIVHere`.

`GetClassValueOnly` *object varName propName* [*Function*]

`GetClassValueOnly` is the same as `GetClassValue`, except that `GetClassValueOnly` does not trigger any active values. `GetClassValueOnly` returns the immediate value of the variable. If this is not an annotated value, `GetClassValueOnly` returns the same value as `GetClassValue`. *object* can be either an instance or a class.

`ObjRealValue` *object varName value propName type* [*Macro*]

If *value* is not an annotated value returns *value*, otherwise returns the value of (←`AV GetWrappedValue` *object varName propName type*). This macro is used by GetValue and

GetClassValue to trigger active values, and can be used by programs that explicitly test for active values.

`PutValueOnly` *object varName newValue propName* [*Function*]

`PutValueOnly` is the same as `PutValue`, except that `PutValueOnly` does not trigger any active values. `PutValueOnly` replaces the immediate value of the variable with *newValue*, even if the old value is an annotated value.

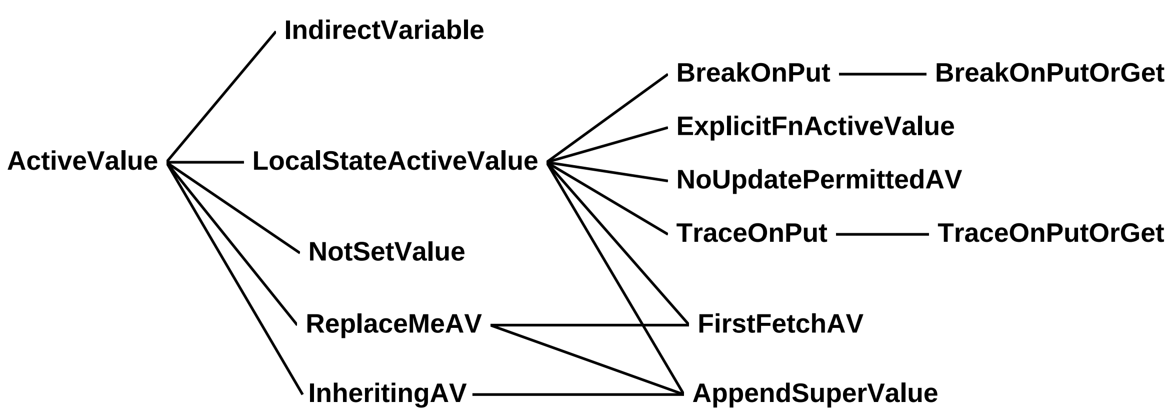
`PutClassValueOnly` *object varName newValue propName* [*Function*]

`PutClassValueOnly` is the same as `PutClassValue`, except that `PutClassValueOnly` does not trigger any active values. `PutClassValueOnly` replaces the immediate value of the variable with *newValue*, even if the old value is an annotated value. *object* can be either an instance or a class.

← *self* `HasAV?` *av* [*ActiveValue method*]

Returns true if the active value (or annotated value) *av* is nested inside in the active value self.

5.3. Specializations of ActiveValue



The ActiveValue Class Hierarchy

5.3.1 NotSetValue and Variable Inheritance

`NotSetValue` [*Variable*]

LOOPS uses annotated values to trigger IV inheritance. When an instance is created, its IVs are initialized to contain (the value of) `NotSetValue`. `NotSetValue` is an annotated value whose active value is the prototype instance of the class `NotSetValue`. The class `NotSetValue` specializes the default `ActiveValue` protocol to trigger IV inheritance. In this way `GetValue` does not need to do any special check to see if a value needs to be inherited — all it needs to do is see if the value is an annotated value. Note that `GetValueOnly` does need to do a special check for `NotSetValue`, but see the function `GetIVHere`.

`NotSetValue` *form* [*Macro*]

Returns true if *form* evaluates to `NotSetValue`, otherwise false. (`NotSetValue` *form*) → `(EQ` *form* `'NotSetValue)`. This is the approved way of testing a value to see if it is `NotSetValue`.

← *self* `AVPrintSource` [*NotSetValue method*]

Returns the pair `("#." . NotSetValue)`. This causes (the value of) `NotSetValue` to print out as `#.NotSetValue`. This will be read in as the value of the variable `NotSetValue`.

← *self* `GetWrappedValue` *containingObj varName propName type* [*NotSetValue method*]

If *type* is `NIL` or `IV`, this evaluates `(←` *containingObj* `IVValueMissing` *varName propName* `'GetValue)` and returns the result; if *type* is `CV`, evaluates `(←` *class* `CVValueMissing` *varName propName* `'GetValue)` (where *class* is the class of *containingObj* if *containingObj* is an instance, else *containingObj* if it is a class) and returns the result; otherwise an error is generated. See the methods `IVValueMissing` and `CVValueMissing` on the class `Object`.

← *self* `PutWrappedValue` *containingObj varName newValue propName type* [*NotSetValue method*]

If *type* is `NIL` or `IV`, this evaluates `(←` *containingObj* `IVValueMissing` *varName propName* `'PutValue` *newValue*`)` and returns the result; if *type* is `CV`, evaluates `(←` *class* `CVValueMissing` *varName propName* `'PutValue` *newValue*`)` (where *class* is the class of *containingObj* if *containingObj* is an instance, else *containingObj* if it is a class) and returns the result; otherwise an error is generated. See the methods `IVValueMissing` and `CVValueMissing` on the class `Object`.

← *self* `CopyActiveValue` *annotatedValue* [*NotSetValue method*]

Returns `#.NotSetValue`. There is only one `NotSetValue`.

← *self* `WrappingPrecedence` [*NotSetValue method*]

Returns `NIL`. `#.NotSetValue` must always be on the inside of any sequence of nested active values.

`GetIVHere` *object varName propName* [*Function*]

If *propName* is `NIL` and there is a local value for the IV *varName* in the instance *object*, that value is returned. If *propName* is not `NIL` and there is a local value for the IV property *propName* of the IV *varName* in the instance *object*, that value is returned. Otherwise, if *propName* is `NIL` `GetIVHere` returns `#.NotSetValue`, and if *propName* is not `NIL` `GetIVHere` returns (the value of) `NoValueFound`.

`GetCVHere` *object varName propName* [*Function*]

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

`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 teh instance variable *varName* in the class *class*. If *varName* is not already local to the class, this will cause an error. Returns *newValue*.

5.3.2. Indirect Variables

In some applications it is important to be able to access values indirectly from other instances. For example, Steele [Steele80] has recommended this as an approach for implementing equality constraints.

`IndirectVariable` [*Class*]

Mumble.

5.3.3. ReplaceMeAV

The active value mixin `ReplaceMeAV` can be used when an active value should be replaced when a variable is first set.

`ReplaceMeAV` [*Abstract class*]

Mumble.

5.3.4. LocalStateActiveValue

Many kinds of active values explicitly store the "real" value of the variable in an IV of the active value.

`LocalStateActiveValue` [*Abstract class*]

Mumble.

5.3.5. InheritingAV

Some kinds of active values want to compute a value based on what would have been inherited if the active value had not been present. For example, it might be desired to append items onto an inherited value (see the class `AppendSuperValue`).

`InheritingAV` [*Abstract class*]

Mumble.

5.3.6. FirstFetchAV

Mumble.

`FirstFetchAV` [*Class*]

Mumble.

5.3.7. Breaking and Tracing Variable Access

Mumble.

`BreakOnPut` [*Class*]

Mumble.

`BreakOnPutOrGet` [*Class*]

Mumble.

`TraceOnPut` [*Class*]

Mumble.

`TraceOnPutOrGet` [*Class*]

Mumble.

`UnBreakIt` *self varName propName type* [*Class*]

Mumble.

5.3.8. NoUpdatePermittedAV

The active value class `NoUpdatePermittedAV` can be used to prevent a value from being updated.

`NoUpdatePermittedAV` [*Class*]

Mumble.

5.3.9. AppendSuperValue

The active value class `AppendSuperValue` can be used to append data to inherited values.

`AppendSuperValue` [*Class*]

Mumble.

5.3.10. ExplicitFnActiveValue

`ExplicitFnActiveValue` [*Class*]

`ExplicitFnActiveValue` explicitly store functions that will be triggered when the variable is fetched or replaced. They have three IVs: `localState`, `getFn`,and `putFn`. The `localState` is the "real" value of the variable (possibly a nested active value), the `getFn` and `putFn` are names of functions that are applied with standard arguments by the GetWrappedValue and PutWrappedValue methods. The `getFn` and `putFn` are called with arguments *containingObj*, *varName*, *oldOrNewValue*, *propName*, *activeValue*, and *type*. `ExplicitFnActiveValue` active values print out as `#.($A` *localState getFn putFn*`)`, where the *localState*, *getFn*, and *putFn* are the values of the corresponding IVs of the active value.

5.4. Compatibility with older versions

The following existed in older versions of LOOPS, which had a different implementation of active values. They are provided for compatibility purposes only. New programs should not use them. They are not fully supported, and will not exist in future releases. The current implementations of these use the new active values. They are fully compatible with the older versions except where noted.

5.4.1. Old Style Active Values

LOOPS used to combine the notion of annotated value and active value. Variable annotations were instances of the INTERLISP datatype `activeValue`.

`activeValue` [*Record*]

In this version of LOOPS, the record `activeValue` is an access record that converts the three fields of the old active values to appropriate functions for accessing annotated values. Forms like `(type? activeValue` *form*`)` and `(fetch localState of` *activeValue*`)` will do the right thing. Reading in old style active values automatically converts them to annotated values wrapping an instance of the class `ExplicitFnActiveValue`.

`GetLocalState` *av self varName propName type* [*Function*]

Works just like in the old LOOPS.

`PutLocalState` *av newValue self varName propName type* [*Function*]

Works just like in the old LOOPS.

`GetLocalStateOnly` *av* [*Function*]

Works just like in the old LOOPS.

`PutLocalStateOnly` *av newValue* [*Function*]

Works just like in the old LOOPS.

`ReplaceActiveValue` *av newVal self varName propName type* [*Function*]

Works just like in the old LOOPS.

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

Works just like in the old LOOPS, except that the interpretation of *newLocalSt* is different. `MakeActiveValue` ignores the value of *newLocalSt*, and always creates a new active value. This is the behavior that the old `MakeActiveValue` produced when *newLocalSt* was `Embed`.

5.4.2. GetFns and PutFns

`DefAVP` *fnName putFlg* [*Function*]

Works just like in the old LOOPS.

`NoUpdatePermitted` *self varname oldOrNewValue propName activeValue type* [*Function*]

Works just like in the old LOOPS.

`FirstFetch` *self varname oldOrNewValue propName activeValue type* [*Function*]

Works just like in the old LOOPS.

`GetIndirect` *self varname oldOrNewValue propName activeValue type* [*Function*]

Works just like in the old LOOPS.

`PutIndirect` *self varname oldOrNewValue propName activeValue type* [*Function*]

Works just like in the old LOOPS.

`ReplaceMe` *self varname oldOrNewValue propName activeValue type* [*Function*]

Works just like in the old LOOPS.

`AtCreation` *self varname oldOrNewValue propName activeValue type* [*Function*]

No longer works. Instead, you can use either the `FirstFetch` function, or the `:initForm` property of IVs.

5.5. Summary of Variable Access Functions

The following tables summarizes the available functions for variable access.

	Inherit/Trigger	Inherit/Don't Trigger	Don't Inherit/Don't Trigger
from instances			
IV	GetValue PutValue	GetValueOnly PutValueOnly	GetIVHere
CV	GetClassValue PutClassValue	GetClassValueOnly PutClassValueOnly	<n.a.> <n.a.>
from classes			
IV	<n.a.> <n.a.>	GetClassIV PutCIVHere	GetClassIVHere PutClassIV
CV	GetClassValue PutClassValue	GetClassValueOnly PutClassValueOnly	GetCVHere PutCVHere

Tailoring MasterScope

Extending analysis of functions

MasterScope maintains a number of tables describing how (analyzed) functions relate to other objects. The template keywords **TEST**, **PROP**, **FUNCTION**, etc. corespond to some of these tables. For a number of applications, the user would like to be able to define new template words and the database tables that go along with them.

(ADDTEMPLATEWORD WORD) [Function]

defines a new table to hold a new MasterScope relation. The name of the table will be **WORD**, and **WORD** can be used in function templates. This is a new function.

Functions are also provided to allow the user to access these new tables inside of a MasterScope command.

(MSADDRELATION RELATION TABLES) [Function]

defines a new relation for MasterScopes parser and command interpreter. For example, **(MSADDRELATION '(FETCH FETCHES FETCHING FETCHED))** could have been used to define the **FETCH** relation.

RELATION is a list of **ROOT PRESENT PARTICIPLE** and **PAST** conjugations of the new relation. **TABLES** is a list of MasterScope database tables that will be **UNION**ed to compute the new relation. (If **TABLES** is an atom it will be coerced to a list containing that atom. If the tables do not already exist, they will automatically be created by **ADDTEMPLATEWORD**). **TABLES** defaults to the **ROOT** of the relation.

(MSADDMODIFIER RELATION MODIFIERS TABLES) [Function]

defines a new modifier for the given relation. For example, the phrase **SET FREE** could have been defined by **(MSADDMODIFIER 'SET 'FREE '(SETFREE))**.

RELATION is a known MasterScope relation (either built-in or defined by **MSADDRELATION** above). **MODIFIERS** is a list of equivilent modifiers. **TABLES** is a list of MasterScope database tables that should be **UNION**ed to compute the new, modified relation. (If any of the tables do not exist, they will be created by **ADDTEMPLATEWORD**). This is a new function.

(MSADDTYPE TYPE TABLES HOWUSED SYNONYMS) [Function] tells MasterScope what it means to use an object of a given type. The phrase **USE THE FIELD** could be defined by **(MSADDTYPE 'FIELD '(FETCH REPLACE) '(USE FETCH REPLACE))**.

TYPE is the type of the object being described. The word **TYPE** can then be used in MasterScope commands. **TABLES** indicates how the relation **USE THE <TYPE> ...** is defined. **HOWUSED** is a list of verb describing how the type can be used. The above example not only lets you use the phrase **USE THE FIELD**, but also the phrases **FETCH THE FIELD** and **REPLACEUSE THE FIELD**. The default value of **HOWUSED** is **(USE)**. Finally, **SYNONYMS** is a list of synonyms for **TYPE**.

In addition to these new functions there are a number of other ways to tailor MasterScope.

Some MasterScope templates, for example **(IF expression template template)** can compute the template to be used. These expressions can access the current form via the free variable **EXPR**. The free variable **PARENT** can be used to access the expression that contains the current expression. This has existed all along, but has not been documented.

ANALYZEUSERFNS [Variable]

is a list each of whose elements is a function that will be **APPLY***ed to the name, definition, and the results of the MasterScope analysis of a function whenever it is analyzed. The result of each application becomes the new result of the MasterScope analysis.

The results of the MasterScope analysis is an ALIST associating relations (**BIND**, **CALL**, etc) with the coresponding data for the function. This can be used to compute relations that are determined by some global context. This has existed all along, but has not been documented.

(SETSYNONYM PHRASE MEANING -) [Function]

defines a new synonym for MasterScope's parser. Both **MEANING** and **PHRASE** are lists of words; anywhre **PHRASE** is seen in a command, **MEANING** will be substituted. For example, **(SETSYNONYM 'GLOBALS '(VARS IN GLOBALVARS OR @(GETPROP X 'GLOBALVAR)))** would allow the user to refer with the single word **GLOBALS** to the set of variables which are either in **GLOBALVARS** or have a **GLOBALVAR** property.

SETSYNONYM includes a small pattern match ability. A **&** in **PHRASE** will match any word; the word will be substitued for **N** in **MEANING** where **N** is the number of **&**'s which have been matched. For example, **(SETSYNONYM '(FOO & &) '(IN 1 OR ON 2))** will take **FOO FIE FUM** into **IN FIE OR ON FUM**. This is an old function, but the documentation (IRM p 13.20) was incomplete—specifically the pattern match stuff was not mentioned.

Extending MasterScope commands

DESCRIBELST [Variable]

is a list each of whose elements is a list containing a descriptive string and a form. The form is evaluated (it can refer to the name of the function being described by the free variable **FN**); if it returns a non-**NIL** value, the description string is printed followed by the value. If the value is a list, its elements are printed with commas between them. For example, the entry **("types: " (GETRELATION FN '(USE TYPE) T)** would include a listing of the types used by each function. From the IRM, p 13.7.

MSCHECKFNS [Variable]

is a list of functions that will be used to extend the **CHECK** MasterScope command. The **CHECK** command will **APPLY*** each function on **MSCHECKFNS** to the list of files being checked. This is new.

Analyzing new types of objects

This doesn't work yet.

(MSADDANALYZE PLURAL SINGULAR ANALYZEFN RELATIONS) [Function]

defines a new type for MasterScope analysis. For example, **(MSADDANALYZE 'CLASSES 'CLASS 'AnalyzeClass ??)** will let you execute MasterScope commands like **. ANALYZE ANY CLASS ON 'MYFILE** and **. WHAT CLASS DEFINES THE IV foo**. The function **ANALYZEFN** should be a function of two arguments, the item name and the flag **REANALYZE?**, and should return an ALIST associating relation names to coresponding data for the object. This is a new function.

Need to do

(DUMPDATABASE) needs to store out all the new words, relation, etc.

Analyzing new types of things needs to store the info someplace, and **ERASE** needs to be able to find it.

Need a **FILEPKGTYPE** for the MasterScope words.

Proposal for Loops manual contents.

This ordering is based on the Interlisp Reference manual. Numbers in *[[]]*'s indicate the coresponding parts in the (new) IRM. Other comments are enclosed in *{}*'s I envision a manual in a single binder, with large tabs marking the IRM volume separation, and smaller tabs for each chapter.

[[Volume 1 -- Language]]

TOC *[[TOC]]*

Introduction *[[1]]*

- Classes and Instances
- MessageSending
- The Golden Braid

Classes *[[2]]*

{Defines the msg protocol for Classes}

- Creating instances
  - `New` msg
  - `NewWithValues` msg
  - `_New` msg sending form
- Destroying instances
  - `DestroyInstance` msg
- Destroying a class
  - `Destroy` msg
  - `Destroy`! msg
- Changing a class
  - `Add` msg
  - `Delete` msg
  - `Put` msg
  - `Rename` msg
  - ReplaceSupers msg
  - `SetName` msg
- Accessing parts of a class
  - `ListAttribute` msg
  - `ListAttribute!` msg
- Enumerating parts of classes
  - `AllInstances` msg
  - `AllInstances!` msg
  - `SubClasses` msg
- Dealing with Methods
  - `SpecializeMethod` msg
  - `CopyMethod` msg
  - `DefMethod` msg

- `MoveMethod` msg
- `MethodDoc` msg
- Misc
  - `NewClass` msg
  - `Subclass` msg
  - `Prototype` msg
    - See also `_Proto`

Instances *[[3]]*

{Defines the protocol for objects}

- Getting the class of an instance
  - Function `Class`
  - `Class` msg
  - `ClassName` msg
  - Classes for lisp entities -- `LispClassTable`
- Accessing variables
  - `GetValue`, `PutValue`, `GetIVHere`
  - `GetClassValue`, `PutClassValue`, `GetClassIVHere`
  - `@`, `@*`, `_@`
    - problems with embedded "."'s in IV names
  - `GetIVValue` msg if not an instance
  - `NoValueFound`
  - CHANGETRAN
  - `IVMissing` msg
  - `CVMissing` msg
- Creating instances
  - `NewInstance` method
- Naming instances
  - `SetName` msg
  - `UnSetName` msg
  - `Rename` msg
  - `ErrorOnNameConflict`
- Changing an instance
  - `AddIV` msg
  - `DeleteIV` msg
- Destroying instances
  - `Destroy` message
- Copying Instances
  - `CopyShallow` message
  - `CopyDeep` message

MetaClasses *[[4]]*

{Defines the msg protocol for MetaClasses}

- `DestroyClass` msg

- MetaClass
- AbstractClass

AnnotatedValues *[[5]]*

- `NotSetValue`

Methods *[[6]]*

- Method objects
  - Categories
- Method functions
  - LAMBDA word `Method`
  - `&OPTIONAL` arguments
  - Advising inherited methods
    - `_Super`
    - `_Super?`
    - `_SuperFringe`
    - `DoMethod`
    - `ApplyMethod`
  - Functions `ClassNameOfMethodOwner`, `SelectorOfMethodBeingCompiled`, `ArgsOfMethodBeingCompiled`
- Defining new methods
  - `DefaultComment`
  - `SubclassResponsibility`

Conditionals and Iterative Statements *[[9]]*

- Date type predicates
  - `Object?`
  - `Class?`
  - `Instance?`
  - `AnnotatedValue?`
- Other predicates
  - `HasIV`, `HasIV!`, `HasCV`
  - `Understands`
  - `InstOf`, `InstOf!`
- Interative statements
  - `in-supers-of`

Message sending forms *[[10]]*

- List all message sending forms (even if described elsewhere)
  - `_`
  - `_!`
  - `_IV`
  - `_Try`
  - `_Proto`
  - `_Super`
  - `_Super?`
  - `_SuperFringe`
  - `_New`
  - `_AV`
- Looking up methods

FetchMethod
FetchMethodOrHelp

Miscellaneous *[[12]]*

System version information
Pattern match function MatchDescr
FEATURES
DELASSOC

[[Volume 2 -- Environment]]

Browsing and Exec Level Commands *[[13]]*

UNDOable versions of Loops fns (like PutValue) if typed at top level?
LASTWORD set by DefineMethods
LASTCLASS set by DefineClass
Loops Icon
Left button menu
Middle button menu
Right button menu
Background menu command?
Browsers, in general
Interaction with GRAPHZOOM
MaxLatticeHeight
MaxLatticeWidth
Shift selectingfrom the background of a LatticeBrowser will COPYINSERT the graph.
Class Browsers
PPDefault
UpdateClassBrowsers?
File Browsers
Other Browsers
Top level commands
DefineClass
LASTWORD
USERWORDS
DefineMethod

Errors and Breaks *[[14]]*

Function HELPCHECK
LoopsDebugFlg
LoopsDebugFlg
ErrorOnNameConflict
IVMissing
IVValueMissing
MessageNotUnderstood

Breaking and Tracing *[[15]]*

Breaking and Tracing IV access
Breaking and Tracing methods

- Gauges
- Editing *[[16]]*
  - EDITDEF
    - When do changes take effect?
  - Copying an AV inside of DEdit really adds a pointer to the same AV!
  - `Edit` msg
  - `MakeEditSource` msg
  - `InstallEditSource` msg
- File Package *[[17]]*
  - Loading Loops files
    - Effect of `LDFLG = PROP`, `ALLPROP`, ect
    - `UNDO`ing
    - Selective loading of classes, etc from a file.
  - Filepackage coms
    - CLASSES
    - METHODS
    - INSTANCES
    - THESE-INSTANCES
  - Noticing changes
    - `ObjectModified` message
  - Moving and adding
    - `MoveToFile` msg
    - `MoveToFile!` msg
  - Saving instances
    - `MakeFileSource` message
    - `DontSave` IV property
    - `SaveInstance` message
    - `SaveInstance?` message
- Compiling *[[18]]*
  - Macros for `GetValue`, `PutValue`
  - Refering to Loops objects in compiled code
- Masterscope *[[19]]*
- DWIM *[[20]]*
  - No method, tries to correct the spelling
  - No method, but a fn=> use the fn. How about other way?
- Performance Issues *[[22]]*
  - Garbage Collection
  - Space
  - Speed
  - Caching
    - `ClearAllCaches`
      - Evals all forms on the global var `ClearAllCaches`
    - Method lookup caches

local cache
global cache hit rate of 97.5% in our tests
IV Lookup caches
local cache
global cache hit rate of 98% in our tests
Processes *[[23]]*
`_Process`, `_Process!`

[[Volume 3 -- Input/Output]]
Reading and Printing *[[25]]*
The `#` read macro character
How Classes print
`#.(¤ className)` and the function `¤`
Class.FileOut
`#.(¤C className)` and the function `¤C`
How Instances print
`Object.PrintOn`
`Object.PP`
`PPObj`
`ObjectAlwaysPPFlag`
`ObjectDontPPFlag`
Interaction with `SYSPRETTYFLG`
How AnnotatedValues print
`DefaultActiveValueClassName`
UIDs
When assigned
Functions `HasUID?`, `UIDP`, `UID`, `GetObjFromUID`, `MapObjectUID`
User Input/Output Packages *[[26]]*
Inspector
Extensions to `?=`
method doc in `PROMPTWINDOW`
select selector from menu
doesn't work in the DEdit menu
Loops windows *[[28]]*
The default menu available on Loops windows is compatible with Lisp. It even encludes a "Hardcopy" item that works correctly with LatticeBrowsers.
Methods `Window.SetRegion` and `Window.SetOuterRegin`

Interaction with ATTACHEDWINDOWS
Index *[[Index]]*

[[Loops Library Packages Manual]]
Virtual Copies

Gauges

[[Release Notes]]

Installation

- Variables `LOOPSDIRECTORY`, `OptionalLispuserFiles`, and `LoadLoopsForms`. Maybe even `LOOPSUSERSDIRECTORIES`?
- The function `LOADLOOPS`
- The file LOOPSSITE
- Just load the file LOOPS
- Initial screen setup

Customer Support

Differences from old Loops

- KnowledgeBases no longer a part of Loops
  - Functions `ReadLeafObj`, `AllGlobalNames`, `RememberName`, and `GlobalName` no longer exist
  - Variables `WritingSummaryFlg`, `WritingLayerFlg`, `LeafInstanceFlg`, `FirstEnvFlg`, `OpenKBFiles`, `DefaultKBName`, `CurrentEnvironment`, `CurrentNameTable`, and `CurrentUIDTable` no longer exist
  - Macro `Modified` no longer exists
- Rules no longer a part of Loops
  - Rules are not distributed. If they are needed, they can be loaded by first recompiling them and then loading the file LOOPSRULES-ROOT. This replaces a few functions and methods in the standard system and then load the rules files.
- Code cleaned up
  - Variables `VarNameIndexes`, `PrintStatusWindow`, and `TTY` no longer exit
  - Functions `DC`, `DE`, `UE`, `EM`, `EI`, `EC`, and `FILE` no longer exit
  - Macro `@@` no longer exists. Change `(@@ foo)` to `(@ ::foo)`, or even `(@ foo)`
  - Usermacros `PU`, `UE`, and `EU` no longer exist
  - The argument that controls updating in the window methods `SetOuterRegion`, `SetRegion`, `Shape`, and `Shape1` have all been changed to `noUpdateFlg`. They used to be a mixture of `updateFlg` and `dontUpdateFlg`.
  - The interpretation of the `left`, `bottom`, `width`, and `height` IVs in the class `Window` is now the same as Lisp windows: `left` and `bottom` refer to the lower left corner of the outside of the window, `width` and `height` refer to the outside dimensions of the window (including title).
  - The fie LoopsMixin has been deleted. The classes `DatedObject`, `IndirectObj`, `Node`, `Perspective`, `NamedObject`, `GlobalNamedObject`, `TextItem`, `VarLength`, `StrucMeta`, `ListMetaClass`, `TempClass`, and `Template` no longer exist.

Old messages `List` and `List!` are now called `ListAttribute` and `ListAttribute!`. This change was required for future migration to CommonLoops. Theold methods for `List` and `List!` are still available in the LOOPSBACKWARDS file.

The functions `DebugLoops`, `LOOPSDIR`, `i/d`, and `TESTLOOPS` have been removed from the system.

The functions `CheckDestroyedObjects` and `RemoveClassDef` have been removed from the system.

I/O

Support for reading in old style read macro (like `#(localState getFn putFn)` or `#¤Mumble`) is available in the file LOOPSBACKWARDS.

UIDs are no longer strings. They are CONSes of session-id's and uid numbers.

Active Values

Most of the old functions for active values have been moved to LOOPSBACKWARDS. The functions `GetLocalState`, `PutLocalState`, `GetLocalStateOnly`, `PutLocalStateOnly`, `ReplaceActiveValue`, `MakeActiveValue`, and `DefAVP` are still around, and seem to work. The exception is `MakeActiveValue` — it now always `EMBED`s.

Functions `GetActiveValueGetFn`, `GetActiveValuePutFn`, `GetActiveValueLocalState`,PtcieauGtn `alueGetFn`, etc no longer exist. Replacements are available in the file LOOPSBACKWARDS .

`NotSetValue` is no longer `?`—it is now an active value. This may require changes to any code that directly refered to `?`.

Virtual Copies

The messages `MakeCopy99`, `MakeCopyActiveValue99`, `MakeCopyList99`, and `MakeCopyObject99` in VirtualCopies have been renamed `\Internal/MakeCopy`, `\Internal/MakeCopyActiveValue`, etc

Method properties no longer exist

Function `PushNewValue` no longer defined. Instead, use CHANGETRAN.

Can't load new Loops files into old Loops.

Conversion of old Loops code to new Loops

Old code will need to be recompiled.

File LOOPSBACKWARDS

Function ConvertLoopsFiles

Future Directions

Proposal for Loops manual contents.

This ordering is based on the Interlisp Reference manual. Numbers in [[]]'s indicate the coresponding sections in the (new) IRM.

[[Volume 1 -- Language]]

Introduction [[1]]

Classes [[2]]

- The hierarchy
- Tofu
- Object
- Class
- LispClassTable

Instances [[3]]

MetaClasses [[4]]

- The Golden Braid
- MetaClass
- AbstractClass

AnnotatedValues [[5]]

- `NotSetValue`

Methods [[6]]

- Categories
- `DefaultComment`
- `SubclassResponsibility`

Conditionals and Iterative Statements [[9]]

- Date type predicates
  - `Object?`
  - `Class?`
  - `Instance?`
  - `(type? annotatedValue` *form*`)`
- Other predicates
  - `HasIV`, `HasIV!`, `HasCV`
  - `Understands`
  - `InstOf`, `InstOf!`
- Interative statements
  - `in-supers-of`

Message sending forms [[10]]

- List all message sending forms (even if described elsewhere)
- Looking up methods
  - `FetchMethod`
  - `FetchMethodOrHelp`

Miscellaneous [[12]]

System version information
Pattern match functions
FEATURES

[[Volume 2 -- Environment]]

Browsing and Exec Level Commands [[13]]
- UNDOable version of PutValue?
- LASTWORD set by DefineMethods
- LASTCLASS set by DefineClass
- Loops Icon
- Browsers, in general
  - MaxLatticeHeight
  - MaxLatticeWidth
- Class Browsers
- File Browsers
- Other Browsers
- Top level commands
  - DefineClass
  - DefineMethod

Errors and Breaks [[14]]
- LoopsDebugFlg
- ErrorOnNameConflict
- IVMissing
- IVValueMissing
- MessageNotUnderstood

Breaking and Tracing [[15]]
- Breaking and Tracing IV access
- Breaking and Tracing methods
- Gauges

Editing [[16]]

File Package [[17]]

Compiling [[18]]
- Macros for GetValue, PutValue
- Refering to Loops objects in compiled code

Masterscope [[19]]

DWIM [[20]]
- No method, but a fn=> use the fn. How about other way?

Performance Issues [[22]]
- Garbage Collection
- Space
- Speed
- Caching

XEROX LOOPS

RELEASE NOTES
REFERENCE MANUAL
LIBRARY PACKAGES MANUAL

XEROX

610E15980
Lyric/Medley Release
July 1988

XEROX LOOPS RELEASE NOTES

XEROX LOOPS REFERENCE MANUAL

XEROX LOOPS LIBRARY PACKAGES MANUAL

610E15980

Lyric/Medley Release

July 1988

This manual is set in Modern and Terminal typefaces with text written and formatted on Xerox Artificial Intelligence workstations. Xerox laser printers were used to produce text masters.

COVER TEXT:

NOTE: USE APPROPRIATE POINT SIZES TO ACHIEVE EFFECT
SHOWN BELOW. USE XEROX STANDARDS FOR <<8-1/2x11>> BINDER COVERS

FONT

ARTIFICIAL INTELLIGENCE SYSTEMS

XEROX LOOPS

HELVETICA BOLD ITALIC

XEROX LOOPS RELEASE NOTES
XEROX LOOPS REFERENCE MANUAL
XEROX LOOPS LIBRARY PACKAGES MANUAL

LYRIC/MEDLEY RELEASE

REGULAR or TRIUMVIRATE BLACK ITALIC

Part Number 610E15980

SPINE (3 INCH):

ARTIFICIAL INTELLIGENCE SYSTEMS

XEROX LOOPS

XEROX LOOPS RELEASE NOTES
XEROX LOOPS REFERENCE MANUAL
XEROX LOOPS LIBRARY PACKAGES MANUAL

LYRIC/MEDLEY RELEASE

HELVETICA BOLD ITALIC

REGULAR or TRIUMVIRATE BLACK ITALIC

ERRATA FOR THE LYRIC / MEDLEY RELEASE OF XEROX LOOPS

Please note the following corrections to the Lyric/Medley release of the manual and software. These descriptions and workarounds supplement the Lyric/Medley *Xerox LOOPS Release Notes*.

Notes and Cautions

Documentation comments in examples

Documentation props are now strings, as mentioned in the Lyric/Medley version of the *Xerox LOOPS Release Notes*. However, many of the manual's examples incorrectly show comments being used for documentation on methods, and instance and class variables.

Conversion of instance files

We have encountered a case where instance files from Koto LOOPS failed to load after conversion under Lyric/Medley LOOPS. This occurs when instances refer to each other (their IVs contain one another).

This type of instance file must be written out in the INTERLISP readtable, since only that readtable handles the "hash dot" reader macro used to write out a reference to an instance. After conversion of such files, place a MAKEFILE-ENVIRONMENT property on the file that will cause it to be written and read in the INTERLISP readtable.

GetIt for missing IV props

There is an inconsistency in **GetIt**'s behavior when retrieval of missing instance variable properties is attempted. If the object being retrieved from is an *instance*, this returns the value of **NoValueFound**, and triggers any active values. However, if the object being retrieved from is a *class*, **GetIt** returns the value of **NotSetValue**, and does not trigger active values. A workaround would be to check for both **NoValueFound** and **NotSetValue** as return values from **GetIt**.

LOOPS Rules

The User's Module "Rules" has changed to track differences between Koto and Lyric/Medley LOOPS. In Koto LOOPS, one could create RuleSets that were not methods; this is no longer possible. RuleSets can no longer reside "bare" in a FNS definition.

Several things follow from this change:

- **RunRS** no longer works.
- **DefRSM** is now the only way to create new RuleSets. The documented **New** method for the class **RuleSet** is used internally by **DefRSM**; software should no longer specialize or depend on it.
- The RuleSet method **CopyRules** no longer works. RuleSets can be copied using **CopyMethod**.

Converting Koto RuleSets

Converting older RuleSets to run in the Lyric/Medley release of Xerox LOOPS is a two-step process. The first step takes place in the Koto release of LOOPS; the second step occurs in the Lyric/Medley release of LOOPS.

While still running the Koto release of LOOPS, you must first pass preKoto RuleSets through the converter in the LOOPSBACKWARDS User's Module. This will upgrade the RuleSets to run in Koto LOOPS. RuleSets which reside in FNS must also be moved into methods at this time. A final ready-to-convert version of the RuleSet files should then be made.

After starting up the Lyric/Medley release of Xerox LOOPS any previously prepared Koto RuleSets can be passed through the converter in the CONVERSION-AIDS User's Module. Some types of Koto converted Buttress RuleSets will print out a message during conversion; these must then be rule-compiled, i.e. translated from RuleSet to executable code by sending the RuleSet the **RE** message and exiting the rule editor with **OK&LispCompile**.

After these steps all RuleSet formats will be correctly updated and the fully converted files can be made.

[This page intentionally left blank]

A. ACTIVE VALUES IN BUTTRESS LOOPS

In the Buttress version of LOOPS, the concept of active values was implemented differently. The current **ExplicitFnActiveValue** acts very much like the old active values, and is used to provide compatibility with existing Xerox LOOPS code. Most of the functions described in this chapter are found only in the LOOPSBACKWARDS user package, work as they did in the Buttress version of LOOPS, and should be used only to bring existing code into the current system.

Descriptions of the functionality in this appendix are written in terms of the new **ActiveValues** wherever possible.

The active value/annotated value system discussed in Chapter 8, Active Values, is a new implementation. Programs developed using the Buttress activeValue system automatically convert into the new system when loaded, using the **ExplicitFnActiveValue** capability described in Chapter 8, Active Values, and in this appendix.

Note: The following functions and records are maintained for compatibility purposes only; they are not fully supported and may not exist in future Xerox LOOPS releases. Programs that use these records and functions should be changed. The LOOPSBACKWARDS user package must be loaded for these functions to work.

A.1 Buttress System of ActiveValues

The Buttress LOOPS implementation combined the notions of annotated value and active value. To annotate a variable, the value was replaced with an instance of an Interlisp-D data type called activeValue, but there were no LOOPS classes with similar names and functions as there are now.

activeValue [Record]

Purpose: Buttress implementation of the active values concept. Specifically, the Lisp data type equivalent to the present annotatedValue.

Behavior: An activeValue placed as the value of a variable invoked evaluation of code on access attempts rather than just returning a stored value.

Field Names:

localState A place for data storage.

getFn The name of a function applied when the program retrieves the value of a variable that contained an activeValue.

putFn The name of a function that was applied when the program replaces the value of a variable that contained an activeValue.

If either the **getFn** or **putFn** fields is NIL, default actions returned or replaced the **localState**, respectively. Nesting was accomplished by the **localState** of an activeValue being itself an activeValue.

ExplicitFnActiveValue [Class]

Purpose: Mimics the behavior of the Buttress-style active values and allows simple changes to the user code triggered by the **ActiveValue** mechanism.

Behavior: **Get** accesses to the wrapped variable cause the **getFn** to be called, and **Put** accesses cause **putFn** to be called. Enables the old style activeValues to look like the new style without changing any functionality.

Instance Variables:

localState A place for data storage.

getFn The name of a function applied when the active variable is read.

putFn The name of a function applied when the active variable is changed.

(**MakeActiveValue** *self varOrSelector newGetFn newPutFn newLocalState propName type*) [Function]

Purpose: Makes the value of some variable an active value.

Behavior: Creates a new activeValue record and installs it according to the arguments.

Arguments:

self Object whose variable is changed to an active value.

varOrSelector
Variable name or method selector where the data type activeValue is placed.

newGetFn and *newPutFn*
If NIL, the old values of **getFn** and **putFn** are not overwritten. If T, the values of **getFn** and **putFn** are changed to NIL. Any other values are placed in the **getFn** and **putFn** fields of the activeValue.

newLocalState
The value of this argument is ignored. A new **ActiveValue** instance is always created. The contents of **localState** is changed to the previous value of the variable or property being made active.

propName Name of the property, if the active value is to be placed on a property list. This is NIL if the active value is associated with a variable or method.

type Indicates the type of the variable *varNameOrSelector*. Must be o ne of IV (or NIL) for instance variable, CV for class variable, CLASS for a class property, or METHOD for a method property.

(**DefAVP** *fnName putFlg*) [Function]

Purpose: Creates a template for defining an active value function.

Behavior: Creates a template and leaves you in the Interlisp-D function editor.

Arguments:

fnName Name of the function.

putFlg T indicates function is a **putFn**; NIL indicates a **getFn**.

Returns: The function name on exit from the editor.

(**GetLocalState** *activeValue self varName propName type*) [Function]

Purpose: Retrieves data from **localState**.

Behavior: Retrieves the value stored in the **localState** of *activeValue*. Nested active values will be triggered.

Arguments:

activeValue An **ActiveValue**.

self The object containing the **ActiveValue**.

varName The name of the variable were the **ActiveValue** is stored.

propName The name of an instance or class variable property. This is NIL if the **ActiveValue** is associated with the value of the variable itself.

type Specifies where the **ActiveValue** was stored. NIL means an instance variable, CV means class variable, CLASS means a class property, METHOD means a method property.

Returns: Contents of the **localState** field of *activeValue*.

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

Purpose: Data replacement.

Behavior: Stores *newValue* in the **localState** field of *activeValue*. Nested active values will be triggered.

Arguments:

activeValue An **ActiveValue**.

newValue A new value to be stored in **localState**.

self The object containing the **ActiveValue**.

varName The name of the variable were the **ActiveValue** is stored.

propName The name of an instance or class variable property. This is NIL if the **ActiveValue** is associated with the value of the variable itself.

type Specifies where the **ActiveValue** was stored. NIL means an instance variable, CV means class variable, CLASS means a class property, METHOD means a method property.

Returns: The value of *newValue*.

(**GetLocalStateOnly** *activeValue*) [Function]

Purpose: Gets a value from **localState** without triggering any nested **ActiveValue**.

Behavior: Retrieves the value stored in the **localState** field of the **ActiveValue** without triggering any nested **ActiveValue**.

Arguments: *activeValue* The **ActiveValue** in which the **getFn** and **putFn** is found.

Returns: The contents of **localState.**

(**PutLocalStateOnly** *activeValue newValue*) [Function]

Purpose: Puts a value into a **localState** without triggering any nested **ActiveValues.**

Behavior: Replaces the value stored in the **localState** of *activeValue* without triggering any nested **ActiveValue**.

Arguments:

activeValue An **ActiveValue**.

newValue Value used for the replacement.

Returns: The value of *newValue.*

(**ReplaceActiveValue** *activeValue newVal self varName propName type*) [Function]

Purpose: In an object's variable which has an **ActiveValue** installed, overwrites *activeVal* with *newVal*, providing a way of removing an **ActiveValue**.

Behavior: Searches arbitrarily deep nesting to replace the occurrence of *activeVal* with *newVal*. If no match is found in the list that is the value of the variable described by the arguments, an error is invoked.

Arguments:

activeValue The **ActiveValue** to be replaced.

newVal A new value to be stored in the object's variable.

self The object containing the **ActiveValue**.

varName The name of the variable were the **ActiveValue** is stored.

propName The name of an instance or class variable property. This is NIL if the **ActiveValue** is associated with the value of the variable itself.

type Specifies where the **ActiveValue** was stored. NIL means an instance variable, CV means class variable, CLASS means a class property, METHOD means a method property.

newValue Value used for the replacement.

Returns: Value of *newVal.*

A.2 GETFNS AND PUTFNS

A.2 getFns and putFns

In the Buttress version of LOOPS, where the only kind of active value was equivalent to **ExplicitFnActiveValue**, specialization of active values was done not the way it is in Xerox LOOPS, but by the equivalent of putting special purpose functions into the **getFn** and **putFn** instance variables. The following functions emulate the behaviors they had in the Buttress version, using the current **ActiveValue** mechanisms.

In all cases, the functions are installed in the **getFn** or **putFn** instance variable of an **ActiveValue**, and are called when an attempt is made to get or put the variable where the **ActiveValue** is stored. The arguments and values returned are irrelevant to the use of these functions.

(**NoUpdatePermitted** *self varName oldOrNewValue propName activeValue type*) [Function]

Purpose: **putFn** for preventing the updating of a variable.

Behavior: LOOPS-defined **putFn** that causes a break if an attempt is made to replace the value of the variable containing the **ActiveValue**.

(**FirstFetch** *self varName oldOrNewValue PropName activeValue type*) [Function]

Purpose: **getFn** for dynamic variable initialization.

Behavior: LOOPS-defined **getFn** that expects the **localState** of *activeValue* to be an Interlisp-D expression to be evaluated. On the first fetch, the expression is evaluated and the variable or property is set to the value of the expression.

(**GetIndirect** *self varName oldOrNewValue PropName activeValue type*) [Function]

Purpose: LOOPS-defined **getFn** that functions as a pointer to another variable.

Behavior: **GetIndirect** and **PutIndirect** together set up an **ActiveValue** whose **localState** contains a pointer to where the actual value is stored. This is used when the value of a variable should always be the same as another.

(**PutIndirect** *self varName oldOrNewValue PropName activeValue type*) [Function]

Purpose: LOOPS-defined **putFn** that functions as a pointer to another variable.

Behavior: See **GetIndirect**.

(**ReplaceMe** *self varName oldOrNewValue PropName activeValue type*) [Function]

Purpose: LOOPS-defined **putFn** which removes both itself and any **getFn**.

Behavior: In some cases, you may want to compute a default value if given, but replace the active value by the value given if you set the value of a variable. For this, you can employ **ReplaceMe**. Any replacement attempt at the variable containing an **ActiveValue** with this as its **putFn** results in the value of the variable being replaced, and the **ActiveValue** disappearing.

(**AtCreation** *self varName oldOrNewValue PropName activeValue type*) [Function]

Purpose: LOOPS-defined **getFn** used to replace the active value with a dynamically computed value at instance creation.

Behavior: This function no longer works.

To achieve the closest functionality, use the **FirstFetchAV** specialization of the class **ActiveValue** (see Chapter 8, Active Values) or the **FirstFetch** function described above.

[This page intentionally left blank]

LOOPS DOCUMENTATION KIT CONFIGURATIONS

New and Existing LOOPS Customers (Koto to Lyric/Medley):

610E15980 *Xerox LOOPS Reference Manual,* Lyric/Medley (7/88)
Xerox LOOPS Release Notes
Xerox LOOPS Library Packages Manual

Errata Sheet

610E16000 *Xerox LOOPS Users' Packages Manual*, Lyric/Medley (7/88)

LYRIC/MEDLEY TAB TEXT

POINT SIZE: 10

FONT: OPTIMA

TABS:

XEROX LOOPS RELEASE NOTES

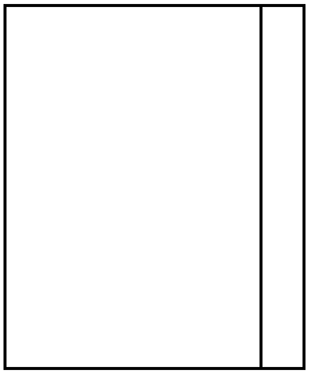
XEROX LOOPS REFERENCE MANUAL

XEROX LOOPS LIBRARY PACKAGES MANUAL

TABS FOR LYRIC/MEDLEY LOOPS (1-1/2" D-RING BINDER)

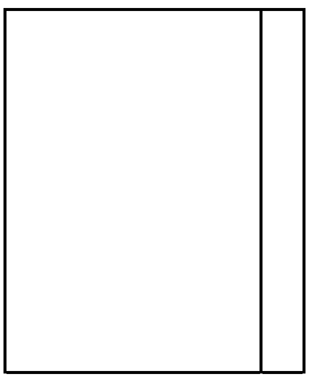
TAB COLOR: PMS GRAY 422-C

BANK 1



XEROX LOOPS RELEASE NOTES

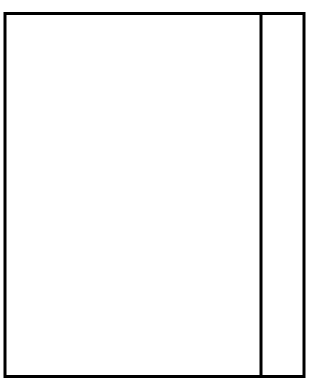
BANK 2



XEROX LOOPS REFERENCE MANUAL

SUBTABS: See following page

BANK 3



XEROX LOOPS LIBRARY PACKAGES MANUAL

SUBTABS FOR LYRIC/MEDLEY LOOPS (1-1/2" D-RING BINDER)

COLOR OF TABS: WHITE

BANK 1

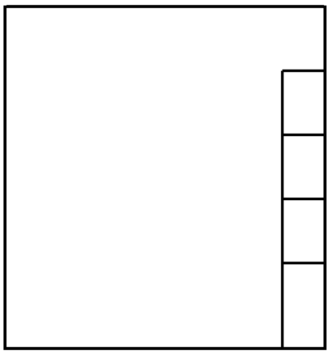
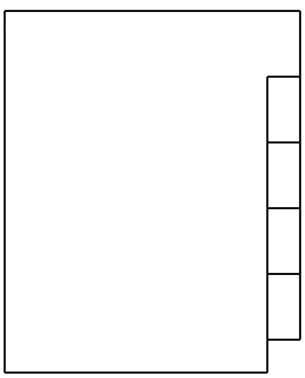


Table of Contents

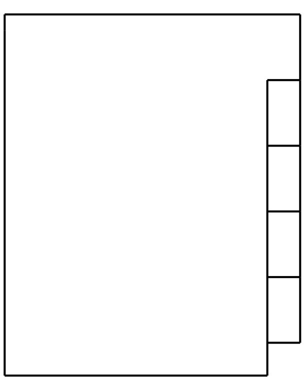
1. Introduction
2. Instances
3. Classes
4. Metaclasses

BANK 2



5. Accessing Data
6. Methods
7. Message Sending Forms
8. Active Values
9. Data Type Predicates

BANK 3



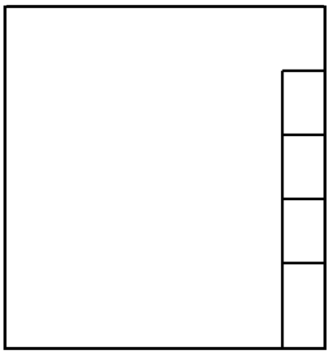
10. Browsers
11. Errors and Breaks
12. Breaking and Tracing
13. Editing
14. File Manager

Continued on next page

SUBTABS FOR LRYIC/MEDLEY LOOPS (1-1/2" D-RING BINDER)

COLOR OF TABS: WHITE

BANK 4



15. Performance Issues

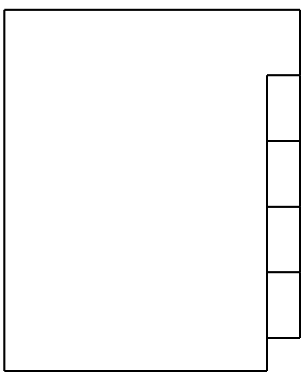
16. Processes

17. Reading and Printing

18. User Input/ Output Modules

19. Windows

BANK 5



20. System Variables and Functions

A. Active Values

Glossary

Index

LYRIC/MEDLEY SUBTABS TEXT

POINT SIZE: 10

FONT: Optima

SUBTABS:

Table of Contents 4. Metaclasses	1. Introduction	2. Instances	3. Classes
5. Accessing Data 9. Data Type Predicates	6. Methods	7. Message Sending Forms	8. Active Values
10. Browsers 14. File Manager	11. Errors and Breaks	12. Breaking and Tracing	13. Editing
15. Performance Issues 19. Windows	16. Processes	17. Reading and Printing	18. User Input/ Output Modules
20. System Variables and Functions	A. Active Values	Glossary	Index

Replace this page with
XEROX LOOPS RELEASE NOTES
tab

Replace this page with
XEROX LOOPS REFERENCE MANUAL
tab

Replace this page with
XEROX LOOPS LIBRARY PACKAGES MANUAL
tab

COVER TEXT:

NOTE: USE APPROPRIATE POINT SIZES TO ACHIEVE EFFECT
SHOWN BELOW. USE XEROX STANDARDS FOR <<8-1/2x11>> BINDER COVERS

FONT

ARTIFICIAL INTELLIGENCE SYSTEMS

XEROX LOOPS

HELVETICA BOLD ITALIC

USERS' PACKAGES MANUAL

LYRIC/MEDLEY RELEASE

REGULAR or TRIUMVIRATE BLACK ITALIC

Part Number 610E16000

SPINE (1 INCH):

ARTIFICIAL INTELLIGENCE SYSTEMS

XEROX LOOPS ***USERS' PACKAGES MANUAL***

LYRIC/MEDLEY RELEASE

HELVETICA BOLD ITALIC

REGULAR or TRIUMVIRATE BLACK ITALIC

LYRIC/MEDLEY LOOPS KIT
on 5¼" floppies
for Xerox 1186 workstation
part number LYMLP1186

Documentation

	Loose material:
150002	Lyric/Medley LOOPS Cover Letter *(single sheet)*
400008	Documentation updates *(4 pages, stapled)*
151010	1186 Lyric LOOPS Floppy Index and Directories *(6 pages, stapled)*
310000	LOOPS MANUAL assembly: (Xerox P/N 610E15980)
	Large binder
	White ENVOS front cover
	Spine label: "LOOPS MANUAL" (Green)
	LOOPS Reference Manual (310001)
	LOOPS Library Packages Manual (310002)
	LOOPS Release Notes (400009)
310003	LOOPS USERS' MODULES, manual assembly: (Xerox P/N 610E16000)
	Small binder
	White ENVOS front cover
	Spine label: LOOPS USERS' MODULES (Green)
	LOOPS Users' Packages Manual

Media

210035	Lyric LOOPS Library
210036	Lyric LOOPS System #1
210037	Lyric LOOPS System #2
210038	Lyric LOOPS Users

Envos Marketing Brochures

ROOMS™ Brochure
ROOMS™ Product Description
LOOPS Product Description

LYRIC/MEDLEY LOOPS KIT
on 8" floppies
for Xerox 1108 workstation
part number LYML1108

Documentation

	Loose material:
150002	Lyric/Medley LOOPS Cover Letter *(single sheet)*
400008	Documentation updates *(4 pages, stapled)*
151009	1108 Lyric LOOPS Floppy Index and Directories *(5 pages, stapled)*

310000 LOOPS MANUAL assembly: (Xerox P/N 610E15980)
Large binder
White ENVOS front cover
Spine label: "LOOPS MANUAL" (Green)
LOOPS Reference Manual (310001)
LOOPS Library Packages Manual (310002)
LOOPS Release Notes (400009)

310003 LOOPS USERS' MODULES, manual assembly: (Xerox P/N 610E16000)
Small binder
White ENVOS front cover
Spine label: LOOPS USERS' MODULES (Green)
LOOPS Users' Packages Manual

Media

212014	Lyric LOOPS Library
212015	Lyric LOOPS System
212016	Lyric LOOPS Users

Envos Marketing Brochures

ROOMS™ Brochure
ROOMS™ Product Description

LOOPS Product Description

LYRIC/MEDLEY LOOPS KIT
on 1/4" tape or 1/2" tape
for Sun Workstations
part number LYMLP3414 or LYMLP3412

Documentation

	Loose material:
150002	Lyric/Medley LOOPS Cover Letter *(single sheet)*
400008	Documentation updates *(4 pages, stapled)*
151012	LOOPS Tape Directory *(for ¼-Inch tape, 1 page)*
or	
151011	LOOPS Tape Directory *(for1/2-Inch tape, 1 page)*
310000	Sun installation of Lyric/Medley LOOPS (Document Update Sheet)

Large binder w/"LOOPS MANUAL" spine label: (610E15980 Xerox)

LOOPS Reference Manual (310001)
LOOPS Library Packages Manual (310002)
LOOPS Release Notes (400009)

Small binder w/"LOOPS USERS' MODULES" spine label:

310003 LOOPS Users' Packages Manual (610E16000 Xerox)

Media

215003	Lyric/Medley LOOPS for the Sun 3, ¼-Inch Tape
or	
216003	Lyric/Medley LOOPS for the Sun 3, 1/2-Inch Tape

Envos Marketing Brochures

ROOMS™ Brochure
ROOMS™ Product Description

LOOPS Product Description

(REV 1/26/89)

LYRIC/MEDLEY LOOPS KIT(OBSOLETE)
on ½" tape
for Sun 3 Workstation
part number LYMLP3412

Documentation

	Loose material:	
150002	Lyric/Medley LOOPS Cover Letter *(single sheet)*	
400008	Documentation updates *(4 pages, stapled)*	
151011	LOOPS Tape Directory *(for ½-Inch tape, 1 page)*	
310000	LOOPS MANUAL assembly: Large binder White ENVOS front cover Spine label: "LOOPS MANUAL" (Green) LOOPS Reference Manual (310001) LOOPS Library Packages Manual (310002) LOOPS Release Notes (400009)	(Xerox P/N 610E15980)
310003	LOOPS USERS' MODULES, manual assembly: Small binder White ENVOS front cover Spine label: LOOPS USERS' MODULES (Green) LOOPS Users' Packages Manual	(Xerox P/N 610E16000)

Media

216003 Lyric/Medley LOOPS for the Sun 3, ½-Inch Tape

Envos Marketing Brochures

ROOMS™ Brochure
ROOMS™ Product Description
LOOPS Product Description

Sun 3 & 4 LOOPS — ¼" tar Tape Manufacturing Instructions

(Envos Internal Use Only)

1. Go to Tree (or any available Sun 3 with a tape drive on it), and log in.
2. Insert a blank ¼-inch cartridge in the tape drive. You must use a 600-foot tape.
3. Type the following commands (the text in bold):

   tree%: **cd /python/loops**

   tree%: **makereleasetape**

   The system will type something like this:

   ```
   You should have a tape in the cartridge drive already, and
   should be connected to the correct directory.  Please verify
   that you are connected to the right place.

   You are connected to:

   /python/loops
   ```
   (or something ending in /python/pcl)

   ```
   Type ^C to Abort

   Type ^D to continue
   ```
4. Type control-D. The tape will now get written, followed by a message telling you that it is finished.
5. When the tape has finished moving (it rewinds automatically), remove it from the drive, and turn the write-protect tab to the "Safe" setting.
6. Label the cartridge and its container (green LOOPS labels). If those were the last 2 labels, make another 10 (Chuck's PC has the files).
7. Don't forget to log out.

PN 409002

Sun 3 & 4 LOOPS — ½“ 9-Track 1600 bpi Manufacturing Instructions

(Envos Internal Use Only)

1. Go to python and load a ½-inch tape into its tape drive. Hit the “Density“ button until the light next to “1600“ is lit. You’ll need to use a small reel of tape for this.
2. At python’s console, log in as yourself.
3. Type the following commands (the text in bold):

   python%: **cd /python/loops**

   python%: **maketape1600**

   The system will type something like this:

   ```
   You should have a tape mounted on python’s mag tape drive, and
   should be connected to the correct directory.  Please verify
   that you are connected to the right place.

   You are connected to:

   /python/loops
   ```
   (or something ending in /python/pcl)

   ```
   Type ^C to Abort

   Type ^D to continue
   ```
4. Type control-D. The tape will now get written, followed by a message telling you that it is finished.
5. When the tape has finished moving (it rewinds automatically), remove it from the drive, and remove the write-protect ring.
6. Label the tape (green Loops label). If that was the last label, make another 10 (Chuck’s PC has the files).
7. Hit the “density” button on the tape drive until the light next to “Host” is lit.
8. Don’t forget to log out.

Address comments to:
Venue
User Documentation
1549 Industrial Road
San Carlos, CA 94070
415-508-9672

LOOPS REFERENCE MANUAL

November, 1991

Text was written and produced with Venue text formatting tools; Xerox printers were used to produce text masters. The typeface is Classic.

1. INTRODUCTION

LOOPS integrates several programming paradigms to facilitate the design of artificial intelligence applications.

- Object-oriented programming, in which information is organized in terms of objects. Every object belongs to a cass, and the classes are arranged in an inheritance lattice which allows complex objects to be described simply. Objects communicate with each other by sending messages. When an object receives a message, it performs some action, which can include sending messages to other objects.
- Procedure-oriented programming, in which smaller subroutines build larger procedures and in which data and instructions are kept separate.
- Access-oriented programming, in which accessing a value triggers an action. This paradgm is useful to monitor certain values.
- Rule-oriented programming, in which programs are organized around recursively composable sets of pattern-action rules. These rules provide a convenient way to describe flexible responses to a wide range of events. This part of LOOPS is included in the users' modules.

As a new user of LOOPS, you first must become familiar with its terminology and with the fundamental concepts described by that terminology. This chapter presents the terminology and related concepts.

1.1 Introduction to Objects

This section shows the LOOPS hierarchy, called a lattice, in Figure 1-1, and describes the key terms in separate subsections. Terms appear in order of increasing complexity, with simpler terms described first and subsequent terms building on these simpler terms.

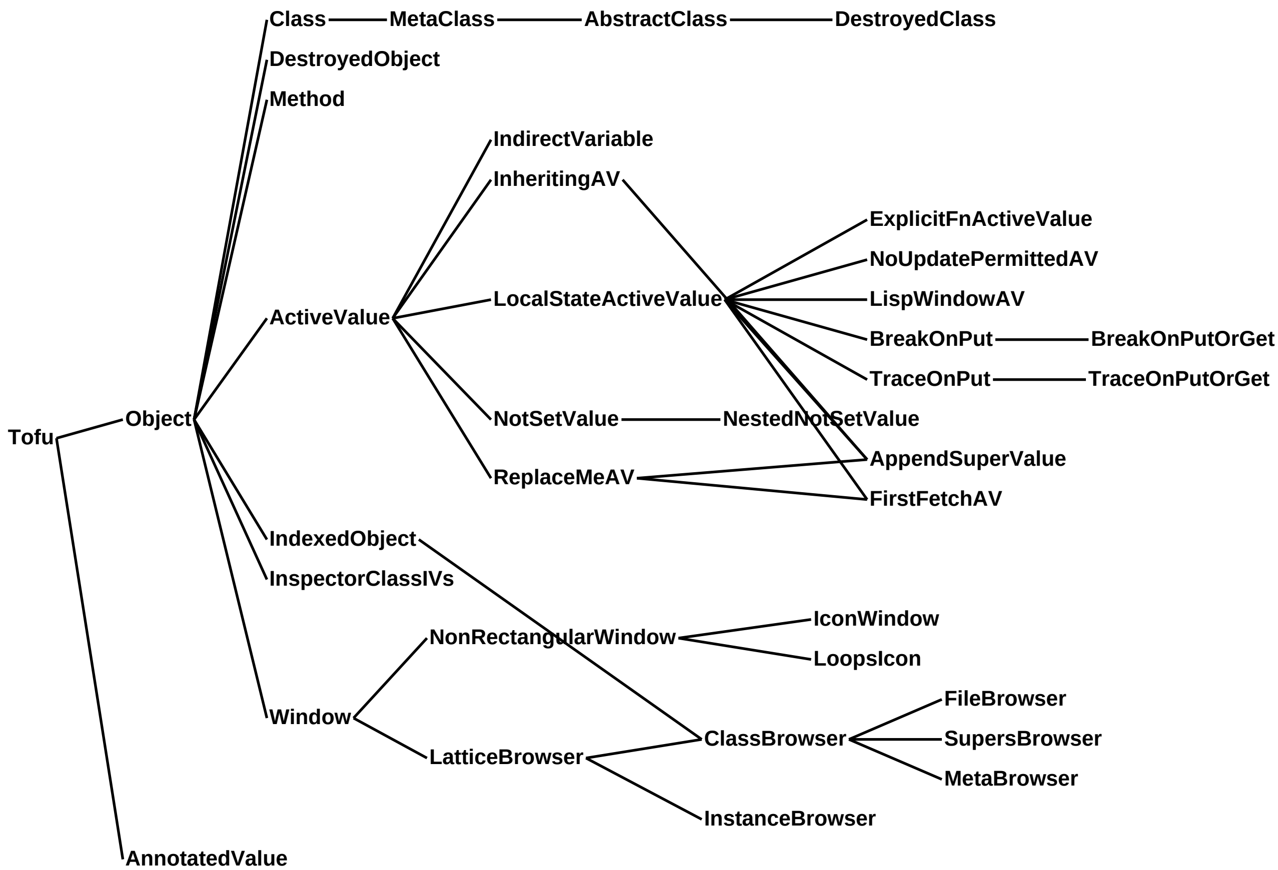


Figure 1-1. LOOPS Lattice

1.1.1 Object

As shown in Figure 1-2, an object is a structure consisting of data and a pointer to functionality that can manipulate the data. In procedure-oriented programming, data and functionality are considered as separate entities.

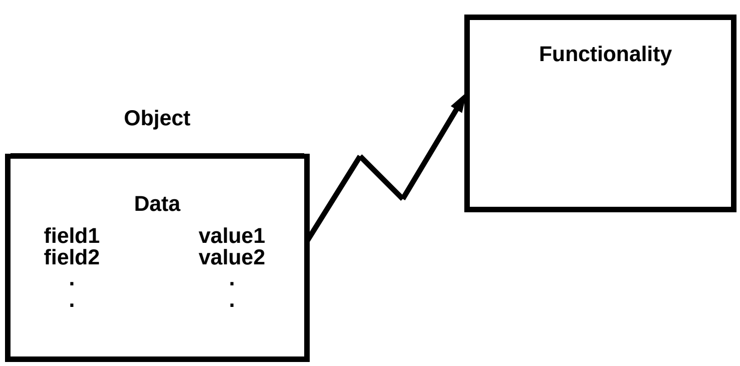


Figure 1-2. An Object

1.1 INTRODUCTION TO OBJECTS

1.1.2 Message

Sending messages to objects provides an alternative to invoking procedures or calling functions. An object responds to a message by computing a value to be returned to the sender of the message, as shown in Figure 1-3. As a side effect, the data within an object may change during the computation. Messages contain a selector for the desired functionality. Messages may also contain arguments, as do procedures.

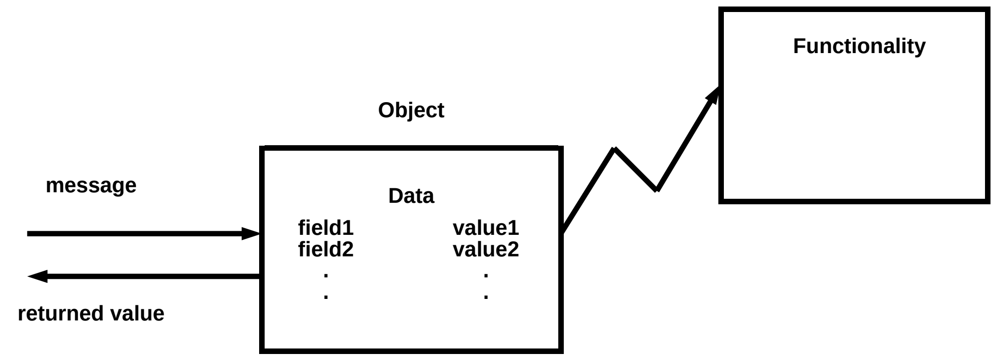


Figure 1-3. An Object Responding to a Message

1.1.3 Method

When an object receives a message, it determines what functionality it must apply to the arguments of the message. This functionality is called a method and is very similar to a procedure. A key concept that distinguishes methods from procedures is that in procedure-oriented programming, the calling routine determines which procedure to apply. In object-oriented programming, you determine the message to send and the object receiving the message determines the method to apply.

1.1.4 Selector

A message that is sent to an object contains a selector. The object uses the selector to determine which method is appropriate to apply to the message arguments. As shown in Figure 1-4, when an object receives a message with a specific selector, the object searches a lookup table containing selectors and methods to find the method associated with that particular selector.

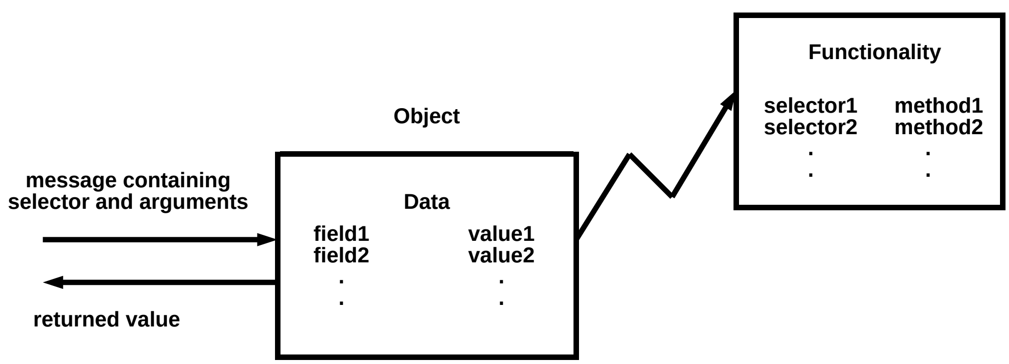


Figure 1-4. A Message Containing a Selector

1.1.5 Class

A class describes objects that are similar; that is, objects containing the same type of data fields and responding to the same messages, as shown in Figure 1-5. Think of the class that describes an object as being a template for the functionality of its objects. When an object is sent a message, the class that describes that object handles the message, not the object itself. Different objects of the same class can respond to messages in the same way; that is, they apply the same method in response to receiving the same message.

To create new objects, send a message to a class requesting that a new object be created. Classes respond to messages because they are also objects.

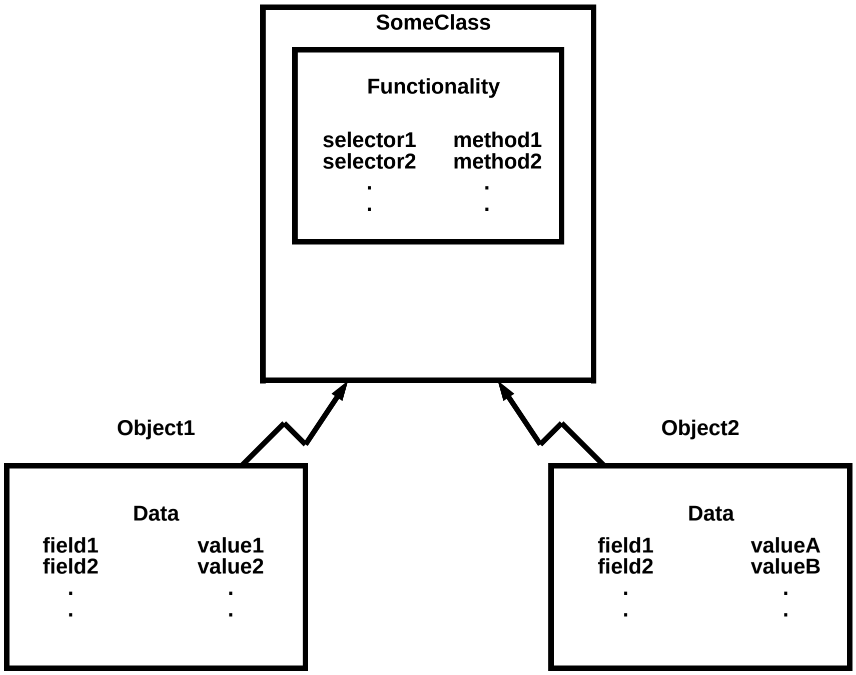


Figure 1-5. Class with Several Objects

1.1.6 Instance

An instance is an object described by a particular class. Every object within LOOPS is an instance of exactly one class.

1.2 Storage of Data in Objects

The data associated with an object is called an object's variables. Methods can change the values of these variables.

1.2 STORAGE OF DATA IN OBJECTS

1.2.1 Class Variables and Instance Variables

LOOPS supports two kinds of variables:

- Instance variables, often abbreviated IVs.

  Instance variables contain the information specific to an instance.

- Class variables, often abbreviated CVs.

  Class variables contain information shared by all instances of the class. A class variable is typically used for information about a class taken as a whole.

Both kinds of variables have names, values, and other properties. For example, the class for **Point** could specify two instance variables, **x** and **y**, and a class variable, **lastPoint**, used by methods associated with all points.

For any particular instance, you can access the values for the instance variables specific to that instance. You can also access the values for the class variables that are available to all instances of the same class.

Determining the value of a class variable requires a similar lookup procedure to that occurring when searching for a method to execute. Instance variable values are stored within the instances, and class variable values are stored within the class.

A class describes the structure of its instances by specifying the names and default values of instance variables, as shown in Figure 1-6. In this way, when a message is sent to a class to create a new instance, LOOPS can determine from the class description the number of instance variables for which it must allocate space the the initial values for those variables.

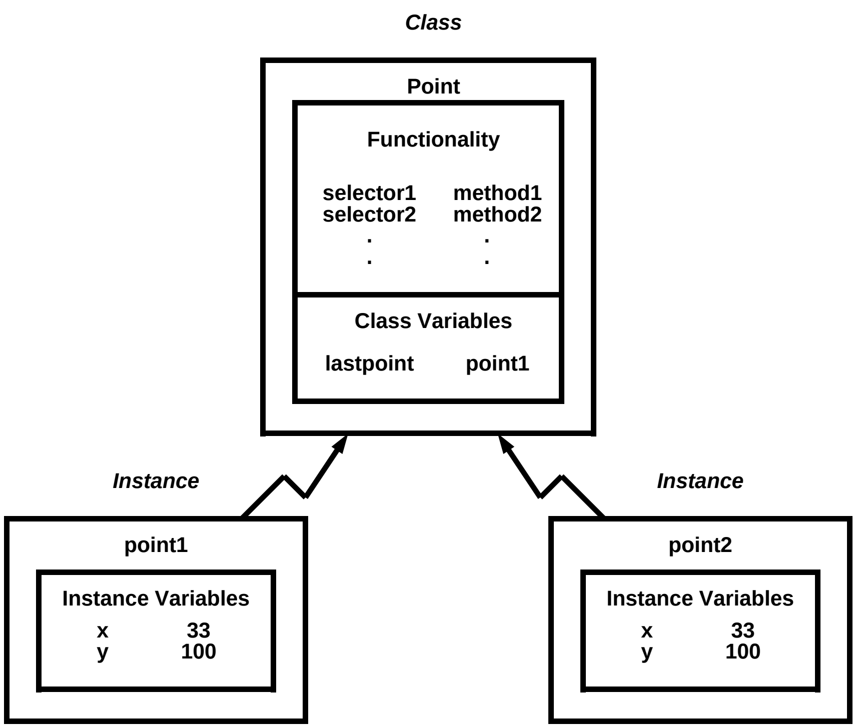


Figure 1-6. Class Variables and Instance Variables

1.2.2 Properties

LOOPS provides extensible property lists for classes, their variables, and their methods. Property lists provide places for storing documentation and additional kinds of information. For example, in a knowledge engineering application, a property list for an instance variable could be used to store the following information:

- Support (reasons for believing a value)
- Certainty factor (numeric assessments of degree of belie[redacted]•Cntanso aints on values
- Dependencies (relationships to other variables)
- Histories (previous values)

1.3 METACLASSES

1.3 Metaclasses

Classes themselves are instances of some class. Metaclasses are classes whose instances are classes. 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**. This message is handled by the class that describes the class receiving the message. For most classes, this method is provided by the standard metaclass for classes **Class**.

To create a new class, send a message to the class **Class**. The class that handles this message is **MetaClass**. Instances of **MetaClass** are classes that describe objects which are classes. Instances of **Class** are classes whose instances are not classes. Figure 1-7 shows an instance of **MetaClass**, which is a class named **Class**, and instances of Class, which are named **Window** and **Point**.

Another class available in the system in **AbstractClass**. This is useful when creating classes that implement general functionality, which must then be specialized into instantiable classes. Instances of this class are classes that are impossible to instantiate. An example of an **AbstractClass** is **ActiveValue**, which is described in Chapter 5, Active Values.

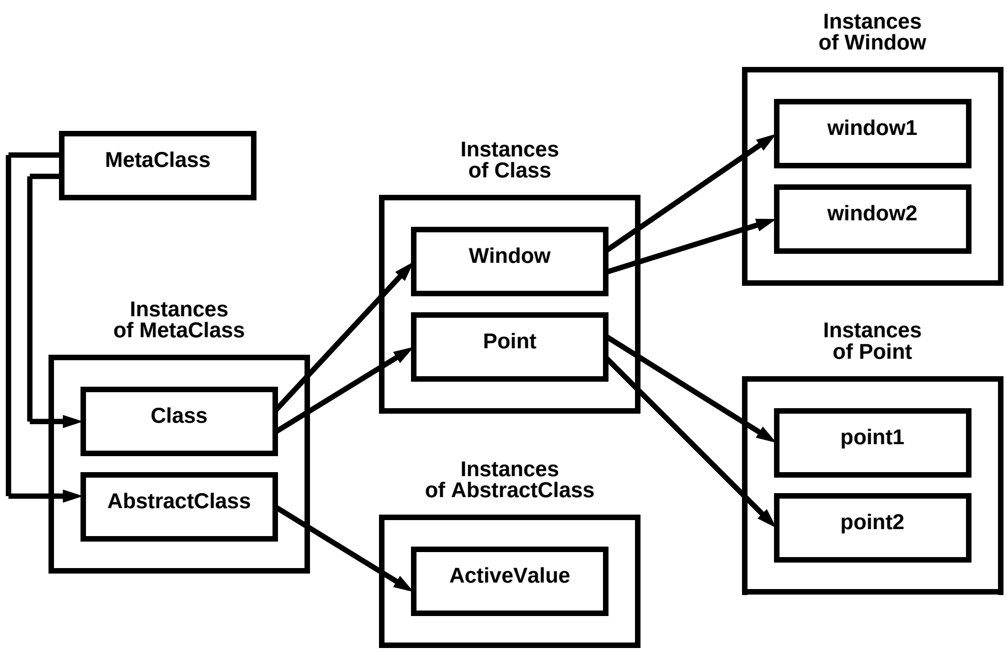


Figure 1-7. A Metaclass and its Instances

1.4 Introduction to Inheritance

Inheritance allows you to organize information in objects. With a few incremental changes, you can use inheritance to create objects that are almost like other objects. Inheritance allows you to avoid specifying redundant information and simplifies updating, since information that is common to several objects need be changed in only one place.

LOOPS objects exist in an inheritance network of classes. Figure 1-8 shows an example in which a class **3DPoint** is a subclass of another class **Point**. Instances of **3DPoint** contain instance variables that are defined in

1.4 INTRODUCTION TO INHERITANCE

Point as well as **3DPoint**. **Point** is referred to as a superclass of **3DPoint.** When an instance of **3DPoint** is created, the instance variables it contains and the messages to which it responds are not limited to those instance variables or methods as defined in the class **3DPoint**. For example, the object **pt2** contains three instance varriables; two of them are inherited from the class **Point** and the other defined in the class **3DPoint.** This instance can also respond to three different messages containing one of the three different selectors: **selector1**, **selector2**, or **selector3**.

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 first in the class, and then at the superclasses specified 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 superclass is changed, thus maintaining the same semantics as the search. Each class can specify inheritance of structure and behavior from any number of superclasses.

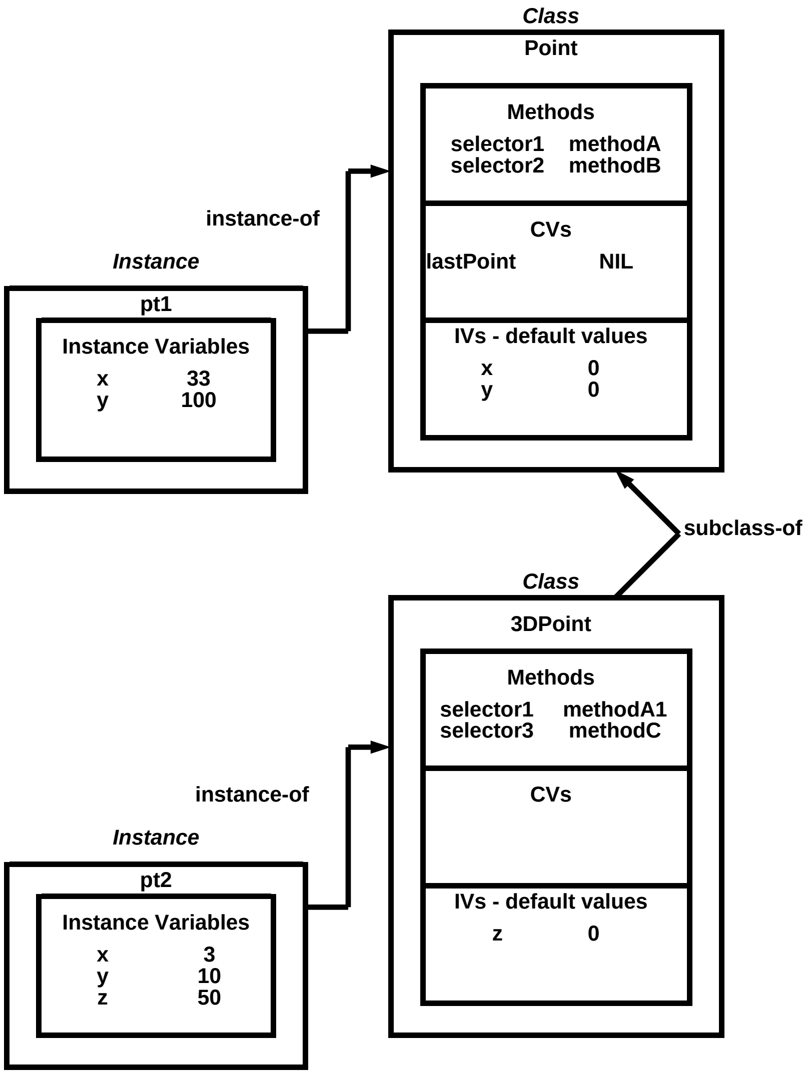


Figure 1-8. A Sample Inheritance Network

1.4.1 Single Superclasses

In the simplest case, each class specifies only one superclass. If the class **A** has the supers list **(B)**, which is a one-element list containing **B**, then all of the instance variables specified local to **A** are added to those specified for **B**, recursively. That is, **A** gets all those instance variables described in **B** and all of **B**'s supers. For example, in Figure 1-9, **A** has instance variables **x**, **z**, and **B1**.

Any conflict of variable names is resolved by using the description closer to **A** in traversing up the hierarchy to its top at the class **Object**. Method lookup uses the same conflict resolutiion. The method to respond to a message is obtained by first searching in **A**, and then searching recursively in **A**'s supers list. For example, in Figure 1-9, the method **selector2** uses **methodA2** instead of **methodB2**.

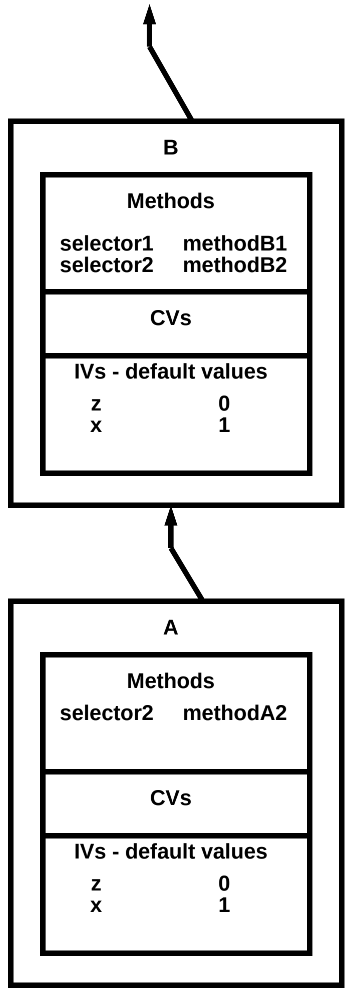


Figure 1-9. A Class with a Single Superclass

1.4.2 Multiple Superclasses

Classes in LOOPS can have more than one class specified on their supers list. Multiple superclasses permit a modular programming style where the following conditions hold:

- Methods and associated variables for implementing a particular feature are placed in a single class.
- Objects requiring combinations of independent features inherit them from multiple supers.

As in Figure 1-10, if **A** has the supers list **(B C)**, first the description from **A** is used, then the description from **B** and its supers is inherited, and finally the description from **C** and its supers. In the simplest usage, the different features have unique variable names and selectors in each super. In case of a name conflict, LOOPS uses a depth first left-to-right precedence.

For example, if any super of **B** had a method for **selector3**, then it would be used instead of the method **methodC3** from **C**. In every case, inheritance from **Object** is only considered after all other classes on the recursively defined supers list. The general rule is left-to-right, depth first, up to where the separate branches of the hierarchy join together; that is, up to any class that is repeated. Alternatively, consider the list as generated by listing all the superclasses in a depth first left-to-right order, eliminating all but the last occurrence of a class in the list.

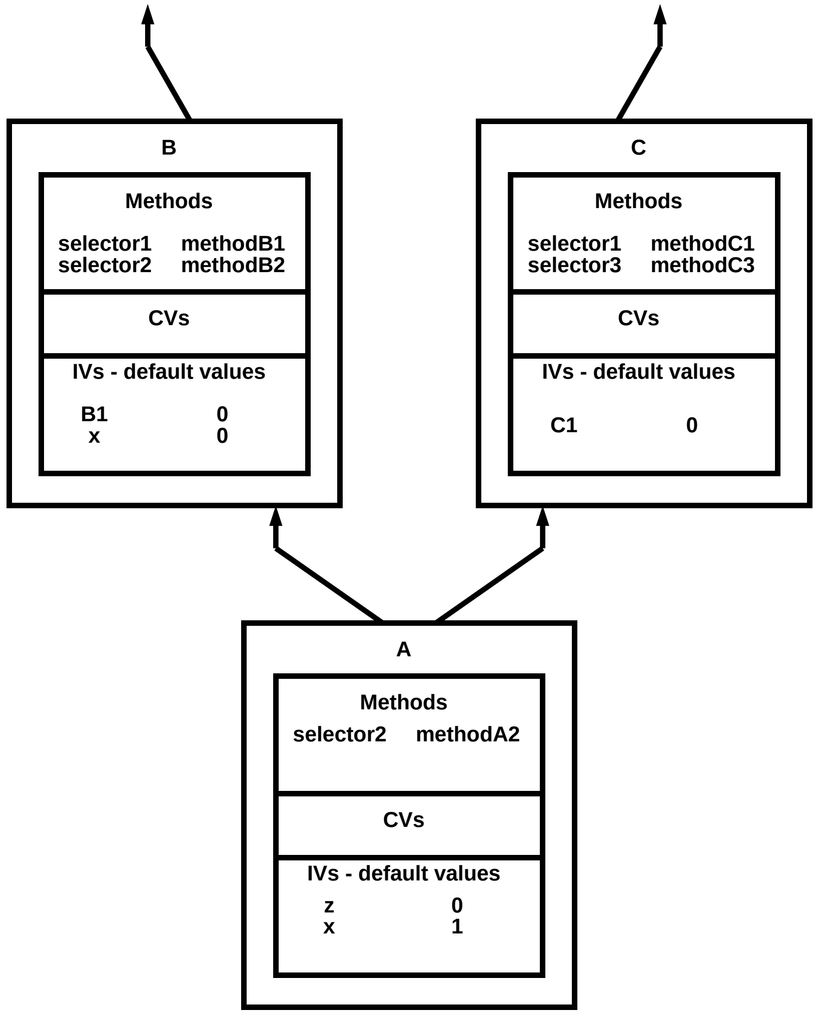


Figure 1-10. A Class with Multiple Superclasses

1.5 Introduction to Access-Oriented Programming: Using Active Values

In access-oriented programming, you can specify a particular procedure to invoke for reqad or write access of any variable of an object. LOOPS checks every object variable access to determine whether the value is marked as an active value. An active value is a LOOPS object. If a variable is marked as an active value, then aa message is sent to the active value object whenever the variable is read or set. This mechanism is dual to the notion of sending 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.

The messages sent to the active value object will depend on the type of access If you try to read a variable, the message **GettingWrapped Value** is sent to the active value object. If you try to set a variable, the message **Putting Wrapped Value** is sent. The object receiving the message may or may not trigger side effects as the result of receiving these messages. In this way, you have control over the side effects that may occur as a result of accessing data.

Active values enable one process to monitor another one. For example, LOOPS has debugging tools that use active values to trace and trap references to object variables. A graphics module 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.

Active values can also be used to maintain constraints among data in a system. As one piece of data changes, the active value associated with that data can contain functionality that updates other data within the system. Examples of this are spreadsheets or electric circuit modeling.

A powerful feature of active values is that they can be nested to yield a natural composition of the access functions.

1.6 Introduction to the LOOPS User Interface

A key feature of LOOPS is its smooth integration with the Venue Medley environment. Many of the tools within Medley have been extended to provide the necessary functionality for manipulating objects. Among these tools are the following:

- SEdit
- The inspector
- Masterscope
- The File Manager
- The Library Module Grapher

This section describes how LOOPS interfaces with each of these Medley tools.

Another aspect of LOOPS is that objects have a name space that is separate from the Lisp name space. LOOPS names are Interlisp symbols. Applying a LOOPS function **$** to a Medley symbol extracts a pointer to a LOOPS object. Objects can also be pointed to as a Lisp value.

1.6.1 SEdit

Class structures are Lisp data types. To change a class structure, LOOPS creates a list structure source for the class definition. This list can then be edited easily by SEdit. Upon exiting SEdit, the list structure is converted back to a data type. This process of converting to and from a list is hidden from view.

1.6.2 Inspector

Inspector macros have been defined within LOOPS that allow you to view the necessary class and instance data while hiding implementation details. Inspectors opened on classes or instances also provide functionality for changing the way one views an object. As an example, you can inspect a class and see or not see information inherited from its superclasses.

1.6.3 Masterscope

The Library Module Masterscope has been extended to a LOOPS Library Module so that message sending and the use of instance and class variables are understood. The functionality of CHECK has been extended to allow consistency checking of LOOPS methods.

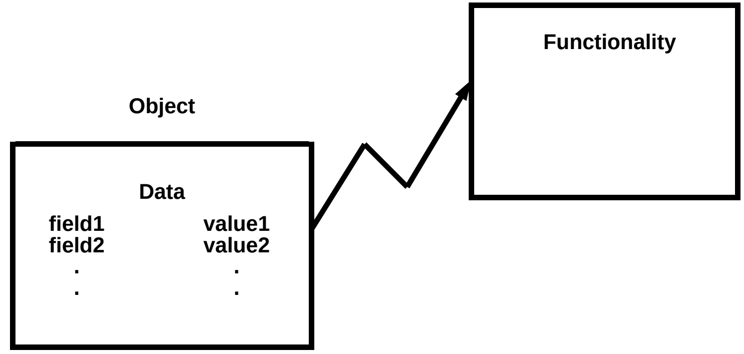
1.6.4 File Manager

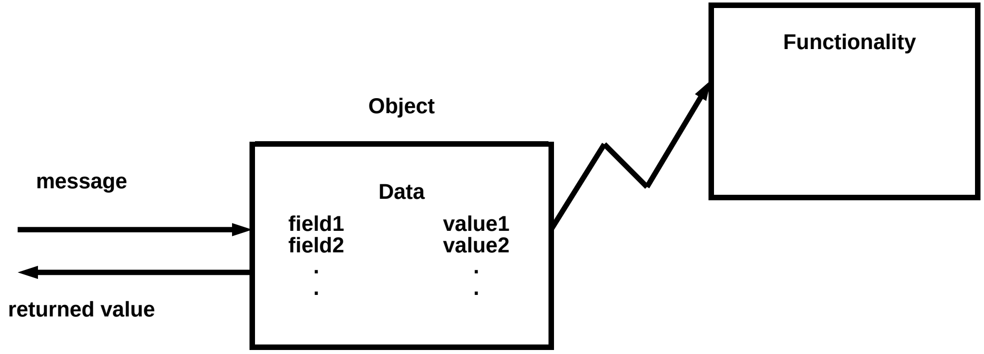
Additional File Manager commands have been added to allow you to save classes, instances, and methods on files.

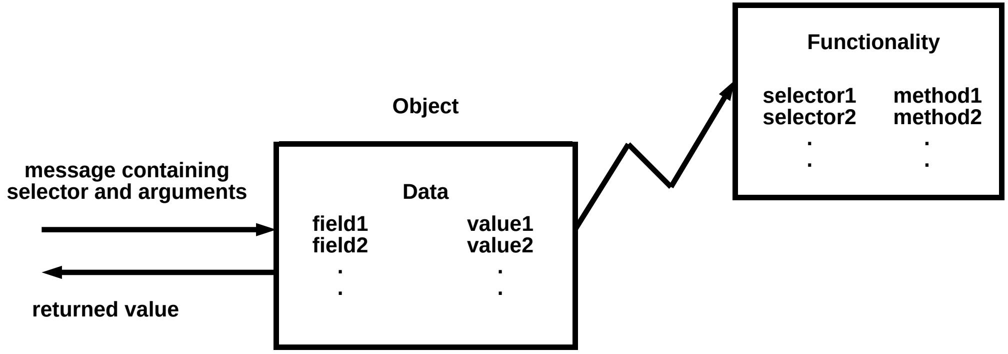
1.6.5 Grapher Module

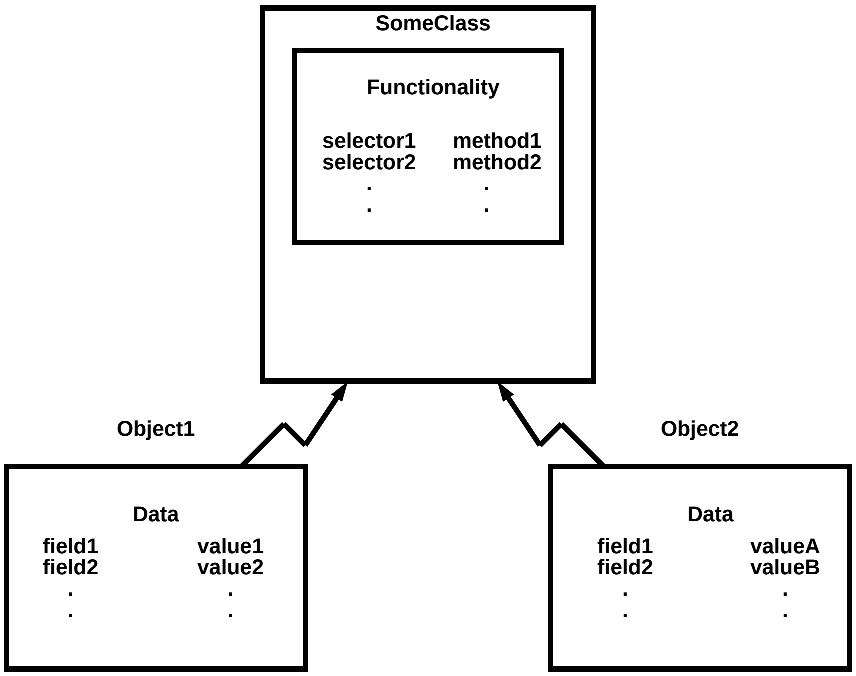
An important part of the LOOPS interface is its ability to show relationships between objects and to enable the programmer to easily manipulate those objects. Browsers of various kinds are in the system to allow you to understand the relationships between classes and how those classes are related to files. The browsers are built upon the Library module Grapher. You can easily extend the built-in browsers to create views onto any object relationship. An example of this is a decision tree where each node was an object representing a particular state of a system.

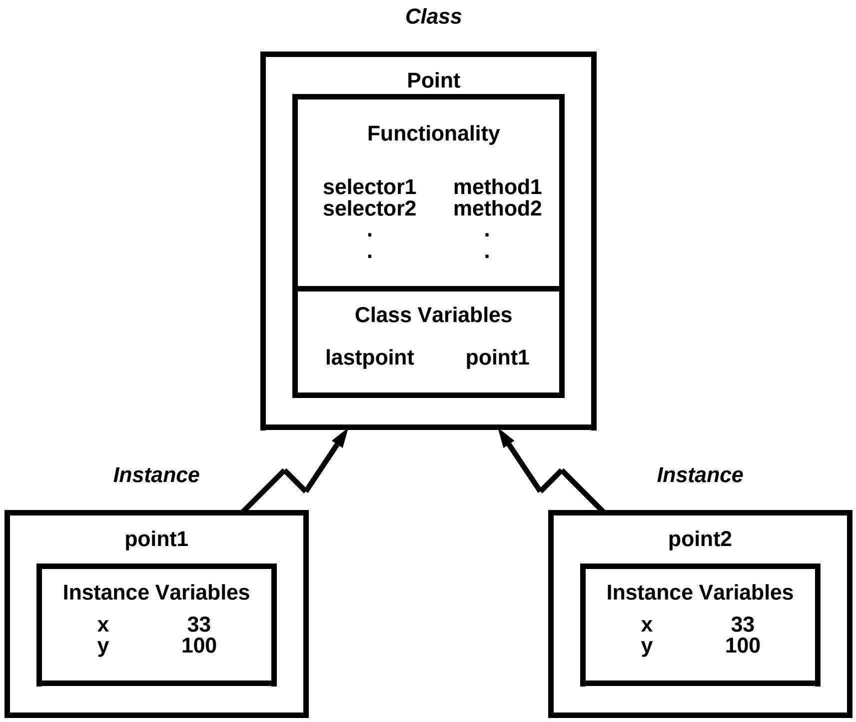
[This page intentionally left blank]

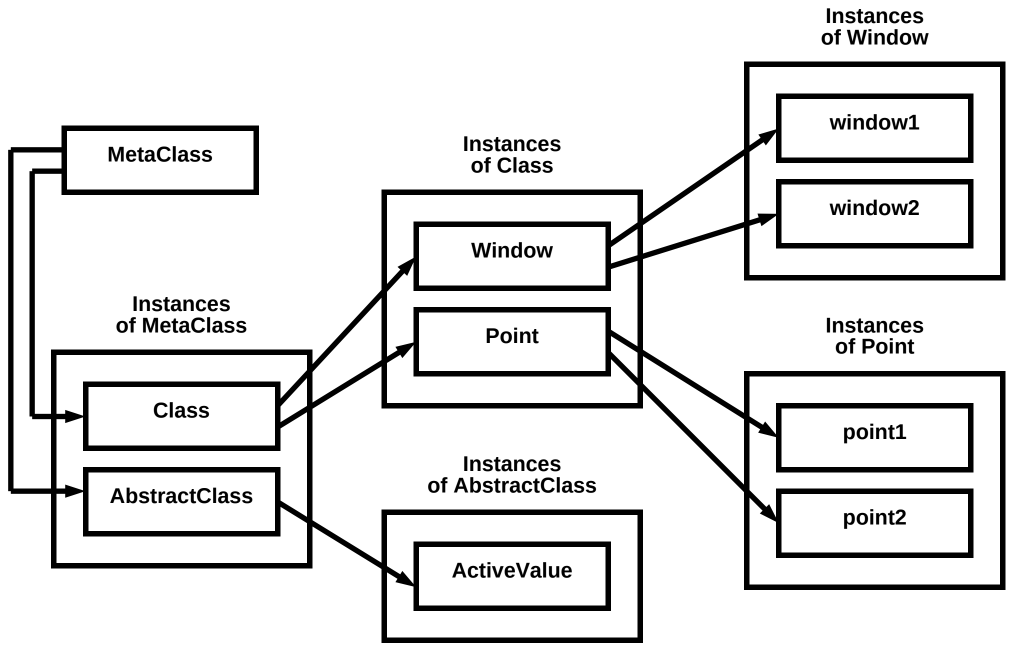

Object
Data
field1 value1
field2 value2
.
.
Functionality


Object
Functionality
message
Data
field1
field2
value1
value2
returned value

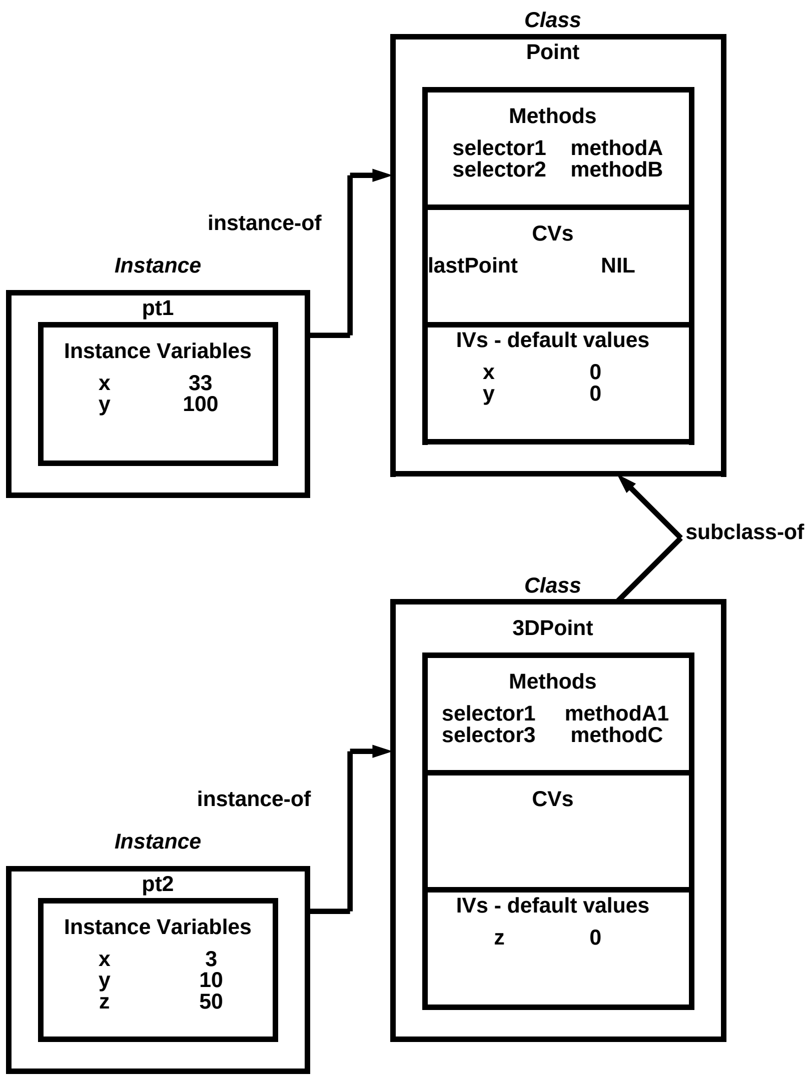

Object
message containing
selector and arguments
returned value
Data
field1
field2
value1
value2
Functionality
selector1
selector2
method1
method2

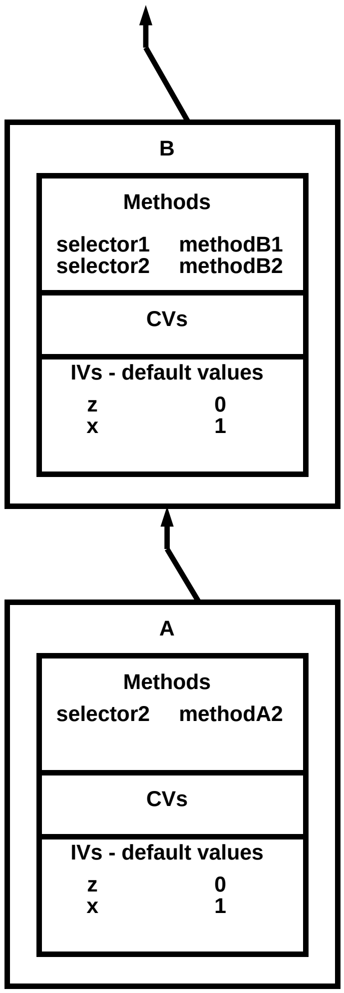

SomeClass
Functionality
selector1 method1
selector2 method2
Object1
Object2
Data
field1 value1
field2 value2
Data
field1 valueA
field2 valueB


Class
Point
Functionality
selector1 method1
selector2 method2
Class Variables
lastpoint point1
Instance
Instance
point1
Instance Variables
x 33
y 100
point2
Instance Variables
x 33
y 100



Instances
of Window
MetaClass
window1
Instances
of Class
window2
Window
Instances
of MetaClass
Point
Instances
of Point
Class
point1
Instances
of AbstractClass
AbstractClass
point2
ActiveValue


Class
Point
Methods
selector1 methodA
selector2 methodB
CVs
lastPoint NIL
IVs - default values
x 0
y 0
instance-of
Instance
pt1
Instance Variables
x 33
y 100
subclass-of
Class
3DPoint
Methods
selector1 methodA1
selector3 methodC
CVs
IVs - default values
z 0
instance-of
Instance
pt2
Instance Variables
x 3
y 10
z 50


B
Methods
selector1 methodB1
selector2 methodB2
CVs
IVs - default values
z 0
x 1
A
Methods
selector2 methodA2
CVs
IVs - default values
z 0
x 1

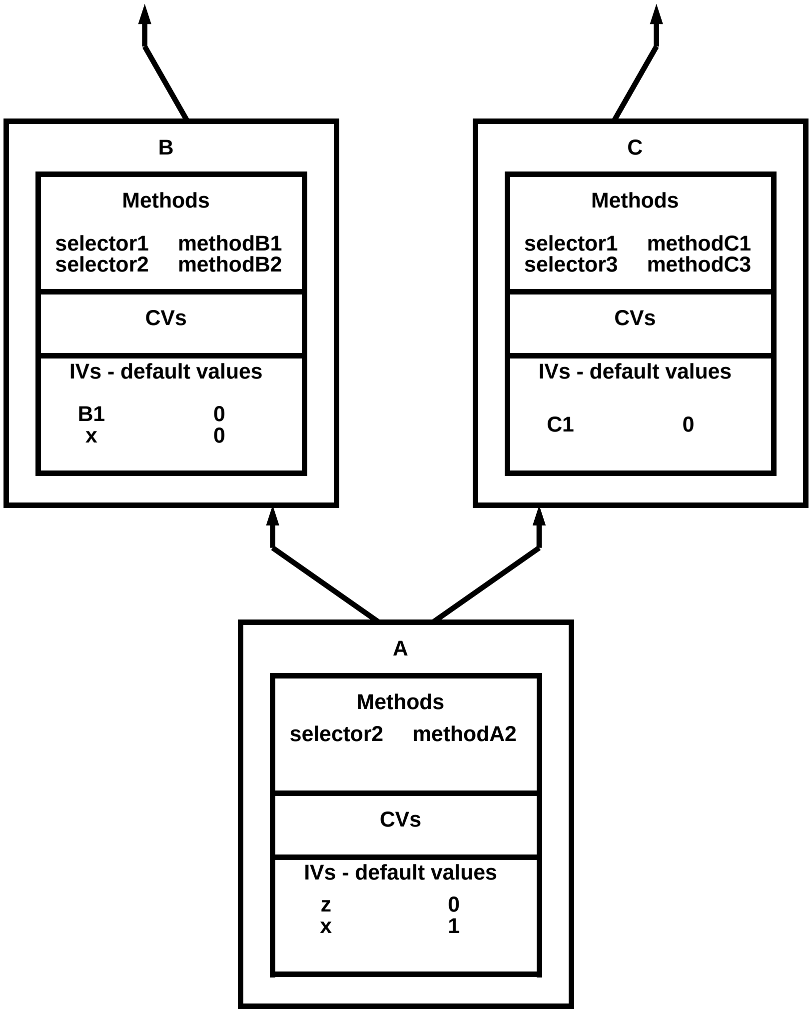

B
Methods
selector1 methodB1
selector2 methodB2
CVs
IVs - default values
B1 0
x 0
C
Methods
selector1 methodC1
selector3 methodC3
CVs
IVs - default values
C1 0
A
Methods
selector2 methodA2
CVs
IVs - default values
z 0
x 1

TABLE OF CONTENTS

2. INSTANCES

3. CLASSES

4. METACLASSES

8. ACTIVE VALUES

9. DATA TYPE PREDICATES AND ITERATIVE OPERATORS

10. BROWSERS

2. INSTANCES

Every object within the LOOPS system is an instance of some class. In this manual, however, the word instance generally refers to objects that are not themselves classes. Instances are a data type that contain local storage for instance variables, a pointer to the class that describes the instance, the Unique Identifier (UID), and other information.

This chapter describes naming and creating instances, accessing data stored within instances or pointed to by instances, and other related topics.

2.1 Instance Naming Conventions

A separate name space for LOOPS objects is maintained by the LOOPS system within a separate object name table. Since Lisp structures and LOOPS objects are stored in separate name tables, you can use the same symbol to refer to both a Lisp structure and a LOOPS object.

Note: The separate name space is not implemented by using the Common Lisp Package System.

Instances are not created with names; therefore, it may be necessary to keep pointers to them. Two ways are available to create pointers:

- Use Lisp variables, as in:

  ```
  (SETQ window1 (← ($ Window) New))
  ```

  This creates an instance of the class **Window** that can be referenced by the Lisp variable **window1**.

- Use a LOOPS name. This can be done in two ways:

  - Assign a name at the same time the instance is created. This can be done by using

    ```
    (← ($ Window) New 'window2)
    ```

    as described above. This creates an instance of the class **Window** that can be referenced by the LOOPS expression `($` window2).

  - Use the message **SetName** if you have a pointer to an object and want to assign a LOOPS name to that object.

The following table shows the items that manipulate LOOPS names.

Name	Type	Description
$	NLambda and Macro	Distinguishes between the Lisp value of a symbol and the LOOPS value of the same symbol; does not evaluate its argument.

$!	Function	Distinguishes between the Lisp value of a symbol and the LOOPS value of the same symbol; evaluates its argument.
SetName	Method	Assigns a LOOPS name to an object.
UnSetName	Method	Removes a name pointer to an object.
Rename	Method	Changes the name of an object.
GetObjectNames	Function	Returns the names of an object, including its UID.
ErrorOnNameConflict	Variable	Causes a break to occur when an attempting to name an object that already has a LOOPS name.

(**$** *name*) [NLambda and Macro]

Purpose/Behavior: Returns a pointer to a LOOPS object specified by the LOOPS name *name*. If no object exists for *name*, NIL is returned.

Arguments: *name* A LOOPS name.

Returns: Pointer to a LOOPS object or NIL; see Behavior.

Example: Given that

```
24←(← ($ Window) New 'window2)
#,($& Window (NEW0.1Y%:.;h.eN6 . 495))
```

then

```
25←($ window2)
#,($& Window (NEW0.1Y%:.;h.eN6 . 495))
```

The returned value is a pointer to the new window instance. For a further explanation, see Chapter 18, Reading and Printing.

(**$!** *name*) [Function]

Purpose/Behavior: Returns a pointer to an object specified by the value of the variable *name*, given that the value is a LOOPS name. If no object exists for *name*, NIL is returned.

Arguments: *name* Evaluates to a valid LOOPS name.

Returns: Pointer to a LOOPS object or NIL; see Behavior.

Example: Given that

```
26←(SETQ foo 'Window)
Window
```

and **Window** is a LOOPS object, then

```
27←($! foo)
#,($C Window)
```

(← *self* **SetName** *name*) [Method of Object]

Purpose: Assigns a LOOPS name to an object.

Behavior: If *name* is NIL, then a break occurs. If *name* is not a symbol, a break occurs. If *name* is already in use as a LOOPS name, and if the variable

ErrorOnNameConflict is non-NIL, then a break occurs, giving you the chance to OK "rebinding" *name*.

Note: If an object has multiple names, (← *self* **SetName** *NewName*) results in both the old name and new name appearing when (FILES?) is executed. The instance is also printed twice on the file if both names are specified to be saved.

Arguments: *self* An object.

name The LOOPS name to be given to the object; must be a symbol.

Returns: *self*

Categories: Object

Specializations: Class

Example: Given the commands

```
28←(SETQ window1 (← ($ Window) New))
#,($& Window (NEW0.1Y%:.;h.eN6 . 496))

29←(← window1 SetName 'window3)
#,($& Window (NEW0.1Y%:.;h.eN6 . 496))
```

the Lisp variable **window1** and the LOOPS expression

```
($ window3)
```

now point to the same object.

(← *self* **UnSetName** *name*) [Method of Object]

Purpose: Removes a LOOPS name pointer to an object.

Behavior: Removes the reference of *name* to *self* from the object name table maintained by the LOOPS system. If *name* is NIL, all names pointing to *self* in the object name table are removed from the files on **FILELST**. If *name* is non-NIL and the instance is associated with any files on **FILELST**, the instance is removed from those files. If *name* is not a valid LOOPS name for the object in question, an error occurs.

Arguments: *self* An object.

name A LOOPS name.

Returns: Used for side effect only.

Categories: Object

(← *self* **Rename** *newName oldNames*) [Method of Object]

Purpose: Changes the name of an object.

Behavior: If *oldNames* is NIL, removes all old names when *newName* is installed as the name for *self*; otherwise replaces only names specified in *oldNames* by *newName*. If *oldNames* is not a valid LOOPS name for the object in question, an error occurs.

Arguments: *self* Evaluates to a LOOPS name.

newName The LOOPS name to be given to the object; must be a symbol.

oldNames List of symbols whose names are to be removed; if NIL, all old names are removed when *newName* is installed as the name for *self*.

Returns: *self*

Categories: Object

Specializations: Class

Example: Examine the following expressions to see the effects of **Rename**.

```
30←($ window2)
#,($& Window (NEW0.1Y%:.H53.G2A . 496))

31←(← ($ window2) Rename 'MyWindow)
#,($& Window (NEW0.1Y%:.H53.G2A . 496))

32←($ window2)
NIL

33←($ MyWindow)
#,($& Window (NEW0.1Y%:.H53.G2A . 496))
```

(**GetObjectNames** *object*) [Function]

Purpose/Behavior: Returns the names of *object*, including its UID.

Arguments: *object* A LOOPS object.

Returns: The names of *object*, including its UID.

Example: The command

```
(PROGN
  (← ($ Window) New 'w1)
  (← ($ w1) SetName 'w1again)
  (GetObjectNames ($ w1)))
```

returns

```
(w1again w1 (NEW0.1Y%:.H53.G2A . 497))
```

ErrorOnNameConflict [Variable]

Purpose/Behavior: Behavior depends on the value.

- If NIL, the existing object is replaced by a new object.
- If non-NIL, a break occurs when an attempt is made to give an object a name that is already in use as a LOOPS name.

Initially, the value for **ErrorOnNameConflict** is NIL.

2.2 CREATING INSTANCES

2.2 Creating Instances

When an instance is created by sending the **New** message to a class, the default behavior for **Class.New** is to send the message **NewInstance** to the newly created object. If you require that special or additional operations occur at instance creation time, specialize the method **NewInstance**. Specializations of the **NewInstance** method should return *self*. You also have the capability to pass arguments to the **NewInstance** method when the **New** message is sent to create the instance. For example, the following defines a class **NamedClass** which adds the instance variable **name** and specializes **New** to set that instance variable to the name of the instance when created.

```
(DefineClass 'NamedClass)
(←($ NamedClass) AddIV 'name)
(DefineMethod ($ NamedClass) 'New '(self name)
'(←@ (←self NewIstance name) name name))
```

You can also indicate whether instances are to be saved on files using the File Manager, which is described in Chapter 14, File Manager.

The following table shows the methods in this section.

Name	Type	Description
New	Method	Creates a new object of a particular class.
←**New**	Macro	Creates an object and sends a message to it.
NewInstance	Method	Allows initialization of newly created instances by class.
NewWithValues	Method	Creates an object with instance variables of assigned values.

(← *class* **New** *name arg1 arg2 ...*) [Method of Class]

Purpose: Creates a new object, which is an instance of the class *class*.

Behavior: Creates a new instance *name* and then sends the message (← "the new instance" **NewInstance** *name arg1arg2 ...*)

In the default case, the **New** method uses the default values for the instance variable values in the newly created instance. These default values are given in the instance variable descriptions of the given class. When that process is finished, the instance can be altered in various ways by sending it messages. Specializations of the **New** method should return the new instance, and can take more arguments after *name*.

The internal data structure of an instance contains a pointer to the class of which it is an instance.

Arguments:

class Pointer to a class.

name Name assigned to the instance; if NIL, object does not have a LOOPS name.

arg1arg2... Arguments passed to the **NewInstance** method.

Returns: Newly created instance of the class.

Categories: Class

Specializations: AbstractClass, MetaClass

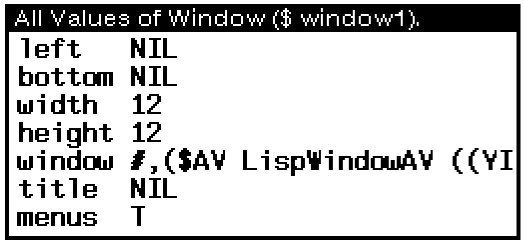
Example: The following command creates a new instance named **window1** of class **Window**.

```
20←(← ($ Window) New 'window1)
#,($& Window (NEW0.1Y%:.;h.eN6 . 515))
```

The command

```
21←(INSPECT (← ($ Window) New))
```

results in the following inspector window:



Some of the values assigned to the various instance variables are default values. These values are defined in the class **Window**.

(←**New** *class selector args*) [Macro]

Purpose: Creates an instance and sends a message to it within one form.

←**New** is pronounced "send new."

Behavior: Is equivalent to the form

(← (← *class* **New**) *selector args*)

Arguments:

class Evaluates to a class.

selector Name of the message to be sent to the new instance.

args Arguments to be sent to the function invoked by the message.

Returns: The new instance.

Example: The command

```
23←(←New ($ Window) Open)
```

creates a new instance of the class **Window** and then sends the message **Open** to the newly created object.

(← *self* **NewInstance** *name arg1 arg2 arg3 arg4 arg5*) [Method of Object]

Purpose: Allows initialization of newly created instances by the class of the instance, as opposed to the metaclass. Subclasses of **Object** that specialize this method should have a ←**Super** form within the method to allow the execution of the default behavior.

Behavior: Not normally called directly, but is sent by method **New**. The default behavior is as follows.

If *name* is non-NIL, the message **SetName** is sent to *self*.

Within *self*, instance variables that are bound to the value of **NotSetValue** and have an **:initForm** property in the class description are filled. This allows you to override the **:initForm** behavior by setting values for instance variables before executing the ←**Super** form. See the discussion of **:initForm** in Section 2.3, "Data Storage in Instances at Creation Time."

Sends the message **SaveInstance** to *self* with the argument *name*.

Note: Specializations of the **NewInstance** method should return *self*.

Arguments: *self* Evaluates to a class.

name LOOPS name given to a new instance.

arg1...arg5 Optional arguments referenced by user-written specialization code.

Returns: LOOPS name of new object created.

Categories: Object

Specializations: IndexedObject

(← *class* **NewWithValues** *valDescriptionList*) [Method of Class]

Purpose: Creates a new object and initializes the instance variables specified in *valDescriptionList*.

Behavior: Creates the object with no other initialization, directly installs the values and property lists specified in *valDescriptionList*, and returns the created object. Variables that have no description in *valDescriptionList* are given no value in the instance and thus inherit the default value from the class.

NewWithValues does not invoke the **NewInstance** method or the **:initForm** properties (see Section 2.3, "Data Storage in Instances at Creation Time"). This means that the instance is not recognized by the File Manager; to be recognized, the instance must be named.

Arguments: *class* Pointer to a class.

valDescriptionList
Evaluates to a list of value descriptions, each of which is a list of variableNames and properties, for example,

*((VarName1 value1 prop1a propVal1a prop1b propVal1b ...)
(VarName2 value2 prop2a propVal2a prop2b propVal2b ...) ...)*

Returns: The created object.

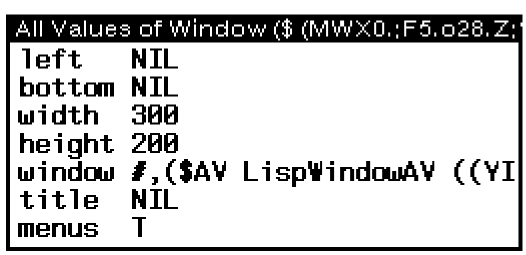
Categories: Class

Specializations: MetaClass

Example: The command

```
22←(INSPECT (← ($ Window) NewWithValues '((width 300)(height 200))))
```

results in the following inspector window:



Contrast the values for the instance variables width and height with the inspector window for **New**, above.

2.3 DATA STORAGE IN INSTANCES AT CREATION TIME

2.3 Data Storage in Instances at Creation Time

When an instance is first created, the value of the variable **NotSetValue** is assigned to its instance variables. **NotSetValue** is initialized to be an active value of the class **NotSetValue** and should not be changed by the user. Trying to access an instance variable triggers this active value which in turn triggers the method **IVValueMissing**.

Data is stored in instances on all Puts and on **GetValues** when the default value is an active value but not **NotSetValue**. Be aware that in reading the value of an instance variable that is not stored in the instance, changes in the default value of the instance variable in the class description are seen in accesses of the instance.

One exception to this method of data storage at creation time is if an instance variable has the property **:initForm** in the class description. In this case, data is stored in the instance at the time of creation.

Testing for whether data is stored locally in the instance can be done in two ways:

- Through the user interface, you can inspect an instance in the local mode. (See Chapter 18, User Input/Output Modules, for more information.) Values not locally stored appear as #,NotSetValue.
- Programmatically, through the function **GetIVHere** with the macro **NotSetValue.**

The following table describes the items in this section.

Name	Type	Description
IVValueMissing	Method	Handles cases when an attempt is made to access the value of an instance variable that is not stored in an instance.
NotSetValue	Macro	Determines if its argument is equivalent to the value of **NotSetValue**.
:initForm	IV Property	Signals a property value that can be evaluated.

(← *self* **IVValueMissing** *varName propName typeFlg newValue*) [Method of Object]

Purpose: Invoked by the system to handle the cases when you try to access the value of an instance variable that is not stored in an instance. This is the mechanism the system uses to access default values.

Behavior: Varies according to the functionality that invoked it.

- **GetValueOnly** accesses return the default value of the instance variable stored in the class.
- **GetValue** accesses return the default value of the instance variable stored in the class if it is not an active value. If the default value is an active value, a copy of the active value is made, stored in the instance, and sent the **GetWrappedValue** message.
- **PutValueOnly** accesses store the new value in the instance.
- **PutValue** accesses store the new value in the instance unless the default value of the instance variable stored in the class is an active value. If this is the case, a copy of the active value is made, stored in the instance, and sent the **PutWrappedValue** message.

Arguments: *varName* Instance variable name.

propName	Property name for instance variable *varName*.
typeFlg	Used internally to indicate the type of access.
newValue	If called by **PutValueOnly** or **PutValue**, this is the value to be placed into the instance variable or property name.

Returns: Value depends on the functionality that invoked this method; see Behavior.

Categories: Object

(**NotSetValue** *arg*) [Macro]

Purpose: Determines if *arg* is **EQ** to the value of **NotSetValue**.

Arguments: *arg* Any value.

Returns: NIL or T.

Example: Given that

```
51←(← ($ Window) New ’w)
#,($& Window (NEW0.1Y%:.;h.eN6 . 515))
```

then

```
52←(NotSetValue (GetIVHere ($ w) ’title))
T
```

:initForm [IV property]

Purpose: This allows instance variables to be initialized at the time of the creation of an instance. The **:initForm** property and its value are in the class definition. Its value is a form that is evaluated when an instance is created. The result of the evaluation is stored as the value of the instance variable containing this property in the newly created instance.

This behavior does not hold if the value of the instance variable is not **NotSetValue**. Refer to the method **Object.NewInstance** in Section 2.2, "Creating Instances," for more information.

Example: Given the commands

```
53←(DefineClass ’testclass)
#,($C testclass)


54←(AddCIV ($ testclass) ’date NIL ’(|:initForm| (DATE)))
date
```

then

```
55←(INSPECT (← ($ testclass) New))
```

returns the following inspector window:

2.4 CHANGING THE NUMBER OF INSTANCE VARIABLES IN AN INSTANCE

2.4 CHANGING THE NUMBER OF INSTANCE VARIABLES IN AN INSTANCE

2.4 Changing the Number of Instance Variables in an Instance

An instance can contain more instance variables than are defined in the class that describes it. It is not possible to remove an instance variable from an instance if the instance variable is defined in the class.

When you try to access the value of an instance variable that is not defined as an instance variable in the instance, the **IVMissing** method is invoked.

The following table shows the functions and methods in this section.

Name	Type	Description
AddIV	Function	Adds an instance variable to an instance.
AddIV	Method	Adds an instance variable to *self*.
DeleteIV	Function	Removes an instance variable or property from an instance.
DeleteIV	Method	Removes an instance variable or property from *self*.
ConformToClass	Method	Makes *self* contain only those instance variables that are defined or inherited by the class of *self*.
IVMissing	Method	Is sent by the system when an attempt is made to access an instance variable that does not exist. It is used for recovery.

(**AddIV** *self name value propName*) [Function]

Purpose: Adds an instance variable to an instance.

Behavior: Varies according to the arguments.

- If *propName* is non-NIL and if *name* already exists, it is added as a property to the instance variable *name* with the value *value*.
- If *name* already exists, and if *propName* is NIL, the value of the instance variable *name* is changed to *value*.
- If *name* does not exist and if *propName* is non-NIL, the instance variable *name* is added to the instance and given the value of the variable **NotSetValue**. It is given the property *propName* with the value *value*.
- If *name* and *propName* already exist, the value of the property *prop* is changed to *value*.

Arguments:

self	A pointer to the instance.
name	The name of the instance variable to be added.
value	The value the new instance variable will be assigned.
propName	Property name of instance variable name; may be NIL.

Returns: Used for side effect only.

Example: Given that

```
55←(← ($ Window) New ’w)
```

the command

```
56←(AddIV ($ w) ’left 1234)
```

changes the value of the instance variable **left** to 1234. The command

```
57←(AddIV ($ w) 'foo 1234)
```

adds the instance variable **foo** to ($ w) and gives it the value 1234.

(← *self* **AddIV** *name value propName*) [Method of Object]

Purpose: Adds an instance variable to *self*.

Behavior: Method form of the function **AddIV**.

Arguments: See the function **AddIV**.

Returns: NIL

Categories: Object

Specializations: Class

Example: Given that

```
58←(← ($ Window) New 'w)
```

the command

```
59←(← ($ w) AddIV 'left 1234)
```

changes the value of the instance variable **left** to 1234. The command

```
60←(← ($ w) AddIV 'foo 1234)
```

adds the instance variable **foo** to ($ w) and gives it the value 1234.

(**DeleteIV** *self varName propName*) [Function]

Purpose: Removes an instance variable or property from an instance.

Behavior: Varies according to the arguments.

- If *self* does not have *varName*, an error occurs.
- If *varName* is defined in the class or a super class of *self*, an error occurs.
- If the instance *self* has *varName*, and *propName* is NIL, the instance variable is deleted.
- If *propName* is non-NIL, it is deleted only if it is a locally stored property, that is, not defined in a class. If *propName* is not a property of *varName* or is defined in a class, no error occurs.

Arguments:

self	A pointer to the instance from which the instance variable is to be deleted.
varName	The name of the instance variable to be deleted.
propName	If non-NIL, specifies that a property, not an instance variable, is to be deleted.

Returns: If no errors occur, this returns *self*.

Example: The following command deletes the instance variable **foo** from ($ w):

```
62←(DeleteIV ($ w) 'foo)
```

(← *self* **DeleteIV** *varName propName*) [Method of Object]

Purpose: Deletes an instance variable or property from *self*.

Behavior: Method version of the function **DeleteIV**.

Arguments: See the function **DeleteIV**.

Returns: If no errors occur, this returns *self*.

Categories: Object

(← *self* **ConformToClass**) [Method of Object]

Purpose/Behavior: Makes *self* contain only those instance variables that are defined in or inherited by the class of *self*.

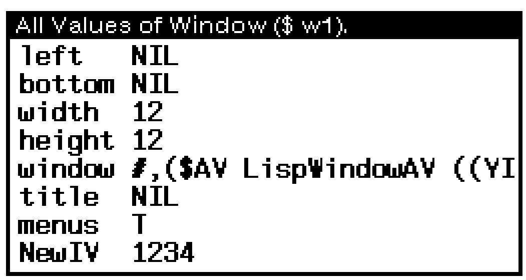
Returns: NIL

Categories: Object

Example: This example adds an instance variable to an instance and shows how **ConformToClass** removes it.

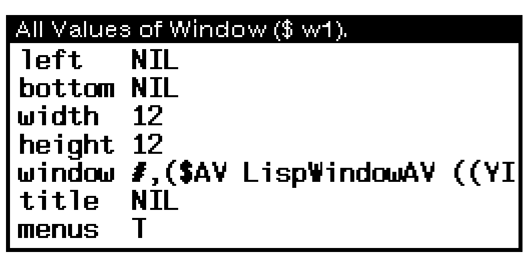
```
63←(← ($ Window) New 'w1)
(#,($& Window (|MXWO.:F5.G18.Z:?|.18))

64←(← ($ w1) AddIV 'NewIV 1234)
1234

65←(INSPECT ($ w1))
```

This produces the following inspector window:




```
66←(← ($ w1) ConformToClass)
NIL

67←(INSPECT ($ w1))
```

This produces the following inspector window:



(← *self* **IVMissing** *varName propName typeFlg newValue*) [Method of Object]

Purpose: This message is sent by the system when an attempt is made to access an instance variable that does not exist. It is used for recovery.

Behavior: Varies according to the arguments.

- If the instance variable *varName* is now defined in the class, copy it to *self*. This can happen if the class was changed after the instance was created.
- If there is a class variable with the name *varName*, use it. The method of use is determined by the **:allocation** class variable property:
  - dynamicCached

    Copy the class variable to *self* on puts or gets.
  - dynamic

    Copy the class variable to *self* on puts. If the access is by **GetValue** or **GetValueOnly**, then get the value from the class. The value retrieved from the class is dependent on the value of *propName* and the class variable property **:initform**. If *propName* is NIL and there is a class variable property **:initform**, then retrieve the value returned from evaluating **:initform**. Otherwise, retrieve the value of the class variable *varName* if *propName* is NIL or the value of the property *propName* if it is non-NIL.
  - class (the default if there is no **:allocation** property)

    Do not copy the class variable *varName* to *self*. On puts, store the value in the class. With gets, do the same as the case when the **:allocation** property is dynamic. Essentially, this allows you to access class variables with the same syntax as instance variables.

  An attempt is made to correct the spelling of *varName* and try the above steps again before breaking.

Arguments:

self	A pointer to the instance.
varName	Instance variable name for *self*.
propName	Property name of instance variable *varName*.
typeFlg	One of **PutValue, PutValueOnly, GetValue, GetValueOnly**.
newValue	Value to be stored in *varName*.

Returns: If doing a put, this returns *NewValue*; else this returns the value of the instance variable name.

Categories: Object

Example: If **w1** is a **Window**, then the following command breaks under **Object.IVMissing** because windows do not have an instance variable named **mumble**.

```
(← ($ w1) Get 'mumble)
```


2.5 MOVING VARIABLES

2.5 Moving Variables

These functions allow you to move variables between classes.

Name	Type	Description
RenameVariable	Function	Changes a variable name in a class.

MoveVariable	Function	Moves an instance variable from one class to another.
MoveClassVariable	Function	Moves an class variable from one class to another.

(**RenameVariable** *className oldVarName newVarName classVarFlg*) [Function]

Purpose: Changes *oldVarName* to *newVarName* in class *className.*

Behavior: Can cause inconsistency without warning; does not test for references to the variable in methods of *className*.

Arguments:

className Class in which function is defined.

oldVarName Old name of variable.

newVarName New name of variable.

classVarFlg If not NIL, then *oldVarName* refers to a class variable.

Returns: If successful, returns *newVarName*; else NIL.

Example: The following command renames the class variable **OldVar** to **NewVar**.

```
27←(RenameVariable ($ MyClass) 'OldVar 'NewVar T)
```

(**MoveVariable** *oldClassName newClassName varName*) [Function]

Purpose: Moves an instance variable from *oldClassName* to *newClassName*.

Behavior: Moves both the *varName* instance variable and its description to *newClassName*. Deletes *varName* from *oldClassName*.

Arguments:

oldClassName Source class.

newClassName Destination class.

varName Variable to be moved.

Returns: Used for side effect only.

(**MoveClassVariable** *oldClassName newClassName varName*) [Function]

Purpose: Moves a class variable from *oldClassName* to *newClassName*.

Behavior: Moves the class variable *varName* and its properties to *newClassName*. Deletes *varName* from the *oldClassName*.

Arguments:

oldClassName Source class.

newClassName Destination class.

varName Class variable to be moved.

Returns: Used for side effect only.

2.6 DESTROYING INSTANCES

2.6 Destroying Instances

A protocol allows you to customize the behavior of the system at instance destruction time. The naming convention is somewhat asymmetrical to that of creation time. To programmatically influence instance creation, specialize the method **NewInstance**. To programmatically influence instance destruction, specialize the method **Destroy**. Include a (←**Super**) in specializations of **Destroy** to guarantee normal system behavior.

The following table describes the methods in this section.

Name	Type	Description
Destroy	Method	Removes an object from the environment.
Destroy!	Method	Removes an object from the environment. If the object is a class, it also destoys all subclasses.
DestroyInstance	Method	Modifies the data structure of an instance as described above.

(← *self* **Destroy**) [Method of Object]

Purpose: Removes an object from the environment.

Behavior: Sends the **DestroyInstance** message with *self* as an argument to the class of *self*. **UnmarkedAsChanged** is called to remove the instance from the notice of the File Manager.

Arguments: *self* A pointer to the instance.

Returns: Used for side effect only.

Categories: Object

Specializations: Class, DestroyedClass, IndexedObject, Window

Example: The following command destroys an instance named **window1**.

```
70←(← ($ window1) Destroy)
```

(← *self* **Destroy!**) [Method of Object]

Purpose/Behavior: Removes an object from the environment. If the object is a class, it also destoys all subclasses.

Arguments: *self* A pointer to the instance.

Returns: Used for side effect only.

Categories: Object

Specializations: Class, DestroyedClass, DestroyedObject

(← *class* **DestroyInstance** *instance*) [Method of Class]

Purpose/Behavior: Destroys *instance* by overwriting its contents. When an instance is destroyed, several things occur:

- The instance is removed from any files on **FILELST**. See the *Interlisp-D Reference Manual*.
- The instance is deleted from system hash tables used for maintaining object identities.
- The class of the instance is changed to **DestroyedObject**.
- Other fields of the internal instance data structure are set to NIL.

If an instance is only pointed to by a LOOPS name, its data structure is freed for garbage collection.

Arguments: *class* Class of *instance.*

instance The instance being destroyed.

Returns: Used for side effect only.

Categories: Class

Specializations: MetaClass, DestroyedClass

2.7 METHODS CONCERNING THE CLASS OF AN OBJECT

2.7 Methods Concerning the Class of an Object

Given an instance, you often need to manipulate the class of an instance. This section describes how to perform this manipulation.

Name	Type	Description
ChangeClass	Method	Changes the class of an instance.
Class	Macro	Determines the class of an object.
Class	Method	Determines the class of an object.
ClassName	Function	Returns the class name of an object.
ClassName	Method	Returns the class name of an object.
InstOf	Method	Determines if *self* is an instance of a class.
InstOf!	Method	Determines if *self* is an instance of a class or any of its subclasses.

You can also compute a class corresponding to a Lisp data type for Lisp objects by using **GetLispClass**, described in Chapter 4, Metaclasses.

(← *self* **ChangeClass** *newClass*) [Method of Object]

Purpose: Changes the class of an instance.

Behavior: Creates a blank instance of the *newClass*. Any instance variables that are locally stored within *self* are added to the new instance.

If *newClass* is not the name of a class or a pointer to the class, an error occurs.

Arguments: *self* A pointer to an instance.

newClass Either the name of a class or a pointer to the class.

Returns: *self*

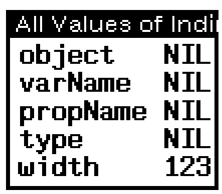
Categories: Object

Specializations: IndexedObject

Example: Create an instance of class **Window** and assign a local value to the instance variable **width** - all other instance variables of ($ w) lack local values. Then, when the class of ($ w) is changed to **IndirectVariable**, ($ w) will have all of the instance variables of its new class, plus the one instance variable of its old class which had a local value, **width**.

```
71←(← ($ Window) New 'w)
#,($& Window (NEW0.1Y%:.;h.eN6 . 501))

72←(←@ ($ w) width 123)
123

73←(← ($ w) ChangeClass 'IndirectVariable)
#,($& IndirectVariable (NEW0.1Y%:.;h.eN6 . 502))

74←(← ($ w) Inspect)
```

This produces the following inspector window:



(**Class** *self*) [Macro]

Purpose: Determines the class of an object.

Behavior: If *self* is a LOOPS object, return its class.

If *self* is not a LOOPS object, evaluate (**GetLispClass** *self*)

Arguments: *self* A pointer to a LOOPS or Lisp object.

Returns: Value depends on the arguments; see Behavior.

Example: Given that

```
75←(← ($ Window) New 'window1)
#,($& Window (NEW0.1Y%:.;h.eN6 . 503))
```

then

```
76←(Class ($ window1))
#,($C Window)
```

Note: If *self* is an annotated value, the method **Class** and the macro **Class** return different values. See Chapter 8, Active Values, for more information on annotated values.

(← *self* **Class**) [Method of Object]

Purpose/Behavior: Method version of the macro **Class**.

Arguments: *self* A pointer to a LOOPS object or a Lisp data structure.

Returns: Value depends on the arguments; see Behavior of the macro **Class**.

Categories: Object

Example: Given that

```
77←(← ($ Window) New 'window1)
#,($& Window (NEW0.1Y%:.;h.eN6 . 504))
```

then

```
78←(← ($ window1) Class)
#,($C Window)
```

(**ClassName** *self*) [Function]

Purpose: Returns the class name of the class of the object.

Behavior: Varies according to the argument.

- If *self* is a class, this returns the name of that class.
- If *self* is an instance, this returns the name of the class that describes that instance.
- If *self* is neither of these, an attempt is made to get the class of *self* by applying the function **GetLispClass** to *self*. If this returns NIL, the function **LoopsHelp** is called with the arguments *self* and "has no class name."

Arguments: *self* Can have multiple values; see Behavior.

Returns: Value depends on the argument; see Behavior.

Example: The command

```
80←(ClassName ($ Window))
```

returns

```
Window
```

(← *self* **ClassName**) [Method of Object]

Purpose/Behavior: Method version of the function **ClassName**.

Arguments: See the function **ClassName**.

Returns: Value depends on the arguments; see Behavior of the function **ClassName**.

Categories: Object

(← *self* **InstOf** *class*) [Method of Object]

Purpose/Behavior: Determines if *self* is an instance of *class*.

Arguments: *self* A pointer to an instance.

class A symbol name of a class or a pointer to a class.

Returns: T or NIL

Categories: Object

Example: Given that

```
83←(← ($ Window) New 'w1)
#,($& Window (NEW0.1Y%:.;h.eN6 . 505))
```

then

```
84←(← ($ w1) InstOf 'Window)
T
```

```
85←(← ($ w1) InstOf ($ Window))
T
```

(← *self* **InstOf!** *class*) [Method of Object]

Purpose: Determines if *self* is an instance of *class* or any of *class*'s subclasses.

Behavior: Tests if class of *self* is a subclass of *class*.

Arguments: *self* A pointer to an instance.

class A symbol name of a class or a pointer to a class.

Returns: Object

Categories: Object

2.8 COPYING INSTANCES

2.8 Copying Instances

This section describes the methods for copying instances.

Name	Type	Description
CopyDeep	Method	Copies all nested objects, annotated values, and lists.
CopyShallow	Method	Creates a new instance of the same class as *oldInstance*. Fills the instance variables of the new instance with the data contained in the old instance.

(← *oldInstance* **CopyDeep** *newObjAList*) [Method of Object]

Purpose: Copies all nested objects, annotated values, and lists. All other values are shared, not copied. This method is similar to the Interlisp function **COPYALL**.

Behavior: Creates a new instance of the same class as *oldInstance*. Fills the instance variables of the new instance with copies of lists, active values, and instances pointed to by *oldInstance*.

Arguments: *oldInstance* A pointer to an instance.

newObjAList

An association list of old copied objects with their associated copies; used to allow copying of circular structures. Users typically let this argument default to NIL.

Returns: The value of the new instance.

Categories: Object

Example: Create the class **CopyTest** with the following structure:

```
SEdit CopyTest Package: INTERLISP
((MetaClass Class Edited%:          ; Edited 22-Mar-88
                                    ; 11:07 by jrb:
  )
 (Supers Object) (ClassVariables)
 (InstanceVariables (var NIL)
        (list NIL)
        (instance NIL))
 (MethodFns))
```

Create the instance **CopyTest1** and initialize it as shown in the following inspector:

```
All Values of CopyTest ($ CopyTest1).
var      123
list     (ABC)
instance #,($ CopyTest2)
```

Now create a copy and inspect it.

```
(INSPECT (SETQ DeepCopy (← ($ CopyTest1) CopyDeep)))
```

```
All Values of CopyTest ($ (MUX0.;F5.G18.?7C . 517)).
var      123
list     (ABC)
instance #,($& CopyTest (MUX0.%:F5.G18.?7C . 518))
```

The value of the instance variable **instance** is different. Also,

```
(EQ (@ ($ CopyTest1) list)(@ DeepCopy list))
```

returns NIL.

(← *oldInstance* **CopyShallow**) [Method of Object]

Purpose/Behavior: Creates a new instance of the same class as *oldInstance*. Fills the instance variables of the new instance with the data contained in the old instance.

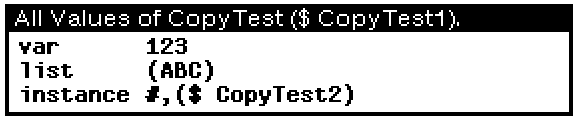
Arguments: *oldInstance*
A pointer to an instance.

Returns: A copy filled with the values shared by the instances.

Categories: Object

Example: Compare this example to the **CopyDeep** example above. Use the same **CopyTest1** instance as above.

```
(INSPECT (SETQ ShallowCopy (← ($ CopyTest1) CopyShallow)))
```



The value of the instance variable **instance** is the same. Also,

```
(EQ (@ ($ CopyTest1) list)(@ ShallowCopy list))
```

returns T.

2.9 QUERYING STRUCTURE OF INSTANCES

2.9 Querying Structure of Instances

At run time, user-written code may need to determine the structure of some object which has been passed into it. This section describes the methods to do this.

Name	Type	Description
HasCV	Method	Determines if a class variable can be accessed via *self*.
HasIV	Method	Determines if an instance variable can be accessed via *self*.
Inspect	Method	Inspects *self* as a class or instance.
ListAttribute	Method	Determines instance variable or instance variable property names contained in an instance.
ListAttribute!	Method	Recursively determines instance variable or instance variable property names contained in an instance.
WhereIs	Method	Searches the supers hierarchy to find a class where a specified name is defined.

(← *self* **HasCV** *cvName propName*) [Method of Object]

Purpose: Returns T if the class variable *cvName* (or its property *propName* if it is non-NIL) can be accessed via *self*; otherwise NIL.

Behavior: Sends the message **HasCV** to the class of *self* passing the arguments *cvName* and *propName*.

Arguments: *self* A pointer to an instance or a class.

cvName Class variable name

propName Property name for class variable *cvName*.

Returns: T or NIL; see Behavior.

Categories: Object

Specializations: Class

Example: The following command checks if an instance **window1** has the class variable **RightButtonItems**:

```
87←(← ($ window1) HasCV 'RightButtonItems)
T
```

(← *self* **HasIV** *ivName propName*) [Method of Object]

Purpose/Behavior: Returns T if the instance variable *ivName* (or its property *propName* if it is non-NIL) can be accessed via *self*; otherwise NIL.

Arguments:
- *self* — A pointer to an instance or a class.
- *ivName* — Instance variable name.
- *propName* — Property name for instance variable *ivName*.

Returns: T or NIL; see Behavior.

Categories: Object

Specializations: Class

(← *self* **Inspect** *INSPECTLOC*) [Method of Object]

Purpose/Behavior: Inspects *self* as a class or an instance. Uses *INSPECTLOC* as the region for the inspector window if it is given.

Arguments:
- *self* — A pointer to an instance.
- *INSPECTLOC* — The region for the inspector window. If NIL, the system prompts you to place the window.

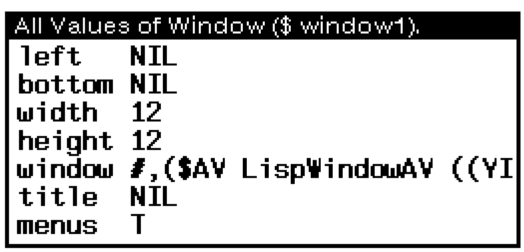
Returns: The Lisp window used by the inspector.

Categories: Object

Example: The following command inspects an instance ($ window1)

```
88←(← ($ window1) Inspect)
```

This results in the following inspector window:



(← *self* **ListAttribute** *type name*) [Method of Object]

Purpose: Determines instance variable or instance variable property names contained in an instance.

Behavior: Converts *type* into uppercase on entry. The remaining behavior varies according to the arguments.

- If *type* is one of IV, IVPROPS, or NIL, and *name* is the name of an instance variable of *self*, this returns a list of property names of *name* that have property values locally stored in the instance.

- If *type* is IVS, this returns a list of the instance variable names of *self*, whether or not the values for the instance variables are locally stored.
- If *type* is none of the above, this evaluates (← (**Class** *self*) **ListAttribute** *type name*).

Note: Using a *type* of **SUPERS** or **SUPERCLASSES** returns a list of the names of the super classes.

Arguments: *self* A pointer to an instance.

type See Behavior.

name If *type* is one of IV, IVPROPS, or NIL, then *name* is an instance variable of *self*; else it is NIL.

Returns: Value depends on the arguments; see Behavior.

Categories: Object

Specializations: Class

Example: Given that

```
90←(← ($ Window) New 'w1)
#,($& Window (NEW0.1Y%:.;h.eN6 . 515))
```

then

```
91←(← ($ w1) ListAttribute 'IVS)
(left bottom width height title menus)

92←(← ($ w1) ListAttribute 'IV 'menus)
NIL
```

After opening ($ w1), positioning the cursor anywhere on the window, and pressing the left and right mouse buttons to create some menus, then

```
93←(← ($ w1) ListAttribute 'IV 'menus)
(LeftButtonItems RightButtonItems)
```

(← *self* **ListAttribute!** *type name verboseFlg*) [Method of Object]

Purpose: Provides a recursive form of **ListAttribute**. Omits inheritance from the classes **Object** and **Tofu** unless *verboseFlg* is T.

Behavior: Converts *type* into uppercase on entry. The remaining behavior varies according to the arguments.

- If *type* is IVS, this is the same as **ListAttribute**.
- If *type* is one of IV, IVPROPS, or NIL, and *name* is the name of an instance variable of *self*, this returns a list of property names of *name*.
- If *type* is none of the above, this evaluates (← (**Class** *self*) **ListAttribute!** *type name*).

Note: Using a *type* of **SUPERS** or **SUPERCLASSES** returns a list of the names of the super classes.

Arguments: *self* A pointer to an instance.

type See Behavior.

name If type is one of IV, PROPS, or NIL, then *name* is an instance variable of *self*; else it is NIL.

verboseFlg T or NIL; if T, inheritance from object and **Tofu** are included. If NIL, they are omitted.

Returns: Value depends on the arguments; see Behavior.

Categories: Object

Specializations: Class

Example: Given that

```
95←(← ($ Window) New 'w1)
#,($& Window (NEW0.1Y%:.;h.eN6 . 515))
```

then

```
96←(← ($ w1) ListAttribute! 'IV 'menus)
(RightButtonItems doc TitleItems ...)
```

(← *self* **WhereIs** *name type propName*) [Method of Object]

Purpose: Searches supers hierarchy to find class where *name* is defined.

Behavior: Performs the method **Class.ListAttribute** for *self* and for each super class of *self*, checking to see if *name* (or *propName* as appropriate) is a member of the list returned. The value returned is the class where *name* (or *propName*) is first found.

The *type* argument is changed to uppercase and then coerced to a valued type argument for **ListAttribute**.

- If *type* is one of METHOD, METHODS, NIL, or T, it becomes METHODS. **WhereIs** then looks for a method with the name *name*.
- If *type* is one of IVPROP or IVPROPS, it becomes IVPROPS. **WhereIs** then looks for an instance variable property with the name *name*.
- If *type* is one of IV or IVS, it becomes IVS. **WhereIs** then looks for an instance variable with the name *name*.
- If *type* is one of CV or CVS, it becomes CVS. **WhereIs** then looks for a class variable with the name *name*.

Arguments:

self	A pointer to an instance.
type	See Behavior.
name	The name of an object attribute being searched for.
propName	Property name for instance variable *name*.

Returns: The class where *name* or *propName* is first found.

Categories: Object

Example: The command

```
97←(← (← ($ LatticeBrowser) New) WhereIs 'left 'IV)
```

returns

```
#,($C Window)
```


2.10 OTHER INSTANCE ITEMS

2.10 Other Instance Items

This section describes other items involved with instances.

NoValueFound [Variable]

Purpose/Behavior: Returned as a result of various accesses; initially set to NIL. When developing code, rebind this to the symbol **NoValueFound** to assist in debugging.

(**NoValueFound** *arg*) [Macro]

Purpose/Behavior: Returns value of (EQ **NoValueFound** *arg*).

Arguments:

arg	Any value.

Returns: T or NIL.

(**ValueFound** *arg*) [Macro]

Purpose/Behavior: Returns value of (NEQ **NoValueFound** *arg*).

Arguments: *arg* Any value.

Returns: T or NIL.

[This page intentionally left blank]

LIST OF FIGURES

[This page intentionally left blank]

3. CLASSES

Classes provide a description of instances within the object domain. The following information is contained within a class:

- The metaclass for this class. See Chapter 4, Metaclasses, for a discussion of metaclasses.
- Class Properties. xamples of class properties are an edit stamp and documentation.
- The supers list for this class. Classes exist in a hierarchy and the supers list places the class within that hierarchy. Instances of the class contain data and respond to messages that are described within the class and superclasses of the class.
- Class variables, their values, and their properties and vlues.
- Instance variables, their default values, and their properties and values.

This chapter covers creating and destroying classes, editing, accessing data stored in classes, inheritance, and related topics. Other chapters that contain information relevant to this chapter are Chapter 4, Metaclasses, since a metaclass is a class of classes, and Chapter 10, Browsers, since the primary user interface for manipulating classes is the browser.

3.1 Creating Classes

Several ways are available to create a class:

- Use the browser interface.
- Use function calling or message sending.
- Use dynamic mixins to dynamically create classes.

The rules for naming classes are the same as those for naming instances. Simply stated, a class name must be a litatom. One exception to this rule is the naming of dynamic mixin classes, which is discussed later in this chapter.

A class is generally referred to with this form: ($ className). See Chapter 2, Instances, for more details regarding LOOPS names.

As discussed in Chapter 2, Instances, the protocol that is followed when instances are created is for the LOOPS system to send the **NewInstance** message to the newly created instance. The **NewInstance** message can be specialized to incorporate behavior specific to the creation time of an instance. Similarly, the system follows a prototol when creating a class using the **New** message. After the class is created, it is sent the **NewClass** message.

3.1.1 Function Calling and Message Sending

The following table shows the items in this section.

Name	Type	Description
DefineClass	Function	Creates a new class.
New	Method	Creates a new class.
CreateClass	Method	Creates a new class.
NewClass	Method	Provides a placeholder for modifying the class creation protocol.

(**DefineClass** *name supers self*) [Function]

Purpose: Creates a new class.

Behavior: If *name* is not a litatom, a break occurs.

- If *supers* is non-NIL, it should be a list of classes or names of classes to be the supers for the newly created class. If the list contains multiple classes, this results in a class that has multiple super classes (see Section 3.3, "Inheritance"). The order of classes in the list specifies the order in which lookup will proceed. If one of the these classes is not a valid class, a break occurs.
- If *supers* is NIL and if *self* is ($ MetaClass), then the supers list is (Class).
- If both *supers* is NIL and *self* is NIL, the supers list is (Object).

If *self* is non-NIL, it is installed as the metaclass for the newly created class. See Chapter 4, Metaclasses.

A class is then built with an **Edited:** property containing the date and time and the value of variable **INITIALS**. (See the *Interlisp-D Reference Manual*.)

The newly created class has no class variables, instance variables, or methods.

The variable **LASTWORD** is set to *name*, which is added to **USERWORDS** for spelling escape completion. (See the *Interlisp-D Reference Manual* for information on **LASTWORD** and **USERWORDS**.)

Arguments:

name	A LOOPS name to be given to the class.
supers	A list of classes.
self	A metaclass.

Returns: The class object.

Examples: The following command defines a subclass of the class **Object**.

```
(DefineClass 'ExampleClass)
```

The following command defines a subclass of the class **Window**.

```
(DefineClass 'MyClass '(Window))
```

The following command defines a class with multiple supers: **ExampleClass** and **Window**.

```
(DefineClass 'AnotherClass '(ExampleClass Window))
```

The following command defines a subclass of the class **Window** that has **AbstractClass** as its metaclass.

```
(DefineClass 'DontMakeMe '(Window) ($ AbstractClass))
```

(← *class* **New** *name supers init1 init2 init3*) [Method of Metaclass]

Purpose: Creates a new class.

Behavior: Sends the message **CreateClass** to *class*, passing the arguments *name* and *supers*. This returns a new class which is then sent the message **NewClass** passing the arguments *init1*, *init2*, and *init3*.

Arguments: *class* A pointer to a class.

name A LOOPS name to be given to the class.

supers A list of classes.

init1, init2, inti3
See Behavior.

Returns: The new class.

Categories: Object

Specializes: Class

Specializations: AbstractClass

Example: The following command creates the class, **AClass**, which is a subclass of the class **Window**. The metaclass of **AClass** is **Class**.

```
(← ($ Class) New 'AClass '(Window))
```

After **AClass** is created, the system sends the `following message:`

```
(← ($ AClass) NewClass)
```

(← *self* **CreateClass** *name supers*) [Method of Metaclass]

Purpose: Creates a new class.

Behavior: Method version of **DefineClass**.

Arguments: *self* A metaclass.

name The name of the newly created class.

supers A list of classes.

Returns: The clsss object.

Categories: MetaClass

(← *class* **NewClass** *init1 init2 init3*) [Method of Class]

Purpose: Provides a hook into class initialization. If you want special actions to occur when creating a class, specialize this method.

Arguments: *class* A pointer to a class.

init1, init2, init3
Dependent on user-defined functionality.

Returns: *class*

Categories: Class

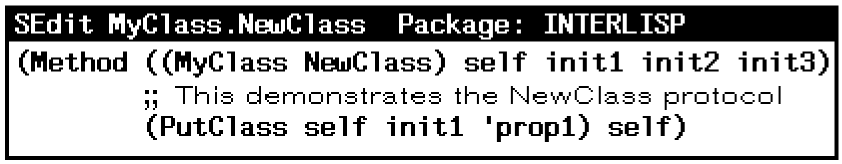
Example: Create a subclass of **Class** called **MyClass**:

```
(DefineClass ’MyClass ’(Class))
```

Give it a method **NewClass:**

```
(DefineMethod ($ MyClass) ’NewClass ’(init1 init2 init3)
’(PROGN (PutClass self init1 ’prop1) self))
```

This looks like the following display editor window:

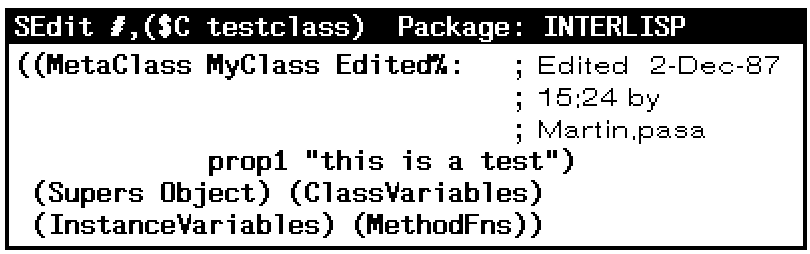


```
SEdit MyClass.NewClass  Package: INTERLISP
(Method ((MyClass NewClass) self init1 init2 init3)
        ;; This demonstrates the NewClass protocol
        (PutClass self init1 'prop1) self)
```

Now send the class **MyClass** the following command:

```
(← ($ MyClass) New ’testclass NIL "this is a test")
```

This results in the creation of the class shown in the following display editor window:



```
SEdit #,($C testclass)  Package: INTERLISP
((MetaClass MyClass Edited%:   ; Edited 2-Dec-87
                               ; 15:24 by
                               ; Martin.pasa
            prop1 "this is a test")
 (Supers Object) (ClassVariables)
 (InstanceVariables) (MethodFns))
```

To display the class, enter

```
(← ($ testclass) Edit)
```

3.1.2 Dynamic Mixins

In some programming situations, you may develop sets of mixins that are designed to be used together. (Mixins are classes that are used only in conjunction with another class to create a subclass, or provide some functionality useful in more than one class.) For example, the class **NamedClass** adds one instance variable **name** and specializes the **New** message to ensure that the instance variable **name** contains the name of the object.

```
(DefineClass ’NamedClass)
(← ($ NamedClass) AddIV ’name)
(DefineMethod ($ NamedClass) ’New ’(self name)
’(←@ (←self NewInstance name) name name))
```

Other classes that want the names of their objects in an instance variable **name** can use **NamedClass** as a mixin.

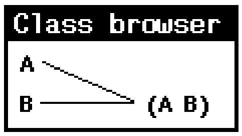
As another example, assume that you have one set consisting of **A1**, **A2**, **A3**, and **A4** and another set containing **B1**, **B2**, and **B3**. Formerly, to allow creation of an instance taking properties from arbitrary combinations of an element from each set, you had to create in advance all 12 combinations of classes with a super from **A** and a super from **B**. This was even more cumbersome if the **A**s and **B**s can also combine with any of a set of 5 **C**s.

What is desired is the ability to create combinations of these classes on the fly, without having to invent a name for each combination and without having each present in the system when only a few may be needed in any given application. To meet this need, LOOPS now provides the dynamic mixin class. The name of such a class is a list, in order, of the classes which are to be the supers of the class. Such a class is automatically created the first time it is referred to. Thus, the following sequence

```
(DefineClass 'A)
(DefineClass 'B)
(← ($ (A B)) New)
```

creates the class whose supers are **A** and **B** (if it did not already exist), and builds an instance of that class.

Dynamic mixins appear in browsers as shown in this sample window.



All of the browser operations still function on dynamic mixin classes.

These classes print as

```
#,($C (A B))
```


3.2 Destroying Classes

The following messages have been provided to destroy a class that has been created. Destroyed classes, if not being pointed to in some fashion, are eventually collected by the garbage collector.

The following table shows the methods in this section.

Name	Type	Description
Destroy	Method	Removes a class from the LOOPS system.
Destroy!	Method	Destroys a class and its subclasses.
DestroyClass	Method	Destroys a class by deleting its contents.

(← *class* **Destroy**) [Method of Class]

Purpose: Removes a class from the LOOPS system.

Behavior: If *self* has any subclasses, a break occurs and you are prompted to determine if you want to use **Destroy!**.

Sends the message **DestroyClass** to the metaclass of *self*.

Specializations of this method may be necessary to undo any actions that might have been performed by user specializations of the **NewClass** method. If you specialize **Destroy**, be sure to include a ←**Super** to guarantee that the functionality of the **Destroy** method is performed.

Arguments: *class* Must be a class.

Returns: NIL

Categories: Object

Specializes: Object

Specializations: DestroyedClass

Example: The following command destroys the class **Datum**:

```
(← ($ Datum) Destroy)
```

(← *class* **Destroy!**) [Method of Class]

Purpose: Destroys a class and its subclasses.

Behavior: Recursively sends the **Destroy** message to *self* and its subclasses.

Arguments: *class* Must be a class.

Returns: NIL

Categories: Object

Specializes: Object

Specializations: DestroyedClass

(← *class* **DestroyClass** *classToDestroy*) [Method of Class]

Purpose: Destroys *classToDestroy* by deleting its contents. This method is invoked by the LOOPS system and should generally not be called directly by user code. However, it can be specialized to change the way classes are destroyed.

Behavior: Performs the following actions:

- Removes *classToDestroy* from any files on **FILELST**.
- Sends the **Destroy!** message to all methods locally associated with *classToDestroy*.
- Removes *classToDestroy* from any subclass data contained in the supers of *classToDestroy*.
- Changes the class name of *classToDestroy* to ***aDestroyedClass*** .
- Changes the supers list of *classToDestroy* to **DestroyedObject** and **Object** .
- Changes the metaclass of *classToDestroy* to **DestroyedClass**.
- Sets other fields of the internal class data structure to NIL.

Arguments: *class* Metaclass of *classToDestroy*.

classToDestroy
Class to destroy.

Returns: NIL

Categories: Class

Specializations: DestroyedClass

3.3 Inheritance

Classes exist in an ordered lattice or hierarchy. Information contained within a class - the supers list - defines where that class is located within the lattice. The supers list specifies the classes immediately above a given class. When an instance of a class is created, it contains not only the instance variables of the defining class, but also the instance variables of all of the classes above the defining class in the class hierarchy. When you try to determine the value of a class variable associated with an instance, all classes above the defining class may be searched. When you send a message to an instance, all classes above the defining class may be searched for the appropriate method.

There are two types of inheritance:

- Simple, in which a class has only one superclass.
- Multiple, in which a class has two or more classes on its supers list.

When an instance is created, it may contain an instance variable that is defined in more than one class. The default value for that instance variable depends on its inheritance. In the case of simple inheritance, the instance variable gets the value from the class that is lowest in the hierarchy. In multiple inheritance, the instance variable gets the value from the class that is lowest in an inheritance list. To create this list,

1. Put the first class that describes the instance.
2. Begin with the first class on its supers list and move up from it, making a list of classes which assume simple inheritance.
3. Build one of these lists for all successive super classes.
4. Append these lists together.
5. Remove all occurrences of any classes that appear in the list a multiplenumber of times except for the last entry.

Another way to think about this, which creates the same inheritance list, is the following:

1. Begin with the first super class and walk up the hierarchy until you reach a class where the inheritance paths merge.
2. Walk up each path leading from each successive super class to where paths merge.
3. Take the class where the paths merge and walk up from there.

As an example of simple inheritance, examine Figure 3-1 which shows some of the class variables and instance variables defined within each class.

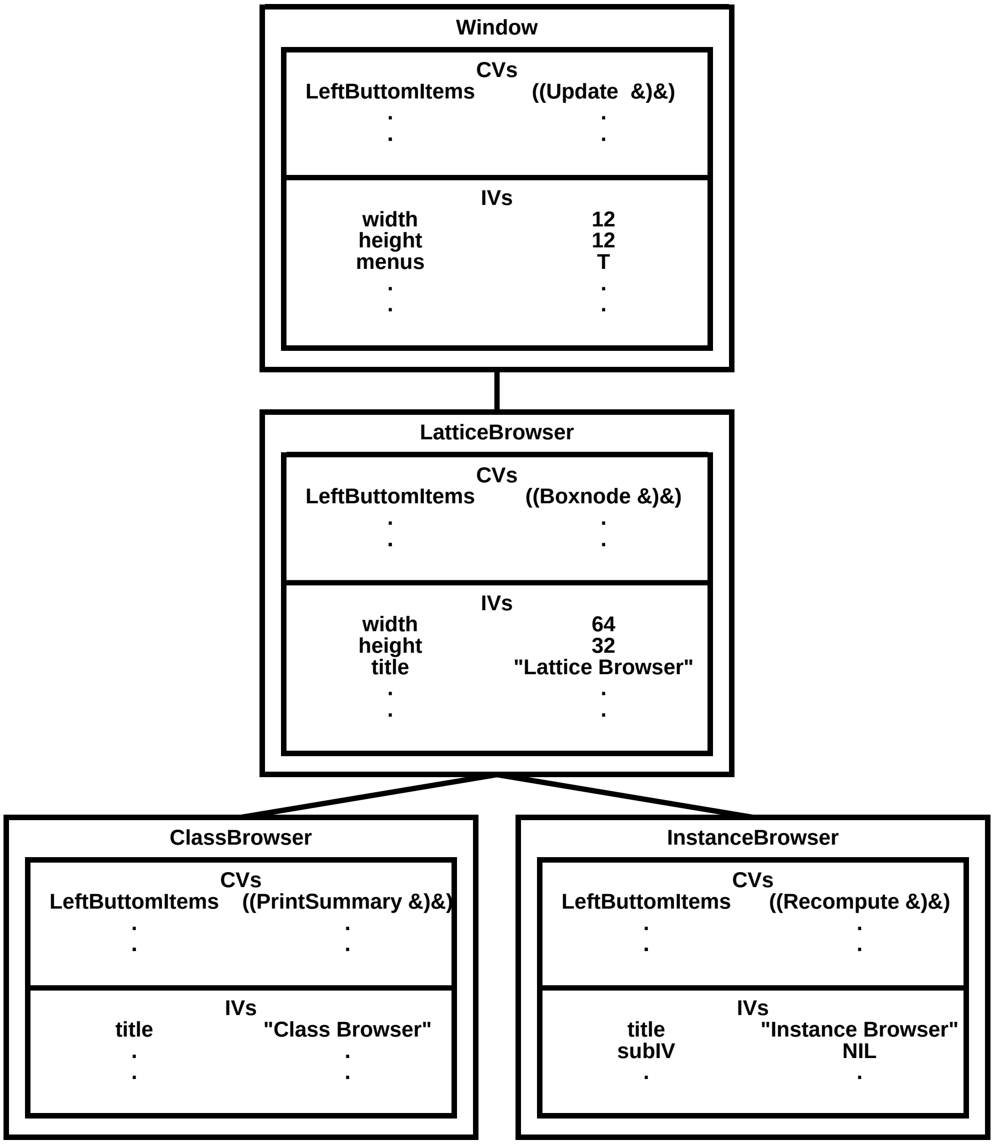


Figure 3-1. Simple Inheritance Lattice

An instance of the class **ClassBrowser** has this as an inheritance list:

ClassBrowser
LatticeBrowser
Window
Object
Tofu

The instance variable values of this instance are as follows:

IV	Value	From Class
title	"Class browser"	**ClassBrowser**
width	64	**LatticeBrowser**
height	32	**LatticeBrowser**
menus	T	**Window**

Accessing the value of the class variable **LeftButtonItems** causes this value to come from the class **ClassBrowser**.

Figure 3-2 shows an example of multiple inheritance.

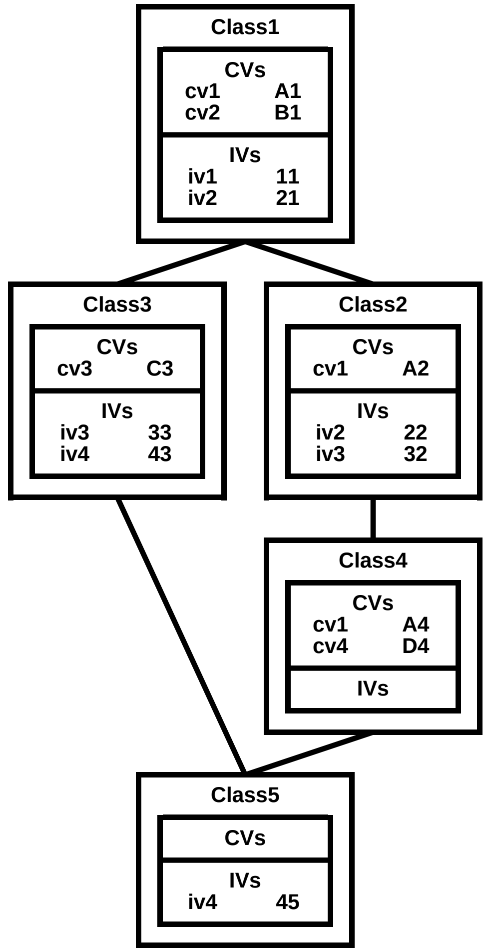


Figure 3-2. Multiple Inheritance Lattice

If the order of the supers for **Class5** is **Class3** and then **Class4** (that is, its supers list is (Class3 Class4)), then the inheritance list for an instance of **Class5** is as follows:

Class5
Class3
Class4
Class2
Class1

The instance variable and class variable values this instance are as follows:

IV	Value	From Class	CV	Value	From Class
iv1	11	**Class1**	**cv1**	A4	**Class4**
iv2	22	**Class2**	**cv2**	B	**Class1**
iv3	33	**Class3**	**cv3**	C	**Class3**
iv4	45	**Class5**	**cv4**	D4	**Class4**

3.4 EDITING CLASSES

3.4 Editing Classes

Changing the contents of a class typically involves using the display editor, although programmatical ways to make these changes are available. To edit a class structure, the LOOPS system first changes the structure to a list and then passes that list to the display editor. Upon exit from the display editor, the system translates the modified list back into the class structure.

The editor is most often called from the browser interface. (See Chapter 10, Browsers.) The following method provides a programmatical way to invoke the editor.

(← *class* **Edit** *commands*) [Method of Class]

Purpose: Edits a class definition.

Behavior: Calls **EDITE** (see the *Interlisp-D Reference Manual*) with the translated class structure passed as the **EXPR** argument and *commands* passed as the **COMS** argument.

This method binds the variable **LASTCLASS** to the class name of *self*.

Arguments: *class* Pointer to a class.

commands Commands passed to **EDITE**.

Returns: Name of the class.

Categories: Object

Specializes: Object

Example: The following command causes a display editor window to appear.

```
(← ($ LoopsIcon) Edit)
```

Calling the editor causes a structure to appear in a display editor window. At this time, you can change the structure of the class by using any of the following techniques:

- Changing the value of the class's metaclass. This is done by changing th class name after the word **MetaClass**.
- Changing the superclasses for the class. The form for this is :

  ```
  (Supers class1 class2 ...)
  ```

  At least one class must be in the supers list. The order of this list determines the order of inheritance; the first class after the word **Supers** on this list is the first class to search for inherited data and methods.
- Adding or removing class properties. Class properties occur within the same list as MetaClass, after the metaclass class name. The form for this is

  ```
  (MetaClass metaclassName classProp1 propVal1 classProp2
  propVal2 ...)
  ```
- Adding or removing class variables or associated properties. The form for class variables is:

  ```
  (ClassVariables
  (cvName1 cvVal1 prop1a propVal1a prop1b propVal1b ...)
  (cvName2 cvVal2 prop2a propVal2a prop2b propVal2b ...)
  ...)
  ```

It is not necessary to have any properties for a class variable. If the length of each class variable list is not an even number, a break occurs under the editor. The message in the break window describes an odd length list the first time you try to exit from the editor.

- Adding or removing instance variables or associated properties. These have the same form as class variables with the distinction that the value listed for each instance variable is not its value, but only its default value for the purposes of instanciation.

For example, examine the display editor window in Figure 3-3.

```
SEdit #,($C IndirectVariable)  Package: INTERLISP
((MetaClass Class doc
            (* Active Value for redirecting references to another
               variable)
            Edited%: (* smL " 9-May-86 09:52"))
 (Supers ActiveValue) (ClassVariables)
 (InstanceVariables
       (object NIL doc (* The object with the "real" variable))
       (varName NIL doc (* The name of the "real" variable))
       (propName NIL doc (* The prop name of the "real" variable))
       (type NIL doc (* The type of the "real" variable)))
 (MethodFns IndirectVariable.GetWrappedValueOnly
            IndirectVariable.PutWrappedValueOnly
            IndirectVariable.WrappingPrecedence))
```

Figure 3-3. Sample Display Editor Window

This figure shows the following information:

- The title bar of the display editor window indicates the class being edited.
- The metaclass of the class **IndirectVariable** in this example is the class **Class**. **IndirectVariable** has two class properties. The first is a **doc** property. The second is an **Edited:** property.
- This class has one super class: **ActiveValue**.
- This class has no class variables. It has four instance variables: object, varName, propName, and type. Each has a **doc** property.
- The **MethodFns** are listed in this structure as a convenience. It is not possible to add or delete elements of this list from the editor and have any changes actually occur. Selecting one of the method function names and then selecting Edit (Meta-O in SEdit) allows you to edit that method either as its method code (METHOD-FNS), its method object (METHODS), or its Interlisp code (FNS).

3.5 MODIFYING CLASSES

3.5 Modifying Classes

In addition to the editing technique for changing a class, you can use programmatic means to modify the structure of a class. This section describes the functions and methods for modifying classes.

Name	Type	Description
Add	Method	Adds a component to a class.
Delete	Method	Deletes a component from a class.

DeleteClassProp	Function	Removes a class property from a class.
AddCV	Function	Adds a class variable to a class.
AddCV	Method	Adds a class variable to a class.
DeleteCV	Function	Deletes a class variable or one of its properties from a class.
AddCIV	Function	Adds an instance variable to a class; can also add properties to a class.
AddIV	Method	Adds an instance variable to a class.
DeleteCIV	Function	Removes an instance variable or property from a class.
ReplaceSupers	Method	Changes the super classes of a class.

(← *class* **Add** *type name value prop*) [Method of Class]

Purpose/Behavior: Adds a component to a class.

Arguments:

- *class* — Pointer to a class.
- *type* — One of IV, IVPROP, CV, CVPROP, METACLASS, or METHOD.
- *name* — The name of the item to be added.
- *value* — The value, or default value if *type* is one of IV or IVPROP.
- *prop* — The name of the property, if a property is to be added.

Returns: NIL

Categories: Class

Example: The following command adds a new instance variable **color** to class **Datum**:

```
(← ($ Datum) Add 'IV 'color)
```

(← *class* **Delete** *type name prop*) [Method of Class)

Purpose: Deletes a component from a class.

Behavior: Varies according to the arguments.

- If *type* is one of IV, IVPROP, or NIL, this calls (**DeleteCIV** *class name prop*).
- If *type* is one of CV or CVPROP, this calls (**DeleteCV** *class name prop*).
- If *type* is META, METACLASS, or CLASS, and if *prop* is NIL, then the metaclass of *self* is changed to the class **Class**.
- If *type* is META, METACLASS, or CLASS, and if *prop* is non-NIL, then this calls (**DeleteClassProp** *class prop*).
- If *type* is METHOD or SELECTOR, this calls (**DeleteMethod** *class name prop*).

Arguments:

- *class* — A pointer to a class.
- *type* — See Behavior.
- *name* — IV, CV, or selector name.

prop A property name.

Returns: NIL

Categories: Class

Example: The following command deletes the instance variable **color** from the class **Datum**:

```
(← ($ Datum) Delete 'IV 'color)
```

(**DeleteClassProp** *classRec propName*) [Function]

Purpose: Removes a class property from a class.

Behavior: Marks *classRec* as changed.

Arguments: *classRec* Pointer to a class.

propName Property to be deleted.

Returns: NIL is *propName* is not found; otherwise *propName*.

(**AddCV** *class varName newValue*) [Function]

Purpose: Adds a class variable to a class.

Behavior: Varies according to the arguments.

- If *varName* is NIL, you are prompted to enter a name.
- If *varName* is already a class variable, its value is changed to *newValue*. NIL is returned.
- If *varName* is not a class variable of *class*, it is added to *class* with the value *newValue*. Also, a **doc** property is added with the following value:

```
'(* CV added by , (USERNAME NIL T))
```

varName is returned in this case.

Arguments: *class* A pointer to a class.

varName Name of the new variable.

newValue The new value.

Returns: Value depends on the arguments; see Behavior.

(← *class* **AddCV** *varName newValue*) [Method of Class]

Purpose: Adds a class variable to a class.

Behavior: Provides a method version of the function **AddCV**.

Arguments: See the function **AddCV**.

Returns: NIL

Categories: Class

(**DeleteCV** *class varName prop*) [Function]

Purpose: Deletes a class variable or one of its properties from a class.

Behavior: Marks *class* as changed.

Arguments:
class Pointer to a class.
varName Class variable name to be deleted.
prop Property to be deleted.

Returns: NIL, if *varName* is not found, else *varName*.

(**AddCIV** *class varName defaultValue otherProps*) [Function]

Purpose: Adds an instance variable, and perhaps properties, to a class.

Behavior: If the length of *otherProps* is odd, an error occurs.

The remaining behavior varies according to the arguments.

- If *varName* is NIL, you are prompted to enter a name.
- If *varName* is already an instance variable of *class*, then change its default value to *defaultValue*. Properties on *otherProps* are added or changed as necessary. NIL is returned.
- If *varName* is not an instance variable of *class*, it is added to *class* and its default value is *defaultValue*. Properties on *otherProps* are also added. If there is no **doc** property, it is added and given the following value:

```
'(* IV added by , (USERNAME NIL T))
```

varName is returned in this case.

Arguments:
class Must be a pointer to a class.
varName New instance variable name.
defaultValue New default value.
otherProps NIL or a list in property list format.

Returns: Value depends on the arguments; see Behavior.

(← *class* **AddIV** *varName defaultValue otherProps*) [Method of Class]

Purpose: Adds an instance variable to a class.

Behavior: Provides a method version of the function **AddCIV**.

Arguments: See the function **AddCIV**.

Returns: NIL

Categories: Object

Specializes: Object

Example: Define a new class **TestClass**, add an instance variable **testIV** with two properties **testProp1** and **testProp2**, all with initial values, and then prettyprint the class's variables.

```
64←(DefineClass 'TestClass)
#,($C TestClass)
```

```
65←(← ($ TestClass) AddIV 'testIV 1234
'(testProp1 1 testProp2 2))
testIV

66←(← ($ TestClass) PPV! T)
#,($ TestClass)
MetaClass and its Properties
  Class  Edited: (* edited: 24-Sep-87 08:41 by mcgill)
Supers
  (Object Tofu)
Instance Variable Descriptions
  testIV 1234 doc (* IV added by MCGILL)
testProp2 2 testProp1 1
Class Variables
```

(**DeleteCIV** *class varName prop*) [Function]

Purpose: Removes an instance variable or property from a class.

Behavior: If *class* does not have *varName*, a break occurs.

Marks *class* as changed.

Arguments:

class	Pointer to a class.
varName	Instance variable to be deleted.
prop	If non-NIL, property to be deleted.

Returns: Value depends on the arguments.

- NIL for removing an instance variable if successful.
- *prop* for removing a property if successful.
- NIL if *prop* is not a property.

(← *class* **ReplaceSupers** *supers*) [Method of Class]

Purpose: Changes the super classes of a class.

Behavior: Checks that no circular lists can be made in the inheritance lattice.

- If the super class of *class* is Tofu, no change occurs.
- If supers is different from the current supers, the supers list of *class* is changed and *class* is marked as changed.

Arguments:

class	Pointer to a class.
supers	A list of class names or classes.

Returns: NIL

Categories: Class

3.6 METHODS FOR MANIPULATING CLASS NAMES

3.6 METHODS FOR MANIPULATING CLASS NAMES

3.6 Methods for Manipulating Class Names

LOOPS classes must have one and only one LOOPS name. The following functions and methods allow you to change and rename class names.

Name	Type	Description
Rename	Method	Changes the name of a class. Prompts for name if not provided, then calls **SetName**.
SetName	Method	Changes the name of a class.
UnSetName	Method	Unnames a class.
ClassName	Function	Finds the class name of an object.

(← *class* **Rename** *newName*) [Method of Class]

Purpose: Changes the name of a class. Prompts for name if not provided, then calls **SetName**.

Behavior: Varies according to the argument.

- If *newName* is NIL, this causes a break and prompts you for a name. **Rename** then sends the message **SetName** passing this name as an argument
- If *newName* is non-NIL, **Rename** sends the message **SetName** passing *newName* as an argument.

Arguments: *class* Pointer to a class.

newName A litatom.

Returns: NIL

Categories: Object

Specializes: Object

Example: The following command renames class **Datum** to **Thing**:

```
(← ($ Datum) Rename 'Thing)
```

(← *class* **SetName** *newClassName*) [Method of Class]

Purpose: Changes the name of a class.

Behavior: Removes the old name of *self* from **ObjNameTable** .

SetName uses the Interlisp-D function **EDITCALLERS** to rename references to the class name or any file that contains the class. If **EDITCALLERS** cannot succeed, for example, when a file is not RANDACCESSP, a message is printed that the class cannot be renamed on that file. For complete information on **EDITCALLERS**, see the *Interlisp-D Reference Manual*.

The names of the method functions of *class* are changed to use *newClassName*.

Arguments: *class* Pointer to a class.

newClassName A litatom.

Returns: NIL

Categories: Object

Specializes: Object

(← *class* **UnSetName**) [Method of Class]

Purpose: Unnames a class, but does not destroy it. Has limited usefulness for keeping a class name from being typed in.

Behavior: Removes *class* from the LOOPS name hash table and from any files on **FILELST**. This method is intended to be used internally only; it is not recommended to create an unnamed class.

Arguments: *class* Pointer to a class.

Returns: NIL

Categories: Object

Specializes: Object

(**ClassName** *self*) [Function]

Purpose: Finds the class name of an object.

Behavior: Varies according to the arguments.

- If *self* is a class, this returns the name of that class.
- If *self* is an instance, this returns the name of the class that describes that instance.
- If *self* is neither a class or an instance, this returns **Tofu**.

Arguments: *self* See Behavior.

Returns: Value depends on the arguments; see Behavior.

Example: Given that

```
(← ($ Window) New 'w1)
```

the commands

```
(ClassName ($ w1))
(ClassName ($ Window))
```

both return **Window**.

3.7 QUERYING THE STRUCTURE OF A CLASS

3.7 Querying the Structure of a Class

The following functions and methods allow you to query what is contained in a class.

Name	Type	Description

GetClassProp	Method	Obtains a class's metaclass or properties.
HasAttribute	Method	Determines whether *self* has an attribute name.
HasAttribute!	Method	Recursive form of **HasAttribute**, but works only on classes.
HasCV	Method	Determines if a class has a class variable with a specified property.
HasItem	Method	Determines if a class has an item of a given type.
HasIV	Method	Determines if a class has an instance variable with a specified property.
HasIV!	Method	Same as **HasIV**, except that **HasIV!** also searches up the supers chain.
ListAttribute	Method	Lists the elements of a class that are local to the class.
ListAttribute!	Method	Lists all the items associated with a class.
WhoHas	Function	Determines what classes contains a specified item.

(← *class* **GetClassProp** *propname*) [Method of Class]

Purpose: Obtains a class's metaclass or properties by following metaclass links.

Behavior: Varies according to the arguments.

- If *propname* is NIL, this returns the *class*'s metaclass.
- If *propname* is non-NIL, this looks first in *class* for that property. If it cannot find it there, it looks through *class*'s metaclass links.
- If no property is found, the value of the variable **NotSetValue** is returned.

Arguments: *class* A pointer to a class.

prop Property name.

Returns: Value depends on the arguments; see Behavior.

Categories: Class

Example: The following commands show the variety of responses.

```
51←(← ($ Window) GetClassProp)
#,($C Class)

52←(← ($ Window) GetClassProp 'doc)
"A LOOPS object which represents a window"

53←(← ($ IconWindow) GetClassProp 'doc)
"An icon window that appears as an irregular shaped image
on the screen -- See the ICONW Library utility"
```

(← *self* **HasAttribute** *type name propname*) [Method of Class]

Purpose: Determines whether *self* has an attribute name, with a property *propname* if supplied.

Behavior: *self* can be an instance or a class. Remaining behavior depends on *type*, which is converted to uppercase on entry:

- If *type* is IV, IVPROP, or NIL, this returns T if *self* has an instance variable of *name*, with a property called *propname* (if *propname* is non-NIL), otherwise it returns NIL.
- If *type* is CV or CVPROP, this returns T if *self* has a CV called *name*, with a property of *propname* (if *propname* is non-NIL), otherwise it returns NIL.
- If *type* is METHOD or SELECTOR, this returns NIL or the name of the method implementing *name*.

HasAttribute applied to an instance reports on the actual state of the instance; it sees all instance variables and class variables whether local, inherited, or specially added to the instance. If only local attributes are required, use (← (Class instance) HasAttribute ...).

Arguments:

self	Can be an instance or a class.
type	See Behavior.
name	A symbol which is looked up as the variable or method name.
propname	A symbol which is looked up as the property name.

Returns: See Behavior.

Categories: Object

Specializations: Class

Example: The command

```
(← ($ LoopsIcon) HasAttribute 'IV 'icon)
```

returns T.

(← *class* **HasAttribute!** *type name propname*) [Method of Class]

Purpose: Recursive form of **HasAttribute**; only works on classes

Behavior: Similar to **HasAttribute**, but will also search through *class* 's supers.

Arguments:

class	A class.
type	See Behavior under **HasAttribute**.
name	A symbol which is looked up as the variable or method name.
propname	A symbol which is looked up as the property name.

Returns: See Behavior.

Categories: Object

Specializations: Class

Example: The command

```
(← ($ LoopsIcon) HasAttribute 'IV 'left)
```

returns NIL, but

```
(← ($ LoopsIcon) HasAttribute! 'IV 'left)
```

returns T.

(← *class* **HasCV** *cvName prop*) [Method of Class]

Purpose: Determines if a class has a class variable *cvName* with a property *prop*.

Note: The preferred form of this method is **HasAttribute** or **HasAttribute!**.

Behavior: Varies according to the arguments.

- If *prop* is NIL, this returns T if *class* contains a class variable called *cvName*, else NIL.
- If *prop* is non-NIL, this returns T if *class* contains a class variable called *cvName* with the property *prop*, else NIL.

Note: **HasCV** does not distinguish between locally defined class variables and inherited class variables. If you need to test a class to see if it has a class variable defined locally, you can use the **HasAttribute** method. For example, the form (←MyClass HasAttribute 'CV 'ABC) will return a non-NIL value if and only if the class **MyClass** has a local definition of the class variable ABC.

Arguments:

class A pointer to a class.

cvName A class variable name.

prop Property name.

Returns: NIL or T; see Behavior.

Categories: Object

Specializes: Object

Example: The command

```
(← ($ Window) HasCV 'TitleItems)
```

returns T.

(← *class* **HasItem** *itemName prop itemType*) [Method of Class]

Purpose: Determines if a class has an item of a given type.

Note: The preferred form of this method is **HasAttribute** or **HasAttribute!**.

Behavior: Varies according to the arguments.

- If *itemType* is IV or IVS, this sends the message (← *class* **HasIV** *itemName prop*).
- If *itemType* is CV or CVS, this sends the message (← *class* **HasCV** *itemName prop*).
- If *itemType* is SELECTOR, METHOD, SELECTORS, or METHODS, this finds the corresponding local method of *class*.
- If *itemType* is not one of the above, this returns NIL.

Arguments:

class Pointer to a class.

prop Property name.

itemType See Behavior.

Returns: Value depends on the arguments; see Behavior.

Categories: Class

(← *class* **HasIV** *IVName prop*) [Method of Class]

Purpose: Determines if a class has an instance variable *IVName* with a property *prop*.

Note: The preferred form of this method is **HasAttribute** or **HasAttribute!**.

Behavior: *class* should point to a class.

- If *prop* is NIL, this returns T if *IVName* is contained in *class*.
- If *prop* is non-NIL, this returns T if *IVName* is contained in *class*, and *prop* is a property of *IVName* in *class* or one of its supers.

Arguments:

class	Pointer to a class.
IVName	Instance variable name.
prop	Property name.

Returns: Value depends on the arguments; see Behavior.

Categories: Object

Specializes: Object

(← *class* **HasIV!** *IVName prop*) [Method of Class]

Purpose/Behavior: Same as **HasIV**, except that **HasIV!** also searches up the supers chain.

Note: The preferred form of this method is **HasAttribute** or **HasAttribute!**.

Arguments: See the method **HasIV**.

Returns: Value depends on the arguments; see Behavior.

Categories: Class

(← *class* **ListAttribute** *type name*) [Method of Class]

Purpose: Lists the elements of a class that are local to the class.

Behavior: *type* is converted to uppercase on entry. The remaining behavior varies according to the arguments.

- If *type* is IVS, this returns the instance variable names (not values) local to *class*. *name* is ignored.
- If *type* is IV, IVPROPS, or NIL, *name* should be bound to an instance variable of *class*. This returns the property names (not values) of the instance variable *name*. If *name* is not an instance variable of *class*, this returns NIL.
- If *type* is CVS, this returns the class variables local to *class*. *name* is ignored.
- If *type* is CV or CVPROPS, *name* should be bound to a class variable of *class*. This returns the property names of the class variable *name*. If *name* is not a class variable of *class*, this returns NIL.

- If *type* is METHODS or SELECTORS, this returns the selectors for the class. *name* is ignored.

Arguments: *class* Pointer to a class.

type See Behavior.

name See Behavior.

Returns: Value depends on the arguments; see Behavior.

Categories: Object

Specializes: Object

Example: The following commands show the variety of responses.

```
55←(← ($ SupersBrowser) ListAttribute ’IVs)
(title)

56←(← ($ Window) ListAttribute ’iv ’menus)
(DontSave Title LeftButtonItems MiddleButtonItems TitleItems doc)

57←(← ($ IconWindow) ListAttribute ’METHODS)
(GetMenuItems)
```

(← *class* **ListAttribute!** *type name verboseFlg*) [Method of Class]

Purpose: Lists all items associated with a class.

Behavior: Provides a recursive version of **ListAttribute**.

If *verboseFlg* is NIL, items that are inherited from Tofu, Object, or Class are omitted, unless *class* is one of Tofu, Object, or Class.

type is converted to uppercase on entry.

- If *type* is META or METACLASS, this returns the same as **ListAttribute.**
- If *type* is IVS or NIL, this returns the instance variables an instance of *class* would have.
- If *type* is SUPERS or SUPERCLASSES, this returns the ordered list of super classes of *class*.
- If *type* is SUBS or SUBCLASSES, this returns all of the subclasses of *class*.
- If *type* is any other option that can be passed to **ListAttribute**, this returns all local and inherited values.

Arguments: *class* Pointer to a class.

type See Behavior.

name A litatom.

verboseFlg See Behavior.

Returns: Value depends on the arguments; see Behavior.

Categories: Object

Specializes: Object

(**WhoHas** *name type files editFlg*) [Function]

Purpose: Determines what classes contain a specified item.

Behavior: Returns a list of classes on *files* that contain *name*. If *editFlg* is non-NIL, then edit the methods (if *type* is METHOD), or the classes before returning.

Arguments:

name	The item specified.
type	One of IV, CV, METHOD, or Method. If *type* is NIL, it defaults to METHOD.
files	A file or a list of files. If *files* is NIL, it defaults to **FILELST**.
editFlg	T or NIL.

Returns: A list of classes on *files* that contain *name*.

3.8 COPYING CLASSES AND THEIR CONTENTS

3.8 Copying Classes and Their Contents

Inheritance lets classes be described in terms of other classes in a hierarchical manner. When it is preferable to duplicate a class description in different parts of a lattice these methods provide the capability.

The following table shows the methods in this section.

Name	Type	Description
Copy	Method	Copies a class.
CopyCV	Method	Copies a class variable to another class.
CopyIV	Method	Copies an instance variable to another class.

(← *class* **Copy** *name*) [Method of Class]

Purpose: Makes a copy of a class.

Behavior: If *name* is NIL, you are prompted to supply a name for the new class. This copies variables and properties and methods.

Arguments:

class	The class being copied.
name	The name of the copy.

Returns: The new class.

Categories: Class

Example: Given that

```
(DefineClass 'Datum)
(← ($ Datum) AddIV 'someThing)
```

the following command makes a copy of class **Datum** and names it **Thing**:

```
(← ($ Datum) Copy 'Thing)
```

(← *class* **CopyCV** *cvName toClass*) [Method of Class]

Purpose/Behavior: Copies a class variable to another class. This also copies the properties of *cvName* to *toClass*.

Arguments:

	class	The source class.
	cvName	The name of the class variable to copy.
	toClass	The destination class.

Returns: NIL

Categories: Class

(← *class* **CopyIV** *ivName toClass*) [Method of Class]

Purpose/Behavior: Copies an instance variable to another class. This also copies the properties of *ivName* to *toClass*.

Arguments:

	class	The source class.
	ivName	The name of the instance variable to copy.
	toClass	The destination class.

Returns: NIL

Categories: Class

3.9 ENUMERATING INSTANCES OF CLASSES

3.9 Enumerating Instances of Classes

New instances may be created without names, or without being tracked. These methods allow you to produce a list of instances according to their classes. **Prototype** instances are a convenience used where the methods defined for a class must be used, but there is no logical instance for the class.

The following table shows the items in this section.

Name	Type	Description
AllInstances	Method	Finds all instances of a class.
AllInstances!	Method	Finds all instances of a class or its subclasses.
IndexedObject	Class	Keeps track of instances so that **AllInstances** searches can proceed more rapidly.
PrintOn	Method	Modifies how instances of **IndexedObject** that do not have LOOPS names will be printed.
Prototype	Method	Returns an instance of a class that is stored on the class's class variable **Prototype**.

(← *class* **AllInstances**) [Method of Class]

Purpose: Finds all instances of a class.

Behavior: Checks if *class* is a subclass of **IndexedObject**. If so, a faster search is used to find all of the instances of *class*. If not, this checks if each object is an instance of *class*. Instances that do not yet have a UID will not be found.

Arguments: *class* A class.

Returns: A list of the instances found.

Categories: Class

Example: The following command produces a list of all the LOOPS window instances:

```
61←(← ($ Window) AllInstances)
```

(← *class* **AllInstances!**) [Method of Class]

Purpose: Finds all instances of a class or its subclasses.

Behavior: Returns a list of instances that are instances of *class* or any of its subclasses. Instances that do not have the class **IndexedObject** as a super class, or that do not yet have a UID are not found. (See Chapter 18, Reading and Printing, for more information on UIDs.)

Arguments: *class* A pointer to a class.

Returns: A list of the instances found.

Categories: Class

IndexedObject [Class]

Purpose: Keeps track of instances so that **AllInstances** searches can proceed more rapidly.

Behavior: This class is to be used as a **Mixin** (an addition superclass), and should be the first class on a supers list for a class.

IndexedObject provides **NewInstance** and **Destroy** protocols that cause instances to be added to or removed from a global list when they are created or destroyed. This global list allows the **AllInstances** protocols to search more quickly.

IndexedObject also provides a **PrintOn** protocol that modifies how instances will be printed if they have no LOOPS name.

MetaClass: Class

Supers: Object

Class Variables: **IdentifierVar**
The name of an instance variable which will contain a string which could provide some identification to the user. Used in **PrintOn** if variable is in object and filled. **shortName**, the value of this class variable, is the default variable name which is used.

(← *self* **PrintOn**) [Method of IndexedObject]

Purpose: Modifies how instances of **IndexedObject** that do not have LOOPS names will be printed.

Behavior: If *self* has a LOOPS name, or if *self* does not have an instance variable with a name equal to (@ *self* ::IdentifierVar), then do a (←**Super**). Otherwise, build a form that incorporates the value of the instance variableIV referenced by (@ *self* ::IdentifierVar).

Arguments: *self* An instance.

Returns: A list ; see example

Categories: Object

Specializes: Object

Example: Create a class, **IndexedObjectTest**, that has this structure.

```
62←(DefineClass 'IndexedObjectTest '(IndexedObject))
#,($C IndexedObjectTest)

63←(← ($ IndexedObjectTest) AddIV 'shortName 'ioTest)
shortName
```

Create an instance.

```
64←(SETQ test (← ($ IndexedObjectTest) New))
#,($& IndexedObjectTest (YMW0.0X%:.>T4.n18 . 36))

65←(←@ test shortName 'changeName)
changeName

66←(← test PrintOn)
("#," $& IndexedObjectTest (changeName (YMW0.0X%:.>T4.n18 . 36)))
```

(← *class* **Prototype** *newProtoFlg*) [Method of Class]

Purpose: Returns a prototype instance of a class.

Behavior: Varies according to the arguments.

- If *class* has a class variable **Prototype** and the variable's value is an instance of *class*, return the value (assuming *newProtoFlg* is NIL).
- If there is no class variable **Prototype**, or if there is a class variable **Prototype** but its value is not an instance of *class*, or if *newProtoFlg* is non-NIL, then create a new instance of *class*, store the instance on the class variable **Prototype**, and return the instance.

See **Proto** in Chapter 7, Message Sending Forms, for more information.

Arguments: *class* A class.

newProtoFlg
If non-NIL, create a new prototype instance.

Returns: The prototype.

Categories: Class

Example: LOOPS defines an icon to make it easy to bring up class browsers and file browsers. The icon is the **Prototype** instance of the class **LoopsIcon**.

To move the icon to the center of the bottom of the screen, enter

```
71←(←Proto ($ LoopsIcon) Move (QUOTIENT SCREENWIDTH 2) 0)
(576 . 0)
```

This places the left edge of the icon at the center of the screen. To move the icon to the center of the screen, enter

```
72←(LET ((icon (← ($ LoopsIcon) Prototype)))
       (← icon Move (QUOTIENT (DIFFERENCE SCREENWIDTH
                                          (@ icon width))
                              2)
                    0))
(544 . 0)
```


3.10 DEALING WITH INHERITANCE

3.10 Dealing with Inheritance

The inheritance lattice for classes shows how methods and variables are shared (see Chapter 10, Browsers, for details on how to graph the lattice on the screen). To programmatically inspect and add to this lattice via **Specialize**, use the following functions and methods:

Name	Type	Description
Fringe	Method	Finds the leaves of a branch of an inheritance tree.
Specialize	Method	Creates a subclass of a class.
SubClasses	Method	Returns a list of subclasses.
Subclass	Method	Determines if a class is a subclass of another class.
AllSubClasses	Function	Computes the subclasses of a class.
SubsTree	Function	Computes all the names of the subclasses of a class.

(← *class* **Fringe**) [Method of Class]

Purpose: Finds the leaves of a branch of an inheritance tree.

Behavior: Returns a list of subclasses of *class*, whether close or distant, that have no subclasses.

Arguments: *class* A class, the root of the tree to explore.

Returns: Names of subclasses of *class* that have no subclasses.

Categories: Class

Example: The following commands show the variety of responses.

```
73←(← ($ Window) Fringe)
(InstanceBrowser MetaBrowser SupersBrowser FileBrowser
LoopsIcon IconWindow)

74←(← ($ ClassBrowser) Fringe)
(MetaBrowser SupersBrowser FileBrowser)
```

(← *class* **Specialize** *newName*) [Method of Class]

Purpose: Creates a subclass of a class.

Behavior: Creates a class with class as its only super.

- If *newName* is non-NIL, this is the name of the new class.

- If *newName* is NIL, this creates a name consisting of the name of *class* followed by an integer.

Arguments: *class* Pointer to a class.

newName Name of the new subclass.

Returns: The new class.

Categories: Class

Example: Given that

```
(DefineClass 'Datum)
```

the following command creates a specialization of the class **Datum** called **DatumX**:

```
(← ($ Datum) Specialize 'DatumX)
```

(← *class* **SubClasses**) [Method of Class]

Purpose: Returns a list of subclasses.

Behavior: The classes returned by this are the immediate subclasses of *class*.

Arguments: *class* A pointer to a class.

Returns: A list of subclasses.

Categories: Class

Specializations: DestroyedClass

Example: The following command gets a list of the subclasses of the class **Window**:

```
(← ($ Window) SubClasses)
```

(← *class* **Subclass** *super*) [Method of Class]

Purpose: Determines if a class is a subclass of another class.

Behavior: If *class* is a subclass of *super*, *super* is returned, else NIL.

Arguments: *class* Pointer to a class.

super Either the LOOPS name of a class or a pointer to a class.

Returns: Value depends on the arguments; see Behavior.

Categories: Class

Example: The command

```
(← ($ DestroyedClass) Subclass 'Class)
```

returns

```
#,($C Class)
```

(**AllSubClasses** *class currentSubs*) [Function]

Purpose: Computes the subclasses of a class.

Behavior: This is a recursive function that computes (without duplicates) all of the subclasses of *class*.

Arguments: *class* Must be a pointer to a class, for example, ($ Window).

currentSubs Used by LOOPS; NIL when called by the user.

Returns: A list of classes.

Example: The command

```
(AllSubClasses ($ LatticeBrowser))
```

returns

```
(#,($C FileBrowser) #,($C SupersBrowser)
#,($C MetaBrowser) #,($C ClassBrowser)
#,($C InstanceBrowser))
```

(**SubsTree** *class currentList*) [Function]

Purpose: Computes the names of the subclasses of a class.

Behavior: Provides a recursive function that computes (without duplicates) all of the names of the subclasses of *class*.

Arguments: *class* Can be a class name or a pointer to a class

currentList Used internally by **SubsTree**; it should be NIL when called by the user.

Returns: A list of class names.

Example: The command

```
(SubsTree 'LatticeBrowser)
```

returns

```
(InstanceBrowser ClassBrowser MetaBrowser SupersBrowser
FileBrowser)
```

[This page intentionally left blank]

PREFACE

Overview of the Manual

The *LOOPS Reference Manual* provides a detailed description of all the methods, functions, classes, and other items available in the Lisp Object-Oriented Programming System, LOOPS. This manual describes the Medley Release of LOOPS, which runs under Medley.

This manual is for people who are familiar with LOOPS programming principles, and is not intended to teach you LOOPS or how to use it. Please contact your LOOPS distributor for information about classes and training material.

Organization of the Manual and How to Use It

This manual is divided into chapters, with most chapters focusing on a particular aspect of LOOPS. The organization of this manual is similar to the *Interlisp-D Reference Manual*.

A Table of Contents is included at the beginning of the manual to help you find specific material. At the end of the manual, a Glossary is included to define terms within the context of LOOPS.

All readers should review Chapter 1, Introduction, before referring to specific material.

Conventions

This manual uses the following conventions:

- Case is significant in LOOPS and Lisp. All selectors, methods, arguments, etc., must be typed as shown. Typically, this means that method names are capitalized and variables are not.
- Arguments appear in italic type. Optional arguments are indicated by a dash (-).
- Selectors, methods, functions, objects, classes, and instances appear in bold type.

  For example, a message sending form appears as follows:

  (← *self* **Selector** *Arg1 Arg2* -)
- Examples appear in the following typeface:

```
89←(←LOGIN)
```

- All examples are typed into an Interlisp Exec. This is the recommended Exec for all LOOPS expressions.

- Methods with an exclamation mark (!) suffix usually perform operations deeply into class structure instead of only on a given object.
- Methods with a question mark (?) suffix usually are predicates; that is, truth functions.
- Method names often appear in the form **ClassName.SelectorName**.
- Cautions describe possible dangers to hardware or software.
- Notes describe related text.

This manual describes the LOOPS items (functions, methods, etc.) by using the following template:

Purpose: Gives a short statement of what the item does.

Behavior: Provides the details of how the item operates.

Arguments: Describes each argument in the following format:

argument Description

Returns: States what the item returns, and does not appear if the item does not return a value. The phrase "Used as a side effect only." means that the purpose of the item is to perform a computation or action that is independent of any returned value, not to return a particular value.

Categories: A way to group related methods. For example, all the methods releated to Masterscope on the class **FileBrowser** have the category Masterscope, not **FileBrowser**. This item appears only for methods.

Specializes: The next higher class in the class hierarchy that contains a method with the same selector; only appear for methods. For example, **the manual entry for RectangularWindow.Open** would say that it specializes **Window.Open**, since **Window** is the first superclass of **RectangularWindow** that implements a method for **Open**.

Specializations: The next lower class(es) in the class hierarchy that contains method(s) with the same selector; only appears for methods. For example, the manual entry for **Window.Open** would say that it has a specialization of **RectangularWindow.Open** since **RectangularWindow** is a subclass of **Window** and has its own version of **Open** method.

Example: An example is often included to show how to use the item and what result it produces. Some examples may appear differently on your system, depending on the settings of various print flags. See Chapter 18, Reading and Printing, for details.

References

The following books and manuals augment this manual.

LOOPS Library Modules Manual

LOOPS Users' Modules Manual

Interlisp-D Reference Manual

Common Lisp: the Language by Guy Steele

Common Lisp Implementation Notes, Medley Release

Lisp Release Notes, Medley Release

*Lisp Library Modules Manual, Medley Releas*e

[This page intentionally left blank]

4. METACLASSES

In object-oriented programming, every object is described by a class. Instances are described by classes and classes are, in turn, described by metaclasses. The methods that an instance inherits are defined in the class definition of that instance and the methods that the class inherits are defined in the metaclass definition of that class's metaclass. Sending a message to an instance invokes method in class. Similarly, sending a message to class invokes method in metaclass.

The two classes **Class** and **MetaClass** are metaclasses of other classes. If **Class** or **MetaClass** refers to the metaclass, it appears in a bold typeface.

One method defined by a class's metaclass is **New**, which returns a new instance of a class. Different classes can initialize their instances in different ways. For example, one class may need to have certain values assigned to instance variables at creation, while another does not. The different forms of **New** are defined in separate metaclasses.

A class's metaclass is assigned when the class is created. A new class is created by sending a metaclass the message **New** or by specializing an already existing class. In the latter case, the metaclass defaults to the metaclass of the class's super class. The class's metaclass can be changed by directly editing the class definition.

This chapter discusses the metaclasses provided with LOOPS, describes pseudoclasses, explains how to define new metaclasses, and discusses the root class **Tofu** .

4.1 Specific Metaclasses

This section describes the metaclasses provided by LOOPS: **Class**, **MetaClass, AbstractClass**, and **DestroyedClass**. These metaclasses are shown in Figure 4-1.

Figure 4-1. Class Browser Showing Metaclasses

4.1.1 Metaclass Class

Class is the default metaclass for LOOPS classes. When a class whose metaclass is **Class** receives the message **New**, it creates a new instance of itself and returns that instance. If this message is sent at the top level, the definition of the created instance is printed in the Executive window.

4.1.2 Metaclass MetaClass

MetaClass is the metaclass for all metaclasses and provides the message **New** to all metaclasses. For metaclasses, the result of sending the message **New** is the definition of a new metaclass. This is discussed in detail in Section 4.3, "Defining New Metaclasses."

4.1.3 Metaclass AbstractClass

If a class's metaclass is **AbstractClass**, then it cannot be instantiated. If an abstract class is sent the message **New**, the following message is printed to the TTY window.

```
#,($C className) Abstract Class cannot be instantiated
```

To make a class an **AbstractClass**, either send the metaclass **AbstractClass** the message **New** or change the metaclass of the class definition directly using the editor.

Use an abstract class to define a class which should not have any instances. For example, consider mixin classes. Mixins are always used in conjunction with another class to create a subclass. Instances are created from the new subclass that has the mixin as one of its parents. Because mixins never have instances, they have **AbstractClass** as their metaclass.

As an example, consider a circuit simulation module that contains various classes such as **Resistors, Inductors, Batteries,** and **Wires**. A possibility is to define a super class for these classes called **AnalogDevice** to contain all the information common to all such classes: current, impedance, voltage drop, etc. This super class also holds all the methods common to the classes, such as **ApplyOhmsLaw**. Since **AnalogDevice** is not itself intended to be instantiated (only its subclasses are), its metaclass can be **AbstractClass** so that an error occurs if it is accidentally instantiated.

Note: Whenever **AnalogDevice** is specialized to create a new subclass, be sure to change its metaclass.

4.1.4 Metaclass DestroyedClass

DestroyedClass is the metaclass for classes that have been sent the message **Destroy** or **Destroy!** Trying to instantiate a **DestroyedClass** causes an error. Attempts to destroy a **DestroyedClass** have no effect.

4.2 PSEUDOCLASSES

4.2 Pseudoclasses

Pseudoclasses provide an object interface to Lisp data types, which are also known as Lisp objects. Pseudoclasses associate a class with the type name of a Lisp object. When messages are sent to Lisp objects of the named type, the messages are actually sent to the pseudoclass. Lisp objects which have pseudoclasses are considered pseudoinstances.

Pseudoclasses provide two special cases in the message-sending mechanism: for lists whose first element is a class, or for ordinary Lisp data types.

In the first case, the list's first element is used as the class to look up the method to be used.

In the second case, the class of the data type is found by using the **GetLispClass** function, which looks in an internal table based on the type name of the data type. If none is found, it is assumed to be **Tofu**. If found, the data type is considered a pseudoclass and instances of it pseudoinstances.

Pseudoclasses also provide special cases in the behavior of **GetValue** and **PutValue**, to allow simulation of variable or property access, as described below.

(**GetValue** *pseudoinstance varName propName*) [Function]

Purpose: A variation on the behavior of GetValue to simulate retrieving variable or property values on pseudoinstances.

Behavior: If **GetValue** is called with *self* bound to a pseudoinstance, then the method associated with the selector **GetValue** in the pseudoclass is called with the arguments:

pseudoinstance varName propName

Arguments: *pseudoinstance*
A Lisp object which has a pseudoclass.

varName The simulated variable name.

propName The simulated property name, or NIL.

Returns: The result of the call to the **GetValue** method in the pseudoclass.

(**PutValue** *pseudoinstance varName propName newValue*) [Function]

Purpose: A variation on the behavior of **PutValue** to simulate setting of variable or property values on pseudoinstances.

Behavior: If **PutValue** is called with *self* bound to a pseudoinstance, then the method associated with the selector **PutValue** in the pseudoclass is called with the arguments:

instance varName newValue propName

Arguments: *pseudoinstance*
A Lisp object which has a pseudoclass.

varName The simulated variable name.

propName The simulated property name, or NIL.

newValue The new value to be placed in the simulated slot.

Returns: The result of the call to the **PutValue** method in the pseudoclass.

(**GetLispClass** *obj*) [Function]

Purpose: Used by the system to compute a class corresponding to a Lisp data type.

Behavior: Gets the hash value for the key (**TYPENAME** *obj*) from an internal hash array.

- If this hash value is NIL, ($ Tofu) is returned.
- If the hash value is not NIL and it is a class, it is returned.

- In all other cases, the hash value, which should be a function, is applied to *obj* and the result is returned.

Arguments: *obj* A Lisp object.

Returns: Value depends on the hash value; see Behavior.

Example: The command

```
79←(GetLispClass (create annotatedValue))
```

returns

```
#,($C AnnotatedValue)
```

LispClassTable [Global Variable]

Purpose: Used by **GetLispClass** to map type names of Lisp objects to pseudoclasses.

Format: This hash table has EQ hashing. It contains pairs of symbol keys (a type name) and either classes, NIL, or a function object to be applied (see **GetLispClass**).

4.2.1 Example

This example creates a pseudoclass from the Lisp data type STRINGP.

1. Define a class **String** that receives its messages:

```
37←(DefineClass 'String)
#,($C String)
```

2. Place an entry in the LispClass hash table to link the Lisp data type STRING to the **String** class.

```
38←(PUTHASH 'STRINGP ($ String) LispClassTable)
#,($C String)
```

3. Add methods to **String** which will operate on Lisp STRINGPs, for example:

```
39←(DefineMethod ($ String) 'UpCase '(self)
    '(U-CASE self))
String.Upcase
```

This allows messages like the following:

```
40←(← "abc" UpCase)
"ABC"
```

4. Specialize **GetValue** and **PutValue** to allow element access in strings, for example:

```
41←(DefineMethod ($ String) 'GetValue '(index)
    '(NTHCHAR self index))
String.GetValue

42←(DefineMethod ($ String) 'PutValue '(index value)
    '(RPLSTRING self index value))
String.PutValue
```

This allows messages to access characters in strings, for example:

```
43←(← "abc" GetValue 2)
b

44←(← "abc" PutValue 2 'p)
"apc"
```


4.3 DEFINING NEW METACLASSES

4.3 Defining New Metaclasses

A new metaclass must be defined if you want to create several classes for which a class message, such as **Destroy**, needs to be specialized. To create a new metaclass, an object of the class **MetaClass** must be instantiated. This is done by sending **MetaClass** the message **New**.

(← (**$ MetaClass**) **New** *metaClassName supers*) [Method of Metaclass]

Purpose: Instantiates a new metaclass with **MetaClass** as its metaclass, *metaClassName* as its name, and *supers* as a list of its super classes.

Behavior: Evaluates *metaClassName*, which must evaluate to a symbol. The default for *supers* is **(Class)**. If used, *supers* must evaluate to a list of classes. The message returns the new metaclass.

Arguments: *metaClassName*
Name of the new metaclass; must evaluate to a symbol.

supers List of classes.

Categories: **MetaClass**

Specializes: Class.New

Specializations: AbstractClass.New

Example: Assume the following **MetaClass** definition:

```
42←(←($ MetaClass) New 'ListMetaClass '(Class))
#,($C ListMetaClass)
```

The message **New** can then be defined for the metaclass, **ListMetaClass.** In this example, it saves all the instances created of a class with the metaclass **ListMetaClass**. The instances are stored as the value of the class property **AllInstances.** Define the message **New** using **DefineMethod** as follows:

```
DefineMethod ($ ListMetaClass) 'New '(name arg1 arg2 arg3
arg4 arg5)
    '((* * Create an instance and add it to
           list in class)
     (LET ((newObj (←Super)))

            (* * newObj is the instance returned by
                sending the New message provided by
                the class CLASS.)

          (PutClass
            self
            (CONS newObj (LISTP (GetClassHere
```

```
                              self 'AllInstances)))
              'AllInstances)

              (* * LISTP returns list or
                  NIL if not a list)
              (* * GetClassHere returns the value
                  of a property of a class.)

            newObj]
```

Now a new class can be defined by sending #,($C ListMetaClass) the message **New**. The result is a new class with **ListMetaClass** as the metaclass.

```
43←(←($ ListMetaClass) New 'Book)
#,($C Book)      [New class called Book]

44←(←($ Book) New 'Book1)
#,($C Book1)

45←(←($ Book) New 'Book2)
#,($C Book2)

46←(GetClass ($ Book) 'AllInstances)
#,($C Book2) #,($C Book1)  [List of all instances created so far]
```


4.4 TOFU

4.4 Tofu

The highest class in the LOOPS hierarchy is **Tofu**, which is an acronym for Top of the Universe. It is the simplest class, having no instance variables and only three defined messages:

- **MessageNotUnderstood**
- **MethodNotFound**
- **SuperMethodNotFound**

Figure 4-2 shows specializations of **Tofu**. The most familiar specialization of **Tofu** is the class **Object**, which is the root of the most of the other classes. Another specialization of **Tofu** is **AnnotatedValue**. **AnnotatedValue** is used with active values (see Chapter 8, Active Values).

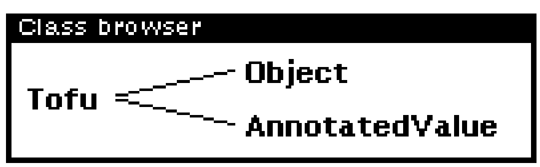


Figure 4-2. Specializations of Tofu

If another evaluation protocol or scheme for catching error conditions is needed, specialize **Tofu** and define all the methods required for handling data, usually some subset of the methods of **Object**. Specializing Tofu should only be necessary on very rare occasions.

The following table shows the methods in this section.

Functionality	Type	Description

MessageNotUnderstood	Method	Provides the error handling mechanism for when a message is sent to an object that cannot respond to that message.
MethodNotFound	Method	Provides some intermediate checking before sending the message MessageNotUnderstood.
SuperMethodNotFound	Method	Provides some intermediate checking before sending the message **MessageNotUnderstood**.

(← *self* **MessageNotUnderstood** *selector messageArguments superFlg*) [Method of Tofu]

Purpose/Behavior: Provides the error handling mechanism for when a message is sent to an object that cannot respond to that message.

Calls **ERROR** with a list which includes *self*, *selector*, and "not understood."

Arguments:
self An object receiving a message with the selector *selector*.

selector A selector.

messageArguments
A list of the arguments to the message.

superFlg Used internally.

Returns: See Behavior.

Categories: Tofu

Specializations: Object

(← *self* **MethodNotFound** *selector*) [Method of Tofu]

Purpose/Behavior: Provides some intermediate checking before sending the message **MessageNotUnderstood**.

Arguments:
self An object receiving a message with the selector *selector*.

selector A selector.

Returns: Used for side effect only.

Categories: Tofu

(← *self* **SuperMethodNotFound** *selector classOfSendingMethod*) [Method of Tofu]

Purpose/Behavior: Provides some intermediate checking before sending the message **MessageNotUnderstood**.

Arguments:
self An object receiving a message with the selector *selector*.

selector A selector.

classOfSendingMethod
The class with the method that contains a ←**Super.**

Returns: Used for side effect only.

Categories: Tofu

[This page intentionally left blank]

5. ACCESSING DATA

This chapter discusses the various ways to access data:

- Generalized Get and Put functions
- Accessing data in instances
- Accessing data in classes

5.1 Generalized Get and Put Functions

These functions support generalized instance variable and property access for LOOPS objects. They can be very useful for implementing methods that support new types of conditional accessing; they have been used to simplify code in the active values system, for example.

This section deals with the following functions:

Name	Type	Description
GetIt	Function	Retrieves values from instance variables and properties.
GetItOnly	Function	Like **GetIt** , but returns active values on a variable/property without triggering them.
GetItHere	Function	Like **GetIt**, but returns active values on a variable/property without triggering them; does not observe **NotSetValue** as **GetItOnly** does.
PutIt	Function	Stores values into instance variables and properties.
PutItOnly	Function	Like **PutIt**, but stores by smashing active values on a variable/property without triggering them.

(**GetIt** *self varOrSelector propName type*) [Function]

Purpose: Retrieves values from instance variables and properties.

Behavior: Varies according to the arguments.

- If *type* is 'IV or NIL
  - If *self* is an instance, this is equivalent to (**GetValue** *self varOrSelector propName*)
  - If *self* is a class, this is equivalent to (**GetClassIV** *self varOrSelector propName*)
- If *type* is 'CV,this is equivalent to (**GetClassValue** *self varOrSelector propName*)

- If *type* is ’CLASS,this is equivalent to (**GetClass** *self* (**OR** *varOrSelector propName*))
- If *type* is ’METHOD,this is equivalent to (**GetMethod** *self varOrSelector propName*)

Arguments: *self* A class or an instance.

varOrSelector An instance variable name or the name of a method.

propName Property name.

type Specifies the type of the object *self*.

Returns: Value depends on the arguments; see Behavior.

Example: The command

```
(GetIt ($ Window) ’doc NIL ’CLASS)
```

returns

```
"A Loops object that represents a window"
```

(**GetItOnly** *self varOrSelector propName type*) [Function]

Purpose: Retrieves values from instance variables and properties without triggering active values.

Behavior: Varies according to the arguments.

- If *type* is ’IV or NIL
  - If *self* is an instance, this is equivalent to (**GetValueOnly** *self varOrSelector propName*)
  - If *self* is a class, this is equivalent to (**GetClassIV** *self varOrSelector propName*)
- If *type* is ’CV,this is equivalent to (**GetClassValueOnly** *self varOrSelector propName*)
- If *type* is ’CLASS,this is equivalent to (**GetClassOnly** *self* (**OR** *varOrSelector propName*))
- If *type* is ’METHOD,this is equivalent to (**GetMethodOnly** *self varOrSelector propName*)

Arguments: *self* A class or an instance.

varOrSelector An instance variable name or the name of a method.

propName Property name.

type Specifies the type of the object *self*.

Returns: Value depends on the arguments; see Behavior.

Example: The command

```
(GetItOnly (GetClassValue ($ LoopsIcon) ’Prototype) ’window)
```

returns the LoopsWindowAV that holds the image of the LOOPS icon. Calling **GetIt** with similar arguments returns the Lisp window object held by that LoopsWindowAV.

(**GetItHere** *self varOrSelector propName type*) [Function]

Purpose: Retrieves values from instance variables and properties without triggering active values; does not observe **NotSetValue** like **GetItOnly**.

Behavior: Varies according to the arguments.

- If *type* is ’IV or NIL

- If *self* is an instance, this is equivalent to (**GetIVHere** *self varOrSelector propName*)

- If *self* is a class, this is equivalent to (**GetClassIVHere** *self varOrSelector propName*)

- If *type* is ’CV,this is equivalent to (**GetCVHere** *self varOrSelector propName*)

- If *type* is’CLASS,this is equivalent to (**GetClassHere** *self* (**OR** *varOrSelector propName*))

- If *type* is ’METHOD,this is equivalent to (**GetMethodHere** *self varOrSelector propName*)

Arguments:

self A class or an instance.

varOrSelector An instance variable name or the name of a method.

propName Property name.

type Specifies the type of the object *self*.

Returns: Value depends on the arguments; see Behavior.

Example: The command

```
(GetItHere (GetClassValue ($ LoopsIcon) ’Prototype) ’title)
```

returns the value of **NotSetValue**. Calling **GetIt** with similar arguments returns the default value for this instance variable, NIL.

(**PutIt** *self varOrSelector newValue propName type*) [Function]

Purpose: Stores values into instance variables and properties.

Behavior: Varies according to the arguments.

- If *type* is ’IV or NIL

- If *self* is an instance, this is equivalent to (**PutValue** *self varName newValue propName*)

- If *self* is a class, this is equivalent to (**PutClassIV** *self varName newValue propName*)

- If *type* is ’CV,this is equivalent to (**PutClassValue** *self varName newValue propName*)

- If *type* is ’CLASS,this is equivalent to (**PutClass** *self newValue* (**OR** *varName propName*))

Arguments:

self A class or an instance.

varName An instance variable name.

propName Property name.

type Specifies the type of the object *self*.

Returns: Value depends on the arguments; see Behavior.

Example: The command

```
(PutIt (GetClassValue ($ LoopsIcon) 'Prototype) 'title "foo")
```

sets the instance variable **title** of the LOOPS icon prototype to "foo". This can be verified by inspecting (GetClassValue ($ LoopsIcon) 'Prototype) and examining the title slot.

(**PutItOnly** *self varOrSelector newValue propName type*) [Function]

Purpose: Stores values into instance variables and properties and smashes any active values it finds in its way without triggering them.

Behavior: Varies according to the arguments.

- If *type* is 'IV or NIL

- If *self* is an instance, this is equivalent to (**PutValueOnly** *self varName newValue propName*)

- If *self* is a class, this is equivalent to (**PutClassIV** *self varName newValue propName*)

- If *type* is 'CV,this is equivalent to (**PutClassValueOnly** *self varName newValue propName*)

- If *type* is 'CLASS,this is equivalent to (**PutClassOnly** *self newValue* (**OR** *varName propName*))

Arguments: *self* A class or an instance.

varName An instance variable name.

propName Property name.

type Specifies the type of the object *self*.

Returns: Value depends on the arguments; see Behavior.

Example: If the inspector from the **PutIt** example is used to set a break on the the instance variable **title** of the LOOPS icon prototype, then doing

```
(PutItOnly (GetClassValue ($ LoopsIcon) 'Prototype) 'title "mumble")
```

will set the instance variable **title** to "mumble" while smashing the trace active value.

5.2 ACCESSING DATA IN INSTANCES

5.2 ACCESSING DATA IN INSTANCES

5.2 Accessing Data in Instances

Two kinds of variables are associated with an instance:

- Its local instance variables, also referred to as IVs.
- The class variables, also referred to as CVs, that it shares with all instances of the class.

The data contained within instances are the values of instance variables and associated properties as well as a pointer to the class that describes the instance. Details of the LOOPS implementation determine exactly when the values of instance variables are stored within an instance. In some cases, the system must look to the class to find the values of instance variables. In general, you do not need to be concerned with this distinction; however, the details of it are covered in Chapter 2, Instances.

The types of data that an instance may contain is not limited. The values for an instance variable or a class variable can be any Lisp or LOOPS data structure.

The active value is a special case of data. When you try to access a variable with an active value as its value, the active value may be returned, depending upon the type of access. Normally, however, data computed by the active value is returned, not the active value. The details of how this computation is performed is described in Chapter 8, Active Values.

Instance variable names and class variable names are symbols and are not necessarily unique to each class. Although it is possible to use the same symbol for both a class variable name and an instance variable name, it is advisable not to do this since some of the LOOPS functionality examines both the instance variables and class variables in the search for data. See the method **IVMissing** in the class **Object**.

This section deals with the following functions and methods. See the *LOOPS Library Modules Manual* for information on how these interact with Masterscope.

Name	Type	Description
GetValue	Function	Finds the value of an instance variable.
Get	Method	Finds the value of an instance variable.
PutValue	Function	Writes the value of an instance variable.
Put	Method	Writes the value of an instance variable.
GetValueOnly	Function	Finds the value of an instance variable without triggering active values.
PutValueOnly	Function	Writes the value of an instance variable without triggering active values.
GetClassValue	Function	Returns the value of a class variable.
PutClassValue	Function	Changes the value of a class variable. The change occurs within the class and therefore causes all instances to access the new value of the variable.
GetClassValueOnly	Function	Returns the value of a class variable; does not trigger active values.
PutClassValueOnly	Function	Changes the value of a class variable. The change occurs within the class and therefore causes all instances to access the new value of the variable. Does not trigger active values.

GetIVHere	Function	Gets the value stored in an instance variable without invoking active values.

(**GetValue** *self varName propName*) [Function]

Purpose: Finds the value of an instance variable when *varName* and *propName* are to be computed.

Behavior: Varies according to the arguments.

- If *self* is an instance and *propName* is NIL, this returns the value of the instance variable *varName*. If there is no instance variable of the name *varName* and there is a class variable of that name, this returns the value of the class variable. See the **IVMissing** message for a complete discussion of this behavior. If there is neither an instance variable or class variable of that name, a break occurs.
- If *self* is an instance and *propName* is non-NIL, this returns the value of the property *propName* of the instance variable or class variable *varName*. If there is no property of the name, *propName*, this returns the value of the variable **NoValueFound**.
- If the value of *varName* (or *propName* if it is non-NIL) is an active value, the active value is activated.
- If *self* is not an instance, this calls `(`**GetIt** *self varName propName* `'IV)`

See the *LOOPS Library Modules Manual* about interaction with Masterscope.

Arguments:

self A class or an instance.

varName Instance or class variable name.

propName Property name.

Returns: Value depends on the arguments; see Behavior.

Example: Given that

```
32←(← ($ window1) Shape '(100 200 300 400))
(100 200 300 400)
```

then

```
33←(GetValue ($ window1) 'width)
300

34←(GetValue ($ window1) 'LeftButtonItems)
((Update ...))
```

(← *self* **Get** *varName propName*) [Method of Object]

Purpose/Behavior: Method version of **GetValue**.

Arguments: See **GetValue**.

Categories: Object

(**PutValue** *self varName newValue propName*) [Function]

Purpose: Writes the value of an instance variable when *varName* and *propName* are to be computed.

Behavior: Varies according to the arguments.

- If *self* is an instance and *propName* is NIL, this changes the value of the instance variable *varName to newValue*. This returns *newValue*. If *varName* is not an instance variable of *self*, this causes a break.
- If *self* is an instance and *propName* is non-NIL, this changes the value of the property *propName* of the instance variable *varName* to *newValue*. If *propName* is not already a property of *varName*, it is added. This returns *newValue*.
- If the value of *varName* (or *propName* if it is non-NIL) is an active value, the active value is activated.
- If *self* is a class, this calls (**PutIt** *self varName newValue propName* `'IV`)

See the *LOOPS Library Modules Manual* about interaction with Masterscope.

Arguments:

self	A class or an instance.
varName	Instance name or class name.
newValue	The new value for *varName* or *propName*.
propName	Property name.

Returns: Value depends on the arguments; see Behavior.

Example: `(PutValue ($ window1) 'width 120)`

(← *self* **Put** *varName newValue propName*) [Method of Object]

Purpose/Behavior: Method version of the function **PutValue**.

Arguments: See **PutValue**.

Categories: Object

Specializations: Class

(**GetValueOnly** *self varName propName*) [Function]

Purpose: Similar to **GetValue**, except that it overrides the active value mechanism.

Behavior: See **GetValue**. If the value found is an active value, it is returned without triggering its side effects.

Arguments: See **GetValue**.

Returns: See Behavior.

Example: The following expressions compare **GetValue** and **GetValueOnly**

```
35←(GetValue ($ window1) 'window)
{WINDOW}#nn,mmmm

36←(GetValueOnly ($ window1) 'window)
#,($AV LispWindowAV ...)
```

(**PutValueOnly** *self varName newValue propName*) [Function]

Purpose: Similar to **PutValue**, except that it overrides the active value mechanism.

Behavior: See **PutValue**. The argument *newValue* overwrites any active value on the slot without triggering it.

Arguments: See **PutValue**.

Returns: Used for side effect only.

(**GetClassValue** *self varName propName*) [Function]

Purpose: Returns the value of a class variable.

Behavior: Varies according to the arguments.

- If *propName* is NIL, this returns the value of the class variable *varName*. If *varName* is not a class variable, a break occurs.
- If *propName* is non-NIL, this returns the value of the property, *prop*, of the class variable *varName*. If *varName* has no property of that name, the value of the variable **NoValueFound** is returned.

See the *LOOPS Library Modules Manual* about interaction with Masterscope.

Arguments:

self	An instance or a class.
varName	Class variable name of *self*.
propName	Property name for class variable *varName*; may be NIL.

Returns: Value depends on the arguments; see Behavior.

Example: The following commands show a variety of retuned values.

```
37←(GetClassValue ($ window1) 'window)
```

This breaks, since window is not a class variable of **Window**.

```
38←(GetClassValue ($ window1) 'LeftButtonItems)
 ((Update ...))
```

```
39←(GetClassValue ($ window1) 'LeftButtonItems 'qwerty)
NIL
```

(**PutClassValue** *self varName newValue propName*) [Function]

Purpose: Changes the value of a class variable. The change occurs within a class and therefore causes a class variable lookup by other instances to find the new value.

Behavior: Varies according to the arguments.

- If *propName* is NIL, this changes the value of the class variable *varName* to *newValue*. If *varName* is not a class variable, this breaks.
- If *prop*Name is non-NIL, this changes the value of the property, *prop*Name, of the class variable *varName* to *newValue*. If *varName* has no property of that name, the property is added.

See the *LOOPS Library Modules Manual* about interaction with Masterscope.

Arguments:

self	An instance or a class.
varName	Class variable name of *self*.
newValue	Value to be assigned to class variable or property name.

propName Property name for class variable *varName*; may be NIL.

Returns: *newValue*

Example: The following command breaks since **left** is not a class variable name of **Window**.

```
40←(PutClassValue ($ window1) 'left 1234)
```

The command

```
41←(PutClassValue ($ window1) 'TitleItems 1234)
```

changes the value of **TitleItems**. The command

```
42←(PutClassValue ($ window1) 'TitleItems 123 'asdf)
```

adds the property **asdf** with the value 123 to **TitleItems**.

(**GetClassValueOnly** *self varName propName*) [Function]

Purpose: Gets the value of a class variable without triggering active values.

Behavior: Varies according to the arguments.

- If *propName* is NIL, this returns the value of the class variable *varName* without triggering active values. If *varName* is not a class variable, this breaks.
- If *propName* is non-NIL, this returns the value of the property, *propName*, of the class variable *varName* without triggering active values. If *varName* has no property of that name, the value of the variable **NotSetValue** is returned.

See the *LOOPS Library Modules Manual* about interaction with Masterscope.

Arguments:

self An instance or a class.

varName Class variable name for *self*.

propName Property name of class variable *varName*; may be NIL.

Returns: Value depends on the arguments; see Behavior.

Example: The following command returns the value of the variable **NotSetValue** since **LeftButtonItems** has no property of the name **qwerty**.

```
43←(GetClassValueOnly ($ window1) 'LeftButtonItems 'qwerty)
#,NotSetValue
```

(**PutClassValueOnly** *self varName newValue propName*) [Function]

Purpose: Changes the value of a class variable without triggering active values. The change occurs within a class and therefore causes a class variable lookup by other instances to find the new value.

Behavior: The behavior is the same as **PutClassValue** except that the value stored does not trigger an active value, but overwrites it instead.

Arguments:

self An instance or a class.

varName Class variable name of *self*.

newValue Value to be assigned to class variable or property name.

propName Property name for class variable *varName*; may be NIL.

Returns: *newValue*

(**GetIVHere** *self varName propName*) [Function]

Purpose: Gets the value stored in an instance without invoking active values.

Behavior: Returns the value of *varName* (or the property, *propName*, if it is non-NIL) without triggering active values. If the value of *varName* (or *propName*) is not yet stored in *self*, the value of the variable **NotSetValue** is returned.

See the *LOOPS Library Modules Manual* about interaction with Masterscope.

Arguments: *self* Must be an instance.

varName Instance variable of *self.*

propName Property name for variable *varName*; may be NIL.

Returns: Value depends on the arguments; see Behavior.

Example: Given that

```
44←(← ($ Window) New ’w2)
#,($& Window (NEW0.1Y%:.;h.eN6 . 496))
```

then

```
45←(GetIVHere ($ w2) ’left)
#,NotSetValue
```

After entering the command

```
46←(PutValue ($ w2) ’left 123)
123
```

then

```
47←(GetIVHere ($ w2) ’left)
123
```

5.2.1 Compact Accessing Forms

When you write methods for classes that you have defined, there are a number of accesses to the data contained in the object bound to the method argument *self*. The following forms have been created to allow a more concise notation for these accesses.

Name	Type	Description
@	Macro	Provides compact **GetValue** and **GetClassValue** forms.
@*	Macro	Provides compact **GetValue** forms.
←@	Macro	Provides compact **PutValue** and **PutClassValue** forms and assigns a new value.

(@ *accessPath*) [Macro]

Purpose: Provides compact **GetValue** or **GetClassValue** forms.

Behavior: The *accessPath* can be one, two, or three arguments.

- If the *accessPath* is one argument, *self* is assumed to be the object and the argument points to an instance variable. This is the most common usage in methods in which you need to get the value of an instance variable contained in *self*. For example,

  ```
  (@ iv1)
  ```

  translates to

  ```
  (GetValue self 'iv1).
  ```

- If the *accessPath* is two arguments, the first argument is an object and the second argument is an instance variable. For example,

  ```
  (@ ($ w) left)
  ```

  translates to

  ```
  (GetValue ($ w) 'left).
  ```

- If the *accessPath* is three arguments, the first argument is an object, the second argument is an instance variable, and the third argument is a property. For example,

  ```
  (@ ($ w) menus DontSave)
  ```

  translates to

  ```
  (GetValue ($ w) 'menus 'DontSave).
  ```

When programming using objects, one object often points to another object. For example, the value of an instance variable is another object. Using different *accessPath* forms allows you to write accesses into objects that are nested inside of other objects. As an example, assume an object ($ pipe) has an instance variable named output with a value ($ tank), which has an instance variable named **level**. The command

```
(@ ($ pipe) output:level)
```

which is equivalent to

```
(@ (@ ($ pipe) output) level)
```

gets the value of the instance variable level of ($ tank).

The "**:**" is a delimiter that indicates instance variable access. The following table shows all the delimiters.

Delimiter	Description
:	Indicates instance variable access.
::	Accesses the value of a class variable whose name follows the double colon.
:,	Accesses the value of a property whose name follows the colon-comma.
.	Sends a message to the object with the selector following the period.
!	Evaluates the next expression.
****	States that the next symbol refers to a Lisp symbol. This is often used in conjunction with the exclamation mark, above.

$ States that the next expression is a LOOPS object.

You can test forms using these delimiters by evaluating `(Parse@ (LIST accessPath) 'IV).`

Arguments: *accessPath* One, two, or three arguments; refer to Behavior.

Returns: See Behavior.

Example: The following examples show the (@ accessPath) form, the Parse@ test, and the translation.

1.
```
(@ foo)
(Parse@ (LIST 'foo) 'IV)
(GetValue self 'foo)
```

2.
```
(@ ::fie:foe)
(Parse@ (LIST '::fie:foe) 'IV)
(GetValue (GetClassValue self 'fie) 'foe)
```

The following three examples are rarely seen in code, but they are additional examples of the expressions that can be interpreted by the system.

3.
```
(@ foo::!::fum)
(Parse@ (LIST 'foo::!::fum) 'IV)
(GetClassValue (GetValue self 'foo)(GetClassValue self 'fum))
```

4.
```
(@ ($ w) fie:,foe.fum)
(Parse@ (LIST '($ w) 'fie:,foe.fum) 'IV)
(← (GetValue ($ w) 'fie 'foe) fum)
```

5.
```
(@ $fie.foe:!\fum.!foo)
(Parse@ (LIST '$fie.foe:!\fum.!foo) 'IV)
(←! (GetValue (← (GetObjectRec 'fie) foe) fum)(GetValue self 'foo))
```

(**@*** *accessPath*) [Macro]

Purpose/Behavior: Provides a concise form for writing embedded **GetValue** forms.

Arguments: *accessPath* An object followed by an arbitrary number of instance variable names.

Returns: The value of a nested instance variable.

Example: The command

```
(@* ($ foo) a b c)
```

translates to

```
(GetValue (GetValue (GetValue ($ foo) 'a) 'b) 'c)
```

(**←@** *accessPath newValue*) [Macro]

Purpose/Behavior: Similar to the @ macro, but used to assign a new value instead of reading a value. Evaluates *newValue*.

Arguments: *accessPath* See Behavior in the @ macro.

newValue Value to be assigned to variable indicated by *accessPath.*

Returns: *newValue*

Example: The following examples show the (←@ accessPath) form, the Parse@ test, and the translation.

1.
```
(←@ foo 1234)
(ParsePut@ (LIST 'foo 1234) 'IV)
(PutValue self 'foo 1234)
```

2.
```
(←@ ($ w) ::left 1234)
(ParsePut@ (LIST ($ w) '::left 1234) 'IV)
(PutClassValue #.($ w) 'left 1234)
```

3.
```
(←@ ($ w) menus DontSave 'Any)
(ParsePut@ (LIST ($ w) 'menus 'DontSave '(QUOTE Any)) 'IV)
(PutValue #.($ w)  'menus  'Any 'DontSave)
```

5.2.2 Support for Changetran

Interlisp uses Changetran to provide an extensive set of facilities for expressing changes to structures, such as push, pushnew, pop, add, change, by using access expressions. You can use any LOOPS access expression in a Changetran context, so that you can now write expressions such as:

```
(push (@ v1) newTop)
(change (@ x)  newValue)
(pushnew (@ colors:,truck) 'red)
(pop (@ ::cv17))
(add (@ x:y:z) 37)
```

The first two are equivalent to:

```
(PushValue self 'v1 (CONS newTop(@ V1)))
(_@ x newValue)
```

This uniform interface allows simpler expressions for changes, and arbitrary extensions through Changetran. See the *Interlisp-D Reference Manual* for more information on Changetran.

5.3 ACCESSING DATA IN CLASSES

5.3 Accessing Data in Classes

A number of functions and methods are available for reading and storing data within classes. Some of these change existing data, and others change the structure of the class by adding variables or properties.

When reading or storing data, some of these functions trigger any active values that are associated with that data. See Chapter 8, Active Values, for a discussion of their behavior.

5.3.1 Metaclass and Property Access

Associated with a class are a metaclass and properties. This section describes the following functions to manipulate their values.

Name	Type	Description
GetClass	Function	Obtains a class's metaclass or properties.

PutClass	Function	Changes the metaclass or class properties of a class.
GetClassOnly	Function	Obtains a class's metaclasses or properties without triggering active values.
PutClassOnly	Function	Changes the metaclass or class properties of a class without triggering active values.
GetClassHere	Function	Obtains a property local to the class.

(**GetClass** *classRec propName*) [Function]

Purpose: Obtains a class's metaclass or properties by following metaclass links.

Behavior: Sends the message **GetClassProp** to *classRec* and passes *propName* as an argument.

Varies according to the arguments.

- If *propName* is NIL, this returns the *class*'s metaclass.
- If *propName* is non-NIL, this looks first in *class* for that property. If it cannot find it there, it looks through *class*'s metaclass links.
- If no property is found, the value of the variable **NotSetValue** is returned.

Arguments: *classRec* Pointer to a class.

propName Property name.

Returns: See Behavior.

Example: The following commands show the variety of returned values.

```
31←(GetClass ($ Window))
 #,($C Class)

32←(GetClass ($ Window) 'doc)
" A LOOPS object which represents a window"

33←(GetClass ($ IconWindow) 'doc)
"An icon window that appears as an irregular shaped image
on the screen -- See the ICONW Library utility"
```

(**PutClass** *classRec newValue propName*) [Function]

Purpose: Changes the metaclass or class properties of a class.

Behavior: Varies according to the arguments.

- If *propName* is NIL, this changes the metaclass of *classRec* to *newValue.* If *newValue* is not a class or the name of a class, this causes a break.
- If *propName* is non-NIL and *classRec* already has this property, this triggers an active value on *propName* if it exists and changes the value of *propName* to *newValue.*
- If *propName* is non-NIL and *classRec* does not have this property, the property is added with the value *newValue*.

Marks the class *classRec* as changed.

Arguments: *classRec* Pointer to a class.

newValue See Behavior.

propName Property name.

Returns: Newly created class object.

Example: The following command changes the **doc** property of class **Datum**:

```
66←(DefineClass 'Datum)
#,($C Datum)

67←(PutClass ($ Datum) '(* this is the updated doc for class Datum) 'doc)
(* this is the updated doc for class Datum)
```

(**GetClassOnly** *classRec propName*) [Function]

Purpose: Obtains a class's metaclass or properties by following superclass links, without triggering active values.

Behavior: Varies according to the arguments.

- If *propName* is NIL, this returns the *classRec*'s metaclass.
- If *propName* is non-NIL, this looks first in *classRec* for that property. If it cannot find it there, it looks through *classRec*'s supers links. This returns the value of the property found without triggering active values.
- If no property is found, the value of the variable **NotSetValue** is returned.

Arguments: *classRec* Pointer to a class.

propName Property name.

Returns: Value depends on the arguments; see Behavior.

Example: The command

```
(GetClassOnly ($ IconWindow) 'doc)
```

returns

```
"An icon window that appears as an irregular shaped image
on the screen -- See the ICONW Library utility"
```

(**PutClassOnly** *classRec newValue propName*) [Function]

Purpose: Changes the metaclass or class properties without triggering active values.

Behavior: Varies according to the arguments:

- If *propName* is NIL, this changes the metaclass of *classRec* to *newValue.* If *newValue* is not a class or the name of a class this causes a break.
- If *propName* is non-NIL and *classRec* already has this property, this changes the value of *propName* to *newValue*. Any active values are replaced.
- If *propName* is non-NIL and *classRec* does not have this property, the property is added with the value *newValue*.

The class *classRec* is marked as changed.

Arguments: *classRec* Pointer to a class.

newValue	A class or the name of a class.
propName	NIL or the name of a class property.

Returns: *newValue*

(**GetClassHere** *classRec propName*) [Function]

Purpose: Obtains property local to class.

Behavior: Gets the class property without triggering active values or inheritance. If there is no local property the value of **NotSetValue** is returned.

Arguments:

classRec	Pointer to a class.
propName	NIL or the name of a class property.

Returns: *newValue*

Example: The command

```
(GetClassHere ($ ActiveValue) 'doc)
```

returns

```
#,NotSetValue
```

5.3.2 Class Variable Access

A class variable can be thought of as being shared by all instances of that class and by all instances of any of its subclasses. This section describes how to access class variables with the functions shown in the following table.

Name	Type	Description
GetClassValue	Function	Returns the value of a class variable or property.
PutClassValue	Function	Stores a value in a class variable or property.
GetClassValueOnly	Function	Returns the value of a class variable or property, without triggering active values.
PutClassValueOnly	Function	Stores a value in a class variable or property, without triggering active values.
GetCVHere	Function	Returns the value of a class variable in a particular class without looking for inherited values.
PutCVHere	Function	Stores a class variable locally with a value if it is not local.

(**GetClassValue** *self varName prop*) [Function]

Purpose: Returns the value of a class variable or property.

Behavior: Varies according to the arguments.

If *self* is an instance, the lookup begins at the class of the instance, since instances do not hav class variables stored locally. If *self* is a class, the lookup is in that class.

- If *prop* is NIL, **GetClassValue** returns the value of the class variable *varName*. If *varName* is not found, this breaks.
- If *prop* is non-NIL, **GetClassValue** returns the value of the property *prop*, associated with the class variable *varName*. If the value is an active value, it is activated. If *varName* has no property *prop*, this returns the value of the variable **NoValueFound**.

If the class does not have a class variable *varName*, **GetClassValue** searches through the super classes of the class until it finds *varName*. Since this is rare, class variables are stored only in the class in which they are defined, and the runtime search is necessary.

Arguments: *self* An instance or a class.

varName A class variable name.

prop Property name.

Returns: Value depends on the arguments; see Behavior.

Example: Given that

```
(← ($ Window) New 'window1)
```

then the command

```
(GetClassValue ($ window1) 'LeftButtonItems)
```

returns the same value as the command

```
(GetClassValue ($ Window) 'LeftButtonItems)
```

The command

```
(GetClassValue ($ Window) 'abcde)
```

breaks. The command

```
(GetClassValue ($ Window) 'LeftButtonItems 'wxyz)
```

returns the value of **NoValueFound**.

(**PutClassValue** *self varName newValue propName*) [Function]

Purpose: Stores a value in a class variable or property.

Behavior: Varies according to the arguments.

If *self* is an instance, the lookup begins at the class of the instance, since instances do not have class variables stored locally. If *self* is a class, the lookup is in that class.

- If *prop* is NIL, **PutClassValue** changes the value of the class variable *varName*.
- If *prop* is non-NIL, **PutClassValue** stores *newValue* as the value of the property, *prop*. If an active value is the current value, it is triggered.

If *varName* is not local to the class, the value is put in the first class in the inheritance list in which *varName* is found. If *varName* is not found, a break occurs.

Arguments: *self* An instance or a class.

varName A class variable name.

newValue A new value.

propName Property name.

Returns: *newValue*

Example: Given that

```
(← ($ Window) New 'window1)
```

then the command

```
(PutClassValue ($ window1) 'LeftButtonItems 2 'number)
```

adds the property number with the value 2 to the class variable **LeftButtonItems** of the class **Window.** The following command performs the same action.

```
(PutClassValue ($ Window) 'LeftButtonItems 2 'number)
```

(**GetClassValueOnly** *classRec varName prop*) [Function]

Purpose: Returns the value of a class variable or property, without triggering active values.

Behavior: Similar to **GetClassValue**, with the following exceptions:

- If **GetClassValueOnly** finds that the value is an active value, the active value is returned without being triggered.
- If *prop* is non-NIL and is not found, **GetClassValueOnly** returns the value of the variable **NotSetValue**.

Arguments: See **GetClassValue**.

Returns: Value depends on the arguments; see Behavior.

Example: The command

```
(GetClassValueOnly ($ Window) 'abcde)
```

breaks. The command

```
(GetClassValueOnly ($ Window) 'LeftButtonItems 'wxyz)
```

returns the value of **NotSetValue.**

(**PutClassValueOnly** *self varName newValue propName*) [Function]

Purpose: Stores the value of a class variable or property, without triggering active values.

Behavior: Similar to **PutClassValue**, except that **PutClassValueOnly** does not trigger an active value, but replaces it with *newValue*.

Arguments: See **PutClassValue** .

Returns: Used for side effect only.

(**GetCVHere** *classRec varName propName*) [Function]

Purpose: Returns the value of a class variable in a particular class without looking for inherited values.

Behavior: Returns the value of the class variable *varName*, or the *propName* property if *propName* is non-NIL.

If the value is an active value, it is returned without being triggered.

If there is no *varName* (or *propName*), this returns the value of the variable **NotSetValue**.

Arguments:

classRec	Must be a class.
varName	A class variable name.
propName	Property name.

Returns: Value depends on the arguments; see Behavior.

Example: The command

```
(GetCVHere ($ NonRectangularWindow) 'LeftButtonItems)
```

returns

```
#,NotSetValue
```

The command

```
(GetCVHere ($ Window) 'LeftButtonItems)
```

returns

```
((Update (QUOTE Update)...)
```

(**PutCVHere** *self varName value*) [Function]

Purpose: Puts a class variable locally with a value if it is not local.

Behavior: Calls (**AddCV** *self varName value*).

Arguments:

self	An instance or a class.
varName	A class variable name.
value	Value for the class variable.

Returns: *value*

5.3.3 Instance Variable Access

An instance variable can be thought of as being local to each instance of a class. The class defines what instance variables and their default values will be in an instance. This section describes the functions that manipulate the default values in the class.

See the *LOOPS Library Modules Manual* for interaction with Masterscope.

Name	Type	Description
GetClassIV	Function	Gets the default value of an instance variable or associated property as defined in a class or one of its supers.
GetClassIVHere	Function	Gets the default value of an instance variable or associated property as defined in a class.

PutClassIV	Function	Changes the default value for an instance variable in a class.

(**GetClassIV** *self varName prop*) [Function]

Purpose: Gets the default value of an instance variable or associated property as defined in a class or one of its supers.

Behavior: If *self* is not bound to a class, an error occurs.

Searches through the supers of *self* to find *varName* or *prop*.

- If *prop* is NIL, this returns the default value for *varName*.
- If *prop* is non-NIL, this returns its default value.

If the default value is an active value, it is returned without being triggered.

Arguments:
self Must be bound to a class.

varName The name of an instance variable.

prop Name of a property associated with *varName*.

Returns: Value depends on the arguments; see Behavior.

Example: The commands

```
(GetClassIV ($ Window) 'window)
(GetClassIV ($ NonRectangularWindow) 'window)
```

both return

```
#,($AV LispWindowAV ...)
```

(**GetClassIVHere** *self varName prop*) [Function]

Purpose: Gets the default value of an instance variable or associated property as defined in a class.

Behavior: Similar to **GetClassIV**. This does not search the super classes of *self* for *varName*. If *varName* or *prop* is not local to *self*, this returns the value of **NotSetValue**.

Arguments:
self Pointer to a class.

varName Name of an instance variable.

prop Name of a property associated with *varName*.

Returns: The default value of *varName* or *prop* or **NotSetValue**.

Example: The command

```
(GetClassIVHere ($ Window) 'window)
```

returns

```
#,($AV LispWindowAV ...)
```

the command

```
(GetClassIVHere ($ NonRectangularWindow) 'window)
```

returns

```
#,NotSetValue
```

(**PutClassIV** *self varName newValue propName*) [Function]

Purpose: Changes the default value for an IV in a class.

Behavior: If *self* is not a class that contains the instance variable *varName*, an error occurs.

- If *propName* is NIL, the default value for the instance variable *varName* is changed to *newValue*.
- If *propName* is non-NIL, the default value for it is changed to *newValue*.

Arguments:

self	Must be a class that contains the instance variable *varName*.
varName	An instance variable name.
newValue	The new default value.
propName	Property name.

Returns: *newValue* (used for side effect only).

Example: After the commands

```
68←(DefineClass 'Datum)
#,($C Datum)

69←(← ($ Datum) AddIV 'id# NIL)
id#
```

the following command changes the default value of the instance variable **id#** to '(7) for all new instances of the class **Datum**:

```
70←(PutClassIV ($ Datum) 'id# '(7))
(7)
```

[This page intentionally left blank]

abstract class	A class which cannot be instantiated, for example, **ActiveValue**.
active value	The mechanism that carries out access-oriented programming for variables in LOOPS. Active values send messages as a side effect of having an object's variable referenced.
activeValue	The previous implementation of the active value concept.
ActiveValue	An abstract class that defines the general protocol followed by all active value objects.
annotatedValue	A special Interlisp-D data type that wraps each **ActiveValue** instance.
AnnotatedValue	An abstract class that allows an annotatedValue to be treated as an object.
browser	A window that allows you to examine and change items in a data structure.
class	A description of one or more similar objects; that is, objects containing the sames types of data fields and responding to the same messages.
class inheritance	The means by which a class inherits variables, values, and methods from its super class(es).
class lattice	A network showing the inheritance relationship among classes.
class variable (CV)	A variable that contains information shared by all instances of the class. A class variable is typically used for information about a class taken as a whole.
inheritance	The means by which you can organize information in objects, create objects that are similar to other objects, and update objects in a simplified way.
Inspector	A Lisp display program that has been modified to allow you to view classes, objects, and active values.
instance	An object described by a particular class. Every object within LOOPS is an instance of exactly one class.
instance variable (IV)	A variable that contains information specific to an instance.
instantiate	To make a new instance of a class.
lattice	An arrangement of nodes in a hierarchical network, which allows for multiple parents of each node.
Masterscope	A Lisp Library Module program analysis tool that has been modified to allow analysis of LOOPS files.
message	A command sent to an object that activates a method defined in the object's class. The object responds by computing a value that is returned to the sender of the message.
metaclass	Classes whose instances are classes or abstract classes.
method	What an object applies to the arguments of a message it receives. This is similar to a procedure in procedure-oriented programming, except that here, you determine the message to send and the object receiving the message

determines the method to apply, instead of the calling routine determining which procedure to apply.

mixin	A class that is used in conjunction with another class to create a subclass. Mixins never have instances, and hence have **AbstractClass** as their metaclass.
object	A data structure that contains data and a pointer to functionality that can manipulate the data.
property list	A place for storing additional information on classes, their variables, and their methods.
selector	Part of a message that is sent to an object. The object uses the selector to determine which method is appropriate to apply to the message arguments.
self	A method argument that represents the receiver of the message.
specialization	The process of creating a subclass from a class, or the result of that process.
subclass	A class that is a specialization of another class.
super class	A class from which a given class inherits variables, values, and methods.
Tofu	An acronym for Top of the universe, which is the highest class in the LOOPS hierarchy.
Unique Identifier (UID)	An alphanumeric identifier that LOOPS uses to store and retrieve objects. Objects do not have UIDs unless they are named, are instances of indexed objects, or are instances printed to a file.
wrap	Objects have fields that can contain data. Some **ActiveValue** can be added so this data is stored within it. When this occurs, the **ActiveValue** wraps the data.

6. METHODS

Methods are the expressions that evaluate when a message is sent to an instance or a class. Methods are analogous to Interlisp-D functions, except that they are defined by a LOOPS class and invoked by sending a message to an instance of that class.

This chapter presents the basic constructs used to create and implement methods. Also included are important methods and functions relevant to the definition and maintenance of methods.

6.1 Categories

LOOPS methods can be divided into categories. This section contains a brief description of each method category. These categories serve as additional documentation only; they do not imply differences in implementation.

Any symbol can be used as a category. Categories can be used as a tool for the organization of methods. Methods may belong to more than one category.

Class [Category]

Messages associated with a class method can only be sent to an object of type class. Methods associated with the class **Class** have this category. See Chapter 3, Classes, for more information on classes.

Object [Category]

The message associated with an object method can only be sent to an object of type object. Methods associated with the class **Object** have this category.

Internal [Category]

Internal methods are low-level system methods, and should not be specialized by users.

Public [Category]

Public methods are defined by the user or the system. These methods can be specialized by users.

Any [Category]

Methods that have not been categorized belong to this category by default.

Masterscope [Category]

Masterscope is an interactive program analysis tool. Methods that are predefined for Masterscope are local only to Masterscope and can be used only when Masterscope has been invoked. Refer to the *Lisp Library Modules Manual* for more information on Masterscope.

(← *self* **AllMethodCategories**) [Method of Class]

Purpose/Behavior: Extracts and lists the categories of all methods defined by the class *self*.

Arguments: *self* Pointer to a class.

Returns: The categories of the methods defined by the class of *self*.

Categories: Class

Example: Line 98 shows the categories of all methods defined in the class *self*.

```
98←(← ($ Class) AllMethodCategories)
(Class Object Masterscope)
```

(← *self* **CategorizeMethods** *categorization*) [Method of Class]

Purpose: Allows you to change how methods are categorized.

Behavior: Varies according to the arguments.

- If *categorization* is NIL, this opens a display editor window with a form that represents the current categorizations. After you have exited from the editor, these new categorizations are installed.
- If *categorization* is non-NIL, it must be of the following form:

  (category1 (selector1 ... selectorN)) (category2 (selector ...)).

  A categorization specified by **CategorizeMethods** deletes any previous categorization; i.e., if method Print for class Thing was in categories Internal and I/O, after doing

  ```
  (← ($ Thing) CategorizeMethods '((Output
  (Print))(Printing (Print))))
  ```

  Print will be only in categories Output and Printing.

Arguments: *self* Pointer to a class.

categorization
A list in the form as described in Behavior, or NIL.

Categories: Class

Example: This example shows how to use **CategorizeMethods** with categorization NIL.

```
1←(← ($ MetaClass) CategorizeMethods)
```

The following display editor window appears:

```
SEdit  Package: INTERLISP
((Any (CreateClass DestroyInstance New NewWithValues))
 (Public (CreateClass DestroyInstance New NewWithValues))
 (Internal NIL)
 (MetaClass (CreateClass))
 (Class (DestroyInstance New NewWithValues)))
```

(← *self* **ChangeMethodCategory** *selector newCategory*) [Method of Class]

Purpose: Changes the category of a selected method.

Behavior: Varies according to the arguments.

- If *selector* is NIL, a menu appears showing the selectors for the class of *self*. This is done using the message **PickSelector** to determine the *selector* that is to have its category changed.
- If *selector* is supplied, but not associated with *self*, this message returns NIL.
- If *newCategory* is an atom, adds *selector* to the category. If *newCategory* is a list of atoms, removes *selector* from all its current categories, then adds it to the categories in the list. If *newCategory* is NIL, pops up a menu showing all of the known categories and an additional item, ***other***. If ***other*** is selected, you are prompted to enter a new category name.

Arguments: *self* Pointer to a class.

selector Method selector for class of *self* or NIL.

newCategory An atom, a list of atoms, or NIL.

Returns: The new category if there was a change made; else NIL.

Categories: Class

Example: The following command changes the categories of the method associated with **Shape1**.

```
2←(← ($ Window) ChangeMethodCategory
'Shape1 '(Window Internal))
(Window Internal)
```


6.2 STRUCTURE OF METHOD FUNCTIONS

6.2 Structure of Method Functions

This section discusses the structure of a LOOPS method.

(**Method** :FUNCTION-TYPE *type ((class selector) self args ...) body...*) [Definer]

Purpose: Similar to **DefineMethod**, but gives more control over the argument list and body syntax. Allows use of Common Lisp lambda argument lists, and Common Lisp syntax in the body of the method. This is the form you will see when editing methods.

Behavior: Defines a method whose argument list is either Interlisp (default) or Common Lisp style. The body of the method may likewise contain either Common Lisp or Interlisp syntax. Common Lisp syntax is distinguished by lexical scoping, etc. (see the *Common Lisp Implementation Notes* for more information).

Arguments:

type The :FUNCTION-TYPE type clause is optional and defaults to :IL.

:IL - The body of the method uses Interlisp syntax, allows CLISP expressions, etc.

:CL - The body of the method uses Common Lisp syntax (is lexically scoped).

class The class to which the method will be attached.

selector The new method's selector.

self This argument must be present and first.

args If type was given as :CL this argument list may contain Common Lisp keys like &OPTIONAL, &KEY and &REST.

body The body of the method. If the type was given as :CL it will be treated as the body of a Common Lisp lambda is, e.g. scoping will be lexical.

Returns: The name of the method function.

Example:

```
12← (Method :FUNCTION-TYPE :CL ((Window Foo) self bar
&OPTIONAL baz &REST glorp)
          (CL:FORMAT T "Bar ~s baz ~s glorp ~s~%%" bar baz
glorp))
13← (← ($ Window) New 'Flarb)
14← (← ($ Flarb) Foo 1 2 3 4)
Bar 1 baz 2 glorp (3 4)
```


6.3 CREATING, EDITING, AND DESTROYING METHODS

6.3 Creating, Editing, and Destroying Methods

This section describes the methods and functions which are used to create, rename, delete, and edit LOOPS methods.

Name	Type	Description
DefineMethod	Function	Defines a new method on a class.
DeleteMethod	Function	Deletes a method from a class.
EditMethod	Method	Invokes the editor on a method of a class.
SubclassResponsibility	Macro	Appears in the template when you create a new method.

(**DefineMethod** *class selector args expr file -*) [Function]

Purpose: Defines a new method on a class.

Behavior: Varies according to the arguments.

- If *args* is a non-NIL symbol and *expr* is NIL, its function definition is installed as the method for (class selector). This definition must accept an appropriate number of arguments and otherwise work as a LOOPS method. Also, *args* must be a symbol of the form **Name1.Name2** for many of the LOOPS internal routines to handle it properly.
- If *args* is a list of arguments and *expr* is a function, its body will be installed as the definition of **class.selector**.

Arguments:

class	Class in which method is defined.
selector	Method selector (message).
args	List of arguments.
expr	Function definition or NIL.
file	Place where method is stored.

Example: The following expression shows how to add a method called **Increment** to a class called **Documentation**.

```
(DefineMethod ($ Documentation) 'Increment '(Number) '(PLUS number 1]
```

(**DeleteMethod** *class selector prop*) [Function]

Purpose: Deletes a method from a class.

Behavior: Varies according to the arguments.

- If *prop* is NIL or T, the method is deleted from the class.
- If *prop* is T, the function definition is also deleted.

Note: You may also delete methods by using the **ClassInheritance Browser**. Position the mouse on the appropriate class, press the middle mouse button, and select **DeleteMethod** from the resulting menu.

Arguments:

class	Class in which method is defined.
selector	Method selector (message).
prop	T or NIL; determines whether the function definition is deleted.

Example: The following command deletes the method associated with '**MyOpen** from **LatticeBrowser**.

```
(DeleteMethod ($ LatticeBrowser) 'MyOpen)
```

(← *self* **EditMethod** *selector commands okCategories*) [Method of Class]

Purpose: Invokes the display editor on a method of a class.

Behavior: Varies according to the arguments.

- If *selector* is NIL, a menu of selectors is presented using the message **PickSelector** in *okCategories* . This can be a list or a symbol.

- If *selector* is non-NIL, and if it corresponds to a method that is in not *self's* class, you are asked whether the method should be created.
- If *selector* cannot be found, the spelling corrector is invoked to find a correct local selector. If it can be corrected, the local method is used, or an inherited method that is made local is used. When the method is finally determined, **EDITF** (refer to the *Lisp Release Notes* and the *Interlisp-D Reference Manual*) is invoked with *commands* passed as the second argument.

Note: You may also edit methods by using the **ClassInheritance Browser**. Position the mouse on the appropriate class, press the middle mouse button, and select **EditMethod** from the resulting menu.

Arguments:

self	Class name.
selector	Refers to the method.
commands	List of **EDITF** commands.
okCategories	Atom or list specifying valid categories.

Categories: Class

(**SubclassResponsibility**) [Macro]

Purpose/Behavior: Appears in the template when you create a new method. It is used to make sure you specialize a method.

6.4 ESCAPING FROM MESSAGE SYNTAX

6.4 Escaping from Message Syntax

The methods described in the previous section manipulate methods in a specific order. Sometimes it may be necessary to invoke multiple inherited methods in some other order. The more general functions in this section have been provided to do this.

CAUTION

These functions do not conform to the conventions of method inheritance and should be used as a last resort and with extreme caution.

The following table shows the items in this section.

Name	Type	Description
DoMethod	Function	Computes the action which should be a method associated with a class and applies it to an object and arguments.
ApplyMethod	Function	Computes the action which should be a method associated with a class and applies it to an object and argument list.

DoFringeMethod	Function	Invokes a method in the class of an object or in each of the super classes for that class.

(**DoMethod** *object selector class arg1 ... argn*) [Function]

Purpose: Computes the action which should be a method associated with *class* and applies it to *object*.

Behavior: All of the arguments are evaluated. If *class* is NIL, **DoMethod** uses the class of *object*. If no method from *class* can be computed from *selector,* an error is generated.

Arguments:

object Instance to which action is applied.

selector Evaluates to a method selector.

class NIL or class in which method name resides.

arg1...argn The arguments for the method.

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

Purpose: Same as **DoMethod**.

Behavior: Applies the selected method to the already evaluated arguments in *argList*; otherwise, this is the same as **DoMethod.**

Arguments:

object Instance to which action is applied.

selector Evaluates to a method name.

arglist The arguments for the method.

class Class in which method name resides.

Example: This example illustrates the MessageNotUnderstood protocol, the function **ApplyMethod**, and the macro **_Super**. This is a specialization of the default **MessageNotUnderstood** message that tries to correct the spelling of the selector. (See Chapter 11, Errors and Breaks, for more information on **MessageNotUnderstood** .)

```
(Method ((DwimObject MessageNotUnderstood)
             self selector mesageArguments superFlg)
  (LET ((correctSelector (FixSelectorSpelling selector)))
    (COND ((correctSelector (ApplyMethod correctSelector mesageArguments))
          (T (_Super))))))
```

Note: *self* is included in the list of **messageArguments**.

(**DoFringeMethods** *object selector arg1 ... argn*) [Function]

Purpose: Invokes method for *selector* in the class of *object* or in each of the super classes for that class.

Behavior: Evaluates all of the arguments. If the method for *selector* in the class of *object* is defined in that class (not through inheritance), **DoFringeMethods** invokes the local method. If there is no local method, **DoFringeMethods** goes down the class of *object,* and for each super invokes its method for selector if one exists. If the supers share supers this can result in the same method being called more than once.

Arguments:

object Class instance.

selector Method selector.

arg1...argn	Arguments to *selector*.

Returns: NIL

6.5 MOVEMENT BETWEEN CLASSES

6.5 Movement between Classes

This section describes functions and methods that are used in moving methods between classes, as well as stack method macros.

6.5.1 Movement of Methods

The following functions and methods are used to move methods, instance variables, and class variables between classes.

Name	Type	Description
RenameMethod	Function	Renames a function used as a method.
MoveMethod	Function	Moves a method from one class to another.
MoveMethod	Method	Moves a method from one class to another.
MoveMethodToFile	Function	Moves a method to this file if it has the same name as a function on a specified file.
CalledFns	Function	Finds names of all functions called from a set of classes.

(**RenameMethod** *classOrName oldSelector newSelector*) [Function]

Purpose: Renames a function used as a method in *classOrName*.

Behavior: This changes the selector for a method. If no method is associated with *oldSelector* or *newSelector*, this generates an error. Explicit references to *oldSelector* such as

```
(←Super self oldSelector))
```

will not be fixed by **RenameMethod**.

Arguments:

classOrName
Class in which function is defined.

oldSelector Old name of method; invokes method before this function is called.

newSelector
New name of method; invokes method after this function is called.

Returns: If successful, returns *newSelector* in the form **ClassName.Selector.**

Example: The following command renames a method named **Foo** to **Fie** in the class **MyClass**.

```
24←(RenameMethod ($ MyClass) 'Foo 'Fie)
```

(**MoveMethod** *oldClassName newClassName selector newSelector files*) [Function]

Purpose: Moves a method from *oldClassName* to *newClassName*. The method is deleted from *oldClassName.*

Behavior: If *newSelector* is a different name than *selector*, **MoveMethod** renames the method. Explicit references to *oldSelector* such as

```
(←Super self oldSelector))
```

will not be fixed by **RenameMethod**.

Note: You may also move methods by using the **ClassInheritance Browser**. Position the mouse on the appropriate class, press the middle mouse button, and select **MoveMethod** from the resulting menu.

Arguments:

oldClassName
Source class.

newClassName
Destination class.

selector Method selector to be moved.

newSelector
New name; if NIL, the existing *selector* is preserved.

files Files in which the change is to occur.

Example: The following command moves the method **Buy** from class **Car** to class **Boat** and renames the method to **Purchase**.

```
25←(MoveMethod ($ Car) ($ Boat) 'Buy 'Purchase)
Boat.Purchase
```

(← *self* **MoveMethod** *newClassName selector*) [Method of Class]

Purpose: Moves a method from the class associated with *self* to *newClassName*.

Behavior: Same as the function **MoveMethod**, except that you cannot rename *selector*.

Arguments:

self Pointer to a class from which the method is taken.

newClassName
Destination class; must be a class, not a class name.

selector Method selector to be moved.

Returns: **NewsClass.Selector**

(**MoveMethodsToFile** *file*) [Function]

Purpose/Behavior: Moves a method to this file if it has the same name as a function on *file*.

Arguments: *file* Name of a file to which methods are moved.

Returns: Normally T; NIL if a method does not have the same name as a function on *file.*

(**CalledFns** *classes definedFlg*) [Function]

Purpose: Finds names of all functions called from a set of *classes*.

Behavior: Varies according to the arguments.

- If *definedFlg* is NIL, all the functions associated with *classes* are returned.
- If *definedFlg* is T, the defined functions are returned.
- If *definedFlg* is 1, the undefined functions are returned.

Arguments: *classes* List of classes to search.

definedFlg NIL, 1, or T.

Returns: NIL or the list of functions.

Example: The following command finds all functions called from the class **Method**.

```
(CalledFns '(Method))
```

6.5.2 Stack Method Macros

This section describes macros that access methods on the stack.

(**ClassNameOfMethodOwner**) [Macro]

Purpose: Uses the stack to perform a help check. Returns the name of the class to which the method on top of the stack belongs.

(**SelectorOfMethodBeingCompiled**) [Macro]

Purpose: Uses the stack to perform a help check. Returns the name of the method being compiled.

(**ArgsOfMethodBeingCompiled**) [Macro]

Purpose: Uses the stack to perform a help check. Returns all arguments associated with the method being compiled.

[This page intentionally left blank]

INDEX

[This page intentionally left blank]

7. MESSAGE SENDING FORMS

Objects in LOOPS communicate with each other by sending messages. This chapter describes the standard message sending forms used in LOOPS.

The following table shows the macros in this section.

Name	Type	Description
←	Macro and Function	Sends a message to an object.
SEND	Macro and Function	Sends a message to an object.
←**!**	Macro and Function	Evaluates the selector and sends a message to an object.
←**IV**	Macro	Invokes the function stored in an instance variable of the object.
←**Try**	Macro	Sends a message to an object only if it has a corresponding method.
←**Proto**	Macro	Sends a message to the prototype instance of a class.
←**Super**	Macro and unction	Combines an inherited method with local code; must appear in the body of a method. F
←**Super?**	Macro	Combines an inherited method with local code; must appear in the body of a method. This does not cause an error if there is no inherited method.
←**SuperFringe**	Macro and Function	Invokes general methods for objects with more than one super class from which to inherit methods; must appear in the body of a method.
←**New**	NLambda NoSpread Macro	Creates an instance of a class and then sends a message to that instance.
FetchMethod	Macro	Finds the function name which implements the method invoked by a selector.

In addition, Chapter 8, Active Values, contains a description of ←**AV**, and Chapter 15, Performance Issues, contains a description of ←**Process** and ←**Process!**.

(← *self sel arg1 ... argn*) [Macro and Function]

Purpose: Sends the message with the selector *sel* to an object *self*. This is the standard way to send a message.

Behavior: Evaluates all arguments except *sel*.

When an object receives a message, it tries to match the selector *sel* with the names of its methods. If the object or the message does not recognize the message, a **Not Understood** error occurs.

The function version does more error checking than the macro and also attempts to convert unbound symbols into names for classes and instances.

Arguments: *self* Pointer to an object.

sel Selector; not evaluated.

arg1...argn Arguments associated with *sel*.

Returns: The value returned by the method associated with *sel*.

Example: In this example, the message **New** is sent to the class **Window**. This returns the newly created instance.

```
76←(← ($ Window) New 'Window1)
#,($& Window (|OZW0.1Y:.;h.Qm:| . 495))
```

(**SEND** *self sel arg1 ... argn*) [Macro and Function]

Purpose: Same as ←, above.

Example: The expression

```
(SEND ($ Window) 'New 'Window1)
```

is equivalent to

```
(← ($ Window) New 'Window1)
```

(←**!** *self sel arg1 ... argn*) [Macro and Function]

Purpose/Behavior: Sends a message with the selector *sel* to an object *self*. It differs from ← in that it evaluates all of its arguments, including *sel.*

Arguments: *self* Pointer to an object.

sel Selector, which is evaluated.

arg1...argn Arguments associated with *sel*.

Example: This example illustrates the fact that ←**!** evaluates the *sel* argument.

The code

```
(for sel in '(Shape Invert)
     do (←! ($ Window1) sel))
```

is equivalent to

```
(←Window1 Shape)(←Window1 Invert)
```

(←**IV** *self IVName arg1...argn*) [Macro]

Purpose: Invokes the function stored in the instance variable *IVName* of the object *self*.

Behavior: Gets a function from *IVName* of self and applies the function to *self* with the arguments *args*. Returns the value of the function or breaks.

←**IV** does not evaluate *IVName*.

Arguments: *self* Pointer to an object.

IVName Instance variable name, which is not evaluated.

arg1...argn Arguments associated with *sel*; bound to arguments specified in the call.

(←**Try** *self sel arg1 ... argn*) [Macro]

Purpose: Sends the message with the selector *sel* to *self*, but only if there is a corresponding method.

Behavior: If *sel* is in fact a selector of *self*, the method is applied and the appropriate value is returned. If the method is not a selector of *self*, the symbol **NotSent** is returned.

Arguments:
self Pointer to an object.

sel Selector; not evaluated.

arg1...argn Arguments associated with *sel*.

Example: The expression `(←($ Window1) abcd)` normally causes a break.

```
79←(←Try ($ Window1) Update)
NIL

80←(←Try ($ Window1) abcd)
NotSent
```

(←**Proto** *class sel arg1 ... argn*) [Macro]

Purpose: Sends a message to the prototype instance of a class.

Behavior: Creates an instance of a class, if necessary, and puts that instance on the class variable **Prototype** of *class*, marking the class as changed. This instance is referred to as the prototype instance. **Proto** then sends the message *sel* to that instance.

Arguments:
class Pointer to a class.

sel Selector; not evaluated.

arg1...argn Arguments associated with *sel*.

Example: Usually only one instance of **LoopsIcon** is needed at a time, so the class **LoopsIcon** keeps one in its class variable **Prototype**.

```
81←(←Proto ($ LoopsIcon) Open)
```

(←**Super** *self sel arg1 ... argn*) [Macro and Function]

Purpose: Can invoke an inherited method within a method. ←**Super** must appear in the body of a method; it cannot be invoked directly.

Behavior: Searches up the class hierarchy and invokes the next more general method of the same name, even if a specialized method is inherited over a distance. It returns the value from that super method. You can use the form (←**Super**) when the arguments are not changed. If no arguments are provided, ←**Super** uses the arguments of the method from which it was called.

←**Super** and the other similar functions are now lexically scoped; that is, it is illegal o call ←**Super** anywhere but within a method body, and any selector given must be the same as the selector for that method.

Arguments:
self Pointer to an object.
sel Selector; not evaluated. Must be the same as the selector of the method in which the ←**Super** appears.

arg1...argn Arguments associated with *sel*.

Example: Two examples of ←**Super** are included:

- One example shows where the arguments are not changed.
- Oneexample shows where the arguments are changed.

Example 1: A use of ←**Super** where the arguments are not changed.

Define a subclass of **Window** that will call **RINGBELLS** before a window is shaped.

```
(DefineClass 'RingingWindow '(Window))
```

Through the browser interface, specialize the method **Shape**, to create the following method.

```
(RingingWindow.Shape
  (Method ((RingingWindow Shape)
           self newRegion noUpdateFlg)
  **COMMENT**    **COMMENT**
          (RINGBELLS)
          (←Super)))
```

Executing the following command calls **RINGBELLS** before the new window is shaped.

```
(←New ($ RingingWindow) Shape)
```

In the method above, if the positions of **RINGBELLS** and (←**Super**) were reversed, **RINGBELLS** would be called after the window was shaped.

Example 2: A use of ←**Super** where the arguments are changed.

Define a subclass of **Window** that will be square.

```
(DefineClass 'SquareWindow '(Window))
```

Through the browser interface, specialize the method **Shape**, to create the following method.

```
(SquareWindow.Shape
  (Method ((SquareWindow Shape)
           self newRegion noUpdateFlg)
  **COMMENT**    **COMMENT**
          (←Super self Shape
                       (create REGION
                          using
                           (SETQ newRegion
                             (OR newRegion
                                 (GETREGION)))
                          HEIGHT ←(fetch WIDTH
                                      of
                                    newRegion))
                       noUpdateFlg)))
```

Executing the following command creates a square window:

```
(←New ($ SquareWindow) Shape)
```

(←**Super?** *self sel arg1 ... argn*) [Macro]

Purpose: Invokes the single next most general method; must appear in the body of a method. This does not cause an error if no inherited method matches.

Behavior: Analogous to ←**Super**. The difference between ←**Super?** and ←**Super** is that ←**Super?** does not break if the *sel* does not have a more general method, whereas ←**Super** generates a break if there is not a more general method.

Arguments:

self Pointer to an object.

sel Selector; not evaluated. Must be the same as the selector of the method in which the ←**Super?** appears.

arg1...argn Arguments associated with *sel*.

(←**SuperFringe** *self sel arg1 ... argn*) [Macro and Function]

Purpose: Invokes general methods for objects with more that one super class from which you wish to inherit methods; must appear in the body of a method.

Behavior: It invokes and executes the next more general method of the same name from each of the classes on the super's list *object's* class. Calling ←**SuperFringe** is analogous to sending ←**Super** up through each item on the super's list. If no arguments are provided ←**SuperFringe** uses the arguments of the method from which it was called.

Arguments:

self Pointer to an object.

sel Selector; not evaluated. Must be the same as the selector of the method in which the ←**SuperFringe** appears.

arg1...argn Arguments associated with *sel*.

(←**New** *class selector arg1 ... argn*) [NLambda NoSpread Macro]

Purpose: Creates an instance of class and then sends *sel* and arguments to that instance.

Behavior: Creates a new instance of a class and sends a message to that instance. It returns the instance as a value and discards any value that may be returned by invoking the method specified by selector. ←**New** is equivalent to (← (← *ClassName* **New**) *selector arg1 ... argn*).

Arguments:

class Pointer to a class.

sel Selector; not evaluated.

arg1...argn Arguments associated with *sel*.

Returns: The new instance.

Example: This example shows an example of ←**New** that creates a new instance of the class **Window** and asks you to shape it.

```
99← (←New ($ Window) Shape)
#,($& Window (|OZW0.1Y:.;h.Qm:| . 497))
```

(← *class* **FetchMethod** *sel*) [Method of Class]

Purpose: Finds the function name which implements the method invoked by sending a message with the selector *sel* to an instance of *class*. The function can be found in either *class* or its supers.

Behavior: Calls the function **FetchMethod**.

Arguments: *class* Pointer to a class.

sel Selector; evaluated.

Returns: The function for *sel* or NIL.

Example: Line 100 shows that the class **Window** implements the method **Update**.

```
100←(← ($ Window) FetchMethod 'Update)
Window.Update
```

Line 1 shows that neither the class **Window** nor any of its supers implements the method **abcd**.

```
1←(← ($ Window) FetchMethod 'abcd)
NIL
```

Line 2 shows that the class **Object** implements the method **PP** which will be triggered when instances of the class **Window** receive the **PP** message.

```
2←(← ($ Window) FetchMethod 'PP)
Object.PP
```

[This page intentionally left blank]

8. ACTIVE VALUES

Active values are used in LOOPS to transpose the access of data within an object to a message being sent to a different object. Typical uses include:

- Causing side effects to occur when data is accessed
- Debugging
- Initializing variables
- Maintaining constraints between variables

An **ActiveValue** is an instance of a subclass of the LOOPS class **ActiveValue**. When an **ActiveValue** instance is installed in the value of a variable, further references to that variable cause messages to be sent to the instance.

LOOPS provides several kinds of active values which are described in this chapter. You can obtain new behavior by specializing one of the existing LOOPS **ActiveValue** subclasses.

When **GetValue** notices that an **ActiveValue** is installed on the variable, it sends the **GetWrappedValue** message to the **ActiveValue**. Similarly, when **PutValue** notices that an **ActiveValue** is installed on the variable, it sends the **PutWrappedValue** message to the **ActiveValue**. The value returned from the get or put is the value returned from the message send. Each specialization of **ActiveValue** either inherits these methods from its superclasses or specializes them to call user code. The messages are received and processed by the **ActiveValue** instances.

For example, assume that you are modeling a simulation that requires the value of an instance variable called windSpeed to be a random value. You can make the value of windSpeed into an active value called ($ RandomWindSpeedAV1). Now, if you try to determine the value of windSpeed by entering

```
(@ ($ SomeAirport) windSpeed)
```

the value returned from this expression is the value returned from

```
(← ($ RandomWindSpeedAV1) GetWrappedValue . otherArgs)
```

This returns the required random value.

The variable containing the **ActiveValue** may still have a current value. Most system **ActiveValue** subclasses are specializations of **LocalStateActiveValue**, which uses an instance variable **localState** in the **ActiveValue** to hold the value.

For efficient implementation, LOOPS uses a special Interlisp data type, the annotatedValue data type, to "wrap" each **ActiveValue** instance when it is installed as a value within an object; the **annotatedValue** contains the **ActiveValue** instance. That is, if the value of an instance variable is said to be an active value, in actuality, the value of the instance variable is an annotatedValue which contains the active value. This allows every **GetValue** or **PutValue** to use Interlisp's microcoded type checking mechanism to see if it should be processed normally or via the **ActiveValue** mechanism. This extra layer is largely invisible in application programs. LOOPS also contains a class

AnnotatedValue to handle the occasional accident when a user forgets about the distinction between annotatedValue and **ActiveValue**, and attempts to treat an annotatedValue as a LOOPS object.

The class **ActiveValue** defines the general protocol followed by all active value objects.   Methods setting up the basic functionality of **ActiveValues** are defined in this class and inherited by all its specializations. Methods defined in this class include **AVPrintSource** to specify how annotatedValues print, **AddActiveValue** to install an **ActiveValues**, and **DeleteActiveValue** to delete an installed **ActiveValue**.

The class **ActiveValue** itself is an abstract class; that is, it is a placeholder in the class hierarchy that is not intended to be instantiated. When this documentation refers to an active value object, it is referring to an instantiation of a specialization of the class **ActiveValue**.

Note: The current **ActiveValue** is different from the **activeValue** implementation in the Buttress version of LOOPS. See Appendix A, Active Values in Buttress LOOPS, for more information.

8.1   Using Active Values

A general template is available when using active values. As with all templates, you should not blindly follow it.   A good understanding of the active value mechanism is necessary to avoid errors in more complicated situations.

- Determine the functionality that you want the active value to provide. For example, will it cause a side effect to occur on access of data? Will it maintain constraints between two pieces of data? The required functionality will give an indication of which active value class you should use.
- Specialize one of the active value classes to satisfy your specific requirements, if necessary.
- Create an instance of the active value class that you have chosen or created.
- Initialize the contents of that instance, if necessary.
- Install that active value instance on the data that you want to become ative. This is accomplished by using the **AddActiveValue** message.

In a number of situations, you may want to install an active value on an instance variable for every instance of a class. One technique for doing this is discussed in Section 8.5, "Active Values in Class Structures."

8.2 SPECIALIZATIONS OF THE CLASS ACTIVEVALUE

8.2   Specializations of the Class ActiveValue

The class **ActiveValue** is an abstract class, and therefore cannot be instantiated. This class contains a number of methods, described in Section 8.3, "ActiveValue Methods," that are necessary for the active value mechanism to function. As a user, you will be making active values which are instances of some subclass of **ActiveValue**, either one of those already provided or one that you created. Figure 8-1 shows the class **ActiveValue** and its specializations. This section describes the subclasses of **ActiveValue** in order of their appearance in this figure. Also included is information on specializing active values.

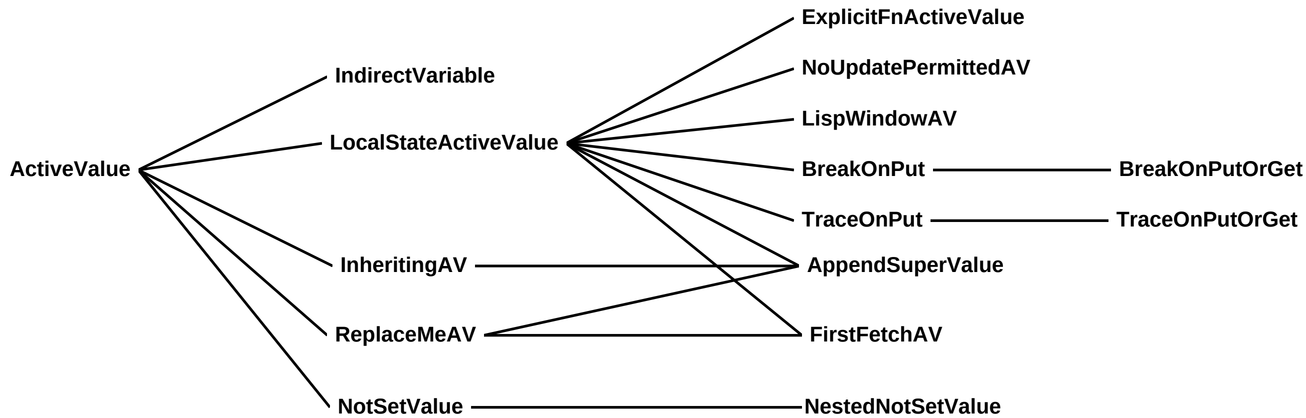


*Figure 8-1. The Class **ActiveValue** and its Specializations*

8.2.1 IndirectVariable

This specialization sets up the functionality of an **ActiveValue** to return the value of another variable as its value. It is analogous to the concept of indirect addressing in other computer languages.

Note: Indirect variables must be the innermost of nested active values. Wrapping precedence (see Section 8.3, "Active Value Methods") insures this.

IndirectVariable [Class]

Purpose: Enables variable values to be accessed indirectly from other variables. This simulates two variables sharing the same memory location. This is a useful technique for implementing simulations and enforcing constraints.

Behavior: When a Fetch is made on the variable containing the **IndirectVariable** instance, this active value retrieves and returns the value of the tracked variable. If a Store is made with the variable containing the **IndirectVariable** instance, this active value stores the new value in the tracked variable. Essentially, this forces the two variables to share the same data.

Instance Variables:

object Object instance containing the tracked value.

varName The name of the variable being tracked.

propName If non-NIL, the name of the variable property being tracked.

type Type of variable being tracked. Value can be CV, IV or NIL, which defaults to IV.

Examples: Several examples are included to show the use of **IndirectVariable**.

Example 1: Consider a chemical reactor simulation where you have a tank draining into a pipe. The output pressure of the tank needs to equal the input pressure of the pipe. The following demonstrates this.

First, build the appropriate pipe and tank classes and make instances of them.

```
78← (DefineClass ’Tank)
#,($C Tank)

79← (DefineClass ’Pipe)
#,($C Pipe)
```

```
80← (← ($ Tank) AddIV 'outputPressure)
outputPressure

81← (← ($ Pipe)  AddIV  'inputPressure)
inputPressure

82← (← ($ Tank) New 'tank1)
#,($& Tank (NYW0.0X%:.aF4.6R8 . 5))

83← (← ($ Pipe) New 'pipe1)
#,($& Pipe (NYW0.0X%:.aF4.6R8 . 6))
```

Create an instance of **IndirectVariable** and initialize its contents to point to the tank's output pressure.

```
84← (← ($ IndirectVariable) New 'indVar1)
#,($& IndirectVariable (NYW0.0X%:.aF4.6R8 . 7))

85← (←@ ($ indVar1) object ($ tank1))
#,($& Tank (NYW0.0X%:.aF4.6R8 . 5))

86← (←@ ($ indVar1)   varName 'outputPressure)
outputPressure
```

Install the active value instance as the value of the pipe's input pressure.

```
87← (← ($ indVar1) AddActiveValue ($ pipe1) 'inputPressure)
#,($AV IndirectVariable (indVar1 (NYW0.0X%:.aF4.6R8 . 7)) (object
#,($& Tank (NYW0.0X%:.aF4.6R8 . 5))) (varName outputPressure))
```

Accesses to either the pipe's input pressure or the tank's output pressure produce the same result.

```
90← (@ ($ pipe1) inputPressure)
NIL

92← (←@ ($ pipe1)  inputPressure 100)
100

94← (@ ($ tank1)  outputPressure)
100

95← (←@ ($ tank1)  outputPressure 200)
200

96← (@ ($ pipe1) inputPressure)
200
```

An inspector window of ($ pipe1) appears as follows:

```
All Values of Pipe ($ pipe1).
inputPressure #,($AV IndirectVariable (indVar1 (
```

Example 2: Consider a conveyor that must be three feet above a bin. Assume both have an instance variable named height.

First, create the classes and instances.

```
53← (DefineClass 'Bin)
#,($C Bin)

54← (DefineClass 'Conveyor)
#,($C Conveyor)

55← (← ($ Bin) AddIV 'height 0)
height
```

```
56← (← ($ Conveyor) AddIV 'height 0)
height

57← (← ($ Bin) New 'bin1)
#,($& Bin (|DAW0.1Y:.H53.]99| . 506))

58← (← ($ Conveyor) New 'conveyor1)
#,($& Conveyor (|DAW0.1Y:.H53.]99| . 507))
```

Create a specialization of **IndirectVariable** and specialize the methods **GetWrappedValue** and **PutWrappedValue**. You need to specialize **IndirectVariable** because you do not want to maintain equality between the two variables, but instead want to maintain a different mathematical relationship. The **_Super**s are used to use the default behavior of **IndirectVariable** which takes care of retrieving or storing the data into the tracked variable.

```
59← (DefineClass '3FeetAbove
'(IndirectVariable))
#,($C 3FeetAbove)
```

```
SEdit 3FeetAbove.GetWrappedValue  Package: INTERLISP
(Method ((3FeetAbove GetWrappedValue) self
          containingObj varName propName type)
         ;; Compute value of indirect variable and add 3 to it
         (PLUS 3
               (←Super self GetWrappedValue
                       containingObj varName propName
                       type)))
```

```
SEdit 3FeetAbove.PutWrappedValue  Package: INTERLISP
(Method ((3FeetAbove PutWrappedValue) self
          containingObj varName newValue propName type)
         ;; Instead of placing newValue here, substract 3
         ;; from it and put it in the indirect variable
         (←Super self PutWrappedValue containingObj
                 varName (DIFFERENCE newValue 3)
                 propName type)
         ;; return newValue
         newValue)
```

Create an instance of **3FeetAbove** and initialize its contents to point to the bin's height.

```
65← (← ($ 3FeetAbove) New '3fa1)
#,($& 3FeetAbove (|DAW0.1Y:.H53.]99| . 505))

66← (←@ ($ 3fa1) object ($ bin1))
#,($& Bin (|DAW0.1Y:.H53.]99| . 506))

67← (←@ ($ 3fa1) varName 'height)
height
```

Install this instance of **3FeetAbove** as the value of the conveyor's height.

```
68← (← ($ 3fa1) AddActiveValue ($ conveyor1) 'height)
#,($AV 3FeetAbove (3fa1 (|DAW0.1Y:.H53.]99| . 505))
     (object #,($& Bin (|DAW0.1Y:.H53.]99| . 506)))
            (varName height))
```

The height of **bin1** defaults to 0, what is the height of **conveyor1**?

```
69← (@ ($ bin1) height)
0
```

```
70← (@ ($ conveyor1) height)
3
```

Set either **bin1's** height or **conveyor1's** height and notice how they track each other.

```
71← (←@ ($ bin1) height 15)
15

72← (@ ($ conveyor1) height)
18

73← (←@ ($ conveyor1) height 21)
21

74← (@ ($ bin1) height)
18
```

8.2.2 LocalStateActiveValue

This specialization sets up a class of **ActiveValue** that contains the instance variable **localState**, which is used primarily for storing the value of the referenced variable.

If you need an active value that will produce your own specific side-effect, you will most likely use your own specialization of **LocalStateActiveValue.** The data that would have been accessed, had an active value not been installed, is stored in the **localState** instance variable.

LocalStateActiveValue [Class]

Purpose: Creates a subclass of **ActiveValue** with an instance variable to hold the current value of the referenced variable.

Behavior: Holds the data that normally is stored in the variable where it is installed. At installation time, the current variable value is placed in the **localState** instance variable of the **ActiveValue**. Subclasses of **LocalStateActiveValues** are the most common **ActiveValue** instances.

The class **LocalStateActiveValue** is commonly specialized. In particular, it is usually desirable to specialize the methods **GetWrappedValue** and **PutWrappedValue** associated with new subclasses of **LocalStateActiveValue**. These methods implement the active value messages sent when the variable is accessed.

Instance Variable: **localState** Stores the value of the referenced variable.

Examples: Several examples are included to show the use of **LocalStateActiveValue**.

Example 1: In this example, an active value will print a message if an attempt is made to store an out-of-range value in an instance variable.

Define a subclass of **LocalStateActiveValue** and give it two instance variables that will store the values of the limits.

```
99← (DefineClass 'WarningAV '(LocalStateActiveValue))
#,($C WarningAV)

100← (← ($ WarningAV) AddIV 'lowTrigger 0)
lowTrigger

101← (← ($ WarningAV) AddIV 'highTrigger 100)
highTrigger
```

Specialize **LocalStateActiveValue**'s **PutWrappedValue** method to create one for **WarningAV**.

```
SEdit WarningAV.PutWrappedValue  Package: INTERLISP
(Method ((WarningAV PutWrappedValue) self containingObj
          varName newValue propName type)
         ;; If newValue is greater than highTrigger or lower
         ;; than lowTrigger, then print a message, but store
         ;; the data anyway
         (if (OR (< newValue (@ lowTrigger))
                 (> newValue (@ highTrigger)))
            then (PRINTOUT T "The value " newValue
                            " is out of range." T))
         (←Super self PutWrappedValue containingObj
                 varName newValue propName type))
```

Create an instance of a Bin and attach an instance of a **WarningAV** to its height.

```
4← (← ($ Bin) New 'bin3)
#,($& Bin (|DAW0.1Y:.H53.]99| . 513))

5← (←New ($ WarningAV) AddActiveValue ($ bin3) 'height)
#,($& WarningAV (|DAW0.1Y:.H53.]99| . 514))
```

Attempt to store various values into the bin's height.

```
7← (←@ ($ bin3) height 10)
10

8← (←@ ($ bin3) height -10)
The value -10 is out of range.
-10

9← (←@ ($ bin3) height 110)
The value 110 is out of range.
110

10← (@ ($ bin3) height)
110
```

Example 2: In this example, an active value will return a random number when it is read from. Puts to it will change the range of the random value returned on gets. This will use **localState** for something other than storing the data for active values that provide only pure side-effect behavior.

```
99← (DefineClass 'RandomAV '(LocalStateActiveValue))
#,($C RandomAV)

100← (← ($ RandomAV) AddIV 'localState '(0 100))
localState
```

Specialize **LocalStateActiveValue**'s **PutWrappedValue** and **GetWrappedValue** methods to create them for **RandomAV**.

```
SEdit RandomAV.GetWrappedValue  Package: INTERLISP
(Method ((RandomAV GetWrappedValue) self containingObj
          varName propName type)
         ;; Get the value of the localState.

         ;; We'll assume for this example that the value returned
         ;; is a list of two numbers.  If it is not, we get an error.
         (APPLY #'RAND (@ localState)))
```

```
SEdit RandomAV.PutWrappedValue  Package: INTERLISP
(Method ((RandomAV PutWrappedValue) self containingObj
          varName newValue propName type)
         ;; We'll assume that the new value is a list of two
         ;; numbers.
         (←@ localState newValue))
```

Now, try to test this.

```
36← (DefineClass 'RandomTest)
#,($C RandomTest)

39← (← IT AddIV 'randomIV)
randomIV

40← (← ($ RandomTest) New 'rt1)
#,($& RandomTest (DCW0.0X%:.aF4.5S; . 518))

41← (←New ($ RandomAV) AddActiveValue ($ rt1) 'randomIV)
#,($& RandomAV (DCW0.0X%:.aF4.5S; . 519))

42← (@ ($ rt1) randomIV)
24

43← redo
32

44← redo
9

45← redo
49

46← (←@ ($ rt1) randomIV '(4.0 5.0))
(4.0 5.0)

47← (@ ($ rt1) randomIV)
4.190201

48← REDO
4.1129

49← REDO
4.380234

50← REDO
4.397278
```

8.2.2.1 ExplicitFnActiveValue

ExplicitFnActiveValue emulates the activeValue implementation from the Butttress version of LOOPS. Users are discouraged from using this particular form of active values within new projects.

See the *LOOPS Users' Modules* for details on LOOPSBACKWARDS, which describes **ExplicitFnActiveValue**. See Appendix A, Active Values in Buttress LOOPS, for details on the compatibility of **ActiveValue** with activeValue.

8.2.2.2 NoUpdatePermittedAV

This specialization sets up a class of **ActiveValue** that prevents the value of a variable from being replaced.

NoUpdatePermittedAV [Class]

Purpose: Prevents the value of a variable from being replaced using the **PutValue** method.

Behavior: Stores the current value of the variable in **localState**, then prevents it from being updated. **GetWrappedValue** requests return the value found in **localState**, but **PutWrappedValue** requests cause a break with the break message **NoUpdatePermitted!**, or a message if sent from the Exec.

Example: Suppose an identification number for a piece of data should never be changed. Installing a **NoUpdatePermittedAV** in the data's ID number will cause a break if a replacement attempt is made.

Start with a user-defined class named **Datum.** Make a **Datum** instance named **Datum1**. Set the instance variable named **idNumber** to the value 999. Look at the instance. Make a new instance of **NoUpdatePermittedAV**, and name it **NumberGuard**. Install the **ActiveValue** in the instance variable **idNumber** of the instance **Datum1**. Look at the **ActiveValue** instance; the **localState** instance variable contains the previous value of **idNumber**. To test this **ActiveValue**, attempt to replace the **idNumber** of **Datum1** with a new value.

```
67← (DefineClass 'Datum)
#,($C Datum)

68← (← ($ Datum) AddIV 'idNumber 0)
idNumber

69← (← ($ Datum) New 'Datum1)
#,($& Datum (|DAW0.1Y:.H53.]99| . 524))

70← (←@ ($ Datum1) idNumber 999)
999

71← (← ($ Datum1) PP)
(DEFINST Datum (Datum1 (|DAW0.1Y:.H53.]99| . 524)) (idNumber 999))
#,($& Datum (|DAW0.1Y:.H53.]99| . 524))

74← (← ($ NoUpdatePermittedAV) New 'NumberGuard)
#,($& NoUpdatePermittedAV (|DAW0.1Y:.H53.]99| . 525))

75← (← ($ NumberGuard) AddActiveValue ($ Datum1) 'idNumber)
#,($AV NoUpdatePermittedAV (NumberGuard (|DAW0.1Y:.H53.]99| . 525))
       (localState 999))

76← (← ($ Datum1) PP)
(DEFINST Datum (Datum1 (|DAW0.1Y:.H53.]99| . 524))
         (idNumber #,($AV NoUpdatePermittedAV (NumberGuard
                (|DAW0.1Y:.H53.]99| . 525)) (localState 999))))
#,($& Datum (|DAW0.1Y:.H53.]99| . 524))

77← (←@ ($ Datum1) idNumber 888)
No update permitted!
NIL
```

8.2.2.3 LispWindowAV

This specialization sets up a class of **ActiveValue** used by the system to guarantee that the **window** instance variable within a LOOPS **Window** instance contains an Interlisp window. This class provides functionality required by the LOOPS system, and should not generally used by LOOPS users.

LispWindowAV [Class]

Purpose: Guarantees that a variable contains a window which has been made into a LOOPS window.

Behavior: Meant to be installed only in the instance variable **window** of instances of class **Window.** A specialization of **LocalStateActiveValue**. Checks to see if its **localState** is a window, and assures that other instance variables of the window instance are set correctly. See Chapter 19, Windows, for further details.

8.2.2.4 Breaking and Tracing Active Values

The following active values are all specializations of **LocalStateActiveValue** and are used for debugging, as described in Chapter 12, Breaking and Tracing. This chapter also describes **UnbreakIt**, which unbreaks or untraces a method of a class. These classes provide functionality required by the LOOPS system, and are not generally used by LOOPS users.

Note: All breaks and traces occur before the variable is read or modified.

BreakOnPut [Class]

Purpose: Breaks when a replacement attempt is made.

Behavior: Breaks when a replacement attempt is made. Local variables bound at the time of the break are **containingObj**, **varName**, and **propName**.

BreakOnPutOrGet [Class]

Purpose: Breaks when a retrieval or replacement of a variable is made. This is a specialization of **BreakOnPut**.

Behavior: Break occurs before any access to the variable where it is installed. Local variables bound at the time of the break are **containingObj**, **varName**, and **propName**.

TraceOnPut [Class]

Purpose: Traces replacements of a variable.

Behavior: Has a specialized **PutWrappedValue** method that causes the values of the arguments **containingObj**, **varName**, and **propName** to print in the trace window when the variable is about to be modified.

TraceOnPutOrGet [Class]

Purpose: Traces retrievals and replacements of a variable. This is a specialization of **TraceOnPut**.

Behavior: The **GetWrappedValue** method is also specialized so that the variable is traced before any access of the variable where it is installed.

8.2.2.5 AppendSuperValue

This specialization allows the value of a variable to reside only partially in the local instance or class. This is a specialization of the **ReplaceMeAV**, **InheritingAV**, and **LocalStateActiveValue** classes.

AppendSuperValue [Class]

Purpose: Allows the value of a variable to be defined by both a local value and an inherited value.

Behavior: When an instance of **AppendSuperValue** is installed in a variable, Get-references return its **localState** appended to the end of the inherited value the variable would have if it had no local value. Any **PutValue** to the variable replaces the active value, not just the **localState**; **InheritingAV** and its specializations are designed for use more in class variables where replacement is infrequent.

Examples: Several examples are included to show the use of **AppendSuperValue**.

Example 1: Append the **localState** of the instance variable **idNumber** to the default value specified in the class description.

```
23←(DefineClass 'Datum)
#,($C Datum)

24←(← ($ Datum) AddIV 'idNumber '(5))
idNumber

25←(← ($ Datum) New 'Datum1)
#,($ Datum1)

26←(@ ($ Datum1) idNumber)
(5)

27←(←@ ($ Datum1) idNumber '(9))
(9)

28←(@ ($ Datum1) idNumber)
(9)

29←(←New ($ AppendSuperValue) AddActiveValue ($ Datum1) 'idNumber)
#,($& AppendSuperValue (45 . 54648))

30←(@ ($ Datum1) idNumber)
(5 9)
```

Example 2: In this example, there are two classes of cars; the **Two-tone-Car** class is a subclass of the class **Car**. Each **Car** class has the instance variable color. The default value for color in the class **Car** is (white).

```
89← (DefineClass 'Car)
#,($C Car)

90← (DefineClass 'Two-tone-Car '(Car))
#,($C Two-tone-Car)

91← (← ($ Car) AddIV 'color '(white))
color
```

```
92← (← ($ Two-tone-Car) AddIV 'color)
color
```

The default value for color in the class **Two-tone-Car** is an instance of **AppendSuperValue** with its localState set to (blue). The technique for adding active values as default values in a class is discussed in Section 8.3.1, "Adding and Deleting Active Values.".

```
100← (← ($ AppendSuperValue) New 'asv1)

1← (←@ ($ asv1) localState '(blue))

2← (PutClassIV ($ Two-tone-Car) 'color
                      (create annotatedValue
                         annotatedValue ← ($ asv1)))
#,($AV AppendSuperValue (asv1 (|DAW0.1Y:.H53.]99| . 528))
                      (localState (blue)))

9← (← ($ Car) New 'car1)
#,($& Car (|DAW0.1Y:.H53.]99| . 531))

10← (← ($ Two-tone-Car) New 'ttcar1)
#,($& Two-tone-Car (|DAW0.1Y:.H53.]99| . 532))

11← (@ ($ car1) color)
(white)
```

When an instance of a **Two-tone-Car** is created the default value for its instance variable color is the combination of the values in both the classes **Car** and **Two-tone-Car**. The first inspector shows the existence of the active value that provides this behavior. As soon as one puts a value for color in this instance, the **AppendSuperValue** active value is replaced by the new value as shown in the second inspector.

```
12← (@ ($ ttcar1) color)
(white blue)

13← (INSPECT ($ ttcar1))
{WINDOW}#50,5000
```

```
All Values of Two-tone-Car ($ ttcar1).
color #,($AV AppendSuperValue (asv1 (|[
```

```
14← (←@ ($ ttcar1) color '(tan brown))
(tan brown)
```

```
All Values of Two-tone-Car ($ ttcar1).
color (tan brown)
```

For another example, see the **TitleItems** class variable of the class **ClassBrowser**, where **AppendSuperValue** is used to add menu items to an inherited menu.

8.2.2.6 FirstFetchAV

This specialization has instances that have an expression as the value of the instance variable **localState**. These active values allow a form to be evaluated the first time that they are read.

FirstFetchAV [Class]

Purpose: This is a specialization of the **ReplaceMeAV** mixin and **LocalStateActiveValue**. Instances of this class have an expression as the value of the instance variable **localState.**

Behavior: On the first get access, the expression in **localState** is evaluated. The resulting value replaces the **FirstFetchAV** so the variable is no longer an active value. On the first put access, the put value replaces the **FirstFetchAV** so the variable is no longer an active value. A **FirstFetchAV** is often used as the default value for a variable. This class also specializes the method **AVPrintSource** so that instances print as follows when wrapped in an annotatedValue:

```
#,(Defer contentsOfLocalState)
```

CAUTION

FirstFetchAVs cannot be shared. Unlike lists, SEdit does not make copies of active values. Hence, if active values are copied in SEdit, they will share structure, and if one is modified, all will be changed.

Workaround: Use **CopyActiveValue** to copy the active value instance and the local state into each instance which uses the **FirstFetchAV**. See Section 8.3.4, "Shared ActiveValues in Variable Inheritance," for information on **CopyActiveValue**.

Example: An example application of **FirstFetchAV** is an instance variable that stores a font descriptor. A font descriptor in a class definition does not save correctly;only the pointer to the descriptor is saved. A **FirstFetchAV** stores the expression used to create the descriptor. So, for example the expression held in the **localState** of the **FirstFetchAV** is

```
(FONTCREATE 'HELVETICA 12 'BOLD)
```

On the first access of the instance variable, the font descriptor produced by calling **FONTCREATE** replaces the **FirstFetchAV**.

The complete example follows.

```
29← (DefineClass 'TextObject)
#,($C TextObject)

30← (← ($ TextObject) AddIV 'font)
font

31← (← ($ FirstFetchAV) New 'ffav1)
#,($& FirstFetchAV (|DAW0.1Y:.H53.]99| . 535))

32← (←@ ($ ffav1) localState '(FONTCREATE 'HELVETICA 12 'BOLD)]
(FONTCREATE (QUOTE HELVETICA) 12 (QUOTE BOLD))

33← (PutClassIV ($ TextObject) 'font
                    (create annotatedValue
                      annotatedValue ← ($ ffav1)))
#,(Defer (FONTCREATE (QUOTE HELVETICA) 12 (QUOTE BOLD)))

34← (← ($ TextObject) Edit)
TextObject
```

```
SEdit #,($C TextObject)  Package: INTERLISP
((MetaClass Class Edited%:            ; Edited  4-Dec-87
                                      ; 14:22 by RBGMartin
  )
 (Supers Object) (ClassVariables)
 (InstanceVariables
   (font #,(Defer (FONTCREATE (QUOTE HELVETICA) 12 (QUOTE BOLD)))
         doc (* IV added by MARTIN)))
 (MethodFns))
```

```
35← (← ($ TextObject) New 'to1)
#,($& TextObject (|DAW0.1Y:.H53.]99| . 536))

36← (INSPECT ($ to1))
{WINDOW}#47,125470
```

```
All Values of TextObject ($ to1).
font #,(Defer (FONTCREATE (QUOTE
```

```
37← (@ ($ to1) font)
{FONTDESCRIPTOR}#74,167334
```

```
All Values of TextObject ($ to1).
font {FONTDESCRIPTOR}#74,167334
```

8.2.3 InheritingAV

This specialization of **ActiveValue** is used as a mixin to add the **InheritedValue** method.

InheritingAV [Class]

Purpose: Used as a mixin to add the **InheritedValue** method.

Behavior: An abstract class, adds a method **InheritedValue** which allows looking at the value a variable would have if it had no local value, as **NotSetValue** would work. Used as a mixin to add this capability to other specializations of **ActiveValue**.

Example: Used as super class of **AppendSuperValue** to provide incremental menus in various parts of LOOPS.

(← *self* **InheritedValue** *containingObj varName propName type*) [Method of InheritingAV]

Purpose/Behavior: Allows viewing the value a variable would have inherited if it had no local value yet assigned. Similar to the way **NotSetValue** works, it is removed by an assignment to the variable.

Arguments:

self — **InheritingAV** instance.

containingObj — The instance or class that contains the variable to be viewed.

varName — In the *containingObj* the variable to be viewed.

propName — Name of an instance variable or class variable property to be looked at. If *propName* is NIL, the variable itself is viewed.

type	*O*ne of IV, CV, or NIL: a *type* of IV or NIL indicates that the variable is an instance variable or an instance variable property of *containingObj*; a *type* of CV indicates a class variable or class variable property of *containingObj*.

Returns: The value which would have been inherited if the variable had no local value.

8.2.4 ReplaceMeAV

This specialization of the class **ActiveValue** sets up the functionality to replace itself on the first Put- access.

ReplaceMeAV [Class]

Purpose: Specializes the method **PutWrappedValue** to simply replace itself on the first Put- request.

Behavior: No variables are defined in this class. It is an abstract class not intended for instantiation. It is a mixin (see Chapter 3, Classes) to be combined in specialization with another class to add its functionality to the subclass.

Example: **FirstFetchAV** combines **LocalStateActiveValue** and **ReplaceMeAV** to get an **ActiveValue** that replaces itself with the value of an expression stored in the instance variable **localState**.

8.2.5 NotSetValue

This section describes where and when instances of this class appear in user-defined objects.

CAUTION

Do not specialize the classes **NotSetValue** and **NestedNotSetValue**. The documentation is provided here only to explain the functionality that these classes provide to the LOOPS system.

NotSetValue [Class]

Purpose: This specialization of the class **ActiveValue** is unique in that it was created primarily for implementing instance variable inheritance. It has no instance variable to hold a local value and is replaced after the first Put- variable access.

Behavior When an instance of any LOOPS object is created, its instance variables are initialized to contain the value of the variable **NotSetValue**. **NotSetValue** is an annotatedValue whose **ActiveValue** is the only instance of the class **NotSetValue**. The value of **NotSetValue** stored in an instance variable may be replaced within other initialization procedures of new instances that are invoked by the methods **NewWithValues** and **NewInstance** and the instance variable property **:initForm**.

The class **NotSetValue** specializes the default **ActiveValue** protocol to trigger instance variable inheritance. An annotatedValue check is always done by **GetValue** and **PutValue**. LOOPS speeds up instance generation by always initializing instance variables to the value **NotSetValue**. If a retrieval attempt is made on the variable, **NotSetValue** finds the inherited value and returns that value. If no requests are made for the value of the variable, there is no overhead for the instance variable.

The term local value refers to the values LOOPS has actually written into that instance's instance variables. The local value is always equal to **NotSetValue** before the first Put- access, and to a new value after the first Put- access.

The annotatedValue **#,NotSetValue** is bound to the Lisp variable **NotSetValue**. It must always be on the inside of any sequence of nested **ActiveValues**. Its **WrappingPrecedence** method returns NIL, ensuring this functionality. **NotSetValue** has no **localState** instance variable to hold any nested **ActiveValues**.

See Section 8.3.4, "Shared ActiveValues in Variable Inheritance," for information on **ActiveValues** as default values.

Example: Consider the class **Datum** with the instance variable **idNumber**. Create a new instance named **Datum2**. A standard **GetValue** or @ call returns the default value of **idNumber**, since nothing else has yet been assigned. The call **GetIVHere** shows that the value is not stored in the instance, but is actually returned by **NotSetValue**.

```
91←(← ($ Datum) New 'Datum2)
#,($ Datum2)

92←(@ ($ Datum2) idNumber)
NIL

93←(GetIVHere ($ Datum2) 'idNumber)
#,NotSetValue
```

8.2.5.1 NestedNotSetValue

This subclass of the class **NotSetValue** is used by the internal of LOOPS to solve the problem of using active values as default values.

8.2.6 User Specializations of Active Values

If new specializations of the class **ActiveValue** are defined, the methods **GetWrappedValueOnly** and **PutWrappedValueOnly** might need to be specialized (LOOPS-defined specializations of **ActiveValue**, such as **LocalStateActiveValue**, have already done this). You may also want to specialize the following methods:

AVPrintSource Prints an **ActiveValue** instance.

GetWrappedValue Method associated with getting an **ActiveValue**.

PutWrappedValue Method associated with putting an **ActiveValue**.

WrappingPrecedence Returns T, NIL, or a number to specify order of **ActiveValue** nesting.

CopyActiveValue Copies an annotatedValue and its wrapped **ActiveValue**.

8.3 ACTIVE VALUE METHODS

8.3 Active Value Methods

Methods defined in the class **ActiveValue** describe how active values work.

8.3.1 Adding and Deleting Active Values

This section describes the methods to install, delete, and replace active values.

Name	Type	Description
AddActiveValue	Method	Makes a variable or property an active value.
WrappingPrecedence	Method	Returns a value which determines how to nest the active value.
DeleteActiveValue	Method	Deletes an active value.
ReplaceActiveValue	Method	Replaces an active value.

(← *self* **AddActiveValue** *containingObj varName propName type annotatedValue*) [Method of ActiveValue]

Purpose: Accomplishes two tasks fundamental in making a variable or property an active value. First, the **ActiveValue** is wrapped inside an annotatedValue. Second, the *annotatedValue* is placed as the value of the variable.

Behavior: **AddActiveValue** associates the annotatedValue with the variable specified by the arguments. If the argument *annotatedValue* is not specified or is NIL, a new annotatedValue is created containing the **ActiveValue** *self*. When the current value of the variable is already an annotatedValue, the **WrappingPrecedence** message determines if it should be nested in the current one or wrapped around it.

Arguments:

self **ActiveValue** instance.

containingObj
The instance or class that contains the variable where the **ActiveValue** is to be added.

varName In the *containingObj* the variable to be made into an **ActiveValue**.

propName Name of an instance variable or class variable property to be made into an **ActiveValue**. If *propName* is NIL, the **ActiveValue** is associated with the variable itself.

type One of IV, CV, CLASS, METHOD, or NIL.

- A *type* of IV or NIL indicates that *varName* is an instance variable or an instance variable property of *containingObj*.
- A *type* of CV indicates a class variable or class variable property of *containingObj*.
- A *type* of CLASS indicates access to a class object's instance variables and properties.
- A *type* of METHOD indicates access to a method object 's instance variables and properties.

annotatedValue
AnnotatedValue object used to contain this **ActiveValue**. If NIL, a new **annotatedValue** is created.

Returns: *annotatedValue*

Example: Adds the **ActiveValue** instance named **($ ActiveValueInstance)** to the object **($ ExampleLoopsWindowInstance)** in the instance variable **width**.

```
71←(← ($ ActiveValueInstance) AddActiveValue ($
ExampleLoopsWindowInstance) 'width)

#,($AV IndirectVariable (ActiveValueInstance (NCV0.0X:.SD7.KR . 8))
(object #,($ ExampleLoopsWIndowInstance))(varName width))
```

(← *self* **WrappingPrecedence**) [Method of ActiveValue]

Purpose: Returns a value which determines how to nest the **ActiveValue** associated with *self*.

Behavior: Varies according to the value returned.

- T

  The **ActiveValue** associated with *self* goes on the outside of any other active values.

- NIL

  This **ActiveValue** goes on the inside.

  If two **ActiveValues** return either T or NIL, a break occurs.

- Number

  Specifies precedence: **ActiveValues** with larger **WrappingPrecedence** values go outside ones with smaller **WrappingPrecedence** values.

CAUTION

It is potentially dangerous to have more than one class with a T or NIL precedence.

ActiveValues that have the instance variable **localState** nest in the following way. When a new **ActiveValue** is added to an existing one with equal **WrappingPrecedence**, the original **ActiveValue** is held in the **localState** of the new one. **ActiveValues** not having an instance variable **localState** must nest inside of ones that do.

To set the **WrappingPrecedence** for a user specialization of **ActiveValue**, specialize this method to return the proper value.

Arguments: *self* **ActiveValue** instance.

Returns: The default method defined in the class **ActiveValue** returns 100. **WrappingPrecedence** for the class **NotSetValue** returns NIL. **WrappingPrecedence** for **IndirectVariable** returns 50.

(← *self* **DeleteActiveValue** *containingObj varName propName type*) [Method of ActiveValue]

Purpose: Deletes an **ActiveValue** from *containingObj*.

Behavior: If the variable defined by the arguments is an **ActiveValue**, it is deleted. If it contains a nested **ActiveValue**, the one matching *self* is deleted; otherwise, nothing happens. No **ActiveValue** messages are triggered. If the deleted **ActiveValue** had a **localState**, it becomes the current value.

Arguments: *self* **ActiveValue** instance.

containingObj
The instance or class that contains the variable where the **ActiveValue** is found.

varName In the *containingObj* the variable that contains the **ActiveValue**.

propName Name of an instance variable or class variable property where the **ActiveValue** resides. If *propName* is NIL, the **ActiveValue** is associated with the variable itself.

type One of IV, CV, CLASS, METHOD, or NIL.

- A *type* of IV or NIL indicates that *varName* is an instance variable or an instance variable property of *containingObj*.
- A *type* of CV indicates a class variable or class variable property of *containingObj*.
- A *type* of CLASS indicates access to a class object's instance variables and properties.
- A *type* of METHOD indicates access to a method object's instance variables and properties.

Returns: The deleted annotatedValue if a match was found, NIL otherwise.

(← *self* **ReplaceActiveValue** *newVal containingObj varName propName type*) [Method of ActiveValue]

Purpose: Replaces an **ActiveValue**.

Behavior: Replaces the **ActiveValue** *self* with *newVal*. The location of the old **ActiveValue** is described by the arguments. No **ActiveValue** messages are triggered.

Arguments: *self* **ActiveValue** instance.

newVal The new value used to replace *self*.

containingObj
The instance or class that contains the variable where the **ActiveValue** is found.

varName In the *containingObj* the variable that holds the **ActiveValue**.

propName Name of an instance variable or class variable property where the **ActiveValue** resides. If *propName* is NIL, the **ActiveValue** is associated with the variable itself.

type *type* is one of IV, CV, or NIL: a *type* of IV or NIL indicates that the variable is an instance variable or an instance variable property of *containingObj*; a *type* of CV indicates a class variable or class variable property of *containingObj*.

Returns: The value of *newVal*.

8.3.2 Fetching and Replacing Wrapped Values

The value of a variable is wrapped in an **ActiveValue**, usually by keeping it in the instance variable **localState**. Specify the behavior of new **ActiveValue** specializations by specializing the methods **GetWrappedValue** and **PutWrappedValue**. For example, **IndirectVariable.GetWrappedValue** just does a **GetValue** on the slot specified by its **object**, **varName**, **propName**, and **type** instance variables. These methods may perform arbitrary work before returning a value, usually that of **localState**. The methods **GetWrappedValueOnly** and **PutWrappedValueOnly** are available for accessing **localState** and bypassing the **ActiveValue** mechanism.

The following table shows the items in this section.

Name	Type	Description
GetWrappedValue	Method	Contains the code to be triggered by a Get- reference to the variable which has been made an **ActiveValue**.
GetWrappedValueOnly	Method	Provides a mechanism to assist in handling nested **ActiveValues**.
PutWrappedValue	Method	Contains the code to be triggered by a Put- reference to the variable which has been made an **ActiveValue**.
PutWrappedValueOnly	Method	Provides a mechanism to assist in handling nested **ActiveValues**.

(← *self* **GetWrappedValue** *containingObj varName propName type*) [Method of ActiveValue]

Purpose: Contains the code to be triggered by a Get- reference to the variable which has been made an **ActiveValue**.

Behavior: Performs arbitrary actions, but when finished, it must return a value which will be returned as the value of the **Get** to the original variable.

This method is fundamental for **ActiveValues**. When **GetValue** or **GetClassValue** finds an annotatedValue in an instance, it does not return that as the value. Instead, it sends the contained **ActiveValue** the **GetWrappedValue** message. This method is not usually called explicitly by users, but is triggered when the **GetValue** function retrieves the value of a variable that contains an **ActiveValue**. It should be specialized when a new subclass of **ActiveValue** is defined.

Arguments:

self **ActiveValue** instance.

containingObj
The instance or class that contains the variable where the **ActiveValue** is found.

varName In the *containingObj* the variable that holds the **ActiveValue**.

propName Name of an instance variable or class variable property where the **ActiveValue** resides. If *propName* is NIL, the **ActiveValue** is associated with the variable itself.

type One of IV, CV, or NIL: a *type* of IV or NIL indicates that the variable is an instance variable or an instance variable property of *containingObj*; a *type* of CV indicates a class variable or class variable property of *containingObj*.

Returns: The value returned from the actions performed by **GetWrappedValue** message.

(← *self* **GetWrappedValueOnly**) [Method of ActiveValue]

Purpose: Enables the **ActiveValue** mechanism to deal with different problems of nested **ActiveValues**. You will generally not need to specialize this method, as most uses of **ActiveValues** will specialize a subclass of **ActiveValue**.

Behavior: Specializations of the class **ActiveValue** may need to specialize this method. (**LocalStateActiveValue**, **IndirectVariable**, and **NotSetValue** all have specialized versions of this method.)

The class **LocalStateActiveValue** specialization simply returns the value of *self*'s **localState** without triggering the active value mechanism.

The class **IndirectVariable** specialization simply returns the value of tracked variable without triggering the active value mechanism.

The class **NotSetValue** specialization simply returns the value of **NotSetValue.**

Arguments: *self* **ActiveValue** instance.

Returns: See Behavior.

(← *self* **PutWrappedValue** *containingObj varName newValue propName type*) [Method of ActiveValue]

Purpose: Contains the code to be triggered by a Put- reference to the variable which has been made an **ActiveValue**.

Behavior: The **PutWrappedValue** message is similar to **GetWrappedValue** except that it is triggered when the local state of the value is to be replaced. When **PutValue** or **PutClassValue** attempts to replace an **ActiveValue**, it instead sends the contained **ActiveValue** the **PutWrappedValue** message.

The default method found in the class **ActiveValue** checks for nested **ActiveValues** by sending the **GetWrappedValueOnly** message to *self*. If the result is an **AnnotatedValue**, **PutWrappedValue** forwards the message on the nested **ActiveValue**; otherwise it sends the message **PutWrappedValueOnly** to *self* and returns the result.

Arguments: *self* **ActiveValue** instance.

containingObj The instance or class that contains the variable where the **ActiveValue** is found.

varName In the *containingObj* the variable that holds the **ActiveValue**.

newValue The value used to replace the **ActiveValue** containing *self*.

propName Name of an instance variable or class variable property where the **ActiveValue** resides. If *propName* is NIL, the **ActiveValue** is associated with the variable itself.

type One of IV, CV, or NIL: a *type* of IV or NIL indicates that the variable is an instance variable or an instance variable property of *containingObj*; a *type* of CV indicates a class variable or class variable property of *containingObj*.

Returns: The value of *newValue*.

(← *self* **PutWrappedValueOnly** *newValue*) [Method of ActiveValue]

Purpose: Enables the **ActiveValue** mechanism to deal with different problems of nested **ActiveValues**. You will generally not need to specialize this method, as most uses of **ActiveValues** will specialize a subclass of **ActiveValue**.

Behavior: Specializations of the class **ActiveValue** may need to specialize this method. (**LocalStateActiveValue**, **IndirectVariable**, and **NotSetValue** all have specialized versions of this method.)

The class **LocalStateActiveValue** specialization simply stores *newValue* into *self*'s **localState** without triggering the active value mechanism.

The class **IndirectVariable** specialization simply stores *newValue* into tracked variable without triggering the active value mechanism.

The class **NotSetValue** specialization causes a break**.**

Arguments:	*self*	**ActiveValue** instance.
	newValue	The new value for **localState**.
Returns:	See Behavior.	

8.3.3 Get and Put Functions Bypassing the ActiveValue Mechanism

ActiveValues normally convert **GetValue**, **GetClassValue**, **PutValue**, and **PutClassValue** accesses into messages which invoke methods to return a value, usually from the **localState** instance variable of the **ActiveValue**. The following functions allow access to class variables and instance variables without triggering any installed **ActiveValue**. See Chapter 2, Instances, for details.

Name	Type	Description
GetValueOnly	Function	Finds the value of an instance variable without triggering active values.
PutValueOnly	Function	Writes the value of an instance variable without triggering active values.
GetClassValueOnly	Function	Returns the value of a class variable; does not trigger active values.
PutClassValueOnly	Function	Changes the value of a class variable and changes the value of a class variable. The change occurs within the class and therefore causes all instances to access the new value of the variable. Does not trigger active values.

8.3.4 Shared ActiveValues in Variable Inheritance

When a **LocalStateActiveValue** is used as the default value for an instance variable in a class, it must be copied into each instance or else all of the instances try to share a single **localState**. This copying is done automatically by LOOPS when the instance variable is first accessed, which means that all instances will share the same ActiveValue until that first access. Copying an **ActiveValue** implies creating a new annotatedValue, so it must be done with the specialized method **CopyActiveValue**.

ActiveValues with no local state may be shared by several variables. In the most extreme case, one instance of **NotSetValue** is the default for the instance variables of all new instances of all classes.

(← *self* **CopyActiveValue** *annotatedValue*) [Method of ActiveValue]

Purpose: Makes a copy of an **ActiveValue** instance.

Behavior: Copies the *AnnotatedValue* and the wrapped **ActiveValue** handling instance variables as follows:

- Instance variables that contain **AnnotatedValues** are copied using the **CopyActiveValue** method.
- The instance variable **localState** is copied so that each copy has its own unique local state.
- All other instance variables are considered shared, and are not copied.

Arguments: *self* **ActiveValue** instance.

annotatedValue
The annotatedValue that surrounds *self*.

Returns: A new annotatedValue wrapped around a copy of the **ActiveValue** *self*.

8.3.5 Creating Your Own Active Value

This example defines a new kind of active value, a **BlippingActiveValue**, which prints out a "blip" of some kind whenever the variable it wraps is read or written.

First, define the new class as a specialization of **LocalStateActiveValue**, then specialize the **PutWrappedValue** and **GetWrappedValue** methods. This is done with the display editor, so in the example they are printed out via the **PPMethod** method. In each case, a **PRINTOUT** function was added before the call to ←**Super**.

Create an instance of **Window** for a location to install a **BlippingActiveValue** for the example. Line 38 is required to set the value of **height** locally to instance **Window1**; if this is not done, its initial value is the active value **#,NotSetValue**, which would remove any active value as soon as it was accessed.

The last few statements in the example show how read and write accesses to height cause a blip character to be printed before **height** is either read or written, with a "!" character representing a write access triggering **PutWrappedValue**, and a "." character representing a read access triggering **GetWrappedValue**.

```
32←(DefineClass 'BlippingActiveValue '(LocalStateActiveValue]
#,($ BlippingActiveValue)

33←(← ($ BlippingActiveValue) SpecializeMethod 'PutWrappedValue]
BlippingActiveValue.PutWrappedValue

34←(← ($ BlippingActiveValue) SpecializeMethod 'GetWrappedValue]
BlippingActiveValue.GetWrappedValue

35←(← ($ BlippingActiveValue) PPMethod 'PutWrappedValue]

(BlippingActiveValue.PutWrappedValue
  (Method ((BlippingActiveValue PutWrappedValue)
           self containingObj varName newValue propName type)
                    **COMMENT** **COMMENT**
          (PRINTOUT PPDefault "!")
          (←Super self PutWrappedValue containingObj varName newValue
                       propName type)))
(BlippingActiveValue.PutWrappedValue)

36←(← ($ BlippingActiveValue) PPMethod 'GetWrappedValue]

(BlippingActiveValue.GetWrappedValue
  (Method ((BlippingActiveValue GetWrappedValue)
           self containingObj varName propName type)
                    **COMMENT**  **COMMENT**
          (PRINTOUT PPDefault ".")
          (←Super self GetWrappedValue containingObj varName propName
type)))
(BlippingActiveValue.GetWrappedValue)

37←(← ($ Window) New 'Window1]
#,($ Window1)

38←(←@ ($ Window1) height 9876]
```

```
9876

39←(←New ($ BlippingActiveValue) AddActiveValue ($ Window1) 'height]
#,($& BlippingActiveValue (46 . 45056))

40←(←@ ($ Window1) height 300)
!300

41←(@ ($ Window1) height]
.300

42←(FOR I TO 20 SUM (@ ($ Window1) height]
....................6000

43←(←@ ($ Window1) height 123]
!123

44←(FOR I TO 20 DO SUM (←@ ($ Window1) height I]
!!!!!!!!!!!!!!!!!!!!210

45←
```


8.4 ANNOTATED VALUES

8.4 Annotated Values

AnnotatedValue is a LOOPS pseudoclass, and instances of it, called pseudoinstances, are Interlisp data type instances.

The structure of the data type is simple. Each annotatedValue contains one field named annotatedValue. This field contains an **ActiveValue** object. The Interlisp record package macros discussed below let you create and work with instances of the data type annotateValue.

There is also a LOOPS class named **AnnotatedValue**. It is an abstract class so it cannot be instantiated, but paradoxically there are objects which consider it their class. (Actually, it is not paradoxical, but this behavior is implemented at a low level within the LOOPS system.) These are the Lisp data type annotatedValue. In normal use this class can be ignored.

AnnotatedValue [Class]

Purpose: LOOPS class equivalent of Lisp data type annotatedValue.

Behavior: This is a LOOPS class, but not a subclass of **Object**. Its super is the LOOPS class **Tofu**. (See Chapter 4, Metaclasses, for a description of **Tofu**.) **AnnotatedValue** is a LOOPS abstract class, and instances are Interlisp data type instances. LOOPS fields messages sent to the annotatedValue data type instances by using the class definition **AnnotatedValue**.

8.4.1 Explicit Control over Annotated Values

This section describes the macros and methods that allow explicit control over annotated values.

Name	Type	Description
type?	Macro	Performs a type check for an instance of the Lisp data type annotatedValue.

create	Macro	Creates a new instance of the data type annotatedValue.
fetch	Macro	Retrieves the contents of the annotatedValue field of an annotatedValue instance.
replace	Macro	Replaces contents of the annotatedValue field of the annotatedValue instance.
_AV	Macro	Sends a message to the **ActiveValue** object wrapped in an annotatedValue.
MessageNotUnderstood	Method	Forwards messages intended for the wrapped **ActiveValue** to that object.

(**type? annotatedValue** *value*) [Macro]

Purpose: Performs a type check for an instance of the Lisp data type annotatedValue.

Arguments: *value* The value to type check.

Returns: T if value is an instance of the data type annotatedValue, NIL otherwise.

(**create annotatedValue annotatedValue ←** *object*) [Macro]

Purpose: Creates a new instance of the data type annotatedValue**.**

Arguments: *object* An **ActiveValue** object to initialize the field annotatedValue of the new annotatedValue instance. This must be an object that has a method **AVPrintSource** (a method of **ActiveValue**) or this form breaks on evaluation. No type checking of object will be performed by the macro.

Returns: An instance of annotatedValue.

(**fetch annotatedValue of** *value*) [Macro]

Purpose: Retrieves the contents of the annotatedValue field of an annotatedValue instance.

Arguments: *value* An annotatedValue instance.

Returns: Contents of field annotatedValue.

(**replace annotatedValue of** *value* **with** *object*) [Macro]

Purpose: Replaces contents of the annotatedValue field of annotatedValue instance with *object*.

Arguments: *value* An annotatedValue instance.

object **ActiveValue** object to be stored in the field. No type checking is done on *object*.

Returns: If value is not an annotatedValue, generates an error; otherwise the previous contents of the field is returned.

(**_AV** *av selector* **.** *args*) [Macro]

Purpose: Sends a message to the **ActiveValue** object wrapped in an annotatedValue.

Behavior: Equivalent to

(_ (**fetch** annotatedValue **of** *av*) *selector* **.***args*)

Arguments:

av Instance of an annotatedValue.

selector Selector for message to send to the **ActiveValue** object.

args Arguments to be passed when the message is sent.

Returns: Result of message.

(← *self* **MessageNotUnderstood**) [Method of AnnotatedValue]

Purpose: Forwards messages intended for the wrapped **ActiveValue** to that object.

Behavior: Messages sent to an annotatedValue are forwarded to its wrapped **ActiveValue**. Users should not explicitly send this message.

8.4.2 Saving and Restoring Annotated Values

The following are methods of the class **ActiveValue** that handle annotated values.

(← *self* **AVPrintSource**) [Method of ActiveValue]

Purpose: Prints **ActiveValues**.

Behavior: An annotatedValue determines how it prints out by sending the **AVPrintSource** message to its wrapped **ActiveValue**.

The default method in **ActiveValue** returns a list of the form:

```
("#," $AV className avNames(ivName value propName value ...)(ivName ...) ...)
```

which causes the annotatedValue to print out as

```
#,($AV className avNames(ivName value propName value ...)(ivName ...) ...)
```

Arguments:

self **ActiveValue** instance.

className Name of the class of the **ActiveValue**.

avNames List of names of *self*; the last element being the unique identifier (UID) of *self*

The list *(ivName value propName value ...)* describes the state of the instance variables of the **ActiveValue**. Including the UID of the **ActiveValue** in the print form enables recovery of the identity of the **ActiveValue**. This enables different annotatedValues to share the same **ActiveValue**, and maintain this sharing across saving to a file and reloading into Lisp.

Returns: A form suitable for use by the Interlisp function **DEFPRINT**. Result should be a pair of the form (item1 . item2); item1 will be printed using **PRIN1**, and item2 will be printed using **PRIN2** (see *Lisp Release Notes* and the *Interlisp-D Reference Manual* description of **DEFPRINT**).

Example:

```
#,($AV IndirectVariable (HeightFromWidth (NCV0.0X:.SD7.KR
. 8))
(object #.($ SquareWindow)) (varName width) (propName NIL)
(type IV))
```

(**$AV** *className avNames . ivForms*) [NLambda, NoSpread Function]

Purpose: Reconstructs an annotatedValue that has been saved to a file.

Arguments:

className Name of the class of **ActiveValue**.

avNames A list of the LOOPS names of **ActiveValue** instances.

ivForms A list describing the state of the instance variables of the **ActiveValue**.

Returns: A new annotatedValue whose **ActiveValue** is reconstructed from the *avNames* and *ivForms*.

8.5 ACTIVE VALUES IN CLASS STRUCTURES

8.5 Active Values in Class Structures

It is possible to have an active value as the default value of an instance variable or the value of a class variable in a class. For example, the following class has an active value installed in the class variable **dontChange** and one installed in the instance variable **firstRead**.

```
SEdit #,($C test)  Package: INTERLISP
((MetaClass Class Edited%:                        ; Edited  2-Dec-87 12:59
  )
 (Supers Object)
 (ClassVariables
   (dontChange
      #,($AV NoUpdatePermittedAV ((|DAW0.1Y:.H53.]99| . 540)) (localState 100))))
 (InstanceVariables (firstRead #,(Defer (DATE)))) (MethodFns))
```

LocalStateActiveValue active values as default IV values are copied down into the instance when their **localState** is smashed, instead of being shared by all instances; this is different from normal default behavior. It is also possible to create a **LocalStateActiveValue** which inherits its **localState** value by giving it a **localState** value of the value of **NotSetValu**e). These copy the inherited value down from the superclass when the **LocalStateActiveValue** is created; if the value in the superclass is changed after the **LocalStateActiveValue** is created, that change will not be reflected in the **LocalStateActiveValue**. Normally inherited values are always tracked by instances that inherit them.

There are two ways to enter active values into the structure of a class: with the editor or programmatically.

It is possible to create active values by typing in a form such as:

($AV activeValueClassName NIL (ivname value propName value ...)(ivname value propName value ...) ...). None of the arguments are evaluated.

To add an active value through the editor, you can type in the above form, select it, and mutate it with the function **EVAL**.

Programmatically, you can use the functions **PutClassIV**, **PutClassValue**, **PutClassValueOnly**, **AddCIV**, **AddCV**, etc. or different methods, such as **Add**, to modify or add class variables and instance variables.

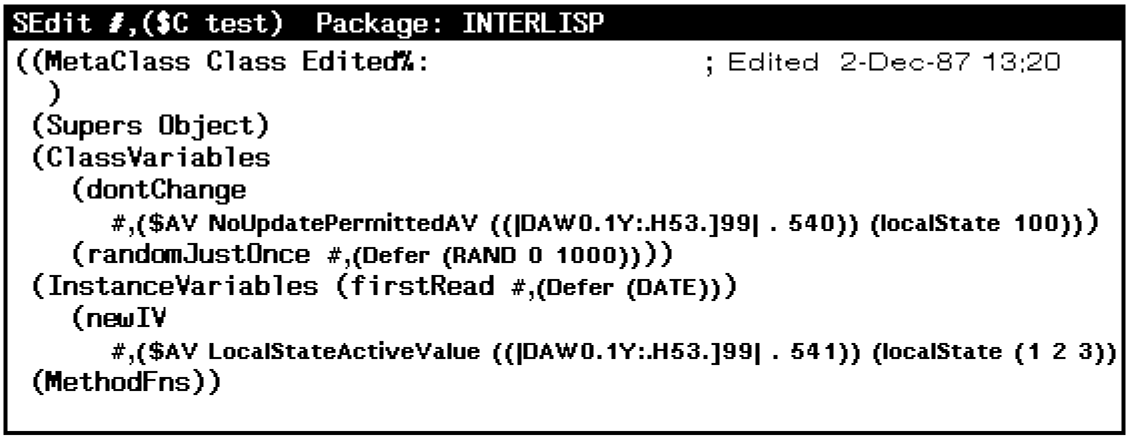
For example, given the above class, **test**:

```
(← ($ test) Add 'CV 'randomJustOnce ($AV FirstFetchAV NIL
(localState (RAND 0 1000))))
```

and

```
(AddCIV  ($ test) 'newIV ($AV LocalStateActiveValue NIL
(localState (1 2 3))))
```

will result in the following:



```
SEdit #,($C test)  Package: INTERLISP
((MetaClass Class Edited%:                    ; Edited  2-Dec-87 13:20
  )
 (Supers Object)
 (ClassVariables
   (dontChange
      #,($AV NoUpdatePermittedAV ((|DAW0.1Y:.H53.]99| . 540)) (localState 100)))
   (randomJustOnce #,(Defer (RAND 0 1000))))
 (InstanceVariables (firstRead #,(Defer (DATE)))
   (newIV
      #,($AV LocalStateActiveValue ((|DAW0.1Y:.H53.]99| . 541)) (localState (1 2 3))
 (MethodFns))
```

An even more general programmatic method that more easily allows customization of an active value uses the annotatedValue data type explicitly. First, you must create an instance of an **ActiveValue** class.

```
(← ($ MyActiveValue) New 'MyAV1)
```

Then the contents of the instance **MyAV1** are initialized. Finally, it is added as the value of a variable in a class structure.

```
(AddCIV  ($ test) 'myNewIV (create annotatedValue
annotatedValue ← ($ MyAV1)))
```

[This page intentionally left blank]

9. DATA TYPE PREDICATES AND ITERATIVE OPERATORS

This chapter describes data type predicates and an operator for iterative statements.

9.1 Data Type Predicates

Data type predicates test the Lisp data type of some datum. For example, some predicates test whether a datum is an object, instance, or class.

LOOPS defines three Lisp data types: annotatedValue, class, and instance. LOOPS provides predicates that enable testing aspects of these types.

Name	Type	Description
Object?	Macro	Determines if a particular datum is a LOOPS object.
Class?	Macro	Determines if a particular datum is a class.
Instance?	Macro	Determines if a particular datum is an instance of a class.
AnnotatedValue?	Macro	Determines if a particular datum is an instance of the annotatedValue Lisp data type.
Understands	Method	Determines if an object will respond to a message.

To determine if a particular datum has an instance variable, class variable, or a property, use **HasIV, HasIV!,** or **HasCV** (see Chapter 2, Instances, and Chapter 3, Classes). To determine if a particular datum is an instance of a class or its superclasses, use **InstOf** or **InstOf!** (see Chapter 2, Instances).

(**Object?** *X*) [Macro]

Purpose/Behavior: Determines if *X* is a LOOPS object. **Object?** returns T for both instances and classes.

Arguments: *X* Possible object.

Returns: Returns T if a name is a pointer to a LOOPS object, and returns NIL otherwise.

Example: This example demonstrates the use of **Object?**.

```
3←(← ($ Window) New 'Window1)
#,($& Window (|OZW0.1Y:.;h.Qm:| . 495))

4←(Object? ($ Window1))
T
```

```
5←(Object? ($ Window))
T

6←(Object? ($ NotAnObject))
NIL

7←(Object? 'NotAnObject)
NIL
```

(**Class?** *X*) [Macro]

Purpose/Behavior: Determines if *X* is a class.

Arguments: *X* Possible class.

Returns: Returns T if *X* is a class; returns NIL otherwise.

Example: This example demonstrates the use of the predicate **Class?** Since **($ Window)** is a class, the function returns T. Since **Window1** and **NotClass** are not class names, NIL is returned. `(Class? X)` is equivalent to `(type? class X)`.

```
8←(Class? ($ Window))
T

9←(Class? ($ NotClass))
NIL

10←(Class? ($ Window1))
NIL
```

(**Instance?** *X*) [Macro]

Purpose/Behavior: Determines if *X* is an instance of some class.

Arguments: *X* Possible instance.

Returns: Returns T if *X* is an instance; returns NIL otherwise.

Example: This example shows the use of **Instance?** `(Instance? X)` is equivalent to `(type? instance X)`.

```
11←(Instance? ($ Window1))
T

12←(Instance? 'Unbound)
NIL

13←(Instance? ($ Window))
NIL
```

(**AnnotatedValue?** *X*) [Macro]

Purpose/Behavior: Determines if *X* is an instance of the annotatedValue data type. For a complete explanation of annotated values, see Chapter 8, Active Values.

Arguments: *X* Possible annotatedValue.

Returns: Returns T if *X* is an annotated value; returns NIL otherwise.

Example: Instances of class Window are created with an active value in the window instance variable. **AnnotatedValue** returns T for the annotatedValue which "wraps" an active value, not for the active value itself.

```
100←(← ($ Window) New 'Window3]
#,($& Window (|OZW0.1Y:.;h.Qm:| . 495))

1←(GetValue ($ Window3) 'window)
{WINDOW}#51,140000

2←(GetValueOnly ($ Window3) 'window)
#,($AV LispWindowAV ((|OZW0.1Y:.;h.Qm:| . 495))
(localState {WINDOW}#51,140000))

3←(AnnotatedValue? (GetValueOnly ($ Window3) 'window))
T

4←(AnnotatedValue? (GetValue ($ Window3) 'window))
NIL

5←(AnnotatedValue? (_ ($ LispWindowAV) New 'LWAV4]
NIL
```

(← *self* **Understands** *selector*) [Method of Object]

Purpose/Behavior: Determines if the object *self* will respond to a message with *selector.*

Arguments: *self* Instance or class in question.

selector Selector in question.

Returns: T if *self* is a class or an instance of a class that understands message selector; NIL otherwise.

Note: If *self* is not a LOOPS object, you get NIL and not an error.

Categories: Object

Example: Given that **Window** is a class, **MyWindow** is an instance, and **SpinAround** is a method of **MyWindow**, **Window** returns NIL, and **MyWindow** returns T. Since **Shape** is a method of **Window**, this also returns T.

```
90←(← ($ Window) Understands 'SpinAround)
NIL

91←(← ($ MyWindow) Understands 'SpinAround)
T

91←(← ($ MyWindow) Understands 'Shape)
T
```


9.2 ITERATIVE OPERATORS

9.2 Iterative Operators

LOOPS provides an iterative operator to be used with Interlisp-D iterative statements.

in-supers-of *X* [Iterative Statement Operator]

Purpose / Behavior: Allows iteration up the supers chain of the object *X*. Used in an Interlisp-D iterative statement. (See the *Interlisp-D Reference Manual* for more information on iterative statements.)

Arguments: *X* A LOOPS class or an instance.

Example: This example shows one way to use this operator.

```
55←(FOR I in-supers-of ($ ClassBrowser) DO (PRINT (← I ClassName]
ClassBrowser
IndexedObject
LatticeBrowser
Window
Object
Tofu
NIL
```

[This page intentionally left blank]

10. BROWSERS

As described in Chapter 1, Introduction, one of the key components in the LOOPS system is inheritance, in which structures have well-defined relationships to other structures. Class inheritance is a typical example of this relationship.

Since inheritance can be described by a two-dimensional graph, it is natural to create a user interface for LOOPS built on the Lisp Library Module, Grapher. This user interface is called a browser. Browsers are tools to assist in the development cycle of a product or vehicles for building user interfaces within a final product.

Much development time is spent building, examining, and modifying the relationships between classes. These tasks include specifying the contents of various classes: class variables, instance variables, properties, and methods. The location of the class within the inheritance structure must also be determined. After a number of classes have been built, the relationships between the classes may need to be reviewed. Often, the initial design is flawed and requires the following changes:

- Moving parts of one class to another class.
- Adding, substracting, or changing data or functionality within classes.
- Adding new classes, or merging different classes.

Browsers are the facility within LOOPS which support these types of operations. This chapter describes how to use browsers both interactively with the mouse, and programmatically.

Browsers are most commonly used on the classes defined for an application. Many of the examples here browse objects which LOOPS uses internally; the functionality is exactly the same.

1

0.1 Types of Built-in Browsers

A number of different types of browsers are already built into LOOPS:

- Lattice browsers
- Class browsers
- File browsers
- Supers browsers
- Metaclass browsers

This section describes these browsers in detail.

10.1.1 Lattice Browsers

The most general class is called **LatticeBrowser**. Figure 10-1 shows a class inheritance lattice with the subclasses of **LatticeBrowser.**

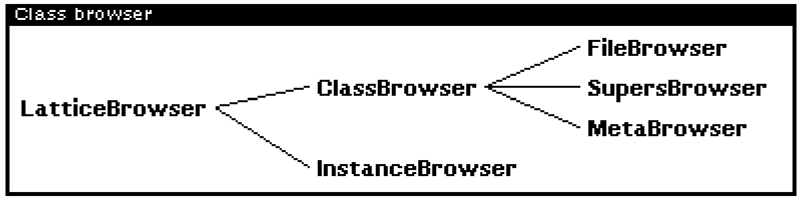


Figure 10-1. Sample Lattice Browser

10.1.2 Class Browsers

A class browser shows the linkages between a class or classes and their subclasses. Super classes are shown on the left (or top) side of the browser window. Subclasses of these are connected by links moving to the right (or down). An example of a class browser is shown in the previous section. The class **LatticeBrowser** is the root object of this example. Subclasses of **LatticeBrowser** are **ClassBrowser** and **InstanceBrowser**. Subclasses of **ClassBrowser** are **FileBrowser**, **SupersBrowser**, and **MetaBrowser**.

10.1.3 File Browsers

A file browser is a class browser containing all classes defined within a file. Additionally, file browsers contain a menu interface to common operations on files.

10.1.4 Supers Browers

A supers browser is an inverted class browser. A class browser is built by following subclass links from a class. A supers browser is built by following superclass links from a class. An example of a supers browser is shown in Figure 10-2.

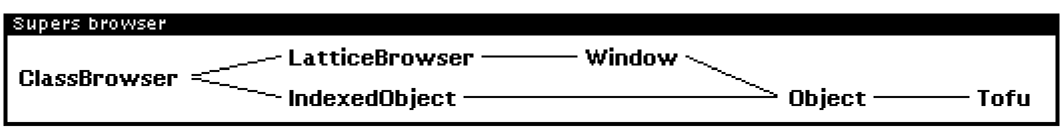


Figure 10-2. Sample Supers Browser

10.1.5 Metaclass Browsers

A metaclass browser is like a supers browsers, but is built by following metaclass links. Figure 10-3 shows two root classes: **ActiveValue** and **ClassBrowser.**

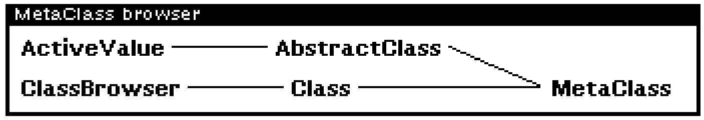


Figure 10-3. Sample Metaclass Browser

10.1.6 Instance Browsers

An instance browser shows the relationships between instances. These relationships may be more dynamic than the inheritance relationships shown by the class browsers. Typically, the relationships are not defined until runtime, and may be changed often. By specializing the instance browser, you can show several relationships between a fixed set of objects.

10.2 OPENING BROWSERS

10.2 Opening Browsers

A browser can be opened in several ways:

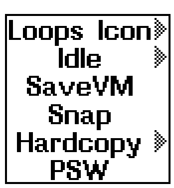
- Selecting a menu option from the Background Menu.
- Selecting a menu option from the LOOPS icon.
- Sending a **Browse** message to an instance of a browser.
- Calling either of the functions **Browse** or **FileBrowse**.

10.2.1 Using Menu Options to Open Browsers

Since browsers are an important part of LOOPS, you can use menus in several ways to create standard browsers. Once opened, via menu or program, any browser can be operated from both the appropriate menus and programmatic commands.

10.2.1.1 Overview of Background Menu and LOOPS Icon

When LOOPS is loaded, the option **Loops Icon** is added to the background menu, as shown in this window:



The **Loops Icon** option has two suboptions:

- **Browse Class**
- **Browse File**

These suboptions are shown in the following windows:

Selecting **Loops Icon** puts a LOOPS icon on the screen; you are prompted to place the icon after it is created. The other commands are discussed in Section 10.2.1.2, "Command Summary."

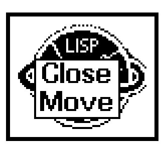
The LOOPS icon, which appears in Figure 10-4, is a prototype instance of the class **LoopsIcon**. It is provided to give you another convenient menu interface to typical programming operations.

Figure 10-4. LOOPS Icon

Pressing the left button while the mouse is on the icon causes the following menu to appear with options appropriate for class browsers:

Pressing the middle button while the mouse is on the icon causes the following menu to appear with choices appropriate for file browsers:

Pressing the right button while the mouse is on the icon causes the following menu to appear with two options for operations on the icon itself:



Close removes the icon from the screen. It can be restored at any time from the background menu. **Move** lets the icon be moved to another location on the screen, just as any icon is moved in Lisp.

10.2.1.2 Command Summary

The background menu and the LOOPS icon provide the same functionalities. This section describes the commands available.

Browse Class, Browse Supers

Selecting either of these options causes the following prompt to appear in the prompt window.

```
Please tell me the name of the root object >
```

Enter the name of a class without using the "$" notation. The system builds the appropriate type of browser and prompts you to move the window containing the browser.

Browse File

Selecting this option causes the following menu to appear:

The top option on this menu is ***newFile*** which has three suboptions; the remaining options are the names of the files that are on **FILELST** (Lisp remembers what files are loaded and how they were loaded; **FILELST** files were loaded normally. See the *Interlisp-D Reference Manual* for a full explanation). Selecting one of the filenames will open a file browser on that file.

newFile Prompts you in the prompt window with the following prompt :

```
Please type in file name: >.
```

Enter the name of a file to create. The system checks to determine if a filecoms exists for that file name. If one exists, the system asks for confirmation before destroying the value of that filecoms and opening up an empty browser window. If no filecoms exists for that filename, an empty file browser window is opened.

loadFile Prompts you with:

```
Please type in file name to load: >
```

The system loads that file and opens a browser on it.

hiddenFile Causes a menu to appear with files that have been loaded but not **SYSLOAD**ed and are not on **FILELST**; that is, the files are on **LOADEDFILELST**, but not on **FILELST**. The LOOPS files, for example, the .LCOMs that add LOOPS to Lisp, are on this list.

Edit Filecoms

Selecting this option brings up the same menu as the option **Browse File**. Instead of opening a browser on the file, a display editor window is opened on the filecoms of that file. If ***newFile*** is selected, you are prompted to enter a file name and an SEdit window is opened with a template containing the File Manager commands **CLASSES, METHODS, FNS, VARS**, and **INSTANCES**.

CleanUp File

Selecting this option first calls **FILES?** and then builds a menu of files in **FILELST** that have changed. From this menu, select a file to be cleaned up; this calls **CLEANUP**.

10.2.2 Using Commands to Open Browsers

You can use the following methods and functions for opening browsers programmatically and from the Lisp Executive window.

Name	Type	Description
Browse	Method	Opens a browser showing the relationships between classes.

BrowseFile	Method	Opens a browser showing the relationships between classes on a file.
Browse	Function	Provides a short way to create a class browser.
FileBrowse	Function	Provides a short way to create a file browser.

(← *self* **Browse** *browseList windowOrTitle goodList position*) [Method of LatticeBrowser]

Purpose/Behavior: Opens a browser showing inheritance relationships between classes.

Arguments:

self — An instance of **ClassBrowser**, **MetaBrowser**, or **SupersBrowser.**

browseList — A LOOPS class name, a LOOPS pointer to a class name, or a list of those.

windowOrTitle — A title to appear on the browser or a window to use (but which will be reshaped to fit the browser.) Title defaults to "Class browser."

goodList — A **goodList** other than the *browselist*. (See Section 10.5.1, "Instance Variables of Class LatticeBrowser," for more information on **goodList**.)

position — Lower left corner of browser. If NIL, position the window with the mouse.

Returns: Pointer to the browser object.

Examples: The following command opens a class browser on **Window**.

```
(←New ($ ClassBrowser) Browse 'Window)
```

The following command opens a supers browser on **InstanceBrowser** and **ClassBrowser**.

```
(←New ($ SupersBrowser) Browse (LIST 'InstanceBrowser ($ ClassBrowser)))
```

(← *self* **BrowseFile** *fileName*) [Method of FileBrowser]

Purpose: Opens a browser showing relationships between classes on a file.

Behavior: Classes defined within *fileName* are displayed within the browser. If *fileName* is NIL, a menu of files on **FILELST** opens. The selected file has a file browser opened on it.

Arguments:

self — An instance of the class **FileBrowser**

fileName — File to browse; should not be a list.

Returns: *self*

Categories: FileBrowser

Example: The following command opens a file browser on the file **LoopsWindow**.

```
(←New ($ FileBrowser) BrowseFile 'LoopsWindow)
```

(**Browse** *classes title goodClasses position*) [Function]

Purpose: Provides a short way to create a class browser.

Behavior: Sends a **Browse** message to a new instance of a **ClassBrowser** passing *classes*, *title*, *goodClasses*, and *position* as arguments. If *goodClasses* is T, it is rebound to the value of *classes* before the message is sent.

Arguments:

classes A LOOPS class name, a LOOPS pointer to a class name, or a list of those.

title A title to appear on the browser. Title defaults to "Class browser."

goodClassses
A **goodList** other than *classes*. (See Section 10.5.1, "Instance Variables of Class LatticeBrowser," for more information on **goodList**.)

position Lower left corner of browser. If NIL, position the window with the mouse.

Returns: A new instance of **ClassBrowser**.

Example: The following command creates a class browser on the class **ActiveValue** and all its subclasses.

```
11←(Browse 'ActiveValue)
```

(**FileBrowse** *filename*) [Function]

Purpose: Provides a short way to create a file browser.

Behavior: Sends a **BrowseFile** message to a new instance of a **FileBrowser** passing *filename* as the argument.

Arguments: *filename* File to browse.

Returns: New instance of **FileBrowser**.

Example: The following command creates a file browser on the file **LoopsWindow**.

```
12←(FileBrowse 'LoopsWindow)
```


10.3 USING CLASS BROWSERS, META BROWSERS, AND SUPERS BROWSERS

10.3 Using Class Browsers, Meta Browsers, and Supers Browsers

Instances of **ClassBrowser, SupersBrowser, MetaBrowser** all have the same menu interface. This section shows examples of the various menus followed by descriptions of the actions performed after selecting particular options.

Three pop-up menus are associated with browsers:

- One menu appears by positioning the mouse on the title bar of the browser window and pressing either the left or the middle mouse button. This menu contains options that control the appearance of the browser.

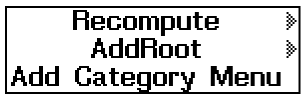
- A second menu appears by positioning the mouse on one of the nodes in a browser and pressing the left mouse button. This menu contains informational options.

- A third menu appears by positioning the mouse on one of the nodes in a browser and pressing the middle mouse button. This menu contains editing options.

These menus differ depending on the browser type. The following sections describe the menus associated with class browsers, supers browsers, and metaclass (or, more simple, meta) browsers. Sections then describe the additional functionality associated with file browser menus.

10.3.1 Selecting Options in the Title Bar Menu

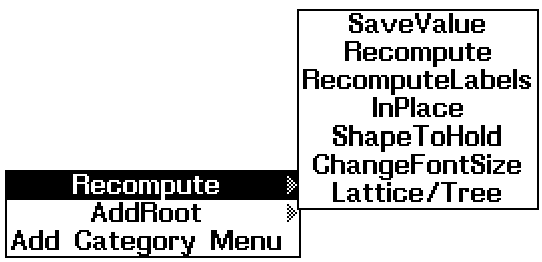
The following menu appears when you position the mouse on the title bar of the browser menu and press either the left or the middle mouse button:



This section describes each menu option.

10.3.1.1 Recompute and its Suboptions

Selecting the **Recompute** option and dragging the mouse to the right causes the following submenu to appear:



Most of the **Recompute** suboptions change the appearance of a browser but not its contents. For example, **SaveValue** provides a pointer to the browser without changing anything in it.

SaveValue The browser instance is stored in the instance variable **savedValue** of the prototype instance of **LoopsIcon** and in the value of **IT** (see the *Interlisp-D Reference Manual*). This value is returned from the function call **SavedValue**.

Recompute Recomputes the entire browser structure from the starting objects. It does not recompute the labels for each item if those labels have been cached in the property **objectLabels** of the instance variable menus.

RecomputeLabels Recomputes the entire browser structure from the starting objects and recomputes the labels for each item.

InPlace Recomputes the browser without affecting the scrolled location of the lattice within the window. This may be necessary for a browser containing a large lattice structure.

ShapeToHold Makes the window for the browser just large enough to hold all of the nodes in the browser, up to a maximum size. Browser windows may also be changed interactively or programmatically with **SHAPEW**.

ChangeFontSize Causes a menu to appear containing 8, 10, 12, and 16. Selecting one changes the font size used to display the nodes to that value. The font family is

```
(@ self browseFont:,FontFamily)
```

The font face is

```
(@ self browseFont:,FontFace)
```

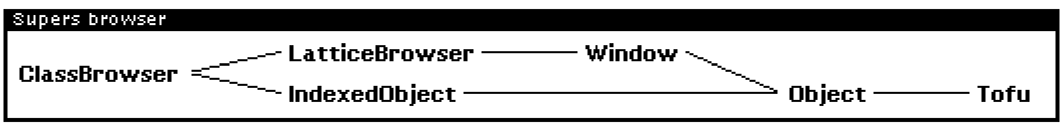
Note: An alternative way to change the font of a browser is to enter:

```
[PROGN (←@ ($ InstanceOfBrowser) browseFont
(FONTCREATE .....))(← ($ InstanceOfBrowser)
RecomputeLabels)]
```

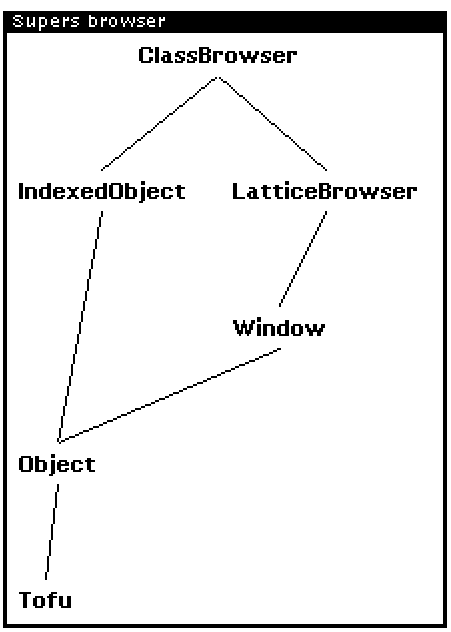
Lattice/Tree Causes the following menu to appear:

Using the example of a supers browser for the class **ClassBrowser**, this browser is drawn for each of the formatting options. A tree does not show branches recombining; a lattice does. A boxed node in a tree indicates the node shows up in more than one location in a tree. When a browser is constructed by the system the default formatting style is HORIZONTAL/LATTICE.

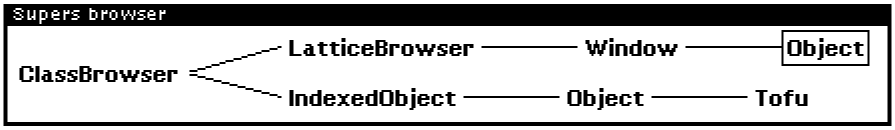
- HORIZONTAL/LATTICE



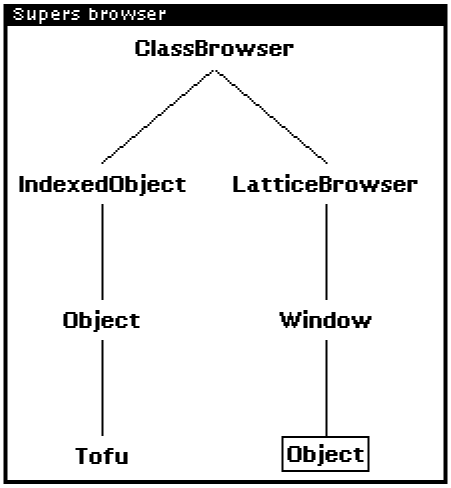
- VERTICAL/LATTICE



- HORIZONTAL/TREE



- VERTICAL/TREE



10.3.1.2 AddRoot and its Suboptions

The **AddRoot** options add items or subtrees to the browser.

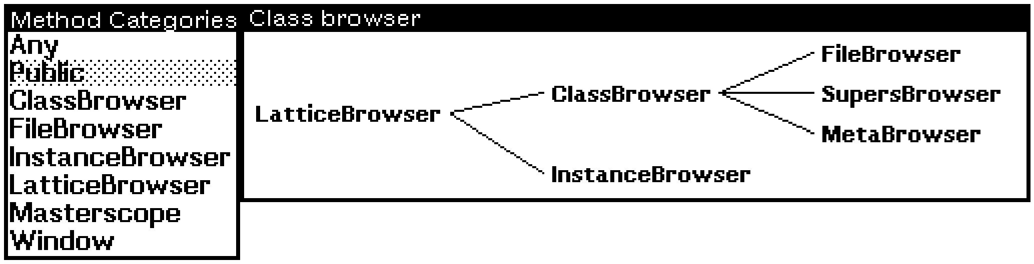
Selecting the **AddRoot** option and dragging the mouse to the right causes the following submenu to appear:

AddRoot A prompt appears in an attached window to enter the name of a class to be added to the browser. If the entered item is not an object, a message that nothing was added to the browser is printed. If the entered item is already in the browser, nothing occurs. If the entered name does not correspond to a class, nothing occurs.

RemoveFromBadList Objects within a browser can be put on the instance variable **badList**. This can be done by positioning the mouse on the node in a browser, pressing the left mouse button, and selecting an option from the menu that appears. Items on the **badList** are not displayed in the browser. If you select the option **RemoveFromBadList,** a menu appears showing any objects on the **badList**. Selecting one of those objects removes it from the **badList** and causes it to be redisplayed in the browser.

10.3.1.3 Add Category Menu

The system searches all methods in all classes shown in the browser and computes the categories for these. These categories are made into a sorted menu with the categories **Any** and **Public** included at the top. This menu is attached to the left side of the browser. Selecting options in this menu acts as a toggle, either highlighting them or returning them to their normal display.

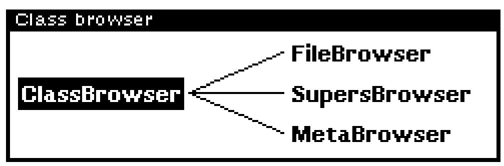


Selected options are stored on the browser instance variable **viewingCategories.** Options on this menu interact with the browser interface for editing methods as described in Section 10.3.2, "Selecting Options in the Left Menu."

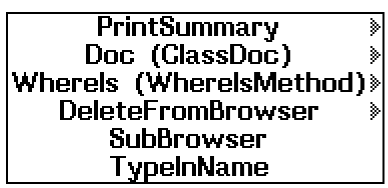
Note: Very often when using a browser, you ask to see what items a class inherits from classes above it in the inheritance lattice. To keep this inherited information more manageable, information inherited from the classes **Tofu**, **Object**, and **Class** are filtered out from the information presented to you. As an example, see the description of **PP** in the following section.

10.3.2 Selecting Options in the Left Menu

When the mouse is inside a browser and you hold down the left mouse button, nodes within the browser become inverted when the cursor moves over them, as shown in the following window:



If you release the left mouse button while the cursor is over a node, the following menu appears:



The options shown on the menu operate on the node (class) selected. Several of these options have associated submenus. Common options are in the main menu, and less common ones are menu suboptions. The actions that occur as a result of selecting one of these options are described in the following subsections. An additional subsection describes extended functionality available with the left mouse button.

10.3.2.1 PrintSummary and its Suboptions

PrintSummary provides a quick way to see object and method definitions. For all printing that occurs as a result of selecting this option or one of its suboptions, the output is sent to the value of the variable **PPDefault**, which is by default the Common Lisp Executive Window.

Selecting the **PrintSummary** option and dragging the mouse to the right causes the following submenu to appear:

```
                          PrintCategories
                                PP
                               PP!
                               PPV!
                             PPMethod
                          MethodSummary
      PrintSummary       > PrintSummary
      Doc (ClassDoc)     >
Wherels (WherelsMethod)  >
    DeleteFromBrowser    >
       SubBrowser
       TypeInName
```

PrintCategories Prints the categories and associated methods for the selected class, as shown in the following window:

```
   #,($ DestroyedObject)

Object
     Destroy!
```

PP Produces a standard **PrettyPrint** of class, as shown in the following window:

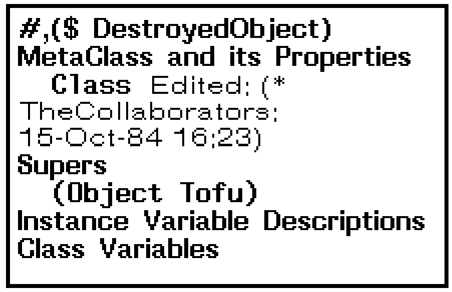
```
(DEFCLASS DestroyedObject
   (MetaClass Class Edited%:
  (* --) )
   (Supers Object))
```

PP! Produces a formatted **Print** of class, as shown in the following window:

```
#,($ DestroyedObject)
MetaClass and its Properties
   Class Edited: (*
TheCollaborators:
15-Oct-84 16:23)
Supers
   (Object Tofu)
Instance Variable Descriptions
Class Variables
Methods
  Destroy! DestroyedObject.Destroy!
 doc NIL args NIL
```

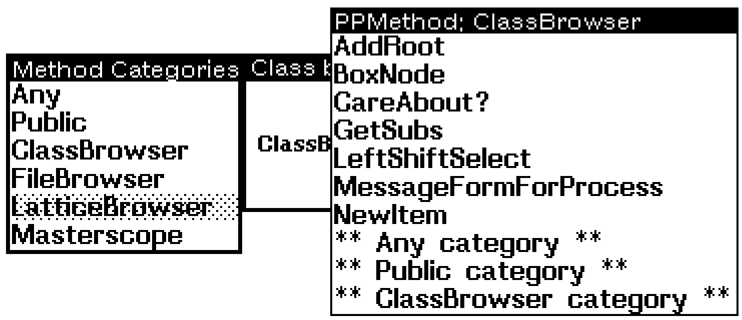
Information that is defined locally within the class is printed in the bold font. Inherited information is printed in the regular font. Inherited information from the classes **Object** and **Tofu** is not printed.

PPV! Same as **PP!**, but does not include **Methods**, as shown in the following window:

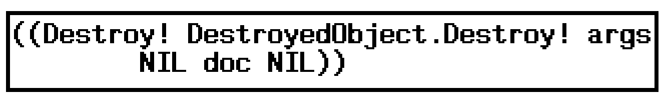


PPMethod Brings up a menu of the methods for this class followed by a list of the known categories. The menu is influenced by the shaded options on the **Method Categories** menu (see Section 10.3.1.3, "Add Category Menu"), whether or not it is opened. Selecting a category will include any methods under that category in the menu. After selecting one of the methods, the Lisp function for that method is prettyprinted.

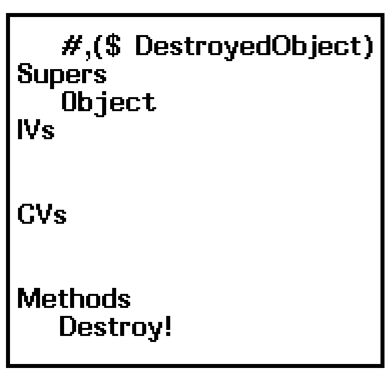
For example, selecting **PPMethod** from the node **ClassBrowser** with the **Method Categories** menu as shown results in the following menu of methods:



MethodSummary Prints a summary of the methods defined for the class, as shown in the following window:



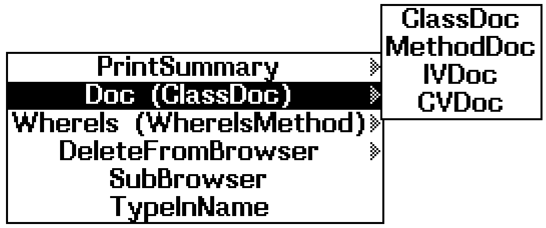
PrintSummary Similar to **PP**, but default values or properties associated with variables and methods are not printed, as shown in the following window:



10.3.2.2 Doc (ClassDoc) and its Suboptions

Each part of an object's definition has a **doc** property containing strings. This menu option is a quick way to see the string for a specific part of a definition. For all printing that occurs as a result of selecting this option or one of its suboptions, the output is sent to the value of the variable **PPDefault**, which is by default the Common Lisp Executive Window.

Selecting the **Doc (ClassDoc)** option and dragging the mouse to the right causes the following submenu to appear:

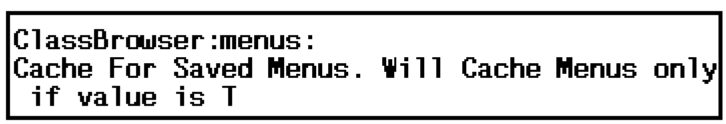


ClassDoc Prints documentation for class, that is, the **doc** property.

MethodDoc Causes a menu to appear in the same manner as **PPMethod** described in Section 10.3.2.1, "PrintSummary and its Suboptions." After selecting a method, information about that method is printed, as shown in the following window:

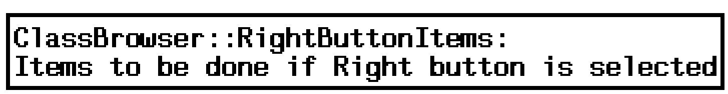
The menu of methods reappears until a selection is made from outside the menu.

IVDoc Causes a menu to appear showing instance variables associated with the class. After selecting one, its documentation is printed, as shown in the following window:



The menu of instance variables reappears until a selection is made from outside the menu.

CVDoc Causes a menu to appear showing class variables associated with the class. After selecting one, its documentation is printed, as shown in the following window:

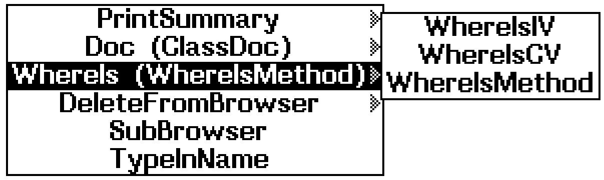


The menu of class variables reappears until no selection is made from the menu.

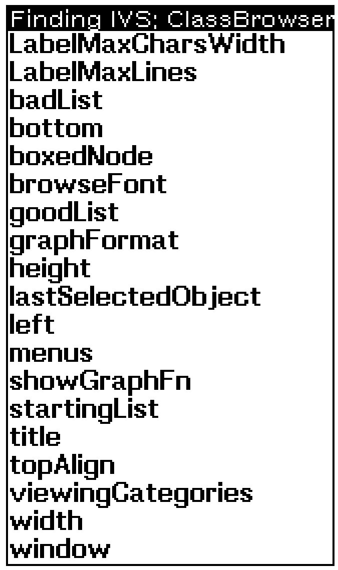
10.3.2.3 WhereIs and its Suboptions

WhereIs describes where each part of the object comes from. For all printing that occurs as a result of selecting this option or one of its suboptions, the output is sent to the value of the variable **PPDefault**, which is by default the Common Lisp Executive Window.

Selecting the **WhereIs** option and dragging the mouse to the right causes the following submenu to appear:

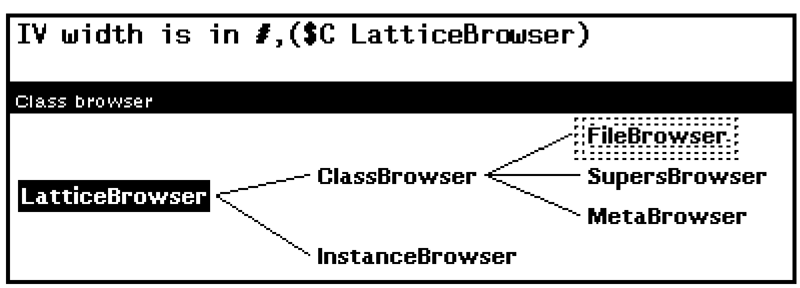


WhereIsIV Causes a menu to appear at the current cursor position, showing the local and inherited instance variables. A sample menu appears here:



When you select an instance variable from this menu, these actions occur, as shown in the following window:

- A search starts with the selected class and then proceeds upwards through its supers. The first class that contains the selected instance variable flashes three times.
- All other classes in the browser that contain, that is, specialize, the instance variable are shaded.
- The name of the topmost class name is printed across the top of the window.

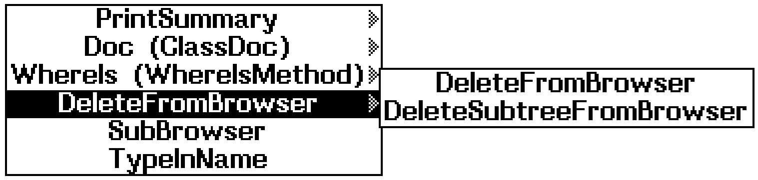


WhereIsCV Same as above, but for class variables.

WhereIsMethod Same as above, but for **Methods**. The menu of methods is not filtered by any category information.

10.3.2.4 DeleteFromBrowser and its Suboptions

Browsers show all of the lattice or tree from the root to the leaves. The **DeleteFromBrowser** option and its suboptions, shown here, allow you to prune the tree.

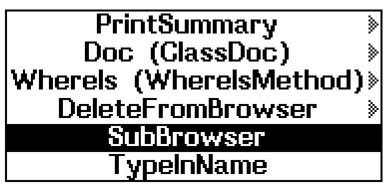


DeleteFromBrowser Deleting a class from a browser adds it to that browser's instance variable **badList**. Items listed in a browser's **badList** are not displayed by the browser. If a class on the **badList** has subclasses, these are made into roots. To redisplay a class once it has been deleted, refer to the command **RemoveFromBadList** in Section 10.3.1.2, "AddRoot and its Suboptions."

DeleteSubtreeFromBrowser Deleting a class with this command places this class on the browser's **badList**. Additionally, any subclasses of the deleted class are also placed on the **badList**. If one uses the command **RemoveFromBadList** to redisplay the deleted class, only that class is redisplayed. Its subclasses remain on the **badList** until they are explicitly removed from it.

10.3.2.5 SubBrowser

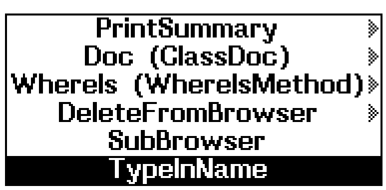
The following window shows the selection of this option:



For class and supers browsers, the SubBrowser option opens a new browser of the same type (for example, a **SubBrowser** of a supers browser is a supers browser) with the class selected becoming the root object of the browser. For file browsers, the **SubBrowser** becomes a class browser.

10.3.2.6 TypeInName

The following window shows the selection of this option:



This option puts the class name in the type-in buffer.

10.3.2.7 Extending Functionality with the Left Mouse Button

Using various keys in conjunction with the left mouse button extends the available options.

- **SHIFT** key

Pressing the **SHIFT** key while selecting a node with the left button causes the name of the class to be typed into the current type-in point. If a node is not selected, but the cursor is in the background of the browser, the entire graph is copied. This can be used to insert browser images into TEdit documents, for example. See the Lisp Library documentation on Grapher for more details.

- Control (**CTRL**) key

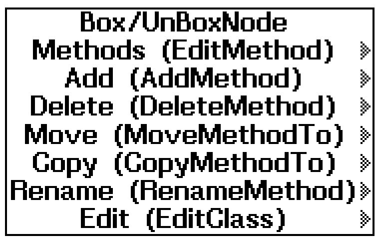
  Pressing the **CTR** key while selecting a node with the left button causes the node to track movement by the cursor. This allows you to temporarily change the layout of the nodes in the graph. The next update of the browser recomputes the node positions.

- **META** key

  Pressing the **META** key while selecting a node with the left button is the same as a **PrintSummary** selection.

10.3.3 Selecting Options in the Middle Menu

When you position the cursor on a browser node and press the middle mouse button, the following menu appears:



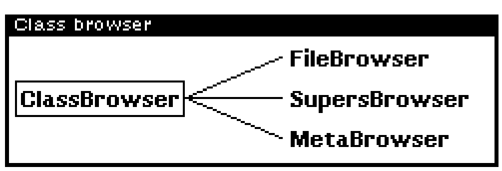
This section describes the actions that occur when you select an option from this menu.

Except for **BoxNode**, all options are followed by a parenthetical suboption. This suboption appears in the option's submenu and performs the same operation as the option itself. For example, selecting **Methods (EditMethod)** performs the same operation as selecting **EditMethod** from its submenu.

The prompt for these options usually appears in a small window at the top of the browser, unless otherwise stated.

10.3.3.1 Box/UnBoxNode

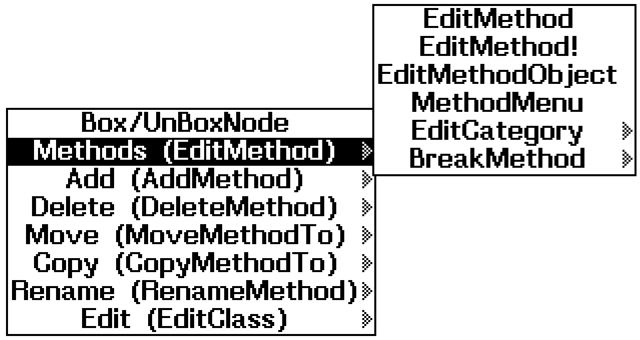
Draws a box around the node, as shown here:



This also selects the node as a target for **Move** and **Copy** options. If a node is already boxed, **Box/UnBoxNode** unboxes it. Only one node in a class browser can be boxed at a time with this menu option.

10.3.3.2 Methods (EditMethod) and its Suboptions

Selecting the **Methods (EditMethod)** option and dragging the mouse to the right causes the following submenu to appear:

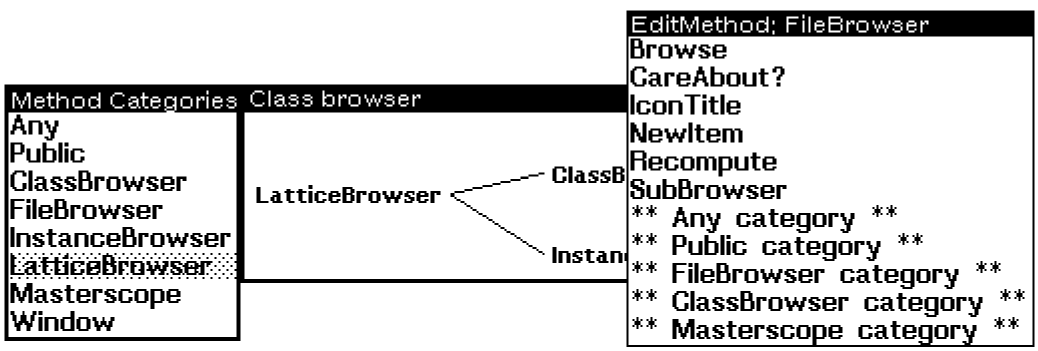


EditMethod Edits a method of class, selected from a menu of local methods. The menu that appears contains some or all of the methods of the class and some category choices. The menu options that appear depend on the shaded options on the **Method Categories** menu (see Section 10.3.1.3, "Add Category Menu").

For example, assume the following actions occur:

- A class browser is opened on the class **LatticeBrowser**.
- A category menu is added and the category **LatticeBrowser** is selected.
- **FileBrowser** under the class **ClassBrowser** is selected.
- The **EditMethod** suboption is selecte.

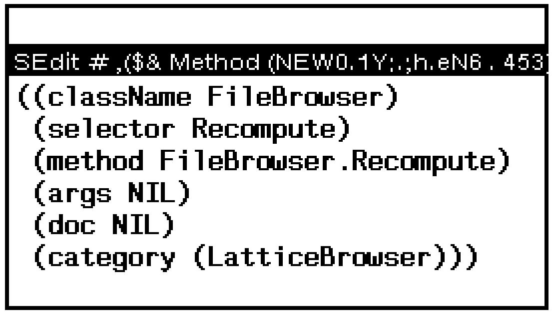
The following menu appears:



If you select one of the method names, an edit window appears containing the source code for the method (assuming the source code was loaded). If you select one of the categories, a similar menu appears showing the changed methods and categories. The method/category menu continues to appear until you select a method or press a mouse button outside of the menu. This operation is provided by the method **PickSelector**.

EditMethod! Edits a method selected from a menu of all the inherited methods, making the method local if necessary.

EditMethodObject Edits the object representing the method, as shown in this display editor window:



This suboption uses the method **PickSelector** to provide the same menu interface for choosing a selector that **EditMethod** uses. Within this edit window, the only items you should change are the **doc** and **category** properties.

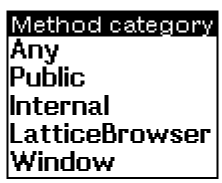
MethodMenu Creates a permanent menu of methods of this class, for example,

After you have placed the menu, pressing the left mouse when the cursor is over a menu option will cause the method to be printed; pressing the middle button will cause the method to be edited.

EditCategory This option has three suboptions:

- **EditCategory**

  Opens a menu of available categories, similar to the following menu example:



  After you select a category, another menu appears containing the methods of that category and a list of categories not chosen (using the method **PickSelector**). When a method is selected, it appears in an edit window.

- **ChangeMethodCategory**

  Using the method **PickSelector**, this prompts you for a method. When one is selected, another menu appears containing the option ***other*** with all of the categories of the selected class.

  If one of the categories is chosen, the category for the method is changed to this value.

  If ***other*** is chosen, this prompts you in the **PROMPTWINDOW** for a symbol or a list of symbols that will become the new category or categories for the method.

  See the method **Class.ChangeMethodCategory**.

- **CategorizeMethods**

  Edits an association list of categories and the local methods they contain via the editor.

  This is an example of the edit window that appears:

```
SEdit  Package: INTERLISP
((Any (CreateClass DestroyInstance New NewWithValues))
 (Public (CreateClass DestroyInstance New NewWithValues))
 (Internal NIL)
 (MetaClass (CreateClass))
 (Class (DestroyInstance New NewWithValues)))
```

BreakMethod This option has three suboptions:

- **BreakMethod**

  Places a break on a method of class, selected from a menu of local methods, by sending the message **BreakMethod**.

- **TraceMethod**

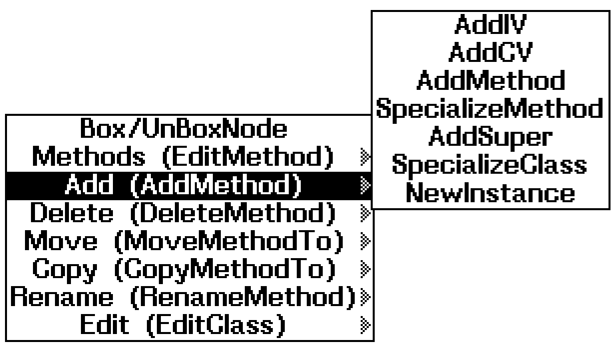
  Places a trace on a method of class, selected from a menu of local methods, by sending the message **TraceMethod**.

- **UnbreakMethod**

  Brings up a menu of methods local to class that have been broken, and removes any breaks or traces on the one selected by sending the message **UnbreakMethod**.

10.3.3.3 Add (AddMethod) and its Suboptions

Selecting the **Add (AddMethod)** option and dragging the mouse to the right causes the following submenu to appear:



AddIV Prompts you for the name of a new instance variable to be added to a class, and opens an editor as shown here:

```
* SEdit DestroyedObject  Package: INTERLISP
(newIV **DefaultValue doc (* IV added by MCGILL))
```

From here, change the default value of the instance variable to the desired value, and change the documentation to add any other properties and values if necessary. When you exit from the editor, the instance variable is added to the class.

AddCV Same as **AddIV**, except that you add a class variable.

AddMethod

Allows you to create and edit a new method for this class. You are prompted to type a selector for the new method. Next, an edit window appears with the following template:

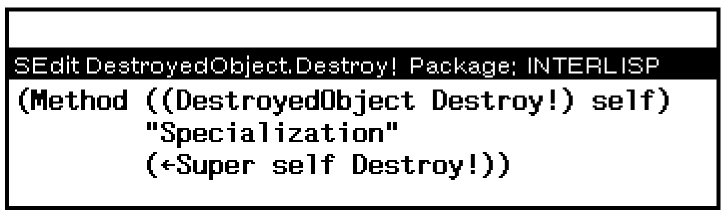
Replace the form (SubclassResponsibility) with the functionality you want.

SubclassResponsibility is a macro that causes a call to **HELPCHECK,** so if you forget to remove it, or want to leave it in for debugging, it will break when the new method is invoked.

SpecializeMethod

Causes a menu to appear, containing the selectors of inherited methods, the option **** Generic Methods ****, and available categories.

If one of the selectors is chosen, a display editor window appears with a template that contains a _**Super** and the same comment as the specialized method. An example of this window appears here:



If **** Generic methods **** is chosen this causes a menu to appear, listing the methods inherited from **Object**, **Class**, or **Tofu**.

AddSuper

Prompts you for a class name to be added to the beginning of the **Supers** list.

SpecializeClass

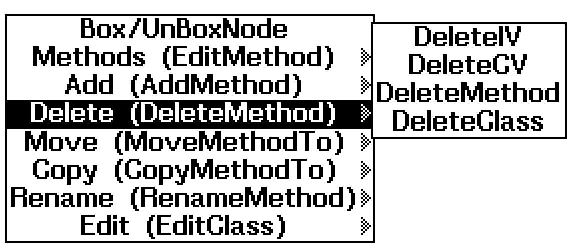
Defines a subclass of this class. This subclass is initialized with no locally defined instance variables, class variables, or methods. You are prompted to give a name for the new class. The new class is added to the browser.

NewInstance

Creates a new instance of this class, calls **PutSavedValue** with the new instance as an argument, and prints the instance.

10.3.3.4 Delete (DeleteMethod) and its Suboptions

Selecting the **Delete (DeleteMethod)** option and dragging the mouse to the right causes the following submenu to appear:



DeleteIV

Opens a menu containing the local instance variables, that is, those defined in the class. Selecting one removes it from the class.

DeleteCV

Same as **DeleteIV**, but for class variables.

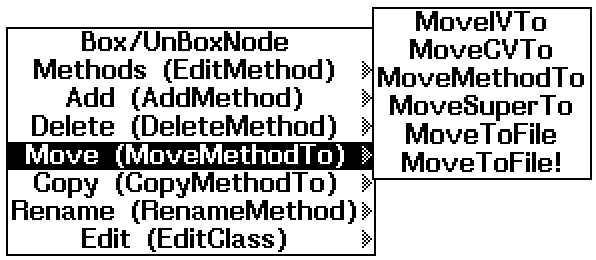
DeleteMethod Opens a menu containing the local selectors. Choosing one opens the following menu for confirmation:

DeleteClass Deletes this class. Opens a menu similar to the following for confirmation:

If the class has no subclasses it will be deleted. If it does have subclasses a **HELPCHECK** break occurs; you can then abort or type OK to destroy the class and all of its subclasses.

10.3.3.5 Move (MoveMethodTo) and its Suboptions

Selecting the **Move (MoveMethodTo)** option and dragging the mouse to the right causes the following submenu to appear:



Moving methods and variables requires that you first have used **BoxNode** (see Section 10.3.3.1, "BoxNode") on the class which is to receive whatever is moved.

CAUTION

Moving methods and variables can have profound effects on the classes that inherit them.

MoveIVTo Opens a menu with the instance variables for the class. Choosing one causes it to be moved to the class that is boxed in the browser. When this operation is completed, the menu opens again prompting you for another instance variable to move, if desired.

MoveCVTo Similar to **MoveIVTo**, but for class variables.

MoveMethodTo Similar to **MoveIVTo**, but for methods.

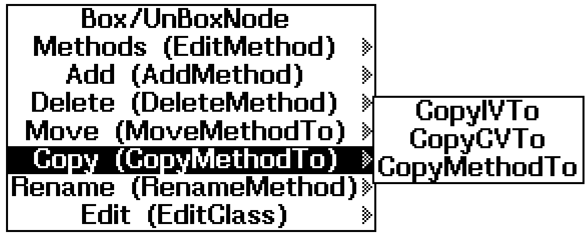
MoveSuperTo Opens a menu of the Supers for the class. Choosing one of these will cause it to be removed as a super class, and the boxed class to be added to the Supers list.

MoveToFile Opens a menu of files on **FILELST** along with ***newFile***. (See Section 10.2.1.2, "Command Summary," for more information on ***newFile***.) The class and its methods are moved to the chosen file.

MoveToFile! Same as **MoveToFile**, but includes subclasses and their methods of the classes.

10.3.3.6 Copy (CopyMethodTo) and its Suboptions

Selecting the **Copy (CopyMethodTo)** option and dragging the mouse to the right causes the following submenu to appear:



Copy options operates similarly to Move options, but leave the original method or variable in its place while adding it to the destination. You can then specialize the original or copy.

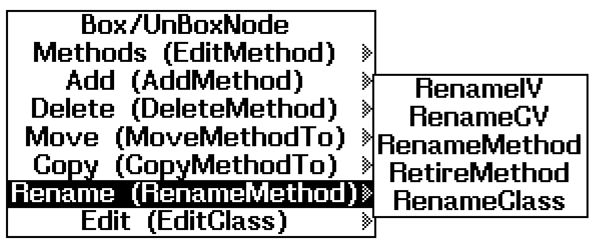
CopyIVTo Similar to **MoveIVTo,** but copies the instance variables instead of moving them.

CopyCVTo Similar to **MoveCVTo**, but copies the class variables instead of moving them.

CopyMethodTo Similar to **MoveMethodTo**, but copies the methods instead of moving them.

10.3.3.7 Rename (RenameMethod) and its Suboptions

Selecting the **Rename (RenameMethod)** option and dragging the mouse to the right causes the following submenu to appear:



RenameIV A menu appears, showing the instance variables of the class. After selecting one of these, you are prompted to give a new name for that instance variable.

RenameCV Similar to **RenameIV**, but for class variables.

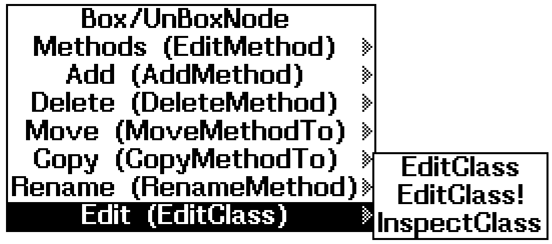
RenameMethod Similar to **RenameIV**, but for methods. You are prompted to give a new name for the selector of the method. The method function name is changed to reflect the change in the name of the selector.

RetireMethod A menu appears, showing the selectors of the class. After selecting one of these, the selector is changed by adding the prefix "Old" to it. The method function is also renamed appropriately.

RenameClass You are prompted for a new name for the class. After the new name is entered, the browser is updated to reflect the change. In addition, the method functions associated with the class are also renamed. For example, given a class **Foo** with selector **Fie**, when the class is renamed to **Fum**, the method function is automatically renamed from **Foo.Fie** to **Fum.Fie**.

10.3.3.8 Edit (EditClass) and its Suboptions

Selecting the **Edit (EditClass)** option and dragging the mouse to the right causes the following submenu to appear:



Editing options allow you to make quick, massive changes to object descriptions, and are sometimes the only menu-driven way to change certain items. See Chapter 13, Editing, for more details.

EditClass Edits the class definition of the class showing only the locally defined class variables, instance variables, and methods.

EditClass! Edits the class definition of the class showing both the locally defined and inherited class variables, instance variables, and methods. Changes to the inherited information that are done during the edit have no effect. The inherited information is included for informational purposes only. For example, you may want to copy the definition of an instance variable from the list of inherited instance variables to the list of local instance variables.

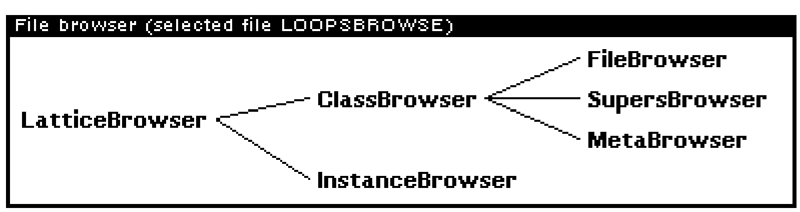
InspectClass Opens an inspector window on the class. See Chapter 18, User Input/Output Modules, for more details.

10.4 Using File Browsers

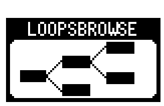
File browsers are a specialization of class browsers. In addition to the capabilities of class browsers, file browsers allow you to manipulate files. This section explains the file browser menu options not available from class browser menus, or those that have been modified from the class browser menu options. This section lists all file browser menu options; references are made to class browser menus where appropriate.

Multiple files can be associated with a file browser. Thus, one of those files can be designated as the "selected" file. There are various options as to which classes should be displayed in a file browser. See Section 10.4.1.4, "Change display mode and its Suboptions," for more information.

When a file browser is opened, the window title displays the selected file:

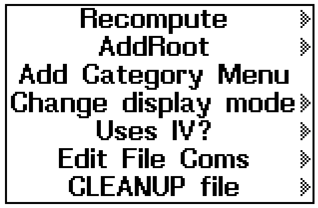


The icon for a shrunken file browser contains the name of the selected file, as shown here:



10.4.1 Selecting Options in the Title Bar Menu

The title bar menu in a **FileBrowser**, shown here, is like the title bar menu in a **ClassBrowser**, but has additional entries for file system and Masterscope functions.



10.4.1.1 Recompute and its Suboptions

Operates identically to the class browser. See Section 10.3.1.1, "Recompute and its Suboptions," for more information.

10.4.1.2 AddRoot and its Suboptions

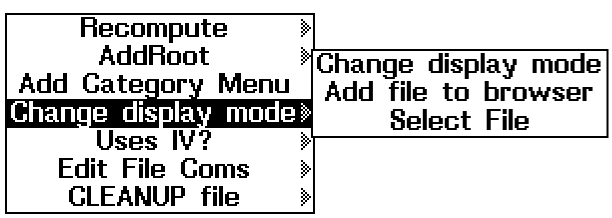
Prompts you for the name of a class to add. If the class already exists, it is added to the browser and shaded. If the class is not contained on the file, it will appear shaded in the browser. If the class does not exist, it is created and added to the selected file.

10.4.1.3 Add Category Menu

Operates identically to the class browser. See Section 10.3.1.3, "Add Category Menu and its Suboptions," for more information.

10.4.1.4 Change display mode and its Suboptions

Selecting the **Change display mode** option and dragging the mouse to the right causes the following submenu to appear:



A **FileBrowser** logically includes more display options than a **ClassBrowser**. A **FileBrowser** can display a class hierarchy as it is stored in the file, or as it exists in combination with other files and the system as a whole.

Change display mode Selecting this option causes a sub-submenu to appear showing three options:

- **selectedFile**

  Displays only the classes contained within the selected file or classes that have been added to the browser by **AddRoot** or **AddSubs** (by setting the browser's instance variable **goodList** to the appropriate value). See

Section 10.5.3, "Methods for the Class LatticeBrowser," for an explanation of **AddSubs**.

- **associatedFiles**

  Same as **selectedFile**, but the browser also includes any classes defined in files associated with the browser. The instance variable **goodList** is bound to this list, slightly differently than the use of **goodList** in a class browser.

- **all**

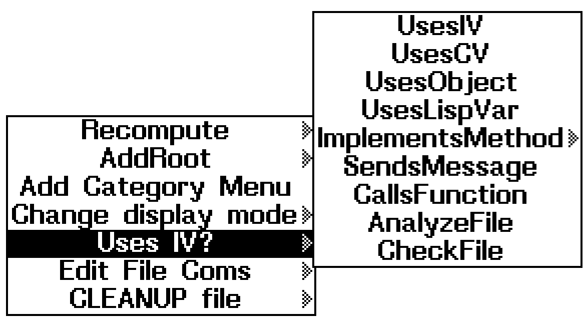
  Same as **associatedFiles,** but any subclasses, even if not defined in the files, are also displayed because the instance variable **goodList** is bound to NIL.

Add file to browser Prompts you with a menu that is similar to the menu that is displayed when you select **Browse File** from the LOOPS Icon or the background menu, except that files already associated with the browser are not displayed on the menu.

Select file Causes a menu to appear, showing the files associated with the browser. Selecting one causes that file to become the "selected file" of the browser. It clears the browser of any classes added to the browser by the **AddRoot** and **AddSubs** menu commands. That is, it resets the starting list of the browser to be only those classes contained within the selected file. The instance variable **badList** of the browser is set to NIL.

10.4.1.5 Uses IV? and its Suboptions

Selecting the **Uses IV?** option and dragging the mouse to the right causes the following submenu to appear:



These menu options trigger various Masterscope operations. Most of these operations prompt you for information that is used in a Masterscope query. The results of this query are used to build a second menu. If the situation occurs that the second menu is empty, a message is printed in the prompt window of the browser similar to "someCV not used as a CV."

CAUTION

Source files being displayed in the file browser must be available or these functions cannot work. In addition, the LOOPS Library Module LOOPSMS must also be loaded (see the *LOOPS Library Modules Manual* for details).

Additionally, the first time one of these options is selected there may be a pause while Masterscope analyzes the file. A window will open, and fill with "blips" as the analysis proceeds, until the file is analyzed and the original question is answered.

UsesIV This first opens a menu of instance variables defined in classes contained in the selected file of the browser. Two additional options are placed at the top of the menu:

- ***other***

  Selecting ***other*** causes a prompt to enter the name of an instance variable.

- ***any***

  Selecting ***any*** creates a menu with all methods that reference any instance variable.

After an instance variable has been chosen, you are prompted to place a menu that contains the following options:

- A list of methods on the selected file that use that instance variable
- A list of classes on the selected file that contain that instance variable.
- ***EditAll***

If one of the methods or classes is selected, it is edited. Suboptions from the methods or classes include:

- **Edit**

  Edits the method.

- **Substitute**

  Prompts for a new name for the instance variable. Changes the instance variable name to the new name in the method and then brings up the display editor for you to edit the method.

- **Check**

  Executes the Masterscope command CHECK <file> on the file associated with the LOOPS FileBrowser. See the *Lisp Library Modules Manual* for details.

EditAll has the following two suboptions:

- ***EditAll***

  Edits each method and class in succession.

- ***SubstituteAll***

  Prompts you for a new name for the instance variable. Substitutes this new name for the old name in all methods and classes listed in the menu.

UsesCV Same as **UsesIV**, but for class variables.

UsesObject Opens a menu of classes or instances defined on the selected file that are used by any of the methods or functions on the selected file. After you choose one of the objects, a menu similar to the one created for **UsesIV** is created, but contains the methods and functions that use the chosen objects.

UsesLispVar Same as **UsesIV**, but the initial menu displays Lisp variables instead of objects.

ImplementsMethod This option has three suboptions:

- **ImplementsMethod**

  Opens a menu of all of the selectors in the selected file. When one is chosen, a menu is created showing the methods that use that selector and the classes that are associated with those methods.

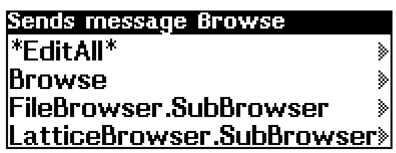
- **OverridesMethod**

  Generates a menu of methods and classes that override (that is, does not invoke **_Super**) the selected method.

- **SpecializesMethod**

  Generates a menu of methods and classes that specialize (that is, invoke **_Super**) the selected method.

SendsMessage Opens a menu of all of the selectors in the selected file. When one is chosen, a menu is created that lists the methods and functions that send messages using that selector. The following window shows a sample of this menu.



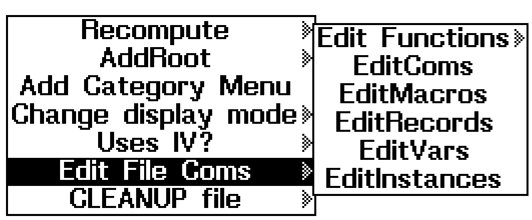
CallsFunction Opens a menu of all of the functions that are called by functions or methods in the selected file. After one is chosen, a menu is opened that contains the methods and/or functions that call the chosen function; the last option on the menu is the chosen function.

AnalyzeFile Begins a separate process analyzing the selected file. When the analysis is completed, "Done analyzing" is printed in the browser's prompt window.

CheckFile Begins a separate process checking the selected file. When the checking is completed, "Done checking" is printed in the browser's prompt window.

10.4.1.6 Edit FileComs and its Suboptions

Selecting the **Edit FileComs** option and dragging the mouse to the right causes the following submenu to appear:



The filecoms are variables that describe the contents of a file, for example, methods, classes, and Lisp functions and variables. LOOPS extends the File Manager to handle object oriented code and data, and the **FileBrowser** gives users a menu driven interface to deal with this extended file functionality.

Edit Functions Opens a sub-submenu giving options dealing with filecoms and with some of the items listed (functions in particular). This sub-submenu contains five suboptions:

- **EditFns**

  Opens a menu of the functions contained within the selected file (those listed under FNS in the filecoms) and the option ***NewFunction***. Selecting one of the functions calls the editor on that function.

  Selecting ***NewFunction*** causes a prompt for a name for the new function. An edit window then opens containing a template for a lambda expression. This newly defined function is added to the FNS list of the selected file's filecoms.

- **MakeFunctionMenu**

  Does an **ADDMENU** (the Interlisp function which adds a permanent menu to the screen) of a menu containing functions on the selected file. Selecting one of the functions opens an editor on it.

- **BreakFunction**

  Opens a menu containing functions on the selected file that are not on **BROKENFNS** (see the *Lisp Release Notes* and the *Interlisp-D Reference Manual*). Selecting one of the functions causes it to break next time it is invoked.

- **TraceFunction**

  Same as **BreakFunction**, except that the selected function is traced.

- **UnbreakFunction**

  Creates a menu of functions that are members of **BROKENFNS** and are contained in the selected file. The selected file is unbroken.

EditComs Edits the filecoms of the selected file.

EditMacros Creates a menu of macros contained in the selected file. Selecting one of them opens an editor on it.

EditRecords Same as **EditMacros,** but for records.

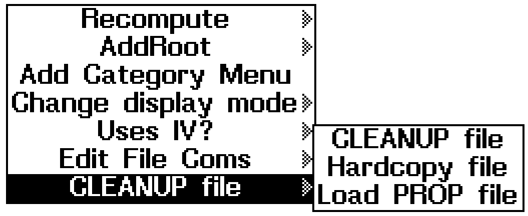
EditVars Same as **EditMacros,** but for variables.

EditInstances Same as **EditMacros,** but for instances.

10.4.1.7 CLEANUP file and its Suboptions

This option invokes some or all of **CLEANUP**, which is the automatic file maintenance utility of Medley.

Selecting the **CLEANUP file** option and dragging the mouse to the right causes the following submenu to appear:



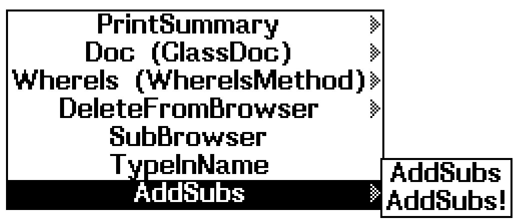
CLEANUP file Calls **FILES?** and then calls **CLEANUP** on the selected file.

Hardcopy file Sends the selected file to the **DEFAULTPRINTINGHOST**.

Load PROP file Loads the selected file with **LDFLG** set to **PROP**, making sources available to Masterscope, but leaving any compiled code in place to execute.

10.4.2 Selecting Options in the Left Menu

When the cursor is inside of a file browser and you press the left mouse button, nodes within the browser are inverted when the cursor moves over them. If you release the left mouse button while the cursor is over a node, the following menu appears:



These options include those in a class browser and add **AddSubs**, an option that expands the lattice of a file browser to look more like that of a class browser by showing related classes not stored in the browsed file.

10.4.2.1 PrintSummary and its Suboptions

Operates identically to the class browser. See Section 10.3.2.1, "PrintSummary and its Suboptions," for more information.

10.4.2.2 Doc (ClassDoc) and Its Suboptions

Operates identically to the class browser. See Section 10.3.2.2, "Doc (ClassDoc) and its Suboptions," for more information.

10.4.2.3 WhereIs (WhereIsMethod) and Its Suboptions

Operates identically to the class browser. See Section 10.3.2.3, "WhereIs (WhereIsMethod) and its Suboptions", for more information.

10.4.2.4 DeleteFromBrowser and Its Suboptions

Operates identically to the class browser. See Section 10.3.2.4, "DeleteFromBrowser and its Suboptions," for more information.

10.4.2.5 SubBrowser

Creates an instance of a class browser with the selected class as the root node, not a file browser. See Section 10.3.2.5, "SubBrowser," for more information.

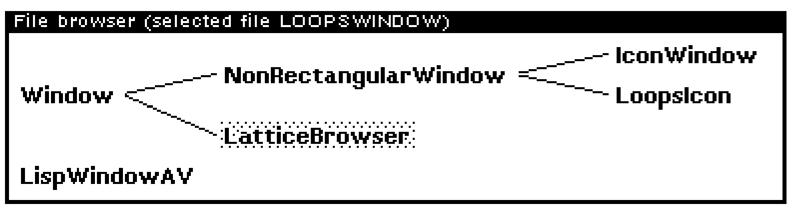
10.4.2.6 TypeInName

Operates identically to the class browser. See Section 10.3.2.6, "TypeInName," for more information.

10.4.2.7 AddSubs and its Suboptions

AddSubs fills out the class lattice in a file browser window. This shows classes, the file they are from (if any) and the inherited methods and variables from classes which are in the file.

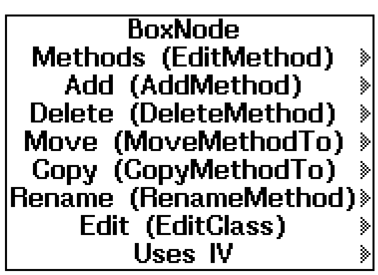
AddSubs Adds the immediate subclasses of the class to the browser and shades the new subclasses, as shown here:



AddSubs! Adds all subclasses of the class to the browser and shades the new subclasses.

10.4.3 Selecting Options in the Middle Menu

When the cursor is over a node and you press the middle mouse button, the following menu appears:



The middle button commands are the same as those on a **ClassBrowser**, with some new functionality for **Add (AddMethod)**, and with Masterscope options added under the option **UsesIV**.

10.4.3.1 BoxNode

Operates identically to the class browser. See Section 10.3.3.1, "BoxNode/UnBoxNode," for more information.

10.4.3.2 Methods (EditMethod) and its Suboptions

Operates identically to the class browser. See Section 10.3.3.2, "Methods (EditMethod) and its Suboptions," for more information.

10.4.3.3 Add (AddMethod) and its Suboptions

Operates identically to the class browser, but with additional functionality to keep the added items associated with the file being browsed in the following suboptions.

SpecializedClass Prompts you to enter a name for the new subclass for the chosen class. If the chosen class is on the selected file, the new subclass is added to that file. If the chosen class is on another file, choose a file from a menu of files to which the new subclass is added.

NewInstance Creates a new instance of the class and calls **PutSavedValue** with the new instance as an argument. You are prompted to give the new instance a name, and the new instance is added to the selected file.

10.4.3.4 Delete (DeleteMethod) and its Suboptions

Operates identically to the class browser. See Section 10.3.3.4, "Delete (DeleteMethod) and its Suboptions," for more information.

10.4.3.5 Move (MoveMethodTo) and its Suboptions

Operates identically to the class browser. See Section 10.3.3.5, "Move (MoveMethodTo) and its Suboptions," for more information.

10.4.3.6 Copy (CopyMethodTo) and its Suboptions

Operates identically to the class browser. See Section 10.3.3.6, "Copy (CopyMethodTo) and its Suboptions," for more information.

10.4.3.7 Rename (RenameMethod) and its Suboptions

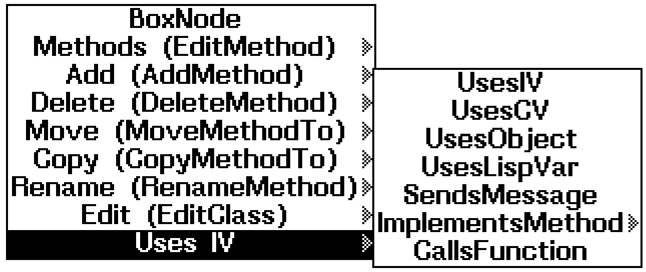
Operates identically to the class browser. See Section 10.3.3.7, "Rename (RenameMethod) and its Suboptions," for more information.

10.4.3.8 Edit (EditClass) and its Suboptions

Operates identically to the class browser. See Section 10.3.3.8, "Edit (EditClass) and its Suboptions," for more information.

10.4.3.9 UsesIV and its Suboptions

Selecting the **UsesIV** option and dragging the mouse to the right causes the following submenu to appear:



These commands operate similarly to the **Uses IV?** commands in the title bar menu. See Section 10.4.1.5, "UsesIV? and its Suboptions," for more information. Here, however, the Masterscope queries are limited to the class in question instead of the entire file.

10.5 PROGRAMMER'S INTERFACE TO LATTICE BROWSERS

10.5 Programmer's Interface to Lattice Browsers

LOOPS browsers are standard LOOPS objects, so their functionality can be exercised programmatically by messages which invoke their methods. Many browser functions are based on the Lisp Library Module, Grapher, but browsers apply only to dealing with LOOPS objects. All of the functionality in the menu-driven interface to the browsers is available programmatically.

Note: Data not a part of LOOPS data can be graphed with LOOPS calls to the Lisp Library Module, Grapher.

ClassBrowsers show the inheritance structure of object classes, a relationship defined at application design time. Browsers can also be used to show dynamic information, not computed until runtime. The LOOPS class **InstanceBrowser** does this, showing links between instances defined at runtime.

InstanceBrowser is derived from **LatticeBrowser** by specializing just two methods, **GetSubs** and **NewPath**. In general, you can specialize browsers to your own purposes by specializing these methods and **GetLabel**. An example is given in Section 10.7, "Class Instance Browser Example," producing a class browser which also shows instances, connected with dashed lines.

10.5.1 Instance Variables for the Class LatticeBrowser

Instance variables appear in alphabetical order.

badList A list of objects that are not displayed in the browser window.

boxedNode The last object boxed, if any.

browseFont The font used for labels. This has two properties: **FontFamily** and **FontFace** which are referenced in the method **ChangeFontSize**.

lastSelectedObject The last object selected.

goodList A list of objects that are displayed in the browser window; if NIL, no objects are displayed.

graphFormat A list indicating the style of layout for the graph. See the method **ChangeFormat** and the Lisp Library Module, Grapher.

LabelMaxCharsWidth Affects the way labels are generated. This limits the width of a label to **LabelMaxCharsWidth** times the width of the character "A". See the method **ChangeMaxLabelSize**. Default value is NIL, which puts no restrictions on label size.

LabelMaxLines Affects the way labels are generated. This limits the number of lines in a label to **LabelMaxLines**. Refer to the method **ChangeMaxLabelSize**. Default value is NIL which puts no restrictions on label size.

startingList List of objects used to compute this browser.

title Title passed to Grapher module.

topAlign This flag is used to indicate whether the graph should be aligned with the top or bottom of the window. If **topAlign** = T (the default), then the Grapher module aligns the graph to the top of the window.

10.5.2 Class Variables for the Class LatticeBrowser

Except for **BoxLineWidth**, the following class variables determine the menus that appear when a mouse is positioned over a node within a browser and the left or middle button is pressed. The default behavior is to send a message to the object represented by the node with the selector returned from the menu selection. The form for the values that these class variables can have is described in Chapter 20, Windows.

Class variables appear in alphabetical order.

BoxLineWidth The width of line that is drawn around a node when it is boxed. See the method **BoxNode** in Section 10.5.3, "Methods for the Class LatticeBrowser."

LeftButtonItems Items for the menu that appears when the mouse is on a node in the browser and the left button is pressed. When an item is selected from a menu, the returned value is sent as a message to an object represented by the node. See **LocalCommands**, below.

LocalCommands When the cursor is positioned over a node in a browser and you press the left or middle mouse button, the default behavior is to bring up a menu from which you select an item. The value returned from that item specifies the selector of a message that is sent to the object which is represented by the node in the browser.

The class variable **LocalCommands** provides a way to override that behavior. If the value returned from the menu selection is on the list that is the value of **LocalCommands**, the message is not sent to the object, but is sent to the browser instead. The object is passed as an argument in that message.

MiddleButtonItems Options for the menu that appears when the mouse is on one of the nodes in a browser and the middle button is pressed. When an option is selected from a menu, the returned value is sent as a message to the object represented by the node. See **LocalCommands**, above.

TitleItems Options for the menu that appears when the mouse is on the title bar in a browser and the left or middle button is pressed. When an option is selected from the menu, the returned value is sent as message to the browser.

The following selectors are associated with this menu:

- **SaveInIT**
- **Recompute**
- **RecomputeInPlace**

- **ShapeToHold**
- **ChangeFontSize**
- **ChangeFormat**
- **AddRoot**
- **RemoveFromBadList**

Examine the class **LatticeBrowser** for more information.

10.5.3 Methods for the Class LatticeBrowser

The following table shows the methods and variables for the class **LatticeBrowser**.

Name	Type	Description
AddRoot	Method	Adds a LOOPS name or an object to the starting list of the browser.
BoxNode	Method	Puts a box around the node representing the object.
Browse	Method	Uses a lattice or tree graph to display the relationship between a number of objects.
BrowserObjects	Method	Returns the list of objects currently in the graph.
ChangeFontSize	Method	Changes the size of the characters in the labels.
ChangeFormat	Method	Changes between lattice and tree graphs.
ChangeMaxLabelSize	Method	Changes how labels are printed.
ClearLabelCache	Method	Recomputes labels.
DeleteFromBrowser	Method	Prunes branches in a graph.
DeleteSubtreeFromBrowser	Method	Prunes branches in a graph.
FlashNode	Method	Changes the label of a node from black-on-white to white-on-black several times.
FlipNode	Method	Changes the label of a node that represents an object from black-on-white to white-on-black.
GetDisplayLabel	Method	Finds the label for a node.
GetLabel	Method	Computes a label for an object.
GetSubs	Method	Computes a list of subnodes of an object.
GraphFIts	Method	Determines if the graph in the browser can be contained within the browser window.
HasObject	Method	Returns T if an object is in the graph.
HighlightNode	Method	Changes the way a node is displayed.
IconTitle	Method	Computes the title to write in the icon.
LeftSelection	Method	Controls the effect of using the left mouse button.

LeftShiftSelect	Method	Sends the message **PP!** to an object.
MiddleSelection	Method	Controls the effect of using the middle mouse button.
MiddleShiftSelect	Method	Invoked by the mouse operations to edit an object in the TTY process context.
NewItem	Method	Gets an object.
NodeRegion	Method	Returns the region occupied in an object in the browser.
ObjectFromLabel	Method	Returns the object in the graph that has a specified label.
PositionNode	Method	Places a node at a particular position in the browser window.
Recompute	Method	Recomputes the browser graph in the same window.
RecomputeInPlace	Method	Recomputes the browser graph in the same window, trying to maintain the same scroll position in the window.
RecomputeLabels	Method	Recomputes the labels in a browser.
RemoveHighlights	Method	Removes all highlights in any node in the graph.
RemoveShading	Method	Removes all shading in any node in the graph.
SaveInIt	Method	Places a pointer in a browser where it can be accessed.
ShadeNode	Method	Adds shading to a node.
ShapeToHold	Method	Reshapes the window to all items.
MaxLatticeWidth	Variable	Restricts the maximum width of a browser window.
MaxLatticeHeight	Variable	Restricts the maximum height of a browser window.
Show	Method	Displays items and their subitems in a browser window.
Shrink	Method	Shrinks a browser window.
SubBrowser	Method	Creates a browser that is an instance of the same class as *self* with a specified object as the root.
TitleSelection	Method	Invokes an action when the mouse is in the title bar on a browser window and the left or middle button is pressed.
UnmarkNodes	Method	Sends the messages **RemoveHighlights** and **RemoveShading** to *self*.

(← *self* **AddRoot** *newItem*) [Method of LatticeBrowser]

Purpose: Adds *newItem*, which is a LOOPS name, to the starting list of the browser.

Behavior: First determines if the name *newItem* points to a LOOPS object. If it does not, a message is printed that nothing has been added to the browser. If *newItem* is NIL, you are prompted to enter a name through the method **NewItem**. If the object pointed to by *newItem* is on the browser's instance variable **badList**, it is removed from **badList.** If the instance variable **goodList** has a value, *newItem* is added to it.

Arguments: *newItem* LOOPS name.

Returns: Class object or NIL.

Categories: LatticeBrowser

Example: The following command adds class **Datum** to a class browser instance named **CB1**:

```
55←(← ($ CB1) AddRoot ($ Datum))
```

(← *self* **BoxNode** *object objName unboxPrevious*) [Method of LatticeBrowser]

Purpose: Puts a box around the node in the graph representing the object.

Behavior: First checks to make sure *object* points to a LOOPS object. If not, nothing happens. The previous value of the instance variable **boxedNode** is returned.

- If the instance variable **boxedNode** is NIL, then a box is drawn around the node with a line width equal to the value of the class variable **BoxLineWidth**. The instance variable **boxedNode** is assigned the value of *object*, and *object* is returned.
- If *object* is EQ to the instance variable **boxedNode**, then the box is erased. That is, calling **BoxNode** twice in succession will draw and then erase the box. The instance variable **boxedNode** is assigned the value NIL, and NIL is returned.
- If none of the above conditions hold, the flag **unboxPrevious** is checked. If it is non-NIL, the previously boxed node is unboxed, and the node represented by *object* is boxed. The instance variable **boxedNode** is assigned the value of *object*, and *object* is returned.

It is possible that *object* is not a node in *self*.

Arguments:

object LOOPS name or object.

objName Used internally; can be NIL.

unboxPrevious
Can be NIL or T.

Returns: The object in the browser that is currently boxed, or NIL if nothing is currently boxed.

Categories: LatticeBrowser

Specializations: ClassBrowser

(← *self* **Browse** *browseList windowOrTitle goodList position*) [Method of LatticeBrowser]

Purpose: Uses a lattice or tree graph to display the relationships between a number of objects. **Browse** is the proper message to use for initializing browsers.

Behavior: Sends the message **Show** to *self* passing the arguments *browseList, windowOrTitle*, and *goodList*. It next sends **ShapeToHold** to *self*. Finally it sends **Move** to *self* with the argument *position*.

Arguments:

browseList A list, elements of which can be a LOOPS name or an object, or a single item which can be a LOOPS name or an object. Used as the starting node(s) of a browser. See the **Show** message later in this section for details.

windowOrTitle
If a window, the browser is displayed in this window. If not a window, this becomes the title of the browser window.

goodList A list, elements of which can be a LOOPS name or an object. See the **Show** message later in this section for details.

position — A position to which the lower left corner of the browser is moved. Can be NIL.

Returns: Used for side effect only.

Categories: LatticeBrowser

Example: The following command gets the class browser instance **CB1** to browse class **Datum** and its subclasses:

```
57←(← ($ CB1) Browse 'Datum)
```

(← *self* **BrowserObjects**) [Method of LatticeBrowser]

Purpose/Behavior: Returns the list of objects currently in the graph of the browser.

Categories: LatticeBrowser

(← *self* **ChangeFontSize** *size*) [Method of LatticeBrowser]

Purpose: Changes the size of the characters of the labels.

Behavior: Changes the font used to display labels in a browser. The browser is redrawn and the window shaped to fit. If no size is given, this lets you select the size from a menu. This menu is bound to the top level binding of the variable **MenuSize** (the first time it is called). The font family used is the value of the **FontFamily** property of the instance variable **browseFont**. The font face used is the value of the **FontFace** property of the instance variable **browseFont**.

This sends the message **RecomputeLabels** to *self*.

Arguments: *size* — Integer size of font for node labels.

Returns: Used for side effect only.

Categories: LatticeBrowser

(← *self* **ChangeFormat** *format*) [Method of LatticeBrowser]

Purpose: Changes between lattice and tree graphs.

Behavior: If *format* is NIL, then select a format from a menu that appears. The items in this menu are determined by the value of the **choices** property of the instance variable **graphFormat**. Changes the value of the instance variable **graphFormat** to *format* or the value selected from the menu.

Arguments: *format* — Describes the format layout. The argument *format* is an unordered list of atoms or lists. The following options control the structure of the graph:

- COMPACT, the default, which lays out the graph as a forest (that is, a set of disjoint trees) using the minimal amount of screen space.
- FAST, which lays out the graph as a forest, sacrificing screen space for speed.
- LATTICE, which lays out the graph as a directed acyclic graph, that is, a lattice.

In addition, the following options control the direction of the graph:

- HORIZONTAL, the default, has roots at the left and links that run left-to-right.
- VERTICAL has roots at the top and links that run top-to-bottom.

See the function **LAYOUTGRAPH** in the Grapher library module documentation for more information.

Returns: Used for side effect only.

Categories: LatticeBrowser

Example: The commands:

```
58← (SETQ b1 (←New ($ SupersBrowser) Browse 'ClassBrowser))
#,($& SupersBrowser (NEW0.1Y%:.;h.eN6 . 506))

59←(← b1 ChangeFormat '(HORIZONTAL REVERSE (MARK BORDER 3 LABELSHADE 1)))
#,($& SupersBrowser (NEW0.1Y%:.;h.eN6 . 506))
```

results in:

(← *self* **ChangeMaxLabelSize** *newMaxWidth newMaxLines*) [Method of LatticeBrowser]

Purpose: Changes how labels are printed.

Behavior: Sets the maximum width of a node label. An argument value of zero means no maximum size, and NIL means no change.

By setting both *newMaxWidth* and *newMaxLines*, you get an abbreviation facility. This binds the values of the instance variables **LabelMaxCharsWidth** and **LabelMaxLines**. The default values for **LabelMaxCharsWidth** and **LabelMaxLines** are both 0, so to return a browser b1 to default performance, send

```
(← ($ b1) ChangeMaxLabelSize 0 0).
```

The resulting labels may be bitmaps or strings. The width of the bitmap is a product of *newMaxWidth* and the width of the character "A" in the current value of the instance variable **browsefont**.

Sends the message **RecomputeLabels** to *self*.

Arguments: *newMaxWidth*
Maximum number of characters per line; default is 0.

newMaxLines
Maximum number of lines; default is 0.

Returns: Used for side effect only.

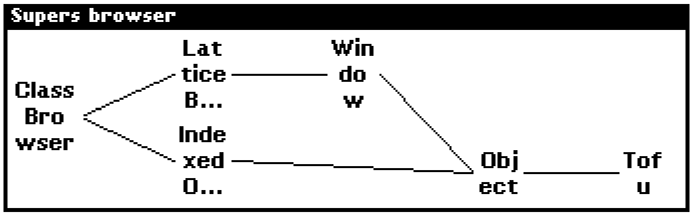
Categories: LatticeBrowser

Example: The commands:

```
60←(SETQ b1 (←New ($ SupersBrowser) Browse 'ClassBrowser))
#,($& SupersBrowser (NEW0.1Y%:.;h.eN6 . 506))
```

```
61←(← b1 ChangeMaxLabelSize 3 3)
#,($& SupersBrowser (NEW0.1Y%:.;h.eN6 . 506))
```

results in:



(← *self* **ClearLabelCache** *objects*) [Method of LatticeBrowser]

Purpose: Forgets cached labels in the browser.

Behavior: Clears the label cache for item(s) in *objects*. If *objects* is the symbol T, then this clears the entire label cache. The cache for the labels is on the **objectLabels** property of the instance variable **menus**.

Arguments: *objects* An object or a list of objects.

Returns: Used for side effect only.

Categories: LatticeBrowser

(← *self* **DeleteFromBrowser** *object objname*) [Method of LatticeBrowser]

Purpose: Prunes branches in a graph.

Behavior: Removes *object* from the browser by putting it on the instance variable **badList** and then sending the **Recompute** message to *self*. The object and its subtree are deleted from the browser.

Arguments: *object* An object in the browser.

objname Used internally; can be NIL.

Returns: Used for side effect only.

Categories: LatticeBrowser

(← *self* **DeleteSubtreeFromBrowser** *object*) [Method of LatticeBrowser]

Purpose: Prunes branches in a graph.

Behavior: Similar to **DeleteFromBrowser**, but the subnodes of *object* are also added to the instance variable **badList**.

Arguments: *object* An object in the browser.

Returns: Used for side effect only.

Categories: LatticeBrowser

(← *self* **FlashNode** *node N flashTime leaveFlipped?*) [Method of LatticeBrowser]

Purpose/Behavior: Changes the label of a node from black-on-white to white-on-black several times.

Arguments: *node* LOOPS name or object.

N Number of times *node* will be flipped.

flashTime The amount of time in milliseconds that the node is held between transitions. If *flashTime* is NIL, this time defaults to 300 milliseconds.

leaveFlipped? Can be NIL or T. If T, *node* is left inverted from its original state.

Returns: Used for side effect only.

Categories: LatticeBrowser

(← *self* **FlipNode** *object*) [Method of LatticeBrowser]

Purpose: Inverts the label of a node.

Behavior: If the node is black-on-white then it is changed to white-on-black, and conversely.

Arguments: *object* LOOPS name or object.

Returns: Used for side effect only.

Categories: LatticeBrowser

(← *self* **GetDisplayLabel** *object*) [Method of LatticeBrowser]

Purpose: Finds the label for a node in the graph.

Behavior: If there is a cached label on the **objectLabels** property of the instance variable **menus**, return it.

If not, this takes the result of **GetLabel** and breaks it into multiple lines to fit in the maximum label size defined by the instance variables **LabelMaxCharsWidth** and **LabelMaxLines**, if these are non-NIL. This placement tries to break the label after special characters such as . , : ; / or space, or at changes from lowercase to uppercase. The resulting bitmap is put into cache so that recomputing the graph is faster.

When a label is broken up into multiple lines, the label is changed from a string to a bitmap, thus causing shading not to work as described later in this section in the method **ShadeNode**.

Arguments: *object* An object.

Returns: Used for side effect only.

Categories: LatticeBrowser

(← *self* **GetLabel** *object*) [Method of LatticeBrowser]

Purpose: Computes a label for *object*. A label may be a symbol or a bitmap; bitmap labels should be freshly created since the method **ShadeNode** may smash them. (The method **GetDisplayLabel** is used internally to fetch labels for display; it caches label bitmaps to minimize the use of **GetLabel.**)

Behavior: Returns (**GetObjectName** *object*).

Arguments: *object* LOOPS name or object, which can be a bitmap.

Returns: (**GetObjectName** *object*)

Categories: LatticeBrowser

(← *self* **GetSubs** *object*) [Method of LatticeBrowser]

Purpose: Computes a list of subnodes of *object*.

Behavior: Determines next level of nodes in lattice. Specializations of **LatticeBrowser** typically specialize this method.

Arguments: *object* A LOOPS object.

Returns: NIL or the value of the instance variable **subs** of *object*.

Categories: LatticeBrowser

Specializations: ClassBrowser, InstanceBrowser, MetaBrowser, SupersBrowser

(← *self* **GraphFits** *snugly*) [Method of LatticeBrowser]

Purpose/Behavior: Determines if the graph in the browser can be contained within the window of the browser.

Arguments: *snugly* If snugly? is non-NIL the graph must fit in the window leaving less than twice the **FONTHEIGHT** of the browser's browseFont as empty space around it.

Returns: T if the entire graph can be displayed within the window; else NIL.

Categories: LatticeBrowser

(← *self* **HasObject** *object*) [Method of LatticeBrowser]

Purpose/Behavior: Returns T if *object* is in the graph of the browser.

Arguments: *object* LOOPS name or object.

Categories: LatticeBrowser

(← *self* **HighlightNode** *object width shade*) [Method of LatticeBrowser]

Purpose: Changes the way a node is displayed.

Behavior: Draws a box around a node for *object* using a given *width* and *shade* for the lines of the box. A shade is a 16-bit number representing a 4x4 bitmap. See **EDITSHADE** in the *Interlisp-D Reference Manual*.

Arguments: *object* LOOPS name or object.

width Integer width of box.

shade 16-bit number representing a 4x4 bitmap.

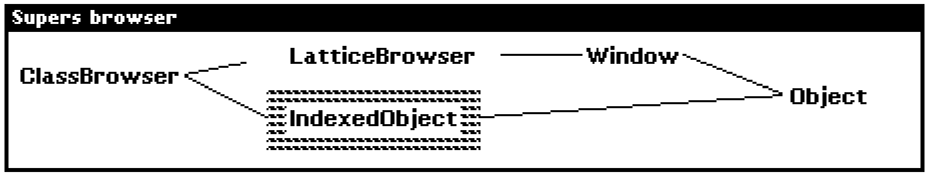
Returns: Used for side effect only.

Categories: LatticeBrowser

Example: The command

```
64←(← b1 HighlightNode 'IndexedObject 10 123)
```

results in the following window:



(← *self* **IconTitle**) [Method of LatticeBrowser]

Purpose/Behavior: Computes the title to write in the icon.

Returns: The label of the first root entry in the lattice, that is, the **CAR** of the instance variable **startingList**. If this is NIL, then use "Browser". If **AddRoot** is called, the new root becomes the first entry on **startingList**.

Categories: LatticeBrowser

Specializations: FileBrowser

(← *self* **LeftSelection**) [Method of LatticeBrowser]

Purpose: Controls the effect of using the left mouse button. LatticeBrowser provides defaults, but allows these methods to be overwritten or specialized.

Behavior: The instance variable **lastSelectedObject** is bound to the object of the node that the left button selected.

The remaining behavior varies according to the key pressed.

- If the **Move** key or the **Control** key is down, this allows you to move the node the mouse is over when you press the left mouse button.
- If the left shift key or **Copy** key is down while the cursor is over a node, the label of the node is copied to the system buffer. If the cursor is not over a node, the entire graph is copied. This allows you to copy browsers into TEdit documents.
- If the **Meta** key is pressed, the message **LeftShiftSelect** is sent to *self*

11. ERRORS AND BREAKS

LOOPS provides an interface to the Medley error system. This allows appropriate detection and recovery from errors that are LOOPS errors rather than Lisp errors. The full power of the Medley error system is available to help you determine and repair the causes of errors. In addition, under certain circumstances, LOOPS will attempt to repair an error and continue if you agree.

This chapter describes the functions and methods LOOPS uses to handle error conditions. It also describes the error messages generated by LOOPS.

11.1 Error Handling Functions and Methods

LOOPS provides several ways to trap and process many common errors. A default processing is available for most errors, and this processing can be specialized for actions you may require.

The following table shows the items in this section.

Name	Type	Description
HELPCHECK	Function	Provides an interface to the Common Lisp error system.
LoopsHelp	NoSpread Function	Generates an error if **LoopsDebugFlg**=NIL, else calls **HELP**.
LoopsDebugFlg	Variable	Controls the behavior of **LoopsHelp**.
ErrorOnNameConflict	Variable	Calls **HELPCHECK** when you attempt to give an object the same name as an existing object.
CVMissing	Method	Sent by access functions when you attempt to access a class variable that does not exist.
CVValueMissing	Method	Sent by access functions when you attempt to access a class variable that has no value.
IVMissing	Method	Sent by access functions when you attempt to access an instance variable that does not exist.
IVValueMissing	Method	Sent by access functions when you attempt to access an instance variable that has no value.
MessageNotUnderstood	Method	Sent when a message has no corresponding selector.

(**HELPCHECK** *mess1 ... messN*) [Function]

Purpose/Behavior: **HELPCHECK** is the LOOPS interface with the Common Lisp error system. When LOOPS detects an error, it generally calls this function with up to four argument messages describing what is wrong and possibly what to do about it. **HELPCHECK** calls **BREAK1** to put you into a break window and returns whatever the call to **BREAK1** returns. For example, if you type `OK`, it returns T. If you type "RETURN ’someValue", it returns that value. In some instances, LOOPS uses such returned values to repair errors and continue execution.

Arguments: *mess1 ... messN*
Messages to print at the break.

Returns: Value depends on what you type in the break window; see Behavior**.**

Example: The following code causes a break window with the message "Are you certain?". If you type "OK" in the break window, the message "He said OK" will print.

```
(IF (HELPCHECK "Are you certain?")
    THEN (PRINT "He said OK"))
```

(LoopsHelp *mess1 ... messN***)** [NoSpread Function]

Purpose/Behavior: Generates an error. Calls **HELP** if **LoopsDebugFlg** is T, otherwise calls **ERROR**. Use **LoopsHelp** whenever you want to give the user a way to recover from errors when **LoopsDebugFlg** is T. For example, have **LoopsHelp** print messages like "FOO is not the name of a class. Type RETURN ’<classname> to continue using <classname>."

Arguments: *mess1 ... messN*
Messages to print at the break.

Returns: Value depends on what you type in the break window; see **HELPCHECK**, above.

LoopsDebugFlg [Variable]

Purpose/Behavior: Controls the behavior of **LoopsHelp**. If it is T, all calls to **LoopsHelp** generate a break. If it is NIL, such calls that occur near the top of the stack or after a short computation cause a message to be printed and a return to the next level. The default value is T. See **BREAKCHK** in the *Interlisp-D Reference Manual* for more information.

ErrorOnNameConflict [Variable]

Purpose/Behavior: If T, an attempt to give an object the same name as an existing object causes a call to **HELPCHECK**. If you type "OK" in the resulting break window, the process continues and the original object is unnamed. The default value is NIL.

(← *self* **CVMissing** *object varName propName typeFlg newValue*) [Method of Class]

Purpose: Sent by access functions when there is an attempt to access a class variable that does not exist.

Behavior: Calls **LoopsHelp** with the message

varName not a CV of *self*

This method can be specialized to take more sophisticated action by using the other arguments which are provided.

When, in an instance, an attempt is made to access a class variable that does not exist, the message **CVMissing** is sent to the instance’s class with the instance in question as *object*.

Note: This method can be invoked if an instance variable is missing.

Arguments: *object* The object upon which the access was attempted.

typeFlg The name of the access function (**GetValue**, **GetValueOnly**, **PutValue**, **PutValueOnly**) which caused this message to be sent. The function name allows the type of access to be determined.

varName The name of the variable on which access was attempted.

propName The name of the property on which access was attempted. If NIL, the value of the class variable *varName* was accessed.

newValue The value to which the class variable was to be set.

Categories: Class

Example: Specialize this method to automatically add the class variable which is missing to the class described by *self*. Assuming the class of *self* is **SomeClass**, the method definition is

```
(Method ((SomeClass CVMissing)
           self object varName propName typeFlg
           newValue)
(← self AddCV varName newValue))
```

(← *self* **CVValueMissing** *object varName propName typeFlg*) [Method of Class]

Purpose: Sent by access functions when there is an attempt to access a class variable that has no value. This method can also be invoked if an instance variable is missing and you attempt to access it.

Behavior: If *propName* is NIL it returns the value of **NotSetValue**, otherwise it returns the value of **NoValueFound**.

The default setting for **NoValueFound** is NIL. The default setting **NotSetValue** is an annotatedValue. See Chapter 8, Active Values, for an explanation of **NotSetValue**.

This method can be specialized to take more sophisticated action by using the other arguments which are provided. See the example for **CVMissing**, above.

Arguments: *object* The object on which the access was attempted.

typeFlg The name of the access function (**GetValue**, **GetValueOnly**, **PutValue**, **PutValueOnly**) which caused this message to be sent. The function name allows the type of access to be determined.

varName The name of the variable on which access was attempted.

propName The name of the property on which access was attempted. If NIL, the value of the class variable *varName* was accessed.

Returns: Value depends on the arguments; see Behavior.

Categories: Class

(← *self* **IVMissing** *varName propName typeFlg newValue*) [Method of Object]

Purpose: Sent by access functions when there is an attempt to access an instance variable that cannot be found in *self*.

Behavior: Tries to remedy the situation, but if it fails, it calls **LoopsHelp** with the message *varName* not an IV of *self*

If the instance variable is present in the object's class, the instance variable will be copied to *self*. This can happen when a class is changed after an instance has been created.

If the instance variable is not present in the class, it attempts to find a class variable of the same name in the class. If one is found, it is used according to its **:allocation** property.

- If the property is **dynamicCached**, the instance variable is added by copying the class variable regardless of the type of access.
- If the property is **dynamic**, the type of access is determined from *typeFlg*, which is the name of the access function. The value of the class variable is returned for a get and the instance variable is created only on a put.
- If the property is **class**, the class variable's value is returned or set and no instance variable is created.

If all else fails, an attempt is made to fix the spelling of *varName* and, if a possible fixed spelling is found, the process starts over.

If an instance variable is not found, the arguments are not used, but could be in a specialization of this method. See the example in **CVMissing** above.

Arguments: *typeFlg* The name of the access function (**GetValue**, **GetValueOnly**, **PutValue**, **PutValueOnly**) which caused this message to be sent. The function name allows the type of access to be determined.

varName The name of the variable on which access was attempted.

propName The name of the property on which access was attempted. If NIL, the value of the instance variable *varName* was accessed.

newValue The value to which the instance variable was to be set.

Returns: Value depends on the arguments; see Behavior.

Categories: Object

(← *self* **IVValueMissing** *varName propName typeFlg newValue*) [Method of Object]

Purpose: Sent by access functions when there is an attempt to access an instance variable which has no value in *self*.

Behavior: Looks up the class hierarchy to find a value. If none is found, **SHOULDNT** (see the *Interlisp-D Reference Manual*) is called with the message

Error in Put or GetValue.

The arguments are not used, but could be in a specialization of this method. See the example in **CVMissing**, above.

This method is used internally to handle inheritance of instance variable values. If this error occurs, the LOOPS system has probably been corrupted.

Arguments: *typeFlg* The name of the access function (**GetValue**, **GetValueOnly**, **PutValue**, **PutValueOnly**) and allows the type of access to be determined.

The other arguments are passed from the access function.

Categories: Object

(← *self* **MessageNotUnderstood** *selector messageArguments superFlg*) [Method of Object]

Purpose: Sent when a message has no corresponding selector in *self*.

Behavior: Attempts to fix the spelling of *selector*. If this fails, it generates an error.

Arguments: *selector* The name of the message that was not understood.

messageArguments
The arguments of the message *selector*.

superFlg If T, an attempt was made to locate the method *selector* in the supers of *self*.

Categories: Object

Example: Define a class that acts as the class of Lisp numbers, and use the **MessageNotUnderstood** message to translate messages into function calls.

```
37←(DefineClass 'Number)
#.($ Number)

38←(← ($ Number) SpecializeMethod 'MessageNotUnderstood)
```

The **MessageNotUnderstood** method is defined in the editor, making the body of the method as follows:

```
(if (GETD selector)
  then (APPLY selector messageArguments))
  else (←Super))
Number.MessageNotUnderstood
```

Use the class **Number** as the LOOPS class for Lisp numbers.

```
39←(PUTHASH 'SMALLP ($ Number) LispClassTable)
#.($ Number)

40←(PUTHASH 'FIXP ($ Number) LispClassTable)
#.($ Number)

41←(PUTHASH 'FLOATP ($ Number) LispClassTable)
#.($ Number)
```

Test it out.

```
42←(← 4 PLUS 5)
9
```


11.2 ERROR MESSAGES

11.2 Error Messages

This section contains the LOOPS error messages along with their explanations. Atoms which are in *italics* are replaced with specific values when the messages are generated. Messages generated by calls to **SHOULDNT** indicate problems in LOOPS system code. Messages generated by direct calls to **ERROR**, that is not via calls to the LOOPS function **LoopsHelp**, may indicate problems with the system or with user code.

Errors appear in their respective categories:

- Errors that occur when accessing classes and instances in LOOPS.

- Errors that occur when sending messages to LOOPS objects.
- Errors dealing with naming objects.
- Errors encountered when using annotated values and active values.
- Other error messages that may be encountered when using LOOPS.

11.2.1 Classes and Instances

This section describes errors that occur when accessing classes and instances.

type not recognized part of class

Explanation: The *type* argument to the method **ListAttribute** does not correspond to one of the parts of a class.

name not a CV of *self*

Explanation: A reference has been made to a class value that does not exist.

Error in Put or GetValue

Explanation: An attempt has been made to access an instance variable that has no value in an object or in any of its supers.

varName not an IV of *self*

Explanation: An attempt has been made to access an instance variable and it does not exist in the object or its supers, and a class variable of the same name does not exist either.

varName is not a local instance variable of class *name*. Type OK to ignore error and go on.

Explanation: An attempt has been made to delete an instance variable which is not in the class.

newValue is not a class. Type OK to replace metaclass of *classRec* with $Class

Explanation: A call has been made to **PutClass** or **PutClassOnly** with either *propName* erroneously set to NIL or left out, or the new metaclass set to something that is not a valid class.

varName is not a CV of *Class* so cannot be moved from there

Explanation: An attempt has been made to move a class variable from a class where it does not exist. Possible causes include wrong source class or misspelled class variable name.

class has subclasses. You cannot **Destroy** classes that have subclasses.
Type OK to use **Destroy!** if that is what you want.

Explanation: Sending the message **Destroy** to a class with subclasses will leave the subclasses referring to nonexistent superclasses. **Destroy!** destroys all of the subclasses as well. Be sure this is what you want before you type "OK".

11.2.2 Methods and Messages

This section describes errors that occur when sending messages to LOOPS objects.

GetValue, PutValue, GetValueOnly, PutValueOnly or **GetIVHere** *self args* not possible

Explanation: An attempt has been made to access a value in an abstract class, which cannot have any values.

← or ←**Super** *self selector* -- not understood

Explanation: Neither the object to which the message was sent nor any of its ancestors has such a method selector.

(← NIL selector --) not understood

Explanation: An attempt has been made to send a message to NIL. One way to do this is to execute (_($ foo) ...), where foo does not name a LOOPS object.

class does not contain the selector *selector*. Type RETURN ’selectorName to try again

Explanation: An attempt has been made to delete a nonexistent method. If the problem is that the wrong method selector was typed or the selector was misspelled, typing "RETURN ’correctName" will fix the problem.

selector is not local for *self* To copy anyway, type OK

Explanation: The object to which **CopyMethod** was sent does not contain *selector*, but one of its supers does. This is not necessarily an error.

selector is not a selector for *self*

Explanation: Neither the object to which **CopyMethod** was sent nor any or its supers contains *selector*.

newClass is not a class. Type OK to use oldClass

Explanation: Something may be missing from the argument to **HELPCHECK,** since nothing is printed after oldClass. Alternatively, the destination class specified in **CopyMethod** is neither a class nor a valid class name.

Typing "OK" causes the method to be copied to the class to which the message was sent. The net result can be to copy a method down from one of the class’s supers or to make a copy within the class with a new selector.

name is not a defined function

Explanation: The selector named in **CopyMethod** exists but it does not have a function defined for it. It is possible the class has been loaded but the method has not or that the function definition for the method was somehow erroneously destroyed.

name not a currently defined class. Cannot add method to class.
Type OK to create class and go on.

Explanation: An attempt has been made to add a method to a nonexistent class.

If the class should exist, but has not been created yet, type "OK" to let LOOPS create it automatically. If the class has yet to be loaded, abort and load it first.

Can't find source for *fn*

Explanation: The source file containing a method of a class that is being moved via **MoveToFile** cannot be found. **WHEREIS** is used to try to find it. Either add the necessary file to **FILELST** or use **LOADFNS** to load the function(s).

11.2.3 Naming Objects

This section describes errors that occur when naming objects.

name is already used as a name for an object

Explanation: **ErrorOnNameConflict** has been set to T and an object with the given name already exists. Typing "OK" will cause the new object to be created anyway.

Can't name object NIL

Explanation: The *name* argument to the method **SetName** has been left out.

name should be a symbol to be a name

Explanation: The method **SetName** has been given a non-symbolic name.

name cannot be a class name. Type OK to ignore

Explanation: A non-symbolic class name has somehow gotten into the CLASSES of a file.

Typing "OK" will continue writing the file, but will not remove the offending name.

Can't rename a class without specifying name.
Type RETURN <newName> to continue and rename class: *self*

Explanation: The *newName* argument has been left out of **Rename**. Classes can not be named NIL.

Typing "RETURN 'aNewName" renames the class.

name not defined as a class or an instance. Type OK to ignore and go on.

Explanation: A name which refers to a nonexistent class or instance is in the **CLASSES** or **INSTANCES** file command of a file.

Typing "OK" continues writing out the file, but does not remove the offending name.

name not the name of an instance! Type OK to proceed.

Explanation: A name that refers to a nonexistent instance is in the **THESE-INSTANCES** file command of a file.

Typing "OK" continues writing out the file, but does not do anything to correct the source of the problem; that is, it does not remove the name from the filecoms or find out why it does not exist.

name is a defined object, but is not a class.

Explanation: The name of some LOOPS object that is not a class has been used as an argument where a class name should have been used.

11.2.4 Annotated and Active Values

This section describes errors that occur when using annotated values and active values.

Active value not found, so can't replace it.

Explanation: The old active value specified in **ReplaceActiveValue** does not exist or has been specifed incorrectly.

Unknown access type *type*

Explanation: An improper *type* has been given to the message **AddActiveValue** or **DeleteActiveValue**.

Invalid type *type*

Explanation: An active value has an incorrect type specifier.

Conflicting active value wrapping precedence *self activeValue otherPrecedence*

Explanation: An attempt has been made to add an annotated value with wrapping precedence T or NIL to an existing annotated value with the same wrapping precedence.

Unknown access type *type*

Explanation: **GetWrappedValue** or **PutWrappedValue** has been given an incorrect type.

Can't set the local state of #.NotSetValue

Explanation: **PutWrappedValueOnly** has been erroneously sent to a #.NotSetValue.

11.2.5 Miscellaneous

This section describes other errors that can occur when using LOOPS.

Use one of METHODS IVS CVS for type. RETURN one of these symbols to go on.

Explanation: An incorrect type has been specified to the method **WhereIs**.

To continue, enter the type into the break window. For example, enter "RETURN 'METHODS".

Name not installed because of error in source

Explanation: The source specification of a class has been corrupted in some way. It may be necessary to manually redefine the class or edit the file.

Time is not set! Call (**SETTIME** dd-mmm-yy hh:mm:ss) and then type in OK

Explanation: LOOPS uses the date and time to create unique internal names for objects; thus, the time must be set before any objects are created. Call **SETTIME** and then type "OK". For example, (**SETTIME** "15-APR-87 12:00:00") sets time at noon on April 15, 1987.

self varName propName not broken. Type OK to go on

Explanation: Either an attempt has been made to unbreak a value which was not broken or the value was specified incorrectly.

[This page intentionally left blank]

12. BREAKING AND TRACING

A number a functions and methods are available in the LOOPS environment to facilitate the process of finding and correcting bugs in user-written LOOPS code. These give you the capability to interrupt or trace methods so that you can examine the state of the computations by using the Interlisp-D Break package; see the *Interlisp-D Reference Manual*.

In addition to being able to break and trace methods, you can break and trace accesses to data within objects. For example, you can determine when a process is attempting to change a class variable or is trying to read the value of an instance variable. This feature gives you a powerful tool to assist in the understanding of the behavior of a piece of code from both a functional view and a data view.

12.1 Breaking and Tracing Methods

The Interlisp-D environment provides a number of features for breaking and tracing functions. LOOPS methods are implemented as Lisp functions, so the breaking and tracing of method invocation is similar to Interlisp-D.

The following table describes the methods in this section.

Name	Type	Description
BreakMethod	Method	Breaks a method of a class.
TraceMethod	Method	Traces a method of a class.
UnbreakMethod	Method	Unbreaks or untraces a method of a class.
SelectorsWithBreak	Method	Returns a list of selectors whose implementations have a break.

(← *self* **BreakMethod** *selector*) [Method of Class]

Purpose: Breaks a method of a class.

Behavior: Varies according to the argument.

- If *selector* is NIL, a menu appears showing the selectors associated with the class *self* that have not already been broken. If you do not make a choice from the menu, this method returns the symbol **NothingBroken**.
- If *selector* is non-NIL and is not associated with *self*, an error occurs stating that *selector* was not found in *self*.

If a method is broken, this fact is printed in the Prompt Window. The broken method function is added to the list **BROKENFNS**. (See the *Interlisp-D Reference Manual* for more information on **BROKENFNS**.)

Arguments: *self* Must be bound to a class.

selector Must be a selector associated with *self* or NIL.

Returns: The symbol **NothingBroken** or NIL.

Categories: Class

Example: The following command causes a break when the message **Open** is sent to any window:

```
12←(← ($ Window) BreakMethod 'Open)
```

(← *self* **TraceMethod** *selector*) [Method of Class]

Purpose: Traces a method of a class.

Behavior: Varies according to the argument.

- If *selector* is NIL, a menu appears showing the selectors associated with the class *self* that have not already been broken. If you do not make a choice from the menu, this method returns the symbol **NothingTraced**.
- If *selector* is non-NIL and is not associated with *self*, an error occurs stating that *selector* was not found in *self*.

If a method is traced, this fact is printed in the Prompt Window. The traced method function is added to the list **BROKENFNS**. (See the *Interlisp-D Reference Manual* for more information on **BROKENFNS**.) Whenever the function is called a message will be printed to a trace window, when it is exited a message will be printed with the returned value.

Arguments: *self* Must be bound to a class.

selector Must be a selector associated with *self* or NIL.

Returns: The symbol **NothingTraced** or NIL.

Categories: Class

(← *self* **UnbreakMethod** *selector*) [Method of Class]

Purpose: Unbreaks or untraces a method of a class.

Behavior: Varies according to the argument.

- If *selector* is NIL, a menu appears showing the selectors associated with the class *self* that have been broken. If you do not make a choice from the menu, this method returns the symbol **NothingUnbroken**.
- If *selector* is non-NIL and is not associated with *self*, an error occurs stating that *selector* was not found in *self*.

If a method is unbroken, its method function is removed the list **BROKENFNS**. (See the *Interlisp-D Reference Manual* for more information on **BROKENFNS**.) The value return is a list containing the name of the unbroken method function.

Arguments: *self* Must be bound to a class.

selector Must be a selector associated with *self* or NIL.

Returns: The symbol **NothingUnbroken** or a list containing the name of the unbroken method function.

Categories: Class

(← *self* **SelectorsWithBreak**) [Method of Class]

Purpose: Return a list of selectors whose implementations have a break.

Behavior: Searches through the list **BROKENFNS** collecting all selectors of method functions that begin with the class name of *self*. (See the *Interlisp-D Reference Manual* for more information on **BROKENFNS**.)

Arguments: *self* Must be bound to a class.

Returns: A list of selectors of *self*.

Categories: Class

12.2 BREAKING AND TRACING DATA

12.2 Breaking and Tracing Data

Breaking or tracing functions or methods cause interruptions to occur in a computation when a function or method is entered. Breaks or traces on data can be made to occur when either the data is to be read or changed. Only data that is contained within objects can be broken; this feature is not available to arbitrary Lisp data. Breaks and traces on data are implemented through the mechanism of active values. The following **ActiveValue** classes contain this mechanism:

- **BreakOnPut**
- **BreakOnPutOrGet**
- **TraceOnPut**
- **TraceOnPutOrGet**

You can use the methods and functions in this section to place or remove breaks on data. You can also add and remove traces and breaks through the inspector interface. See Chapter 18, User Input/Output Modules, for more information on the inspector.

Note: Breaking or tracing a variable effectively breaks or traces any IndirectVariable that points to it.

The following table describes the items in this section.

Name	Type	Description
BreakIt	Method	Puts a break on data within an object.
BreakIt	Function	Sends the message **BreakIt** to *self*.
TraceIt	Method	Puts a trace on data within an object.
TraceIt	Function	Sends the message **TraceIt** to *self*.
UnBreakIt	Function	Unbreaks broken data; untraces traced data.
BrokenVariables	Global Variable	Contains a list of broken or traced variables.

(← *self* **BreakIt** *varName propName &OPTIONAL (type 'IV) breakOnGetAlsoFlg*) [Method of Object]

Purpose: Puts a break on data within an object.

Behavior: Adds an entry to the list **BrokenVariables**.

- If *breakOnGetAlsoFlg* is T, creates an instance of the class **BreakOnPutOrGet** and adds the active value to the data specified by *self*, *varName*, *propName*, and *type*.
- If *breakOnGetAlsoFlg* is NIL, the active value instance is of the class **BreakOnPut**.

When a break occurs, the break window shows the nature of the break and which object and what variable is broken. See examples below.

Arguments:

self Points to the object that contains the data to be broken.

varName The name of the variable.

propName If a property access is to be broken, this is the name of the property.

type The type of the data. This can be IV, CV, or METHOD; the default is IV.

breakOnGetAlsoFlg
If this is non-NIL, breaks will occur when data is read. If this is NIL, breaks will occur only on attempts to write the data.

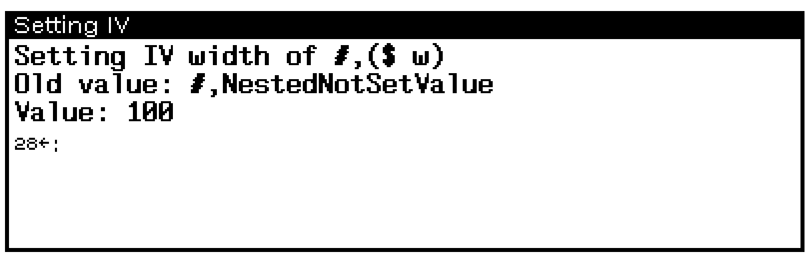
Returns: *self*

Categories: Object

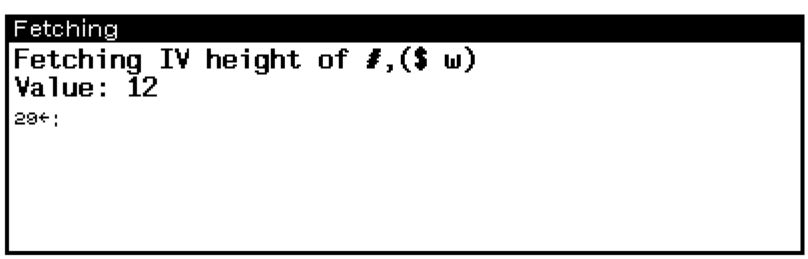
Example: The following commands check if a window's width and height are going to change.

```
(← ($ Window) New 'w)
(← ($ w) BreakIt 'width)
(← ($ w) BreakIt 'height NIL NIL T)
```

Trying to change the width causes this break:



Trying to read the height causes this break:



(**BreakIt** *self varName propName type breakOnGetAlsoFlg*) [Function]

Behavior: Sends the message **BreakIt** to *self* passing the remainder of the arguments. See the method **BreakIt**, above, for details.

(← *self* **TraceIt** *varName propName &OPTIONAL (type 'IV) traceGetAlsoFlg*) [Method of Object]

Purpose: Puts a trace on data within an object.

Behavior: Adds an entry to the list **BrokenVariables**.

- If *traceGetAlsoFlg* is T, creates an instance of the class **TraceOnPutOrGet** and adds the active value to the data specified by *self*, *varName*, *propName*, and *type*.
- If *traceGetAlsoFlg* is NIL, the active value instance is of the class **TraceOnPut.**

When a trace occurs, a trace window appears if necessary, with the traced information printed in it. See examples below.

Arguments:

self Points to the object that contains the data to be traced.

varName The name of the variable.

propName If a property access is to be traced, this is the name of the property.

type The type of the data. This can be IV, CV, or METHOD; the default is IV.

traceOnGetAlsoFlg
If this is non-NIL, trace messages will occur when data is read. If this is NIL, trace messages will occur only on attempts to write the data.

Returns: *self*

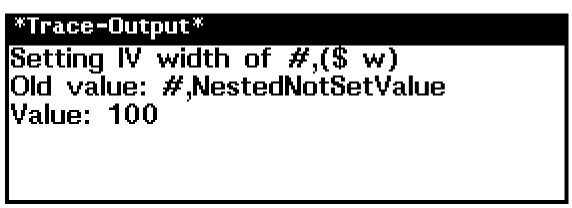
Categories: Object

Examples: To monitor if a window's width and height are going to change, enter

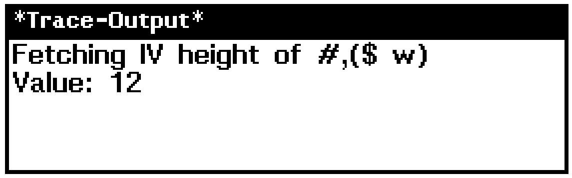
```
97←(← ($ Window) New 'w)
#,($& Window (NEW0.1Y%:.;h.eN6 . 495))

98←(← ($ w) TraceIt 'width)
#,($& Window (NEW0.1Y%:.;h.eN6 . 495))

99←(← ($ w) TraceIt 'height NIL NIL T)
#,($& Window (NEW0.1Y%:.;h.eN6 . 495))
```

Trying to change the width or the height causes a trace.

```
100←(change (@ ($ w) width) 100)
```




```
100

101←(@ ($ w) height)
```




```
12
```

(**TraceIt** *self varName propName type breakOnGetAlsoFlg*) [Function]

Purpose/Behavior: Sends the message **TraceIt** to *self* passing the remainder of the arguments. See the method **TraceIt**, above, for details.

(**UnBreakIt** *self varName propName type*) [Function]

Purpose: Unbreaks broken data; untraces traced data.

Behavior: Varies according to the argument *self*.

- If *self* is NIL, iterates through the list **BrokenVariables** and removes the active values from the objects on that list. **BrokenVariables** is set to NIL.
- If *self* is not NIL, removes the active value from the data described by *self*, *varName*, *propName*, and *type*. The corresponding entry is removed from **BrokenVariables**. If there is no active value on the specified data, a break occurs saying that the specified data is not broken and type OK to continue.

Arguments:

self	Points to the object that contains the data to be traced.
varName	The name of the variable.
propName	If a property access is to be traced, this is the name of the property.
type	The type of the data. This can be IV, CV, or METHOD; the default is IV.

Returns: Value depends on the arguments.

- If *self* is NIL, the value of **BrokenVariables** before it was bound to NIL is returned.
- If *self* is non-NIL and there were no errors, the list containing *self*, *varName*, and *propName* is returned.

Example: The following command removes a break from the instance variable **id#** in the instance named **Datum12**:

```
(UnBreakIt ($ Datum12) 'id#)
```

BrokenVariables [Global Variable]

Purpose/Behavior: This is initialized to NIL. As data within objects is traced or broken, an entry is added to this list. Each entry contains the object, the variable name, the active value created to implement the break, the property name, and the type.

[This page intentionally left blank]

13. EDITING

LOOPS has an interface to the display-based editor, SEdit. This editor is most often used for modifying classes, functions, and methods, but it can also be invoked to modify instances. Instances are typically modified through the inspector interface (see Chapter 18, User Input/Output Modules).

The technique for editing methods is exactly the same as that for editing functions. LOOPS uses Method, an extension of the lambda form that is documented in Chapter 6, Methods, which also gives details about the form of methods and how to invoke the editor upon them.

The process of editing classes and instances is different from editing methods in that you are not editing structure directly. The data structures representing the objects are translated into list structures, those list structures are then edited, and, finally, on exiting from the editor, the list structure is translated back into LOOPS objects. Because of this process, changes in objects do not take effect until you have exited from the editor.

13.1 Editing Classes

The **Edit** and **Edit!** methods provide a screen-based way to modify class structure. You can quickly add and delete local class and instance variables, make inherited variables local, and change initial values. The other methods listed are used to interface LOOPS to the display editor.

Name	Type	Description
Edit	Method	Edits the structure of a class.
Edit!	Method	Edits a class in a form that includes inherited information.
InstallEditSource	Method	Makes a class conform to a description.
MakeEditSource	Method	Makes a list structure for editing a class.
MakeFullEditSource	Method	Makes a list structure, including inherited information, for editing a class.

(← *self* **Edit** *commands*) [Method of Class]

Purpose: Edits the structure of a class.

Behavior: Translates *self* from a data type to a list structure that is then passed to the editor through **EDITE** (see the *Lisp Release Notes* and the *Interlisp-D Reference Manual*). If *commands* is non-NIL, this is passed as the second argument to **EDITE**.

The variable **LASTCLASS** is bound to the class name of *self*.

Generally, *commands* is NIL, which causes you to enter the editor interactively. From this point, you can perform the following actions:

- Change the metaclass of *self*
- Add or delete class properties
- Add or delete class variables, instance variables, and their properties.

Also, as a user convenience, the edited form has a list of methods that you can select and edit, although you cannot delete items from this **MethodFns** list and have that action disassociate the methods from the class.

Arguments: *commands* A list of editing commands to be passed to EDITE.

Returns: The class name of *self*.

Categories: Object

Specializes: Object

Example: The following editing window is generated as a result of

```
(← ($ IndirectVariable) Edit)
```

```
SEdit #,($C IndirectVariable) Package: INTERLISP
((MetaClass Class doc
            (* Active Value for redirecting references to
               another variable)
            Edited%: (* smL " 9-May-86 09:52"))
 (Supers ActiveValue) (ClassVariables)
 (InstanceVariables
       (object NIL doc
               (* The object with the "real" variable))
       (varName NIL doc (* The name of the "real" variable))
       (propName NIL doc
                 (* The prop name of the "real" variable))
       (type NIL doc (* The type of the "real" variable)))
 (MethodFns IndirectVariable.GetWrappedValueOnly
            IndirectVariable.PutWrappedValueOnly))
```

The following information describes this window:

- The title of the window contains the name of the class being edited and package it uses for displaying symbols. This package should be INTERLISP when using LOOPS.
- It has the metaclass **Class**.
- It has the two class properties **doc** and **Edited%:**.
- It has one super class, the class **ActiveValue**.
- It has no class variables.
- It has four instance variables: **object, varName**, **propName**, and **type**. Each instance variable has a **doc** property.
- It has two local methods: **GetWrappedValueOnly** and **PutWrappedValueOnly**.

(← *self* **Edit!** *commands*) [Method of Class]

Purpose: Edits a class in a form that includes inherited information.

Behavior: This is similar is behavior to the method **Class.Edit**.

The form you are editing includes not only items local to *self* but also class variables and instance variables that are inherited. This allows you to easily move inherited information into *self*. Editing operations that modify the inherited values have no effect.

Arguments: *commands* A list of editing commands to be passed to **EDITE**.

Returns: The class name of *self*.

Categories: Class

Example: The following command puts you into the display editor. Compare this display with the previous one.

```
(← ($ IndirectVariable) Edit!)
```

```
SEdit #,($C IndirectVariable) Package: INTERLISP
((MetaClass Class doc
            (* Active Value for redirecting references to
               another variable)
            Edited%: (* smL " 9-May-86 09:52"))
 (Supers ActiveValue) (ClassVariables) (CVsInherited)
 (InstanceVariables
       (object NIL doc
               (* The object with the "real" variable))
       (varName NIL doc (* The name of the "real" variable))
       (propName NIL doc
                 (* The prop name of the "real" variable))
       (type NIL doc (* The type of the "real" variable)))
 (IVsInherited))
```

(← *self* **InstallEditSource** *editedDescription*) [Method of Class]

Purpose: Makes a class conform to a description.

Behavior: Called by the system to change a class data structure to correspond to a list structure you have edited. If there are errors in the structure, the editor is activated again. If there are errors in the edited structure, an error message is printed in the prompt window and you are returned to the editor to fix it.

If there are no errors in the structure, this successfully translates the structure into the class data type structure. In addition, a class property **Edited:** is added to *self* with the value returned by (**EDITDATE** NIL INITIALS).

Arguments: *editedDescription*
A list structure similar to that returned by the message **MakeEditSource**.

Returns: Used for side effect only.

Categories: Object

Specializes: Object

(← *self* **MakeEditSource**) [Method of Class]

Purpose: Makes a list structure for editing a class.

Behavior: This builds a list structure containing metaclass, super, class variable and instance variable information. In addition, the method function names are included in this list.

Returns: List expression of class structure.

Categories: Object

Specializes: Object

Example: The command

```
(← ($ BreakOnPutOrGet) MakeEditSource)
```

returns

```
((MetaClass Class Edited%: (* nbm  " 5-May-87 17:53")
doc "This is the default metaClass for all classes")
 (Supers BreakOnPut)
 (ClassVariables)
 (InstanceVariables)
 (MethodFns BreakOnPutOrGet.GetWrappedValue))
```

(← *self* **MakeFullEditSource**) [Method of Class]

Purpose: Makes a list structure, including inherited information, for editing a class.

Behavior: This is similar to **MakeEditSource**. The constructed list also includes instance variables and class variables that are inherited.

The list does not contain the method functions associated with self that **MakeEditSource** includes.

Returns: List expression of class structure.

Categories: Class

Example: The command

```
(← ($ BreakOnPutOrGet) MakeFullEditSource)
```

returns:

```
((MetaClass Class Edited%: (* nbm  " 5-May-87 17:53")
doc "This is the default metaClass for all classes")
 (Supers BreakOnPut)
 (ClassVariables)
 (CVsInherited)
 (InstanceVariables)
 (IVsInherited
   (localState NIL doc (* The local state of the active value))))
```

13.1 EDITING INSTANCES

13.1 EDITING INSTANCES

13.2 Editing Instances

LOOPS instances can also be edited by the standard Medley code editor. From this editor, you can change the values of instance variables and properties, and add new instance variables. Instances follow the same basic editing protocol that classes do.

The following table shows the methods in this section.

Name	Type	Description
Edit	Method	Allows you to change the values contained in an instance.
InstallEditSource	Method	Makes an instance conform to a description.
MakeEdit	Method	Makes a list structure for editing an instance.

(← *self* **Edit** *commands*) [Method of Object]

Purpose: Allows you to change the values contained in an instance.

Behavior: Changes the data structure of *self* to a list and passes that list to **EDITE** (see the *Lisp Release Notes* and the *Interlisp-D Reference Manual*.) If *commands* is non-NIL, this is passed as the second argument to **EDITE**.

Deleting a variable does not delete it from the instance.

Arguments: *commands* A list of editing commands to be passed to **EDITE**.

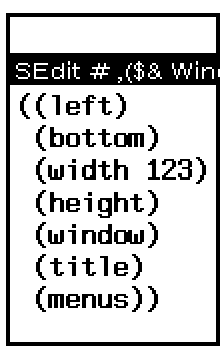
Returns: *self*

Categories: Object

Specializations: Class

Example: The following commands create the edit window shown.

```
71←(← ($ Window) New 'w1)
#,($& Window (NEW0.1Y%:.;h.eN6 . 495))

72←(←@ ($ w1) width 123)
123

73←(←($ w1) Edit)
```



(←*self* **MakeEditSource**) [Method of Object]

Purpose: Makes a list structure for editing an instance.

Behavior: Returns a list showing all instance variables, values, and properties.

Returns: A list showing all instance variables, values, and properties.

Categories: Object

Specializations: Class

Example: The following shows **MakeEditSource** results as values are assigned to the instance variables of ($ w1).

```
38←(← ($ Window) New 'w1)
#,($& Window (NEW0.1Y%:.;h.eN6 . 495))

39←(← ($ w1) MakeEditSource)
((left) (bottom) (width) (height) (window) (title) (menus))

40←(←@ ($ w1) width 123)
123

41←(← ($ w1) MakeEditSource)
((left) (bottom) (width 123) (height) (window) (title) (menus))

42←(←@ ($ w1) menus Title T)
T

43←(← ($ w1) MakeEditSource)
((left) (bottom) (width 123) (height) (window) (title) (menus
#,NotSetValue Title T))

44←(@ ($ w1) window)
{WINDOW}#74,25554

45←(← ($ w1) MakeEditSource)
((left 31) (bottom 407) (width 123) (height 12) (window #,($AV
LispWindowAV ((NEW0.1Y%:.;h.eN6 . 495)) (localState {WINDOW}#74,25554)))
(title) (menus #,NotSetValue Title T))
```

(← *self* **InstallEditSource** *editedDescription*) [Method of Object]

Purpose: Makes an instance conform to a description.

Behavior: This is called by the system to change an instance data structure to correspond to a list structure you have edited.

Arguments: *editedDescription*
A list structure similar to that returned by the message **MakeEditSource**.

Returns: Used for side effect only.

Categories: Object

[This page intentionally left blank]

14. FILE MANAGER

LOOPS data structures are fully integrated into Medley. This includes the definition of new File Manager commands so that any LOOPS object or method can be saved on files and loaded into the environment in exactly the same way that normal Medley data types are saved and loaded.

In addition, the LOOPS file browser provides a menu-driven interface to the File Manager. When using a LOOPS file browser, newly created objects are associated with files automatically. If you are not familiar with LOOPS file browsers see Chapter 10, Browsers.

This chapter describes the functions, methods, and variables used to load and store files containing LOOPS objects. It describes the File Manager commands related to LOOPS objects. It also describes how to add objects to files, delete them from files, and move them from file to file. These are primarily of interest when customizing either the File Manager or LOOPS file browser.

14.1 Manipulating Files

LOOPS takes advantage of the ability to create user-defined File Manager commands to fully integrate LOOPS into the Medley environment. As a result, the same steps used to manipulate files containing Medley data structures are used to manipulate files containing LOOPS data structures. Furthermore, both LOOPS and Medley data structures can be saved together in the same file. This section contains a brief review of the three basic functions used to manipulate files. For a more detailed description which includes additional functions, see the *Lisp Release Notes* and the *Interlisp-D Reference Manual*.

In addition, there is a LOOPS file browser which provides a convenient way of loading files and guaranteeing that newly created classes and methods are associated with files during the development of LOOPS programs. The LOOPS file browser is different from the Lisp Library Module **FILEBROWSER**. Files can be loaded and put into new or existing file browsers by a series of menu selections.

You can manipulate files with these basic steps:

- Assign data structures to a specific file using **FILES?**.
- Write data structures to a file using **MAKEFILE**.
- Enter data structures stored in a file into the environment using **LOAD**.

The following example shows these steps.

```
30←(FILES?)
PIPEANDTANK,LOOPSPRINT,LOOPSUTILITY...to be dumped.
   plus the instances: FFAV1,Datum1,TestW
   plus the class definitions: Datum
   want to say where the above go ? Yes
(instances)
FFAV1  File name: LOOPSFILE
create new file LOOPSFILE ? Yes
Datum1   File name: LOOPSFILE
TestW   File name: LOOPSFILE
(class definitions)
Datum   File name: LOOPSFILE
NIL

31←(MAKEFILE 'LOOPSFILE)
Copyright owner for file LOOPSFILE: XEROX
{DSK}<LISPFILES>LOOPSFILE.;1

32←(LOAD 'LOOPSFILE)
{DSK}<LISPFILES>LOOPSFILE.;1
FILE CREATED  7-Jan-87 16:25:24
LOOPSFILECOMS
{DSK}<LISPFILES>LOOPSFILE.;1
```

See the *Lisp Release Notes* and the *Interlisp-D Reference Manual* for more information on **FILES?** and **MAKEFILE**. See the following section for details on **LOAD**.

14.2 LOADING FILES

14.2 Loading Files

The following table shows the functions and commands described in this section.

Name	Type	Description
LOAD	Function	Loads Medley symbolic files which includes LOOPS objects and methods.
LOADFNS	Function	Allows selective loading from Medley symbolic files.
UNDO	Prog. Asst.	Undoes previous entries into the Medley Executive which are stored on a history list, including calls to **LOAD**.

(**LOAD** *FILE LDFLG*) [Function]

Purpose/Behavior: Loads Medley symbolic files which includes all LOOPS objects and methods; see the *Lisp Release Notes* and the *Interlisp-D Reference Manual*.

Arguments: *FILE* File to be loaded.

LDFLG Alters the effect of loading a file.

- If it is set to **PROP**, the definitions of functions, including **METHOD** functions, are stored on the property **EXPR** of the function name. Thus, any existing definitions are not overwritten.

- If it is set to **ALLPROP**, the values of variables are also saved on property lists.

Returns: Full file name.

(**LOADFNS** *FNS FILE*) [Function]

Purpose/Behavior: Allows selective loading from Medley symbolic files including LOOPS files . The most likely use for this facility is to load the source code for method functions when the compiled versions are already loaded. The methods must be specified by their explicit function names in the form **ClassName.Selector**, for example,

```
(LOADFNS '(SomeClass.AMethod OtherClass.AMethod) '{DSK}<LISPFILES>SOMEFILE 'PROP)
```

It is not recommended that LOOPS objects be selectively loaded by using **VARS** (see the *Lisp Release Notes* and the *Interlisp-D Reference Manual*), because it is not possible to guarantee that all necessary related objects, such as superclasses or methods of a class, are also loaded.

Arguments:

FNS	Selected functions to be loaded.
FILE	File from which functions specified in **FNS** are to be loaded.

Returns: List of functions that have been loaded

UNDO [Program Assistant Command]

Purpose/Behavior: LOOPS saves enough information about objects that are created as a result of loading a file to allow the call to **LOAD** to be undone. The objects are destroyed and any preexisting objects that were deleted by the load are restored. See the *Lisp Release Notes* and the *Interlisp-D Reference Manual*.

14.3 LOOPS FILE PACKAGE COMMANDS

14.3 LOOPS File Manager Commands

Four File Manager types are defined to allow LOOPS objects to be stored in Medley files:

- CLASSES
- METHODS
- INSTANCES
- THESE-INSTANCES

These types and the functions and methods used by LOOPS to process these types are described in this section.

Note: The order of items in the filecoms is important. In particular, class definitions must appear in the file before any methods on that class or any instances of that class. Similarly, methods on a class must appear before any instances of that class.

Name	Type	Description
CLASSES	File Mgr Command	Writes the appropriate **DEFCLASSES** and **DEFCLASS** expressions for the named classes.
DEFCLASSES	NLambda NoSpread	Creates a series of empty classes in preparation for reading their definitions via **DEFCLASS**.

DEFCLASS	NLambda NoSpread	Takes a source specification of a class from a file and causes the appropriate internal representation to be constructed.
METHODS	File Mgr Command	Writes the appropriate **METH** and **DEFINEQ** expressions for each method object and its associated function.
METH	NLambda NoSpread	Creates a method object and attaches it to the appropriate class.
INSTANCES	File Mgr Command	Writes the appropriate **DEFINST** expressions for each instance in the list.
THESE-INSTANCES	File Mgr Command	Appears as a sublist in a filecoms.
DEFINSTANCES	NLambda NoSpread	Creates empty structures for each instance name in a list.
DEFINST	NLambda NoSpread	Creates internal representations for source specifications of an instance.
FileIn	Method	Creates internal representations for source specifications of an instance.

(**CLASSES** *ClassName1...ClassNameN*) [File Manager Command]

Purpose/Behavior: Appears as a sublist in a filecoms. The keyword **CLASSES** tells the File Manager to use the appropriate **DEFCLASSES** and **DEFCLASS** expressions for the named classes when writing to a file.

Arguments: *ClassName* Accepts any symbol, but only gives meaningful result when you use **DEFCLASS** to actually create the class.

Example: `(CLASSES Myclass)`

(**DEFCLASSES** *CLASSES*) [NLambda NoSpread Function]

Purpose/Behavior: Used in a file to create a series of empty classes in preparation for reading in their definitions via **DEFCLASS**. This allows the classes to be read in any order. Otherwise, superclasses would have to be read in before their subclasses.

Arguments: *CLASSES* Accepts any symbol, but only gives meaningful result when you use **DEFCLASS** to actually create the class.

Returns: NIL

Example: The command

```
(DEFCLASSES MyClass)
```

returns `NIL`.

(**DEFCLASS** *FORM*) [NLambda NoSpread Function]

Purpose/Behavior: Takes a source specification of a class, such as produced by the method **MakeFileSource**, from a file and causes the appropriate internal representation to be constructed.

Arguments: *FORM* The source specification of a class.

Returns: NIL

Example:

```
(DEFCLASS MyClass
(MetaClass Class doc (* Something for my project)
          Edited:  (* nbm "18-Oct-87 13:20"))
(Supers Object)
(InstanceVariables ( Iv1 (22) doc
          (* Initial value for my instances)]
```

(**METHODS** *ClassName.Message1...ClassName.MessageN*) [File Manager Command]

Purpose/Behavior: Appears as a sublist in a filecoms. The keyword **METHODS** tells the File Manager to use the appropriate **METH** and **DEFINEQ** expressions for each method object and its associated function.

Arguments: *ClassName.Message*
The source specification of a class.

Example:

```
(METHODS  MyClass.Method1)
```

(**METH** *methDescr*) [NLambda NoSpread Function]

Purpose/Behavior: Creates a method object and attaches it to the appropriate class.

Arguments: *methDescr* Method object to create.

Returns: NIL

Example:

```
(METH MyClass  MyClass.Method1 NIL
(category (Datum)))
```

(**INSTANCES** *InstName1...InstNameN*) [File Manager Command]

Purpose/Behavior: Appears as a sublist in a filecoms. The keyword **INSTANCES** tells the File Manager to use the appropriate **DEFINST** expressions for each instance in the list and also for any other instances that are referenced inside any instances in the list. This assures that there are no references to nonexistent instances when read back in. The method **SaveInstance?** can be specialized to prevent instances from being saved in more than one file when they are referred to by instances in different files.

Example:

```
(INSTANCES  TestW)
```

(**THESE-INSTANCES** *InstName1...InstNameN*) [File Manager Command]

Purpose/Behavior: Appears as a sublist in a filecoms. The keyword **THESE-INSTANCES** tells the File Manager to use the appropriate **DEFINST** expressions for each instance in the list. Unlike the **INSTANCES** File Manager command, **THESE-INSTANCES** does not recursively dump instances that are pointed by *InstName1...InstNameN*.

(**DEFINSTANCES** *Instances*) [NLambda NoSpread Function]

Purpose/Behavior: Takes a list of instance names and creates empty structures for them in preparation for reading in their structures from a file.

Arguments: *Instances* Accepts any symbol but result is useless unless you use **DEFINST** to actually create the *Instance*.

Returns: NIL

Example:
```
(DEFINSTANCES TestW)
```

(**DEFINST** *DEFINST% FORM*) [NLambda NoSpread Function]

Purpose/Behavior: Takes a source specification of an instance and causes the appropriate internal representation to be created. It does this by sending the message **FileIn** to the instance's class. It creates the class if it does not exist.

Arguments: *DEFINST% FORM* The source specification of an instance.

Returns: NIL

Example:
```
[DEFINST Window
  (TestW (JEW0.0X:.H<4.NZ9 . 532))
  (left 179)
  (bottom 446)
  (width 12)
  (height 12)]
```

(← *self* **FileIn** *fileSource*) [Method of Class]

Purpose/Behavior: Takes a source specification for an instance as it appears in a file and causes the appropriate internal representation to be constructed.

Arguments: *self* Class of the instance to be created.

fileSource Loadable form of an instance as stored in a file.

Returns: *self*

Categories: Class

14.4 SAVING LOOPS OBJECTS ON FILES

14.4 Saving LOOPS Objects on Files

Adding LOOPS classes, methods and instances to files can be done in the same way that functions and variables are saved in Medley. In addition, the LOOPS browser allows newly created objects to be automatically associated with files. LOOPS also provides the means for moving objects from file to file.

Whenever a class, method, or named instance is created or edited, it is marked as changed. This allows the File Manager to prompt for a file in which to store new objects and see to it that changed objects are written out when **MAKEFILE** is called.

The following table shows the items in this section.

Name	Type	Description
FILES?	Function	LOOPS adds a prompt for classes, methods and instances along with the normal Medley types.
ObjectModified	Method	Notifies the File Manager that an object has been changed or created.
OnFile	Method	Determines if a class is in **FILELST**.
SaveInstance	Method	Causes newly created instances to be noticed by the File Manager.
SaveInstance?	Method	Determines if an instance needs to be added to the list of instances to be saved.
DelFromFile	Method	Deletes an object from any file in **FILELST** in which it appears.
MoveToFile	Method	**Class**.**MoveToFile** moves a class and its methods from one file to another. **Object**.**MoveToFile** moves an instance from one file to another.
MoveToFile!	Method	Moves a class, all of its methods, and all of its subclasses and their methods from one file to another.
DontSave	IVProperty	Controls what parts of an instance are saved in a file.
OldInstance	Method	Sends a message to an object after it is loaded from a file.

(**FILES?**) [Function]

Purpose/Behavior: The File Manager types have been extended so that, when a call is made to **FILES?**, you are prompted to add classes, methods and instances to files along with the normal Medley. For an example of **FILES?**, see Section 14.1, "Manipulating Files."

After a class is associated with a file, any methods that are added to it are automatically added to that file as well. Thus, it makes sense to put classes in files as soon as possible. This could be done by repeated calls to **FILES?**, but the LOOPS file browser allows classes to be automatically added to files as they are created. Any class that is created by adding a root to a file browser or by specializing a class in a file browser is added to that brower's file. If more than one file is associated with the browser, a menu appears to prompt you to specify a file for the new class. The LOOPS browser also can be used to create a new file and associate it with a file browser. Thus, there is never any need to wait until the end of a session to put classes and methods in files.

You can also save instances on files. Of course, only those instances which should be present after a file is first loaded should be saved. Instances which are constructed "on the fly" as a consequence of running a LOOPS program should not be saved. Only named instances are marked as changed so many such temporary instances may never be noticed. However, if named instances which should not be saved are created, then you are prompted to put them into files after a call to **FILES?** and must respond by typing a right square bracket (]) to each one. Alternatively, it is possible to specialize the method **ObjectModified** so that it does not call **MARKASCHANGED**. Then any instances of classes which have or inherit the specialized method are not noticed by the File Manager regardless of whether or not they are named.

(← *self* **ObjectModified** *name*) [Method of Object]

Purpose: Notifies the File Manager that an object has been changed or newly created.

Behavior: Uses the File Manager command **MARKASCHANGED**. It does nothing if *name* is not given, thus unnamed objects are never marked.

Arguments: *self* A LOOPS object.

name Name of object specified in *self*.

reason Reason is **MARKEDASCHANGED** (see the *Interlisp Reference Manual* for information on **MARKEDASCHANGED**).

Returns: *self*

Categories: Object

Specializations: Method

(← *self* **OnFile** *file*) [Method of Class]

Purpose: Determines if an object is in a file in **FILELST**.

Behavior: Calls **WHEREIS** (see the *Lisp Release Notes* and the *Interlisp-D Reference Manual*).

- If *file* is not given, it returns the name of the file in **FILELST** that the object is contained in or NIL if *self* is not in a file.
- If *file* is given, it must still be a member of **FILELST**, and T or NIL is returned.

Arguments: *self* A LOOPS object.

file The file to be searched.

Returns: Value depends on the arguments; see **Behavior**.

Categories: Class

(← *self* **SaveInstance** *name reason*) [Method of Object]

Purpose: Causes newly created instances to be noticed by the File Manager.

Behavior: Sends *self* the message **ObjectModified**.

Arguments: *self* A LOOPS object.

name Name of object specified in *self*.

reason Reason is **MARKEDASCHANGED** (see the *Interlisp Reference Manual* for information on **MARKEDASCHANGED**).

Returns: *self*

Categories: Object

(← *self* **SaveInstance?** *file outInstances*) [Method of Object]

Purpose: Determines whether an instance needs to be added to the list of instances to be saved in *file*.

Behavior: Checks to see if the current instance is a member of *outInstances*. It is used by the LOOPS File Manager command **INSTANCES** to guarantee that the same instance does not appear more than once in a given file.

This method must be specialized to be used; it cannot be used directly by the user.

Arguments: *self* A LOOPS object.

file The file to be searched.

outInstances A list of LOOPS names. See Behavior.

Returns: T if the instance should be saved on the file; NIL if it should not be saved.

Categories: Object

(← *self* **DelFromFile**) [Method of Object]

Purpose: Deletes an object from any file in **FILELST** in which it appears.

Behavior: Searches through the filecoms of all files in **FILELST** and deletes the object everywhere it appears.

Arguments: *self* A LOOPS object.

Returns: Used for side effect only.

Categories: Object

Specializations: Class, Method

(← *self* **MoveToFile** *file*) [Method of Class]

Purpose: Moves an object from one file to another. If an object is a class, it, and all its methods, move.

Behavior: Adds the object to the filecoms of *file* so that the object will be saved on that file. If *file* is NIL, it prompts for a file form **FILELST** via a menu.

Arguments: *self* A class or method.

file File to which object is moving.

Returns: NIL

Categories: Object

Specializes: Object

(← *self* **MoveToFile!** *file fromFiles*) [Method of Class]

Purpose: Moves a class, all of its methods, and all of its subclasses and their methods from one file to another.

Behavior: Similar to **MoveToFile**.

Arguments: *self* A LOOPS class.
file File to which object is moving.
fromFiles A list of files from which classes may be moved.

Returns: NIL

Categories: Class

DontSave [IV Property Name]

Purpose/Behavior: Controls what parts of an instance are saved in a file. Its value is a list of property names of the instance variable which should not be written out when the instance is dumped. If **Value** is in the list, the instance variable's value is not saved. If the property is **Any**, nothing is saved except the instance variable name. (Must be added by the user.)

(←*self* **OldInstance** *name arg1 arg2 arg3 arg4 arg5*) [Method of Object]

Purpose: Sends a message to an object after it is loaded from a file. This method can be specialized by applications that need to perform some operation on every object when it is created.

Behavior: If *name* is non-NIL, the message **SetName** is sent to *self*.

Instance variables with an **:initForm** property are filled. See the discussion of **:initForm** in Chapter 2, Instances.

Sends the message **SaveInstance** to *self* with the arguments *name*, *arg1*, and *arg2*.

Arguments:

self	Evaluates to a class.
name	LOOPS name of the class or instance.
arg1...arg5	Optional arguments referenced by user-written specialization code.

Categories: Object

Specializations: IndexedObject

14.5 STORING FILES

14.5 Storing Files

This section describes the functions and methods used by LOOPS and Medley to store files.

Name	Type	Description
MAKEFILE	Function	Writes files that contain Medley data types which include LOOPS objects and methods.
PrettyPrintClass	Function	Prints classes in a file in a form that can be read back in.
PrettyPrintInstance	Function	Prints instances in a file in a form that can be read back in.
MakeFileSource	Method	Constructs the representation of an object that is appropriate for printing in a file.
FileOut	Method	Controls the printing of a LOOPS object in a file.

(**MAKEFILE** *FILE*) [Function]

Purpose/Behavior: When all LOOPS objects are associated with their files, the files are written by a call to **MAKEFILE** or **MAKEFILES**. This is identical to the standard use of **MAKEFILE** in Medley. See the *Lisp Release Notes* and the *Interlisp-D Reference Manual*.

Arguments: *FILE* Name of file to be written out.

Returns: Full file name

(**PrettyPrintClass** *className file*) [Function]

Purpose/Behavior: Used by the File Manager command **CLASSES** to print out classes in a file in a form that can be read back in. It checks to make sure the class exists and then sends it the message **FileOut**. It is also used by the method **PP** to print classes to a display stream.

Arguments: *className* The name of the class to be printed on the file *file*.

file The file on which the class *className* is to be printed.

Returns: Pointer to class in the form #,($ className)

(**PrettyPrintInstance** *instanceName file*) [Function]

Purpose: Used by the File Manager command **INSTANCES** to print instances in a file in a form which can be read back in. Sends the message **FileOut** to instance.

Arguments: *instanceName*
Name of a LOOPS instance.

file The file on which the instance i*instancename* is to be printed.

Returns: NIL

(← *self* **MakeFileSource** *file*) [Method of Object]

Purpose: Constructs the representation of an object that is appropriate for printing in a file.

Behavior: Uses the relevant access functions to obtain the parts of the object and then stores them into a list structure.

Arguments: *self* A LOOPS object.

file The file on which *self* is to be printed.

Returns: Loadable form of a LOOPS object.

Categories: Object

Specializations: Class, Method

Example:

```
63←(← ($ TestW) MakeFileSource)
       (DEFINST Window
       (TestW (NEW0.1Y%:.;h.eN6 . 501)))
```

(← *self* **FileOut** *file*) [Method of Object]

Purpose: Controls the printing of a LOOPS object in a file.

Behavior: Gets the appropriate source representation by sending the object the message **MakeFileSource** and prettyprints the result.

Arguments: *self* A LOOPS object.

file The file on which self is to be printed on if T prints to the Lisp Executive window.

Returns: *self*

Categories: Object

Specializations: Class, Method

Example:

```
62_(_ ($ TestW) FileOut T)
(DEFINST Window (TestW (NEW0.1Y%:.;h.eN6 . 501)) )
#,($& TestW (NEW0.1Y%:.;h.eN6 . 501))
```


14.6 COMPILING FILES

14.6 Compiling Files

LOOPS uses the new XAIE compiler and its macrolet facilities. When doing **CLEANUP** on LOOPS files your ***DEFAULT-CLEANUP-COMPILER*** should be set to 'CL:COMPILE-FILE. More information on this cleanup flag and the new compiler are available in the *Lisp Release Notes*.

[This page intentionally left blank]

15. PERFORMANCE ISSUES

Three main areas in LOOPS can affect performance:

- Garbage collection
- Instance variable access
- Method lookup

This chapter describes the impact of these areas on LOOPS. Also included is a section on cache clearing.

15.1 Garbage Collection

The Interlisp-D garbage collector maintains reference counts of each piece of data in the system. (Refer to the *Lisp Release Notes* and the *Interlisp-D Reference Manual* for information on reference counts.) There is potential for noticeable performance degradation if many items have reference counts greater than one. Object-oriented systems in general, and LOOPS in particular, can easily create objects that have multiple references.

The LOOPS system uses a number of methods to avoid creating items with large reference counts. Classes, for example, can easily have large reference counts since each instance of the class points to the class. Because of this, LOOPS does not maintain reference counts of classes. Performance is enhanced, but classes in LOOPS are not garbage collected. This should not present a problem as classes are not often destroyed.

Unique Identifiers (UIDs) also have multiple references: from the instance they name and from the table used by the LOOPS system to associate UIDs with instances. LOOPS avoids this problem by storing copies of the instance UID in the instance. This complicates testing for equality of UIDs, which is a rare event, but removes a potential garbage collection problem.

These and other implementation details substantially reduce the impact of LOOPS on the Interlisp-D garbage collector. In a typical running system, LOOPS objects accounted for less than 16% of the data items with reference count greater than one.

15.2 INSTANCE VARIABLE ACCESS

15.2 Instance Variable Access

LOOPS uses macros to speed the instance variable access from compiled code. Instance variable property access is compiled differently from instance variable value access, and various caching schemes are used to speed up repeated access to a given slot.

LOOPS uses two layers of caching to speed up instance variable access:

- Local cache.

  Instance variable access from compiled code uses a local cache. This cache remembers the class of *self* and the instance variable index the last time this piece of code was executed. If the class of *self* on the next pass through the code matches the stored value, then the stored instance variable index is used. In this case, instance variable access is very fast.

- Global cache.

  A global cache is used by the instance variable access functions when the local cache fails. This global cache is a fixed size table of instance variable pairs. Looking in this cache for a given class is typically faster than computing the instance variable index.

You should be aware that instance variable access is optimized to be faster than accessing the properties of instance variables. Also, be aware that when instances are first created, the data for an instance variable may need to be found by performing a lookup through the class hierarchy. If the lookup goes through several classes, this can be slow. By guaranteeing that the instance variable data is stored in the instance, this lookup delay can be avoided.

The following macros are used to access instance variables. They are mentioned here to point out that calls to **GetValue** and **PutValue** could result in the compilation of any one of several different functions.

(**GetValue** *self varName* &OPTIONAL *propName*) [Macro]

Purpose/Behavior: Compiles to a call to one of the functions **Cached-GetIVValue**, **Cached-GetIVProp**, **GetIVValue**, or **GetIVProp**. The particular function depends on details of the arguments to **GetValue**.

Arguments:

self A class or an instance.

varName Instance or class variable name.

propName Property name.

Returns: Used for side effect only.

(**PutValue** *self varName value* &OPTIONAL *propName*) [Macro]

Purpose/Behavior: Compiles to a call to one of the functions **Cached-PutIVValue**, **Cached-PutIVProp**, **PutIVValue**, or **PutIVProp**. The particular function depends on details of the arguments to **PutValue**.

Arguments:

self A class or an instance.

varName Instance or class variable name.

value The new value for *varName* or *propName*.

propName Property name.

Returns: Used for side effect only.

15.3 METHOD LOOKUP

15.3 METHOD LOOKUP

15.3 Method Lookup

LOOPS uses two layers of caching to speed the method lookup:

- Local cache

  Method lookup from compiled code uses a local cache when the selector can be determined at compile time. This cache remembers the class of *self* and the computed method the last time this message was sent. If the class of *self* on the next pass through the code matches the stored value, then the method is used. In this case, method lookup is very fast.

- Global cache

  A global cache is used by the method lookup functions when the local cache fails. This global cache is a fixed size table of class / selector / method triples. Looking in this cache for a given class and selector is typically faster than searching the class hierarchy for the appropriate method.

15.4 CACHE CLEARING

15.4 Cache Clearing

Code that directly manipulates the structure of LOOPS objects sometimes needs to invalidate the caches used for instance variable access and message sending.

The following functions can be used to clear these caches if you suspect that they might be invalid.

(**ClearAllCaches**) [Function]

Purpose/Behavior: Clears all LOOPS and Interlisp-D runtime caches. This includes local and global instance variable access caches, local and global method lookup caches, and the system CLISP translations hash array.

Returns: NIL

[This page intentionally left blank]

16. PROCESSES

LOOPS provides two special versions of message sending that start a separate process to run LOOPS methods. These are ←**Process** and ←**Process!** which are analogous to ← and ←**!**.

(←**Process** *obj sel arg1 ... argn*) [Macro]

Purpose: Starts a new process to run the selected method on the object, *obj*.

Behavior: The method indicated by *sel* is run in a separate process for the given instance or class, *obj*. See the *Interlisp-D Reference Manual* for a discussion of processes.

Arguments:
obj A LOOPS object.

sel Name of the method to be executed as a process.

arg1 ... argn
Arguments for the method specified in *sel.*

Returns: Pointer to a process data type.

Example: Assume the method **ClockTime** is added to the class **LCD**, as follows:

```
[Method ((LCD ClockTime
        self WaitTime DisplaySeconds?)
        (while T
                do (←@ self reading
                    [MKATOM (DATE (if DisplaySeconds?
                                  then (DATEFORMAT NO.DATE)
                                  else (DATEFORMAT NO.DATE NO.SECONDS])
                   (← self Update)
                   (BLOCK (OR WaitTime 1000])
        (LCD.ClockTime)
```

ClockTime takes two arguments: **WaitTime**, the wait time between updates of the **LCD** reading, and **DisplaySeconds?**, a flag used to determine if seconds are to be displayed on the **LCD**. **ClockTime** runs an infinite loop which sets the **LCD** reading, updates the **LCD** display, and blocks the **ClockTime** loop to allow other system processes to run. The command

```
(←Process ($ LCDInstance1) ClockTime 60000)
```

adds the process **ClockTime** to the process list and **($ LCDInstance1)** becomes a digital clock which updates itself every minute.

(←**Process!** *obj sel arg1 .... argn*) [Macro]

Purpose: Starts a new process to run the selected method on the object *obj*. Like ←**Process**, except the argument *sel* is evaluated.

Behavior: Evaluates *sel* returns a selector for a method of *obj*. This method is run on a separate process for the given instance or class, *obj*.

Arguments: *obj* A LOOPS object.

sel Name of the method to be executed as a process.

arg1 .... argn Arguments needed for the method.

Returns: Pointer to the process data type.

Example: Assume the variable **LCDClock** is set to **ClockTime**, which is the method added to the **LCD** class as described for ←**Process**. The command

```
(←Process! ($ LCDInstance1) LCDClock 2000 T)
```

adds the process **LCDClock** to the process list and **($ LCDInstance)** becomes a digital clock with a seconds display which updates itself every two seconds.

18. USER INPUT/OUTPUT MODULES

This chapter presents two modules that have been developed for displaying and allowing you to enter information. The Interlisp-D Inspector module and ?= handler have been enhanced to support LOOPS data types and message sending.

18.1 Inspector

The LOOPS interface uses and extends the capabilities of the Medley Inspector module. Instances and classes can be easily examined and modified through the interface that the Inspector module provides. This section describes the operations available with the LOOPS interface. For information on the Inspector module, see the *Interlisp-D Reference Manual.*

An inspector is a window opened on a specific piece of data, which for LOOPS means a class or an instance. Figure 18-1 shows an inspector on an instance of a window.

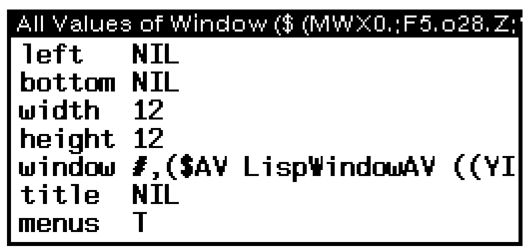


Figure 18-1. Sample Inspector

Inspector windows contain two columns of information; the left column is called the property column, and the right column is called the value column.

You can scroll inspector windows, but these windows are not reshaped by actions such as switching from instance variables inspection to property inspection or adding new instance variables. This may cause some confusion, for example, if you create an inspector to be the correct size and add an instance variable, and that instance variable fails to appear.

An inspector is primarily an interactive facility. A programmatic interface is also available, which uses the LOOPS methods to customize the generic Interlisp-D functionality.

18.1.1 Overview of the User Interface

LOOPS provides two ways to create an inspector:

- Call the Lisp function **INSPECT** with the object to be inspected as the first argument.

- Use the LOOPS method **Inspect**, which is described below.

The user interface to the inspector is the same as that for the Medley environment; that is, you select an option from the left or right column with the left mouse button, and then trigger an action with the middle mouse button. The action opens a menu from which you can choose further options, like assigning a new value or adding an active value for breaking.

Another menu appears when you position the cursor on the title bar of an inspect window and press the middle mouse button. This menu allows you to change the inspector's contents, for instance to show all values or local ones for instance variables.

Three types of inspectors are available in LOOPS:

- Instance inspector
- Class inspector
- Class instance variable inspector

The following sections describe the user interface for each inspector.

(← *self* **Inspect** *INSPECTLOC*) [Method of Object]

Purpose: This provides a message form of the function to inspect the item *self.*

Behavior: Calls (INSPECT *self* NIL *INSPECTLOC*).

Arguments: *INSPECTLOC*
A region where the inspector window should appear. If it is NIL, you are prompted to place a ghost image.

Categories: Object

Example: The following command inspects an instance ($ W1).

```
17←(← ($ W1) Inspect)
```

18.1.2 Using Instance Inspectors

Using an inspector window on an instance provides a clear, direct interface to all of the instance's variables and values. This interface also provides the mouse and keyboard options to change the contents of the inspector to show various aspects of the instance.

18.1.2.1 Titles of Instance Inspector Windows

When you inspect an object, the title of the inspector window reflects the contents of the inspector. If the object is an instance, the title contains the name of the class of the instance and the LOOPS name of the instance, if it has one, or the UID of the instance. Other types of inspectors have different title bars, as described in Section 18.1.3, "Using Class Inspectors," and Section 18.1.4, "Using ClassIV Inspectors."

Contrast the title bar of the following two examples.

`(INSPECT (← ($ Window) New))` generates

```
All Values of Window ($ (MWX0.;F5.o28.Z;
left   NIL
bottom NIL
width  12
height 12
window #,($AV LispWindowAV ((YI
title  NIL
menus  T
```

`(INSPECT (← ($ Window) New 'w1))` generates

```
All Values of Window ($ w1).
left   NIL
bottom NIL
width  12
height 12
window #,($AV LispWindowAV ((YI
title  NIL
menus  T
```

18.1.2.2 Menu for the Title Bar

The following menu appears when you position the cursor on the title bar of the inspector window and press the middle mouse button.

```
Local Values of Window ($ w1).
left   468
bottom 472
width  116
height 74
window #,($AV LispWindowAV ((|MWX0.:F5
title  #,NotSetValue
menus  #,NotSetValue
```

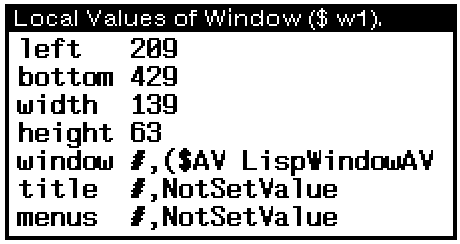
The rest of this section describes the actions that occur as a result of selecting one of the menu options.

Class Opens a second inspector, a Class Inspector as described in Section 18.1.3, "Using Class Inspectors," which inspects the class of the instance within this inspector.

AllValues The default mode for instance inspectors. The values displayed in the right column of the inspector are determined by the function **GetValueOnly**, so active values (except #,NotSetValue) can be seen. The title of the inspector states that all values are being displayed.

```
All Values of Window ($ w1).
left
   NIL
bottom
   NIL
width
   12
height
   12
window
   #,($AV LispWindowAV ((YIV0.C=N5.W←7 . 10)))
title
   NIL
menus
   T
```

LocalValues The values displayed in the right column of the inspector are determined by the function **GetIVHere**. The title of the inspector states that only local values are being displayed. As with **AllValues**, active values are seen, and values that are not yet stored locally in the instance show a value of #,NotSetValue.



Add/Delete Allows you to add or delete instance variables. Selecting this option pops up a new menu with two options:

- **Add**

  If you select **Add**, you are prompted to enter a name for the new instance variable. That instance variable is added locally to the instance and given the value of the variable **NotSetValue**. If you enter a name for an instance variable that currently exists, its value is reset to the value defined in the class.

- **Delete**

  If you select **Delete**, a menu appears with options that are the instance variables of the instance. If you select one that is not defined within the class, it is deleted. If the selected instance variable is defined by the class, a break occurs.

If the inspector is viewing the properties of an instance variable as opposed to all of the instance variables (see **IVs** below), the name entered under **Add** will be added as a property to that instance variable and given the value of the variable **NotSetValue**. If you try to delete an existing property, the menu that appears is a menu of property names.

IVs Changes the view to be one that shows all of the instance variables and their values, not the properties. It is possible to change the view an inspector has on an instance to show only the properties of a given instance variable. This is described in Section 18.1.2.3, "Using Commands in the Left Column," in the description of the Properties option.

Save Value Calls **PutSavedValue** with its value argument bound to the instance being inspected.

Refetch Refreshes the inspector. Inspectors do not automatically update when a change is made to an instance, unless made with the **Edit** command.

Edit Opens a display editor window in which you can modify the value of instance variables and properties, and add or delete instance variables local to the instance.

18.1.2.3 Menu for the Left Column

The following menu appears when the view of the inspector is all of the instance variables of the instance, and you select an item in the left column and press the middle mouse button.

```
PutValue
Properties
BreakIt
TraceIt
UnBreakIt
```

If the view of the inspector is only of properties, this menu contains only one option: **PutValue**.

The rest of this section describes the actions that occur as a result of selecting one of the menu options.

PutValue Allows you to assign a new value to the variable selected. Selecting this option and dragging the mouse to the right causes a submenu with the following options to appear:

- **PutValue**

  Prompts you to enter a new value for this instance variable. The new value is stored using **PutValue**.

- **PutValueOnly**

  Prompts you to enter a new value for this instance variable. The new value is stored using **PutValueOnly**.

- **Use saved value**

  The new value to be stored using **PutValueOnly** is the value of **(SavedValue)**.

Properties Changes the view of the inspector to include only the value and properties of the selected instance variable as shown here:

```
All IVProps of Window ($ w1).menus

Value             T
doc               "Cache For Saved Menus. Will Cache
TitleItems        NIL
MiddleButtonItems NIL
LeftButtonItems   NIL
Title             NIL
DontSave          Any
```

The title bar changes to indicate that the properties of an instance variable are being displayed, which instance is being displayed, and which instance variable of that instance is being displayed.

The Value item is provided purely as a convenience in this view and its menu options will only allow Putting a new value in it.

The following menu appears when the view of the inspector is an instance variable's properties, and you select an item in the left column and press the middle mouse button.

```
PutValue
BreakIt
TraceIt
UnBreakIt
```

When the inspector's view is limited to IV properties the menu options act in a manner similar to that of IVs. This allows Putting, Breaking, Tracing and unBreaking of the properties instead of the IVs themselves.

If you now select the option **LocalValues** from the title bar menu, the title of the inspector changes to indicate that fact, and only properties that are stored locally in the instance appear, as shown here:

To return to a view that shows all instance variables, choose the **IVs** option from the title bar menu.

BreakIt Wraps a **BreakOnPutOrGet** active value around the value of an instance variable. Any read or write accesses to this instance variable will cause a break (see Chapter 12, Breaking and Tracing).

Note: Breaking a variable effectively breaks any IndirectVariable that points to it.

Selecting this option and dragging the mouse to the right causes a submenu with the following options to appear:

- **Break on Access**

  Performs the same action as **BreakIt**.

- **Break on Put**

  Installs a **BreakOnPut** active value. Trying to store a new value into this instance variable will cause a break, but reading the variable will not.

TraceIt Wraps a **TraceOnPutOrGet** active value around the value of this field. Any read or write accesses to this instance variable will be traced (see Chapter 12, Breaking and Tracing).

Note: Tracing a variable effectively traces any IndirectVariable that points to it.

Selecting this option and dragging the mouse to the right causes a submenu with the following options to appear:

- **Trace on Access**

  Performs the same action as **TraceIt**.

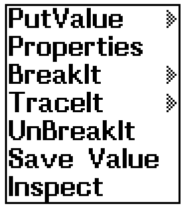
- **Trace on Put**

  Installs a **TraceOnPut** active value. All writes into this instance variable will be traced, but reads will not.

UnBreakIt Removes any of the breaks or traces that have been installed on an instance variable. If there are multiple traces or breaks, this will remove the outermost one.

18.1.2.4 Menu for the Right Column

The following menu appears when the view of the inspector is all of the instance variables of the instance, and you select an item in the right column and press the middle mouse button.



If the view of the inspector is only of properties of an instance variable, this menu contains only three options: **PutValue**, **Save Value**, and **Inspect**.

The only differences between these menus and the ones associated with the left column is the addition of two more options: **Save Value** and **Inspect**. The remaining options trigger identical behaviors as those of the menu associated with the left column.

Save Value Calls **PutSavedValue** with the selected value as its argument.

Inspect Calls the Lisp function **INSPECT** with the selected value as its argument, opening an additional inspector window.

In an inspector viewing the properties of an IV the right hand column middle button menu allows only the Put Value, SaveValue and Inspect options.

18.1.3 Using Class Inspectors

Classes can be inspected by using the Lisp **INSPECT** function or the LOOPS **Inspect** method (see Section 18.1.1, "Overview of the User Interface"). For example, to inspect the class **Window**, enter either of the following commands:

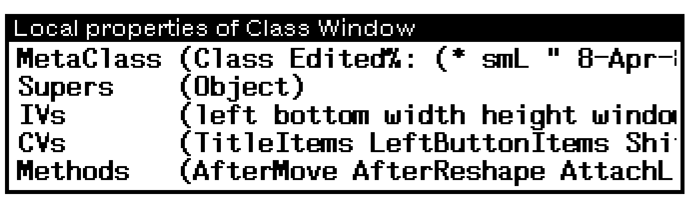
```
(INSPECT ($ Window))
(← ($ Window) Inspect)
```

The contents of a class cannot be changed from within an inspector window, so it is generally used for display as opposed to editing. However, the menu interface does provide ways to edit the contents of a class.

18.1.3.1 Titles of Class Inspector Windows

When you inspect a class, the title states that you are inspecting only local properties, and contains the name of the class.

The title contains the name of the class. The following example shows an inspector on the class **Window**. Since the value column is quite long, it has been truncated here.



18.1.3.2 Menu for the Title Bar

The title bar menu is associated with each inspector of a class. This menu appears when you position the cursor inside the title bar of the class inspector window and press the middle mouse button.

The rest of this section describes the actions that occur as a result of selecting one of the menu options.

Browse Provides a quick way to open a class browser on the class being inspected. Selecting this option and dragging the mouse to the right pops up a submenu with the following options:

- **Browse**

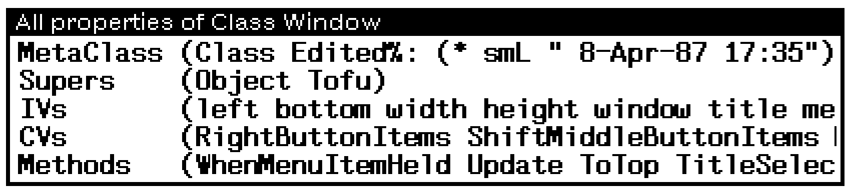
  Opens a class browser with the class being inspected as the root class.

- **BrowseSupers**

  Open a supers browser on the inspected class.

Edit Opens an editing window on the class.

All Causes the values shown in the right column to contain inherited as well as locally defined information. The title bar of the inspector changes to indicate this, as shown here:



Local The default mode for class inspectors. This causes the values shown in the right column to contain only locally defined information, which is indicated in the title bar of the inspector.

Refetch Refreshes the inspector. Inspectors do not automatically update when a change is made to an instance, unless made with the **Edit** command.

18.1.3.3 Menu for the Left Column

No actions occur when you select an item in the left column of a class inspector and press the middle mouse button.

18.1.3.4 Menu for the Right Column

Only one option, **Inspect**, is in the menu that appears when you select an item in the right column of a class inspector and press the middle mouse button.

For the fields **MetaClass**, **Supers**, **CV**s, and **Methods**, selecting **Inspect** from the menu allows a choice of Interlisp-D inspectors. If the selected item in the class inspector is the values of the **IVs** field, then a ClassIVs inspector, described below, is created.

18.1.4 Using ClassIVs Inspectors

ClassIV inspectors provide an interface to the default values for all of the instance variables defined in a class. To create a ClassIV inspector,

- Open a class inspector.
- Select the values of the **IVs** field.
- Press the middle mouse button. This pops up and selects and **Inspect** menu, and automatically opens a ClassIVs inspector.

18.1.4.1 Titles of ClassIVs Inspector Windows

The title for a ClassIVs inspector indicates that the instance variables of a particular class are being inspected. This example shows how a ClassIVs inspector looks for the class **ClassBrowser**.

```
All IVs of Class ClassBrowser
left                NIL
bottom              NIL
width               64
height              32
window              #,($AV LispWindowAV ((YIV0.C=N5.W←7 . 10)))
title               "Class browser"
menus               T
topAlign            NIL
startingList        NIL
goodList            NIL
badList             NIL
lastSelectedObject  NIL
browseFont          #,(Defer (FONTCREATE (QUOTE (HELVETICA 10 BOLD)
LabelMaxLines       NIL
LabelMaxCharsWidth  NIL
boxedNode           NIL
graphFormat         (LATTICE)
showGraphFn         SHOWGRAPH
viewingCategories   (Public)
```

18.1.4.2 Menu for the Title Bar

The following title bar menu is associated with each inspector of an instance. This menu appears when you position the cursor inside the title bar of the inspector window and press the middle mouse button.

```
AllValues
LocalValues
Add/Delete
IVs
Refetch
```

The rest of this section describes the actions that occur as a result of selecting one of the menu options.

AllValues The default mode for ClassIV inspectors. Causes the inspector to show all instance variables, whether inherited or locally defined for the class, and states "AllIVs" in the title.

LocalValues Causes the inspector to show only locally defined instance variables. The following window shows how the title changes to indicate this:

```
Local IVs of Class ClassBrowser
title               "Class browser"
viewingCategories (Public)
```

Add/Delete Allows you to add or delete a ClassIV.

Selecting this option and dragging the mouse to the right causes a submenu with the following options to appear:

- **Add**

If you select **Add**, you are prompted to enter a name for the new instance variable. That instance variable is added to the class and given the default value NIL and a **doc** property with the value (* IV added by (USERNAME)). If you enter a name for an instance variable that currently exists, its default value is reset to NIL.

- **Delete**

  If you select **Delete**, a menu appears with the locally defined instance variables of the class. Selecting one deletes it from the class.

If the inspector is viewing the properties of an instance variable as opposed to all of the instance variables (see **IVs** below), the name entered under **Add** is added as a property to that instance variable and given the value NIL. If you try to delete an existing property, the menu that appears is a menu of property names.

IVs Returns the view to show the instance variables and their values, not the properties.

It is possible to change the view a ClassIVs inspector has on the instance variables of a class to show only the properties of a given instance variable. This is described in Section 18.1.4.4, "Menu for the Right Column."

Refetch Refreshes the inspector.

18.1.4.3 Menu for the Left Column

No actions occur if you select an item in the left column of a ClassIVs inspector with the middle mouse button.

18.1.4.4 Menu for the Right Column

The following menu appears when you select an item in the right column and press the middle mouse button:

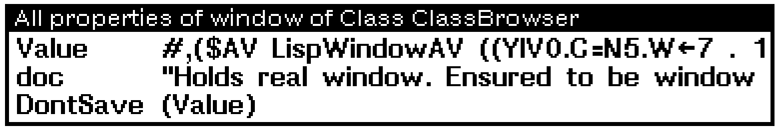
```
Inspect
Save Value
Properties
```

The rest of this section describes the actions that occur as a result of selecting one of the menu options.

Inspect Calls the Lisp function **INSPECT** with the selected value as its argument.

Save Value Calls **PutSavedValue** with the selected value as its argument.

Properties Changes the view of the inspector to display the value and properties of the selected instance variable, as shown in this example:



The title changes to include the following information:

- The properties of an instance variable.
- The name of the instance variable.
- The name of the class.

If you now select either **AllValues** or **LocalValues** from the title menu, the title of the inspector changes to indicate that fact, and the appropriate information is displayed.

18.1.5 Functional Interface for Instance Inspectors

The methods described in this section belong mostly to the classes **Class** or **Object**. Inspectors are not LOOPS objects, so these methods are invoked indirectly within the system functionality of the inspector as a customization of the Interlisp-D inspectors. These methods are meant to be called only from within the context that you create interactively by pressing a mouse button when the cursor is on some portion of an inspector window; you do not invoke them directly. Many of the parameters are simply passed along in case the method creates a menu, and the option selected from the menu needs additional arguments.

In these methods, the arguments *self* and *datum* may the same; that is, the item being inspected. The message is sent to the item being inspected, so its position in the inheritance lattice determines which method from the classes **Class** or **Object** is invoked.

The following table shows the items in this section.

Name	Type	Description
InspectFetch	Method	Returns the value of a left column inspector property that is displayed in the right column of an inspector window.
InspectStore	Method	Stores the value for an instance variable or its property.
InspectPropCommand	Method	After an item is selected in the left column of an inspector window, this triggers an action when the middle mouse button is pressed.
InspectProperties	Method	Determines what is displayed in the left column of an inspector window.
InspectTitle	Method	Creates a string to be used for an inspector window's title.
InspectValueCommand	Method	After an item is selected in the right column of an inspector window, this triggers an action when the middle button is pressed.
TitleCommand	Method	Triggers an action when the cursor is inside the title bar of the inspector window and the middle mouse button is held down.

(← *self* **InspectFetch** *datum property window*) [Method of Object]

Purpose/Behavior: Message sent by inspector to get the value of a left column inspector property that is displayed in the right column of an inspector window. Either **GetValueOnly** or **GetIVHere** is used to determine the value.

Arguments:

self — The object being inspected.

datum — This may or may not be a list. If it is not a list, it is bound to the object being inspected; that is, *self*. It is set to a list within various methods associated with the inspectors. The contents of this list are interpreted by a number of the methods to control what data is displayed within the inspector window.

- The first element of the list is the object being inspected.
- The second element of the list, if not NIL, is typically the name of an instance variable. In the terminology of the inspector, it is an inspector property. For inspectors of instances, the inspector properties (the items in the left column) are the instance variables of the object being inspected. (There can be some confusion here caused by using the word properties either when referring to the left column data of an inspector or when referring to the properties associated with an instance variable).
- The third element of the list, if NIL, indicates that inherited values are to be displayed in the inspector window; if its value is LocalValues, then only locally stored information is displayed.

The value of *datum* is stored on the inspector window property **DATUM**.

property — Used if *datum* is not a list. Refers to an element (instance variable or property name) contained within the left column of an inspector. For an instance inspector, this could be either the name of an instance variable or the name of a property, depending upon the state of the inspector; that is, whether you are viewing instances variable or the properties of a particular variable.

window — Lisp window of the inspector.

Returns: The value of a left column *property* that is displayed in the right column.

Categories: Object

Specializations: Class, InspectorClassIVs

Example: The following command fetches the value of instance **W1'**s instance variable **bottom**:

```
15←(← ($ W1) InspectFetch (LIST ($ W1) 'window))
#,($AV LispWindowAV ((YIV0.C=N5.W←7 . 10)))
```

The following command fetches the value of class **Window**'s supers:

```
16←(← ($ Window) InspectFetch ($ Window) 'Supers)
(Object)
```

(← *self* **InspectStore** *datum property newValue window*) [Method of Object]

Purpose/Behavior: Stores *newValue* as the value for an instance variable or its property using **PutValueOnly**. Where the value is stored, whether in the instance variable or one of its properties, depends upon the values for *datum* and *property*.

Arguments:

datum Instance or class being inspected. See **InspectFetch**, above, for details.

property Instance variable or property where value is to be stored. See **InspectFetch**, above, for details.

newValue New value for *property*.

window Lisp window of the inspector.

Categories: Object

Specializations: Class, InspectorClassIVs

Example: The following command changes the value of instance **W1**'s instance variable **height**:

```
17←(← ($ W1) InspectStore ($ W1) 'height 400)
400
```

or

```
18←(← ($ W1) InspectStore 'W1 'height 400)
400
```

(← *self* **InspectPropCommand** *datum property window*) [Method of Object]

Purpose: This method is an interface between LOOPS and the mouse functions of the inspector, and should only be called through the inspector. It is invoked when an item is selected in the left column of an inspector window and the middle mouse button is pressed.

Behavior: Opens a menu with a number of options. See Section 18.1.2.3, "Menu for the Left Column."

Arguments:

datum Instance or class being inspected. See **InspectFetch**, above, for details.

property Instance variable or property where value is to be stored. See **InspectFetch**, above, for details.

window Lisp window of the inspector. A prompt window will be attached to this window if you ask to **PutValue**, and the window's **INSPECTW.FETCH** function will be called, so the window must be an inspector window.

Categories: Object

Specializations: Class

(← *self* **InspectProperties** *datum*) [Method of Object]

Purpose: Determines what should be displayed in the left column of an inspector.

Behavior: Depending on the value of *datum* as described above, this will return either the instance variables of the object being inspected, or the properties of a particular instance variable.

Arguments: *datum* Instance or class being inspected. See **InspectFetch**, above, for details.

Returns: Value depends on the arguments; see Behavior.

Categories: Object

Specializations: Class, InspectorClassIVs

Example: The following command first shows the instance variables of instance **W1**, then the properties of the instance variable **height**:

```
27←(← ($ W1) InspectProperties 'W1)
(left bottom width height window title menus)

28←(← ($ W1) InspectProperties (LIST 'W1 'height))
(Value doc)
```

(← *self* **InspectTitle** *datum*) [Method of Object]

Purpose: Creates a string to be used as a title for an inspector window.

Behavior: If *datum* is not a list, this sets *datum* to (*datum* NIL NIL).

Depending on the values within the list *datum*, this creates a title showing whether all values or local values are shown and whether all instance variables or the properties of an instance variable are shown. The title also contains the LOOPS name or UID of *self*.

Arguments: *datum* Instance or class being inspected. See **InspectFetch**, above, for details.

Categories: Object

Specializations: Class, InspectorClassIVs

Example: Some examples of the behavior of **InspectTitle**:

```
35←(← ($ Window) InspectTitle)
"Local properties of Class Window"

36←(← ($ W1) InspectTitle)
"All Values of Window ($ W1)."

37←(← ($ Window) InspectTitle (LIST 'Window T))
"All properties of Class Window"

38←(← ($ W1) InspectTitle (LIST 'W1 'height))
"All IVProps of Window ($ W1).height"
```

(← *self* **InspectValueCommand** *datum property value window*) [Method of Object]

Purpose: This method is an interface between LOOPS and the mouse functions of the inspector, and should only be called through the Inspector. It is invoked when an item is selected in the right column of an inspector window and the middle mouse button is pressed.

Behavior: Opens a menu with several options. See Section 18.1.2.4, "Menu for the Right Column."

Arguments: *datum* Instance or class being inspected. See **InspectFetch**, above, for details.

property Instance variable or property being inspected. See **InspectFetch**, above, for details.

	value	This is inspected only if you select **Inspect** from menu.
	window	Lisp window of the inspector.
Categories:	Object	
Specializations:	Class, InspectorClassIVs	
Example:	When the value of the instance variable **height** in instance **W1** is selected in an inspector window, and the middle mouse button is pressed, a message like the following is sent:	

```
(← ($ W1) InspectValueCommand ($ W1) 'height 200 (WHICHW))
```

(← *self* **TitleCommand** *datum window*) [Method of Object]

Purpose: This method is an interface between LOOPS and the mouse functions of the inspector, and should only be called through the Inspector. It is invoked when the cursor is in the title bar of an inspector window and the middle mouse button is pressed.

Behavior: Brings up a menu with several options. See Section 18.1.2.2, "Menu for the Title Bar."

Arguments:

datum Instance or class being inspected. See **InspectFetch**, above, for details.

window Lisp window of the inspector.

Categories: Object

Specializations: Class, InspectorClassIVs

Example: If you position the cursor inside the title bar of the inspector window for instance **W1** and press the middle mouse button, you send a message like the following:

```
(← ($ W1) TitleCommand NIL (WHICHW))
```

18.1.6 Customizing the Inspector

The methods in Section 18.1.5, "Functional Interface for Instance Inspectors," have been specialized in the classes **Class** and **InspectorClassIVs** to create the behavior of the inspectors described in Section 18.1.1, "Overview of the User Interface."

If you want to create a specialized inspector, you need to create a subclass of **Object** or perhaps **Class** and specialize the methods within that new class. The class **InspectorClassIVs** has an instance variable named **class** that contains the name of the class being inspected within a particular instance of **InspectorClassIVs**. Similarly, the user-created inspector class may need an instance variable which contains the object being inspected so that the methods of this class can easily access it.

The methods that you need to specialize will depend upon how the behavior of the newly created inspector class should differ from those of an instance or class inspector.

As an example, assume that you want an inspector to show a subset of the instance variables of an instance. You could specialize the method **InspectProperties** to return that subset. To make **Window** show only the dimensions of a window, define the following method:

```
SEdit Window.InspectProperties  Package: INTERLISP
(Method ((Window InspectProperties) self)
   "Have Window inspectors show only the window dimensions"
   '(width height left bottom))
```

`(←New ($ Window) Inspect)` creates the following:

```
All Values of W
width  12
height 12
left   NIL
bottom NIL
```


18.2 EXTENSIONS TO ?=

18.2 Extensions to ?=

The Interlisp-D environment allows you to begin typing a function to be evaluated in the Executive and pause in the midst of typing the arguments. At this point, if you type a "?=" followed by a carriage return, Interlisp-D prints the arguments to the pending function call and shows the bindings. LOOPS has extended this facility to include similar functionality for message sending and for record creation.

18.2.1 Message Sending

The ?= interface works with the following message-sending forms:

- ←
- ←Super
- ←New
- ←Proto
- ←Process
- SEND

LOOPS first tries to determine the class of the object receiving the message by examining the form following one of the above. If the message form begins with one of ←**New** or ←**Proto**, the object receiving the message is the class desired.

- If the system cannot determine the class of the object, you are prompted in the Prompt Window to enter in the name of the class or to type a right square bracket (]) to evaluate the form and determine that class from that. This handles cases such as

  ```
  (← (← ($ Window) New) ?=<CR>
  ```

- If the class can be determined and if you have not typed in a selector, a menu appears containing the options ***generics*** and ***inherited*** and any selectors local to the class. A submenu is associated with ***generics*** that contains selectors from the classes **Tofu**, **Object**, or **Class**, depending upon the class previously determined. The submenu associated with ***inherited*** are those selectors that are neither generic nor local to the class.

This is the menu that appears when you type

```
(←New ($ NonRectangularWindow) ?= <CR>
```

If you choose one of these options, it is placed into the input buffer and the system prints the binding for *self*, what method will be executed, and the arguments expected. In the prompt window, the system prints the documentation of the method to be executed.

As an example, if you create an instance of a class browser **cb1**, type

```
22←(← ($ cb1) ?= <CR>
```

and then choose **Shape** from the ***inherited*** drag-through menu, the Executive changes as shown here.

```
22←(←($ cb1) Shape
(←
self = ($ cb1)
Method = Window.Shape
 newRegion noUpdateFlg)
```

A similar output occurs if you type in a selector and "?=" instead of choosing a selector from the menu.

If you type "?=" after entering one or more of the arguments, the arguments are printed with the bindings, as shown here.

```
23← (←($ cb1) Shape '(100 150 200 250) ?= <CR>
(←
self = ($ cb1)
Method = Window.Shape
newRegion (QUOTE (100 150 ...))
noUpdateFlg)
```

This interface also works when you are typing to the edit buffer window when using the display editor. It does not work to pick a selector within a display editor window and choose the **?=** item from the **EditCom** submenu.

18.2.2 Record Creation

The same mechanism the LOOPS uses to handle ?= for LOOPS objects is also used to extend it for the Interlisp Record module.

If you begin an input to the Executive with one of **CREATE**, **Create**, or **create**, type the record name or data type to be created next, and then type

?=<CR>

the system prints the names but not the bindings of the fields within the record being created.

For example, when you type

```
48←(CREATE POSITION XCOORD ← 123 ?=<CR>
```

the response is

```
(XCOORD YCOORD)
```

on the next line. The caret moves to the position of the ?= in the original line, and waits for you to enter a value.

[This page intentionally left blank]

19. LOOPS WINDOWS

This chapter describes the ways to manipulate LOOPS windows.

19.1 The Class Window

The class **Window** is the LOOPS interface to the Medley environment window system, which is used by LOOPS browsers and inspectors.

Window [Class]

Description: This class provides a mechanism for manipulating Lisp windows through an object-oriented interface. See Section 19.4, "Mouse and Menu Functionality," for a discussion of how menus work with LOOPS Windows.

When an instance of a LOOPS window is created, it has an instance variable that points to a Lisp window. This Lisp window is initialized with various window properties:

- The property **LoopsWindow** points to the window object.
- The property **RIGHTBUTTONFN** is set to **WindowRightButtonFn**.
- The property **BUTTONEVENTFN** is set to **WindowButtonEventFn**.
- The property **AFTERMOVEFN** is set to **WindowAfterMoveFn**.
- The property **RESHAPEFN** is set to **WindowReshapeFn**.

MetaClass: Class

Supers: Object

Class Variables:

TitleItems A list that defines the menu that will appear when the left or middle mouse button is pressed and the cursor is in the title bar of the window. The default value is NIL.

LeftButtonItems
A list that defines the menu for the left button in the main window. The default value is ((Update ...)).

ShiftLeftButtonItems
A list that defines the menu for the left button in the main window when the **Meta** key is down. The default value is NIL.

MiddleButtonItems
A list that defines the menu for the middle button in the main window. The default value is NIL.

ShiftMiddleButtonItems
A list that defines the menu for the middle button in the main window when the **Meta** key is down. The default value is NIL.

RightButtonItems
A list that defines the menu for the right button in the main window. The default value is ((Close ...)).

Instance Variables:	**left**	The location of the left side of the outside of the window in screen coordinates. The default value is NIL.
	bottom	The location of the bottom side of the outside of the window in screen coordinates. The default value is NIL.
	width	The outside width of the window. The default value is 12.
	height	The outside height of the window. The default value is 12.
	window	An active value that contains the Lisp window. The default value is #,($AV LispWindowAV ...).
	title	The title of the window. Default Value: NIL.
	menus	Also has the properties **Title**, **LeftButtonItems**, **MiddleButtonItems**, and **TitleItems**. These properties are caches for menus only if the value of the instance variable is T. Default Value: T.

19.2 BASIC WINDOW METHODS

19.2 Basic Window Methods

This section describes the basic methods to operate on windows.

Name	Type	Description
AfterMove	Method	Updates the instance variables left and bottom.
AfterReshape	Method	Updates the instance variables left, bottom, width, and height.
Blink	Method	Causes the window to blink.
Bury	Method	Buries the window.
Clear	Method	Clears the window.
Close	Method	Closes the window.
CursorInside?	Method	Determines if the cursor is inside a window.
Destroy	Method	Destroys the window instance.
GetProp	Method	Gets a property from a specified window.
Hardcopy	Method	Makes a hardcopy on the default device.
HardcopyToFile	Method	Makes a hardcopy on a file.
HardcopyToPrinter	Method	Makes a hardcopy to a printer.
Invert	Method	Inverts the window; that is, reverses its black-white pattern.
Move	Method	Moves the window.

MousePackage	Method	Returns a package (defined to return the INTERLISP package).
MouseReadtable	Method	Returns a readtable (defined to return the INTERLISP readtable).
Open	Method	Opens the window.
Paint	Method	Calls **PAINTW** on the window.
ScrollWindow	Method	Scrolls the window.
SetProp	Method	Sets the property in the specified window.
Shape	Method	Reshapes the window.
Shape?	Method	Returns the current region for the window.
Shrink	Method	Shrinks the window to an icon.
Snap	Method	Takes a snapshot of the screen.
ToTop	Method	Opens the window and brings it to the top.
Update	Method	Makes the window consistent with the instance variables.
WindowAfterMoveFn	Function	Sends the message **AfterMove**.
WindowReshapeFn	Function	Sends the message **AfterReshape**.

(← *self* **AfterMove**) [Method of Window]

Purpose/Behavior: Updates the instance variables **left** and **bottom** of *self*.

Arguments: *self* An instance of a window.

Returns: Used for side effect only.

Categories: Window

(← *self* **AfterReshape** *oldBitmapImage oldRegion oldScreenRegion*) [Method of Window]

Purpose/Behavior: Updates the instance variables **left**, **bottom**, **width**, and **height** of *self*. Calls **RESHAPEBYREPAINTFN**; see the *Interlisp-D Reference Manual*.

Arguments: *self* An instance of a window.

Returns: Used for side effect only.

Categories: Window

(← *self* **Blink** *numBlinks*) [Method of Window]

Purpose/Behavior: Inverts the window; that is, reverses its black-white pattern, and then returns to normal *numBlinks* times.

Arguments: *self* Pointer to a window instance.

numBlinks Number of times for window to blink.

Returns: NIL

Categories: Window

Example: The command

```
(← self Blink 5)
```

sends a message to *self* to blink five times.

(← *self* **Bury**) [Method of Window]

Purpose/Behavior: Calls **BURYW** to bury the specified window.

Arguments: *self* Pointer to a window instance.

Returns: The LOOPS window.

Categories: Window

(← *self* **Clear**) [Method of Window]

Purpose/Behavior: Calls **CLEARW** to clear the specified window.

Arguments: *self* Pointer to a window instance.

Returns: NIL

Categories: Window

Specializations: LatticeBrowser

(← *self* **Close**) [Method of Window]

Purpose/Behavior: Closes the specified window and prompt window, if there is one.

Arguments: *self* Pointer to a window instance.

Returns: NIL

Categories: Window

(← *self* **CursorInside?**) [Method of Window]

Purpose/Behavior: Determines if the cursor is inside the window.

Arguments: *self* Pointer to a LOOPS window.

Returns: Returns T if the cursor is inside the window, otherwise returns NIL.

Categories: Window

(← *self* **Destroy**) [Method of Window]

Purpose/Behavior: Destroys the calling instance, removes all **ButtonFns**, and closes the window.

Arguments: *self* Pointer to a window instance.

Returns: NIL

Categories: Object

Specializes: Object

(← *self* **GetProp** *prop*) [Method of Window]

Purpose/Behavior: Gets a property from a window.

Arguments: *self* Pointer to a window instance.

prop Property to get.

Returns: The value of the specified property, if it exists; else NIL.

Categories: Window

Example: To determine the value of the window property **BUTTONEVENTFN**, enter

```
(← ($ window)  GetProp  'BUTTONEVENTFN)
```

(← *self* **Hardcopy**) [Method of Window]

Purpose/Behavior: Makes a hardcopy of the window on the default printer.

Arguments: *self* Pointer to a window instance.

Categories: Window

(← *self* **HardcopyToFile**) [Method of Window]

Purpose/Behavior: Makes a hardcopy of the window to a file. You are prompted for the file name.

Arguments: *self* Pointer to a window instance.

Categories: Window

(← *self* **HardcopyToPrinter**) [Method of Window]

Purpose/Behavior: Makes a hardcopy of the window on a printer. You are prompted for the name of the printer.

Arguments: *self* Pointer to a window instance.

Categories: Window

(← *self* **Invert**) [Method of Window]

Purpose/Behavior: Inverts the window; that is, reverses its black-white pattern.

Arguments: *self* Pointer to a window instance.

Returns: T if successful.

Categories: Window

Specializations: NonRectangularWindow

(← *self* **Move** *xOrPos* *y*) [Method of Window]

Purpose/Behavior: Moves the specified window. If no arguments are supplied, you will be prompted to position the window.

Arguments: *self* Pointer to a window instance.

	x	New **left** in screen coordinates or a new position for **left** and **bottom**. If *x* is a position, *y* is ignored.
	y	New **bottom** in screen coordinates.

Returns: (*x* . *y*)

Categories: Window

Example: The command
`(← ($ window) Move)`
causes window to become attached to the cursor, prompting for new location.

The command
`(← self Move 200 100)`
moves the lower left corner of the window to (200 . 100).

(←*self* **MousePackage**) [Method of Window]

Purpose/Behavior: Returns the package used during mouse interactions with the window *self*. (LOOPS now uses the INTERLISP package exclusively.) The **MousePackage** method protects LOOPS windows from the new packages. To remove this protection, specialize this method to return *PACKAGE*.

Arguments: *self* An instance of a window.

Returns: The INTERLISP package.

(←*self* **MouseReadtable**) [Method of Window]

Purpose/Behavior: Returns the readtable used during mouse interactions with the window *self*. Medley now has many different readtables; some readtables do not work well with LOOPS. (The Common Lisp readtables are not case-sensitive.) This method protects LOOPS windows from the new readtables. To remove this protection, specialize this method to return *READTABLE*.

Arguments: *self* An instance of a window.

Returns: The INTERLISP readtable.

(← *self* **Open**) [Method of Window]

Purpose/Behavior: Opens the specified window instance.

Arguments: *self* Pointer to a window instance.

Returns: NIL

Categories: Window

(← *self* **Paint**) [Method of Window]

Purpose/Behavior: Calls **PAINTW** on the specified window. You are prompted for instructions in the prompt window.

Arguments: *self* Pointer to a window instance.

Returns: NIL

Categories: Window

(← *self* **ScrollWindow** *dspX dspY windowX windowY*) [Method of Window]

Purpose/Behavior: Scrolls the window to move the point *dspX, dspY* to *windowX, windowY*. If *windowX* and *windowY* are NIL, the default is to scroll so that the point *dspx, dspy* appears in the lower left corner of the window. Any of the arguments can be **FIXP** or **FLOATP**. If the value is **FIXP**, then it is treated as an absolute coordinate. If the value is **FLOATP**, then it is treated as a relative position.

Arguments: *self* Pointer to a window instance.

dspX The x point in the given window to move; x is in window coordinates if **FIXP**. If **FLOATP**, the value to move is based upon the width of the **EXTENT** property of the window; see the *Interlisp-D Reference Manual*.

dspY The y point in the given window to move; y is in window coordinates if **FIXP**. If **FLOATP**, the value to move is based upon the height of the **EXTENT** property of the window; see the *Interlisp-D Reference Manual*.

windowX The x point to scroll to in window coordinates if **FIXP**. If **FLOATP**, the value to move is based upon the width of the window.

windowY The x point to scroll to in window coordinates if **FIXP**. If **FLOATP**, the value to move is based upon the height of the window.

Returns: The lower left corner of the new **DSPCLIPPINGREGION**; see the *Interlisp-D Reference Manual*.

Categories: Window

(← *self* **SetProp** *prop value*) [Method of Window]

Purpose/Behavior: Sets the Interlisp window property of the specified LOOPS window, passing its *prop* and *value* arguments through Interlisp function **WINDOWPROP**.

Arguments: *self* Pointer to a window instance.

prop Property to set.

value New value for property.

Returns: Previous value of *prop* if it existed; else NIL.

Categories: Window

(← *self* **Shape** *newRegion noUpdateFlg*) [Method of Window]

Purpose/Behavior: Reshapes the specified window. If *newRegion* is not specified, you are prompted to reshape the window with the cursor.

Arguments: *self* Pointer to a window instance.

newRegion A list specifying the new outer dimensions; the format for the list is (left bottom height width).

noUpdateFlg
If NIL, reshapes the window.

Returns: A list specifying the new region.

Categories: Window

Specializations: NonRectangularWindow

Example: The command

```
(← ($ window1) Shape '(100 200 300 400))
```

returns

```
(100 200 300 400)
```

(← *self* **Shape?**) [Method of Window]

Purpose/Behavior: Returns the current region for the window.

Arguments: *self* Pointer to a window instance.

Returns: A list specifying outer dimensions of the window.

Categories: Window

Example: The command

```
(← ($ window1) Shape?)
```

returns

```
(100 200 300 400)
```

(← *self* **Shrink** *toWhat iconPos expandFn*) [Method of Window]

Purpose/Behavior: Shrinks the window to a given icon.

Arguments:

self	Pointer to a window instance.
toWhat	The icon to shrink to; if NIL, an icon is created.
iconPos	Position of icon on screen.
expandFn	Function to be called on expansion.

Returns: The icon.

Categories: Window

Specializations: LatticeBrowser

(← *self* **Snap**) [Method of Window]

Purpose/Behavior: Calls **SnapW** to take a snapshot of the window.

Arguments: *self* Pointer to a window instance.

Returns: The window.

Categories: Window

(← *self* **ToTop**) [Method of Window]

Purpose/Behavior: Opens the window and brings it to the top of the screen.

Arguments: *self* Pointer to a window instance.

Returns: The window.

Categories: Window

(← *self* **Update**) [Method of Window]

Purpose/Behavior: Makes the window consistent with the instance variables.

Arguments: *self* Pointer to a window instance.

Returns: NIL

Categories: Window

Specializations: NonRectangularWindow

(**WindowAfterMoveFn** *window*) [Function]

Purpose/Behavior: This function is installed as the **AFTERMOVEFN** property of the Lisp window pointed to by a window object. This function extracts the window object from the property **LoopsWindow** and sends it the message **AfterMove.**

This **AFTERMOVEFN** is installed automatically by the system.

Arguments: *window* The window just moved.

Returns: Used for side effect only.

(**WindowShapeFn** *window oldBitmapImage oldRegion*) [Function]

Purpose/Behavior: This function is installed as the **RESHAPEFN** property of the Lisp window pointed to by a window. This function extracts the window object from the property **LoopsWindow** and sends it the message **AfterReshape** with the arguments *oldBitmapImage oldRegion*.

This **RESHAPEFN** is installed automatically by the system.

Arguments: *window* The window just reshaped.

oldBitmapImage See the *Lisp Release Notes* and the *Interlisp-D Reference Manual* for a discussion of window **ReShapeFns**.

oldRegion See the *Lisp Release Notes* and the *Interlisp-D Reference Manual* for a discussion of window **ReShapeFns**.

Returns: Used for side effect only.

19.3 PROMPT WINDOWS

19.3 Prompt Windows

Prompt windows are windows attached to other windows and are used for displaying messages and for getting input. In LOOPS, these operate similarly to prompt windows in Lisp. Prompt windows are not instances of the class Window; they are only instances of the Interlisp data type **Window**.

The following table lists the methods and functions described in this section.

Name	Type	Description
Clear PromptWindow	Method	Clears the prompt window.
ClosePromptWindow	Method	Closes the prompt window.
GetPromptWindow	Method	Associates a prompt window with a LOOPS window.
PromptEval	Function	Prompts for, reads, and evaluates an expression.
PromptForList	Method	Prompts for a list of items.
PromptForString	Method	Prompts for a string.
PromptForWord	Method	Prompts for a word.
PromptPrint	Method	Prints a message in the prompt window.
PromptRead	Function	Prompts for and reads data.
NiceMenu	Function	Creates a menu.
SelectFile	Lambda NoSpread	Prompts for a file name.

Note: The methods **PromptForList**, **PromptForString**, and **PromptForWord**, as well as the functions **PromptRead** and **PromptEval** when called with a prompt window for a LOOPS window, all disable normal mouse button events in the prompting browser and will not allow it to close until the prompt is completed.

(← *self* **ClearPromptWindow**) [Method of Window]

Purpose/Behavior: Clears the prompt window associated with the window *self*.

Arguments: *self* Evaluates to a window instance.

Returns: NIL

Categories: Window

(← *self* **ClosePromptWindow**) [Method of Window]

Purpose/Behavior: Closes the prompt window associated with the window *self*.

Arguments: *self* Evaluates to a window instance.

Returns: The symbol **CLOSED** if a prompt window existed; else NIL.

Categories: Window

(← *self* **GetPromptWindow** *lines fontDef*) [Method of Window]

Purpose/Behavior: Gets a prompt window for window *self*. If one exists, it is returned; else a prompt window is created.

Arguments:

self Pointer to a window instance.

lines Number of lines in window; default is 2.

fontDef Font used in the window; if NIL, this defaults to DEFAULTFONT.

Returns: Pointer to a prompt window.

Categories: Window

(**PromptEval** *promptString window sameLine?*) [Function]

Purpose: Prompts for, reads, and evaluates an expression.

Behavior: Temporarily moves the TTYDISPLAYSTREAM to *window*, if *window* is non-NIL, else to the system prompt window.

The *promptString* is printed followed by a carriage return and the string "The expression read will be EVALuated."

The prompt "> " is printed on the same line as the above if *sameLine?* is non-NIL, else it is printed on a new line. Data entered by the user is evaluated and returned. **LISPX** and **LISPXREAD** are used so that the entered data is placed on the **LISPX** history list. (See the *Lisp Release Notes* and the *Interlisp-D Reference Manual*).

Note: When called with a prompt window for a LOOPS window, **PromptEval** disables normal mouse button events in the prompting browser and will not allow it to close until the prompt is completed.

Arguments: *promptString* A string to be printed.

window A window where the prompting and reading should occur. Defaults to the system prompt window.

sameLine? If non-NIL, the data is read from the same line as the string "The expression read will be EVALuated."

Returns: The data entered by the user after it has been evaluated.

Example: The command

```
26←(← ($ Window) New (PromptEval "Specify new window for object name."))
```

causes the following to appear in the Prompt Window:

```
Specify new window for object name.
The expression read will be EVALuated.
```

>

Entering

```
'NewWindow
```

after the > causes the following return in the Executive Window.

```
#,($& Window (NEW0.1Y%:.;h.eN6 . 501))
```

(← *self* **PromptForList** *promptStr initialString*) [Method of Window]

Purpose/Behavior: Prompts you in prompt window for a list of symbols. If prompt window does not exist, one is created. Input is terminated by a carriage return.

TTYIN is used for editing the user’s input; see the *Interlisp-D Reference Manual*.

Note: **PromptForList** disables normal mouse button events in the prompting browser and will not allow it to close until the prompt is completed.

Arguments: *self* Pointer to a window instance.

promptStr Displayed in prompt window.

initialString Can be used as the default or the first item of the list.

Returns: The list of words entered in prompt window.

Categories: Window

Example: If ($ Window1) is a window, then the command

```
27←(← ($ Window1) PromptForList  "ENTER THE CODES ")
```

causes the prompt ENTER THE CODES to be displayed in an attached prompt window. The system waits for user input.

(← *self* **PromptForString** *promptStr initialStr*) [Method of Window]

Purpose/Behavior: Prompts you in prompt window for a string. If a prompt window does not exist, one is created. Input is terminated by a carriage return.

TTYIN is used for editing the user's input; see the *Interlisp-D Reference Manual*.

Note: **PromptForString** disables normal mouse button events in the prompting browser and will not allow it to close until the prompt is completed.

Arguments: *self* Pointer to a window instance.

promptStr Displayed in prompt window.

initialStr Can be used as the default or the prefix to the string.

Returns: The string entered in prompt window.

Categories: Window

Example: If ($ Window1) is a window, then the command

```
28←(← ($ Window1) PromptForString  "ENTER THE CODES ")
```

causes the prompt ENTER THE CODES to be displayed in an attached prompt window . The system waits for user input.

(← *self* **PromptForWord** *promptStr initialWord*) [Method of Window]

Purpose/Behavior: Returns (CAR (← *self* **PromptForList** *promptStr initialWord*))

Arguments: *self* Evaluates to a window instance.

promptStr Displayed in prompt window.

initialWord Can be used as the default.

Returns: See Behavior.

Categories: Window

Example: If ($ Window1) is a window, then the command

```
29←(← ($ Window1)  PromptForWord  "NEW WORD " )
```

prompts you with NEW WORD in an attached prompt window.

(← *self* **PromptPrint** *msg*) [Method of Window]

Purpose/Behavior: Displays a message in the prompt window associated with the specified window instance. Creates the prompt window if it does not exist.

Arguments: *self* Evaluates to a window instance.

msg Message displayed.

Returns: The message printed.

Categories: Window

(**PromptRead** *promptString window sameLine?*) [Function]

Purpose: Prompts for and reads data.

Behavior: Temporarily moves the **TTYDISPLAYSTREAM** to *window*, if *window* is non-NIL, else to the system prompt window.

The *promptString* is printed. The prompt "> " is printed on the same line as the above if *sameLine?* is non-NIL, else it is printed on a new line. Data that you entered is read and returned.

This contrasts with **PromptEval** in that the entered data is not placed on the **LISPX** history list (see the *Lisp Release Notes* and the *Interlisp-D Reference Manual*).

Note: When called with a prompt window for a LOOPS window **PromptRead** disables normal mouse button events in the prompting browser and will not allow it to close until the prompt is completed.

Arguments: *promptString*
A string to be printed.

window A window where the prompting and reading should occur. Defaults to the system prompt window.

sameLine? If non-NIL, the data is read from the same line as the *promptString*.

Returns: The data entered by the user.

(**NiceMenu** *items title*) [Function]

Purpose: Provides an interface to create a menu and displays the menu.

Behavior: Varies according to the arguments.

- If *items* is NIL, prints "No items for *title*" in the system prompt window and returns NIL.
- If *items* is non-NIL, this builds a menu with the **TITLE** *title*, with the **ITEMS** *items*, and with **CHANGEOFFSETFLG** set to T (see the *Interlisp-D Reference Manual*). If the length of *items* is more than 35, the menu has multiple columns.

Arguments: *items* A form that can be placed in the **ITEMS** field of a menu.

title A value that will be placed in the **TITLE** field of a menu.

Returns: Value depends on the arguments; see Behavior.

(**SelectFile** *prompts*) [Lambda NoSpread Function]

Purpose: Prompts you for a file name.

Behavior: Takes on unlimited number of arguments and will **PROMPTPRINT** all the arguments.

Builds a menu with the items ***newFile*** and the files found on the variable **FILELST**.

If you select one of the files, that is returned.

If you select ***newFile***, you are prompted to enter a file name. An empty filecoms is built for that file name, and the file name is returned.

newFile has three subitems:

- ***newFile***

  See Behavior.

- ***loadFile***

  You are prompted to enter a file name. A search is performed to try to find and load the compiled file. If that is not found, an attempt is made to load the source file. Returns NIL if the file is not found.

- ***hiddenFile***

A menu is displayed containing files that are on the variable **LOADEDFILELST** but not on **FILELST**.

Arguments: *prompts* A number of expressions to be printed in the system prompt window.

Returns: Value depends on the arguments; see Behavior.

19.4 MOUSE AND MENU FUNCTIONALITY

19.4 Mouse and Menu Functionality

When a LOOPS window is instantiated, its instance variable **window** points to an instance of a Lisp window. This window has several properties set, among which are the following that are described in this section:

- The property **RIGHTBUTTONFN** that is set to **WindowRightButtonFn**.
- The property **BUTTONEVENTFN** that is set to **WindowButtonEventFn**.

This section will also explain the functionality of the above and how menus associated with LOOPS windows operate. For more information on Medley windows, see the *Lisp Release Notes* and the *Interlisp-D Reference Manual*.

Name	Type	Description
ButtonEventFn	Method	Sends either the message **TitleSelection**, **LeftSelection**, or **MiddleSelection**.
ClearMenuCache	Method	Deletes menus saved on the menus field of a browser.
ItemMenu	Method	Creates a simple one-level menu.
LeftSelection	Method	Triggers functionality when the cursor is in a window and the left button is pressed.

MiddleSelection	Method	Triggers functionality when the cursor is in a window and the middle button is pressed.
RightButtonFn	Method	Sends the message **RightSelection.**
RightSelection	Method	Triggers functionality when the cursor is in a window and the right button is pressed.
TitleSelection	Method	Triggers functionality when the cursor is in a window's title bar and the left or middle button is pressed.
WhenMenuItemHeld	Method	Displays in the prompt window what happens when option is selected.
WindowButtonEventFn	Function	Invokes the method **ButtonEventFn**.
WindowRightButtonFn	Function	Invokes the method **RightButtonFn.**

(← *self* **ButtonEventFn**) [Method of Window]

Purpose/Behavior: If the cursor is not inside of the window pointed to by *self*, this sends the message **TitleSelection** to *self*.

If the left mouse button is pressed, this sends the message **LeftSelection** to *self*.

If the left middle button is pressed, this sends the message **MiddleSelection** to *self*.

Arguments: *self* An instance of a window.

Returns: Used for side effect only.

Categories: Window

(← *self* **ClearMenuCache**) [Method of Window]

Purpose/Behavior: Deletes menus saved in any properties of the instance variable **menus** of a window. Use this method if you ever change the class variables describing a menu, and you want the new menu to take effect.

Arguments: *self* Pointer to a window instance.

Returns: *self*

Categories: Window

(← *self* **ItemMenu** *items title*) [Method of Window]

Purpose/Behavior: Creates a simple one-level menu guaranteed not to be more than 750 bits high. A large number of menu options will cause a multiple column menu to be formed.

Arguments: *self* Pointer to a window instance.

items The value of this is passed to the **ITEMS** field when the menu is created.

title The title for the menu's window.

Returns: A menu.

Categories: Window

Example: The command

```
32←(← (←New ($ Window)) ItemMenu '(a b c))
```

will create a menu with the three options.

(← *self* **LeftSelection**) [Method of Window]

Purpose/Behavior: Invokes a number of internal methods of **Window**. A menu will pop up. The options in the menu will be defined by the class variable **LeftButtonItems** (or **ShiftLeftButtonItems** if the **Meta** key is also pressed). If an option is selected from the menu, a message will be sent to *self* with a selector as specified by the chosen menu option.

Arguments: *self* Pointer to a window instance.

Returns: Used for side effect only.

Categories: Window

Specializations: LatticeBrowser

(← *self* **MiddleSelection**) [Method of Window]

Purpose/Behavior: Invokes a number of internal methods of **Window**. A menu will pop up. The options in the menu will be defined by the class variable **MiddleButtonItems** (or **ShiftMiddleButtonItems** if the **Meta** key is also pressed). If an option is selected from the menu, a message will be sent to *self* with a selector as specified by the chosen menu option.

Arguments: *self* Pointer to a window instance.

Categories: Window

Specializations: LatticeBrowser

(← *self* **RightSelection**) [Method of Window]

Purpose/Behavior: Invokes a number of internal methods of **Window**. A menu will pop up. The options in the menu will be defined by the class variable **RightButtonItems**. If an option is selected from the menu, a message will be sent to *self* with a selector as specified by the chosen menu option.

Arguments: *self* Pointer to a window instance.

Returns: The menu.

Categories: Window

(← *self* **TitleSelection**) [Method of Window]

Purpose/Behavior: Invokes a number of internal methods of **Window**. A menu will pop up. The options in the menu will be defined by the class variable **TitleItems**. If an option is selected from the menu, a message will be sent to *self* with a selector as specified by the chosen menu option .

Arguments: *self* Pointer to a window instance.

Returns: The choice if selected; else NIL.

Categories: Window

Specializations: LatticeBrowser

(← *self* **WhenMenuItemHeld** *item* - -) [Method of Window]

Purpose/Behavior: Displays in the system prompt window what will happen when the menu item is chosen. The information displayed will either be the help string for the item or the documentation for the method pointed to by the item.

Arguments: *self* Pointer to a window instance.

item The menu item selector.

Returns: NIL

Categories: Window

WindowButtonEventFn [Function]

Purpose/Behavior: Invokes the method **ButtonEventFn**.

WindowRightButtonFn [Function]

Purpose/Behavior: Invokes the method **RightButtonFn.**

(**WindowButtonEventFn** *window*) [Function]

Purpose/Behavior: This retrieves the value of the Lisp window property **LoopsWindow**. It sends the message **ButtonEventFn** to that window object. If the window object is an instance of **NonRectangularWindow** or one of its subclasses and if the cursor is not within the icon bitmap, nothing occurs.

This is invoked automatically by the system when the cursor is inside of a window object and the left or middle mouse button is pressed.

Arguments: *window* The window that contained the cursor when the mouse button was pressed.

Returns: Used for side effect only.

(**WindowRightButtonFn** *window*) [Function]

Purpose/Behavior: This retrieves the value of the Lisp window property **LoopsWindow**. It sends the message **RightButtonFn** to that window object. If the window object is an instance of **NonRectangularWindow** or one of its subclasses and if the cursor is not within the icon bitmap, nothing occurs.

This is invoked automatically by the system when the cursor is inside of a window object and the left or middle mouse button is pressed.

Arguments: *window* The window that contained the cursor when the mouse button was pressed.

Returns: Used for side effect only.

19.4.1 Menu Item Structure

The default behavior for the methods **LeftSelection**, **MiddleSelection**, and **RightSelection** causes a menu to pop up. The options that will appear in a menu are defined in various class variables of the window object being selected. (**IconWindows** are an exception). When an option is selected from a menu, a message is sent to the window with no arguments.

The value of the various class variables can be an item list, such as (item1....itemn) where each item can be one of:

- *selector*

  In this case, the *selector* appears in the menu, and it is the selector of the message sent to the window.

- *(prompt selector help-string)*

  In this case, the *prompt* appears in the menu, and *selector* is the selector of the message sent to the window. *help-string* is printed when the cursor is over the item and the mouse is pressed.

- *(prompt subitemStructure help-string)* where *subitemStructure* = *(defaultSelector itemlist)*

  This form allows a menu to contain submenus. In this case, the *prompt* appears in the main menu, and *defaultSelector* is the selector of the message sent to the window if the main menu item is selected. *itemlist* defines the submenu behavior.

For example, in the class **Window**, the class variable **LeftButtonItems** has the following value:

```
((Update (QUOTE Update) "Update window to agree with object IVs"))
```

The class variable **RightButtonItems** has the value:

```
((Close (Close (Close Destroy))) Snap Paint Clear Bury Repaint
(Hardcopy (Hardcopy (HardcopyToFile HardcopyToPrinter))) Move Shape
Shrink)
```

19.4.2 Caching Menus

When a menu is created by pressing a mouse button on a LOOPS window, the menu is cached on a property of the instance variable **menus** if **menus** has the value T. The name of the property where it is stored has the same name as the class variable that describes the menu. The method **ClearMenuCache** will set these properties to NIL, causing the menus to be deleted from the window instance.

19.5 SUBCLASSES OF WINDOW

19.5 Subclasses of Window

This section describes the classes **NonRectangularWindow**, **IconWindow**, and **LoopsIcon** and functionality associated with them.

Name	Type	Description
NonRectangularWindow	Class	Provides the capability for windows to act as icons.
CreateWindow	Method	Creates a window that acts like an icon.
EditIcon	Method	Edits an icon bitmap.
EditMask	Method	Edits a mask bitmap.
Invert	Method	Inverts the image of an icon; that is, reverses its black-white pattern.
Shape	Method	Prevents the window from being shaped by calling **LoopsHelp**.
IconWindow	Class	Provides some menu options for icon windows.
LoopsIcon	Class	Provides an icon that is part of the LOOPS user interface.
PutSavedValue	Function	Stores a value. This is called from within browser and inspector menu events.
SavedValue	Function	Retrieves a saved value.

NonRectangularWindow [Class]

Description: Provides the capability for windows to act as icons.

MetaClass: Class

Supers: Window

Instance Variables:

icon Allows a bitmap to be used as an icon to be stored in the instance. If the **bitMap** property is set to a symbol whose value is a bitmap, then that bitmap will be used. The default value is NIL.

mask Allows a bitmap to be used as an icon mask to be stored in the instance. If the **bitMap** property is set to a symbol whose value is a bitmap, then that bitmap will be used. The default value is NIL.

(← *self* **CreateWindow**) [Method of NonRectangularWindow]

Purpose: Creates a window that acts like an icon.

Behavior: Determines if **icon** and **mask** or the property **bitMap** have values. If so, it uses those within a call to **ICONW**. If not, it sends the messages **EditIcon** and **EditMask**.

This method is invoked by the system if the instance variable **window** is not yet bound to a value and it is accessed.

Arguments: *self* A window instance.

Returns: Value returned from **ICONW**.

Categories: Window

Specializes: Window

(← *self* **EditIcon**) [Method of NonRectangularWindow]

Purpose: Edits an icon bitmap.

Behavior: Calls **EDITBM** with the value of the instance variable **icon** and assigns **icon** to the returned value.

Arguments: *self* A window instance.

Returns: Value returned from **EDITBM**.

Categories: NonRectangularWindow

(← *self* **EditMask**) [Method of NonRectangularWindow]

Purpose: Edits a mask bitmap.

Behavior: Calls **EDITBM** withthe value of the instance variable **mask** if it is non-NIL, or a copy of **icon,** and assigns **mask** to the returned value.

Arguments: *self* A window instance.

Returns: Value returned from **EDITBM**.

Categories: NonRectangularWindow

(← *self* **Invert**) [Method of NonRectangularWindow]

Purpose: Inverts the image of an icon; that is, reverses its black-white pattern.

Behavior: Modifies the bitmap of the **ICONIMAGE** window property of the Lisp window pointed to by *self*.

Arguments: *self* A window instance.

Returns: Used for side effect only.

Categories: Window

Specializes: Window

(← *self* **Shape**) [Method of NonRectangularWindow]

Purpose/Behavior: Prevents the window from being shaped by calling **LoopsHelp**.

This method is provided to restrict the shaping of this class of window, not to provide additional functionality.

Arguments: *self* A window instance.

Returns: Used for side effect only.

Categories: Window

Specializes: Window

IconWindow [Class]

Description: Provides some menu options for icon windows.

The menu behavior of this class is different from the class **Window** in that the item lists are stored on instance variables and not class variables.

MetaClass: Class

Supers: NonRectangularWindow

Instance Variables:

RightButtonItems
A list that defines the menu that will appear when the right mouse button is pressed when the cursor is in the window. The default value is (Move).

MiddleButtonItems
A list that defines the menu that will appear when the middle mouse button is pressed when the cursor is in the window. The default value is NIL.

LeftButtonItems
A list that defines the menu that will appear when the left mouse button is pressed when the cursor is in the window. The default value is (Move).

ShiftMiddleButtonItems
A list that defines the menu that will appear when the middle mouse button is pressed when the cursor is in the window and the **Meta** key is pressed. The default value is NIL.

ShiftLeftButtonItems
A list that defines the menu that will appear when the left mouse button is pressed when the cursor is in the window and the **Meta** key is pressed. The default value is (Move).

LoopsIcon [Class]

Description: Implements the LOOPS icon which is part of the LOOPS user interface to LatticeBrowsers (see Chapter 10, Browsers).

MetaClass: Class

Supers: NonRectangularWindow

Class Variables:

RightButtonItems
A list that defines the menu that will appear when the right mouse button is pressed when the cursor is in the window. The default value is (Close Move).

MiddleButtonItems
A list that defines the menu that will appear when the middle mouse button is pressed when the cursor is in the window. The default value is (("Browse File" (...))).

LeftButtonItems
A list that defines the menu that will appear when the left mouse button is pressed when the cursor is in the window. The default value is (("Browse Class" (...))).

Instance Variables:

savedValue Used by the functions **PutSavedValue** and **SavedValue.** The default value is NIL.

icon The property **bitMap** has the value **BlackLoopsIconBM.** The default value is NIL.

mask The property **bitMap** has the value **LoopsIconShadow.** The default value is NIL.

(**PutSavedValue** *value*) [Function]

Purpose: Stores a value. This is called from within browser and inspector menu events.

Behavior: Sets the instance variable **savedValue** of the prototype instance of the class **LoopsIcon** to *value*. Also sets the top level binding of **IT** to *value*; see the *Interlisp-D Reference Manual* for information on **IT**.

Arguments: *value* Any arbitrary data.

Returns: *value*

(**SavedValue**) [Function]

Purpose: Retrieves a saved value.

Behavior/Returns: Gets the value of the instance variable **savedValue** of the prototype instance of the class **LoopsIcon**.

19.6 LISP WINDOWS

19.6 Lisp Windows

The methods in this section define the interface between LOOPS windows and Lisp windows. These methods are used internally by the system, and will rarely be used or specialized by users.

Name	Type	Description
AttachLispWindow	Method	Gives a LOOPS window a Lisp window.
CreateWindow	Method	Creates a Lisp window.
DetachLispWindow	Method	Forgets about the current Lisp window.
GetWrappedValue	Method	Gets the value wrapped in the active value.
HasLispWindow	Method	Checks if a Lisp window has ever been created for the LOOPS window.
PutWrappedValue	Method	Replaced the value wrapped in the active value.

(← *self* **AttachLispWindow** *window*) [Method of Window]

Purpose/Behavior: Used to associate a LOOPS window with a Medley window. This detaches any currently attached window before attaching a new one, and fills in the instance variables **left**, **bottom**, **width**, **height**, and **title** from the Lisp window.

Arguments: *self* An instance of a LOOPS window.

window Must be a window pointer.

Returns: Used for side effect only.

Categories: Window

(← *self* **CreateWindow**) [Method of Window]

Purpose/Behavior: Creates a Lisp window for a LOOPS window but does not open it.

Arguments: *self* Pointer to a LOOPS window.

Returns: The window.

Categories: Window

Specializations: NonRectangularWindow

(← *self* **DetachLispWindow**) [Method of Window]

Purpose/Behavior: Removes the pointer from the LOOPS window to the Lisp window.

Arguments: *self* Pointer to a LOOPS window.

Returns: Used for side effect only.

Categories: Window

(← *self* **GetWrappedValue** *containingObj varName propName type*) [Method of LispWindowAV]

Purpose/Behavior: Used by the system to fetch the Medley window from a LOOPS window. If the local state of this active value is not a window, it is made a window.

Arguments: *self* An instance of **LispWindowAV**.

containingObj A LOOPS window.

varName Variable associated with the wrapped value.

propName Used internally.

type Used internally.

Returns: A Lisp window.

Categories: LispWindowAv

Specializes: LocalStateActiveValue

(← *self* **HasLispWindow**) [Method of Window]

Purpose/Behavior: Checks if Lisp window has ever been created for this LOOPS window.

Arguments: *self* A LOOPS window.

Returns: The window pointer, if one exists; else NIL.

Categories: Window

(← *self* **PutWrappedValue** *containingObj varName newvalue propName*) [Method of LispWindowAV]

Purpose/Behavior: Places the window *newvalue* as local state of the active value.

Arguments: *self* An instance of **LispWindowAV.**

containingObj A LOOPS window.

varName Variable associated with the wrapped value.

propName Used internally.

type Used internally.

Returns: The window pointer, if one exists.

Categories: LispWindowAV

Specializes: LocalStateActiveValue

[This page intentionally left blank]

20. SYSTEM VARIABLES AND FUNCTIONS

This section describes the following system variables and functions. These variables are set within the file **LOADLOOPS** or when the function **LOADLOOPS** is executed.

Name	Type	Description
LoopsVersion	Variable	Identifies a release of LOOPS.
LoopsDate	Variable	Identifies the date when the function **LOADLOOPS** is executed.
FEATURES	Variable	Has the symbol LOOPS added to it when the function **LOADLOOPS** is evaluated.
LoadLoopsForms	Variable	Contains a list of forms that are evaluated when LOOPS is loaded.
LispUserFilesForLoops		Variable Contains a list of files required by LOOPS.
OptionalLispuserFIles	Variable	Contains a list of files that is loaded when LOOPS is loaded.
LOOPSDIRECTORY	Variable	Contains the connected directory when **LOADLOOPS** is loaded.
LOOPSLIBRARYDIRECTORY	Variable	Contains the directory where the LOOPS library files reside.
LOOPSUSERSDIRECTORY	Variable	Contains the directory where the LOOPS Users' Modules files reside.
LOOPSUSERSRULESDIRECTORY	Variable	Contains the directory where the LOOPS Rules User Module file resides.
LoopsPatchFiles	Variable	Contains a list of LOOPS files that are loaded when LOOPS is loaded.
LOOPSFILES	Variable	Contains a list of LOOPS files that are loaded by the function **LOADLOOPS**.
ClearAllCaches	Function	Clears all caches used by LOOPS.
ClearAllCaches	Variable	Contains a list of forms that are evaluated within a call to the function **ClearAllCaches**.

LoopsVersion [Variable]

Set to uniquely identify a release of LOOPS.

LoopsDate [Variable]

Set to the value of (DATE) when the function **LOADLOOPS** is evaluated.

FEATURES [Variable]

Has the symbol LOOPS added to it when the function **LOADLOOPS** is evaluated. See the *Common Lisp: the Language* for more information on ***FEATURES***.

LoadLoopsForms [Variable]

Contains a list of forms that are evaluated when LOOPS is loaded. Initialized to NIL using the File Manager command **INITVARS** (see the *Lisp Release Notes* and the *Interlisp-D Reference Manual*).

LispUserFilesForLoops [Variable]

Contains a list of files required by LOOPS. Initialized to (**GRAPHER**).

OptionalLispuserFiles [Variable]

Contains a list of files that is loaded when LOOPS is loaded. Initialized to NIL using the File Manager command **INITVARS**.

LOOPSDIRECTORY [Variable]

Initialized to the directory from which the file **LOADLOOPS** is loaded using the File Manager command **INITVARS**.

LOOPSLIBRARYDIRECTORY [Variable]

Contains the directory where the LOOPS library files reside.

LOOPSUSERSDIRECTORY [Variable]

Contains the directory where the LOOPS Users' Modules files reside.

LOOPSUSERSRULESDIRECTORY [Variable]

Contains the directory where the LOOPS Rules User Module file resides.

LoopsPatchFiles [Variable]

Contains a list that can be passed to **FILESLOAD** (see the *Lisp Release Notes* and the *Interlisp-D Reference Manual*) and is used during the loading of LOOPS. Initialized to ((**LOAD FROM VALUEOF LOOPSDIRECTORY**) **MASTERSCOPE MSPARSE**).

LOOPSFILES [Variable]

Contains the list of LOOPS files loaded by **LOADLOOPS** when building a LOOPS sysout.

(**ClearAllCaches**) [Function]

Purpose: Clears all caches used by LOOPS

Behavior: In addition to clearing some caches used to speed up method and instance variable lookup, this clears the hash array **CLISPARRAY** (see the *Lisp Release Notes* and the *Interlisp-D Reference Manual*) and sends the **ClearMenuCache** message to any open LOOPS windows or their icons.

Returns: NIL

ClearAllCaches [Variable]

Contains a list of forms, each of which is evaluated within a call to the function **ClearAllCaches**. Initially set to NIL.

[This page intentionally left blank]

XEROX LOOPS REFERENCE MANUAL

XEROX

Lyric/Medley Release
July 1988

XEROX LOOPS REFERENCE MANUAL

Lyric/Medley Release

July 1988

TABLE OF CONTENTS

2. INSTANCES

3. CLASSES

4. METACLASSES

8. ACTIVE VALUES

9. DATA TYPE PREDICATES AND ITERATIVE OPERATORS

10. BROWSERS

18. USER INPUT/OUTPUT MODULES

19. WINDOWS

LIST OF FIGURES

[This page intentionally left blank]

Overview of the Manual

The *Xerox LOOPS Reference Manual* provides a detailed description of all the methods, functions, classes, and other items available in Xerox's Lisp Object-Oriented Programming System, Xerox LOOPS (TM). This manual describes the Lyric/Medley Release of Xerox LOOPS, which runs under the Lyric and Medley (with a small patch) releases of Xerox Lisp.

This manual is for people who are familiar with Xerox LOOPS programming principles, and is not intended to teach you Xerox LOOPS or how to use it. Please contact your LOOPS distributor for information about classes and training material.

Organization of the Manual and How to Use It

This manual is divided into chapters, with most chapters focusing on a particular aspect of Xerox LOOPS. The organization of this manual is similar to the *Interlisp-D Reference Manual*.

A Table of Contents is included at the beginning of the manual to help you find specific material. Chapters that have four levels of headings also have internal Tables of Contents. At the end of the manual, a Glossary is included to define terms within the context of Xerox LOOPS.

All readers should review Chapter 1, Introduction, before referring to specific material.

Conventions

This manual uses the following conventions:

- Case is significant in Xerox LOOPS and Lisp. All selectors, methods, arguments, etc., must be typed as shown. Typically, this means that method names are capitalized and variables are not.

- Arguments appear in italic type. Optional arguments are indicated by a dash (-).

- Selectors, methods, functions, objects, classes, and instances appear in bold type.

  For example, a message sending form appears as follows:

  (_ *self* **Selector** *Arg1 Arg2* -)

- Examples appear in the following typeface:

```
89_(_LOGIN)
```

- All examples are typed into an Interlisp Exec. This is the recommended Exec for all Xerox LOOPS expressions.

- Methods with an exclamation mark (!) suffix usually perform operations deeply into class structure instead of only on a given object.
- Methods with a question mark (?) suffix usually are predicates; that is, truth functions.
- Method names often appea in the form **ClassName.SelectorName**.
- Cautions describe possible dangers to hardware or software.
- Notes describe related text.

This manual describes the Xerox LOOPS items (functions, methods, etc.) by using the following template:

Purpose: Gives a short statement of what the item does.

Behavior: Provides the details of how the item operates.

Arguments: Describes each argument in the following format:

argument Description

Returns: States what the item returns, and does not appear if the item does not return a value. The phrase "Used as a side effect only." means that the purpose of the item is to perform a computation or action that is independent of any returned value, not to return a particular value.

Categories: A way to group related methods. For example, all the methods releated to Masterscope on the class **FileBrowser** have the category Masterscope, not **FileBrowser**. This item appears only for methods.

Specializes: The next higher class in the class hierarchy that contains a method with the same selector; only appear for methods. For example, **the manual entry for RectangularWindow.Open** would say that it specializes **Window.Open**, since **Window** is the first superclass of **RectangularWindow** that implements a method for **Open**.

Specializations: The next lower class(es) in the class hierarchy that contains method(s) with the same selector; only appears for methods. For example, the manual entry for **Window.Open** would say that it has a specialization of **RectangularWindow.Open** since **RectangularWindow** is a subclass of **Window** and has its own version of **Open** method.

Example: An example is often included to show how to use the item and what result it produces. Some examples may appear differently on your system, depending on the settings of various print flags. See Chapter 18, Reading and Printing, for details.

References

The following books and manuals augment this manual.

Xerox LOOPS Release Notes

Xerox LOOPS Library Modules Manual

Xerox LOOPS Users' Modules Manual

Interlisp-D Reference Manual

Common Lisp: the Language by Guy Steele

Xerox Common Lisp Implementation Notes, Lyric and Medley Releases

Xerox Lisp Release Notes, Lyric and Medley Releases

Xerox Lisp Library Modules Manual, Lyric and Medley Releases

[This page intentionally left blank]

Writer's Notes -- Conventions

This file includes notes on conventions for *Xerox LOOPS Reference Manual*, Koto Release. This manual is packaged with the *Xerox LOOPS Release Notes* and *Xerox LOOPS Library Packages Manual* to form one binder, part number 3103310.

Writer: Rosie (Raven) Kontur

Printing Date: <DD> <MM> 1987

Directories and Files

The directory {ERIS}<Doc>LoopsKoto>Ref> contains the files for the manual. This directory has the following subdirectories:

- {ERIS}<Doc>LoopsKoto>Ref>X-Index> contains IMPTR files needed to produce the index as well as the index itself.

- {ERIS}<Doc>LoopsKoto>Ref>Z-ReleaseInfo> contains this file on writing conventions and a file on production details.

Filenames describe the contents of the file. For example, the filename

{ERIS}<Doc>LoopsKoto>Ref>Ch01-Intro.tedit

contains Chapter 1, Introduction.

Some chapters (for example, 5, 10, and 19) have internal tables of contents. Chapter 10, which is the largest chapter and has the most bitmaps, is divided among three files.

Assemble the files in the following order for the manual:

{ERIS}<Doc>LoopsKoto>Ref>A1-TitlePage.tedit
{ERIS}<Doc>LoopsKoto>Ref>A2-TOC.tedit
{ERIS}<Doc>LoopsKoto>Ref>A3-LOF.tedit
{ERIS}<Doc>LoopsKoto>Ref>A4-Preface.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch01-Intro.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch02-Instances.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch03-Classes.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch04-Metaclasses.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch05-TOC.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch05-ActiveValue.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch06-Methods.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch07-MsgForms.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch08-IterStmts.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch09-Misc.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch10-TOC.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch10a-Browsers1.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch10b-Browsers2.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch10c-Browsers3.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch11-Errors.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch12-Breaking.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch13-Editing.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch14-FilePkg.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch15-Masterscope.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch16-Performance.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch17-Processes.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch18-ReadPrint.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch19-TOC.tedit
{ERIS}<Doc>LoopsKoto>Ref>Ch19-UserIOPkgs.tedit

{ERIS}<Doc>LoopsKoto>Ref>Ch20-Windows.tedit
{ERIS}<Doc>LoopsKoto>Ref>PndxA-Previous.tedit
{ERIS}<Doc>LoopsKoto>Ref>ZZ-Glossary.tedit
{ERIS}<Doc>LoopsKoto>Ref>X-Index>Index-Final.tedit

About the Index

Creating a properly formatted index takes a certain amount of work (about 2 hours).

- Type in your Executive `(TEDIT (MAKE.IM.INDEX T NIL ’(filenames) NIL))`
- Save the resulting file in {ERIS}<Doc>LoopsKoto>Ref>RefIndex-Raw.tedit.
- Now start dithering. The task is the change all the separators in the file. For example, the index program returns pages in the form 1.2; 10.5,38; 18.3-4. The Index format we use is in the form 1-2; 10-5; 10-38; 18-3. Unfortunately, you can’t make all the necessary changes in a global substitute, since many of the separators are also in the text and the comma requires that you repeat the chapter number. Here goes:

  -- Fix the commas. Do a Find on a comma (,). Everyhwere a comma occurs in a page number, change it to a semicolon (;) and repeat the chapter number and a dot before the page number. For example, 10.5,38 becomes 10-5; 10-38.

  -- Fix the dashes. Do a Find on a dash (-). Everyhwere a dash occurs in a page number, delete it and the following page number.

  -- Fix the period. Except for he form #., you can use a global substitute from a period to a dash (. to -). This takes about 20-30 minutes, as there about 700 substitutions.

  I really recommend saving the file at this point.

- Hardcopy the index. Make whatever page breaks and other changes you feel are necessary. Make a final hardcopy and save the file.

Conventions

This manual uses the following conventions:

- Case is significant in Xerox LOOPS and Lisp. All selectors, methods, arguments, etc., must be typed as shown. Typically, this means that method names are capitalized and variables are not.
- Arguments appear in italic type.
- Selectors, methods, functions, objects, classes, and instances appear in bold type.

  For example, a method appears as follows:

  (_ *self* **Selector** *Arg1 Arg2*)

- Examples appear in the following typeface:

  `89_(_LOGIN)`

- Methods with an exclamation mark (!) suffix usually perform operations deeply into class structure instead of only on a given object.
- Methods with a question mark (?) suffix usually are predicates; that is, truth functions.
- Methods often appear in the form **ClassName.SelectorName**.
- Cautions describe possible dangers to hardware or software.
- Notes describe related text.

This manual describes the Xerox LOOPS items (functions, methods, etc.) by using the following template:

Purpose: Gives a short statement of what the item does.

Behavior: Provides the details of how the item operates.

Arguments: Describes each argument in the following format:

argument Description

Returns: States what the item returns, and does not appear if the item does not return a value. The phrase "Used as a side effect only." means that the purpose of the item is to perform a computation or action that is independent of any returned value, not to return a particular value.

Categories: A way to group related methods. For example, all the methods releated to Masterscope on the class **FileBrowser** have the category Masterscope, not **FileBrowser**. This item appears only for methods.

Specializes: The next higher class in the class hierarchy that contains a method with the same selector. For example, **RectangularWindow.Open** can specialize **Window.Open**. This appears only for methods.

Specializations: The next lower class in the class hierarchy that contains a method with the same selector. For example, **Window.Open** is a specialization of **RectangularWindow.Open**. This appears only for methods.

Example: An example is often included to show how to use the item and what result it produces. Some examples may appear differently on your system, depending on the settings of various print flags. See Chapter 18, Reading and Printing, for details.

Style Sheet Addenda

Here are some guidelines I used when writing the LOOPS manuals. Items appear in rather random order.

- Avoid contractions.
- Avoid subscripts. Use WORD1 rather than WORD to avoid inconsistent line leading.
- Avoid wording that starts "Note that..." or "Notice that...". Either make it a note with correct format or eliminate the "Note that".
- Use semicolons rather than m-dashes.
- Each item in the template starts with an initial capital letter; e.g., "Describes..."
- The arguments are identical in the call and in the argument description.
- Parenthesies appear around expressions and square brackets appear around the name of the functionality.
- The arrow in the expression is the NS character ←, not _. These characters appear similarly when printed, but differently on the screen. See the section, "Special Notes and Cautions," for details.
- A period appears after the word None, after argument descriptions, and Returns: item.
- Items are set to or return T (instead of true).

- Menus contain options, not items or selections.
- You drag (not roll) the mouse to the right of a menu option to see its submenu.
- Use "above" and "below" when referrering to things in the same section, section numbers and names when referrering to things in the same chapter, and chapter numbers and names when referrering to things in another chapter.
- Please study the following style sheet carefully before you start to edit. The various appearances of active value and annotated values are especially crazy making.

  These things appear in **bold**:
  class variables
  functions
  instance variables
  messages
  methods
  variables

  ActiveValue - specific class/instance
  active value - general information
  activeValue - previous implementation of **ActiveValue**

  annotatedValue - data type
  AnnotatedValue - specific class
  annotated values - general information

  bitmap

  data type

  file package
  filecoms

  inspector

  Lisp Library package
  localState - instance variable

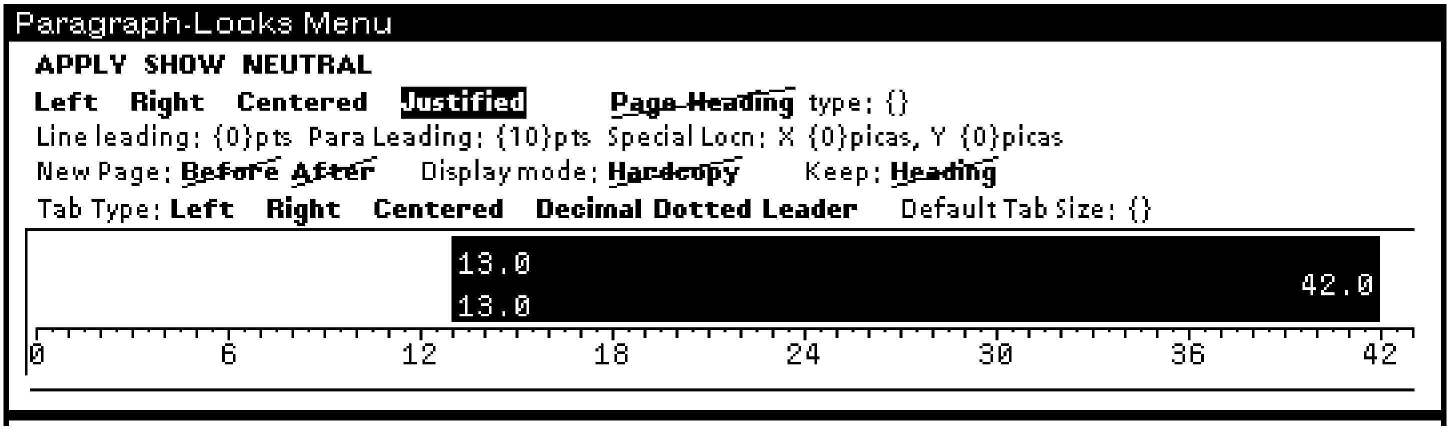
  non-NIL

  prettyprints

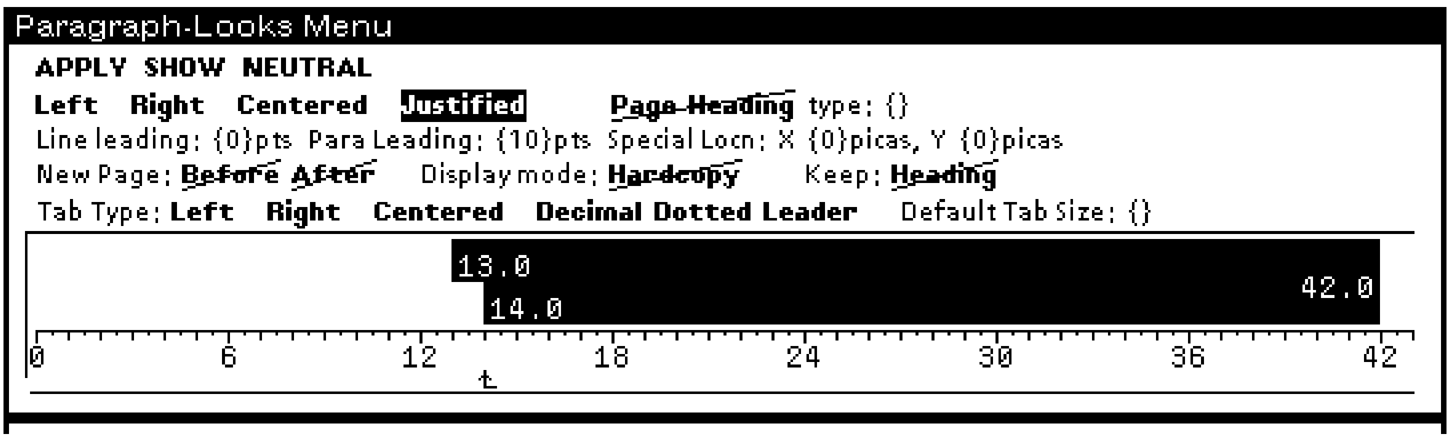
  supers list

Paragraph Formatting

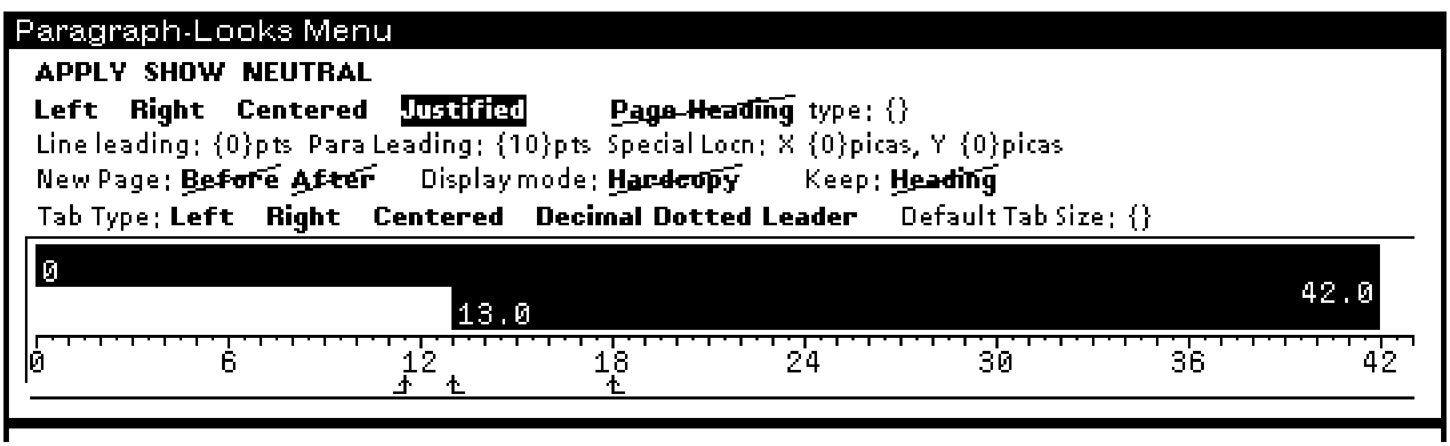
The text has the following format:



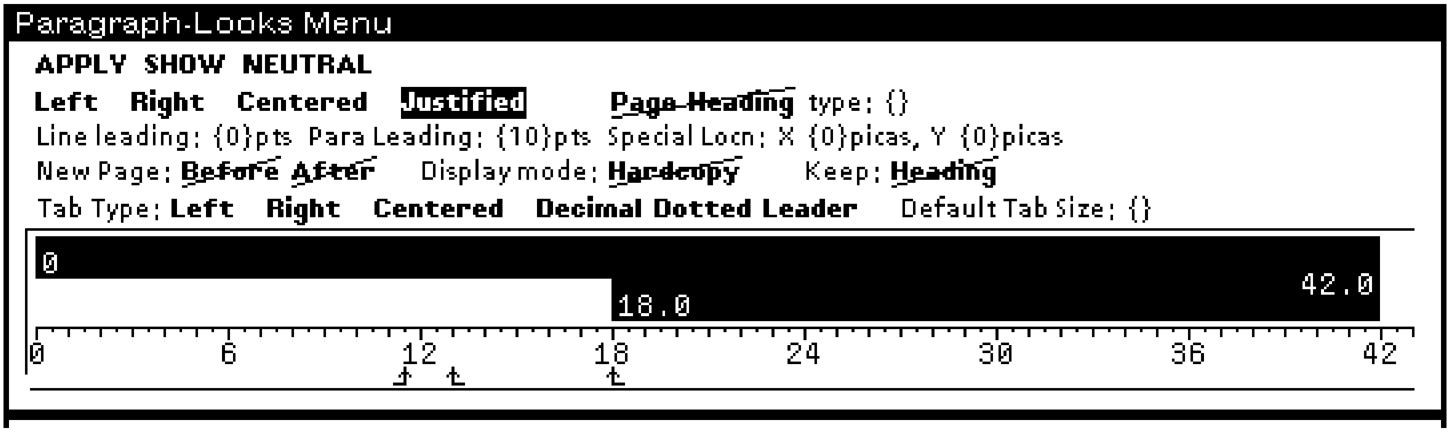
Bulleted lists have the following format:



The template has the following format:

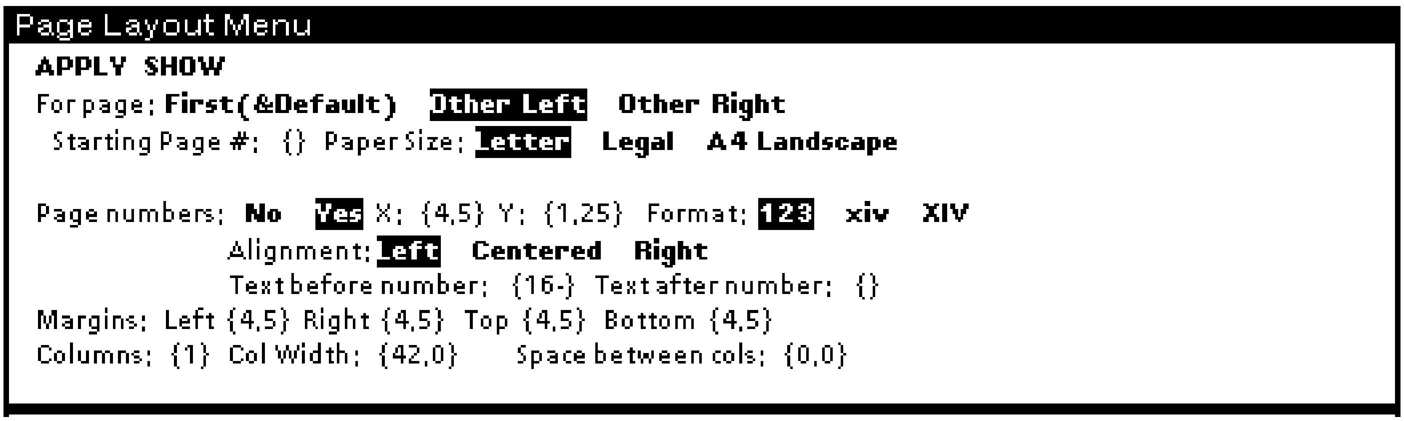


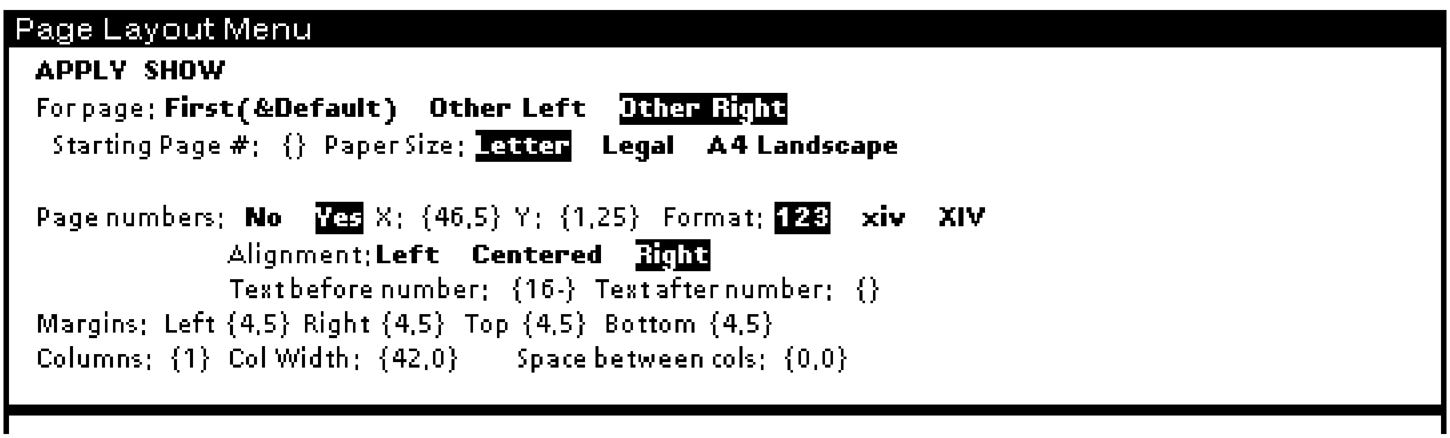
The Arguments section of the template has the second line start at 18 instead of 13.



Page Layout

Page numbering, especially "text before", varies with the chapter.





Bitmaps, Graphs, and Sketches

Scale for bitmaps is 0.8.

Special Notes and Cautions

Make sure you have changed the underscore to be a left arrow before loading any files. To do this,

- Enter the following commands into your Executive:

```
(GETCHARBITMAP (CHARCODE _) '(MODERN 10 MRR))
(EDITBM IT)
```

- When the bitmap editor apears, delete the underscore and insert the following left arrow:

```
..........
..........
..........
..........
..........
....X.....
...XX.....
..XXXXXX..
...XX.....
....X.....
..........
..........
..........
..........
```

- Finally, enter the following commands into your Executive to store the pattern:

```
   (PUTCHARBITMAP (CHARCODE _) '(MODERN 10 MRR) IT)
   (PUTCHARBITMAP (CHARCODE _) '(MODERN 10 BRR) IT)
   (PUTCHARBITMAP (CHARCODE _) '(TERMINAL 10 MRR) IT)

(PUTCHARBITMAP (CHARCODE _) '(TERMINAL 10 BRR) IT)
   (PUTCHARBITMAP (CHARCODE _) '(TERMINAL 12 BRR) IT)
```

[This page intentionally left blank]

Writer’s Notes -- Production Details

This file includes notes on the production of *Xerox LOOPS Reference Manual*, Koto Release. This manual is packaged with the *Xerox LOOPS Release Notes* and *Xerox LOOPS Library Packages Manual* to form one binder, part number 3103310.

Writer: Rosie (Raven) Kontur

Printing Date: <DD> <MM> 1987

Files Needed

To edit or print the manual, make sure you have the following files loaded:

IMTOOLS
SKETCH
GRAPHER

Fonts Used

{ERIS}<LISP>FONTS>

Modern font
18-point bold
14-point bold
12-point bold
10-point regular
10-point italic
10-point bold

Terminal font
12-point bold
10-point regular
10-point bold

Printing Information

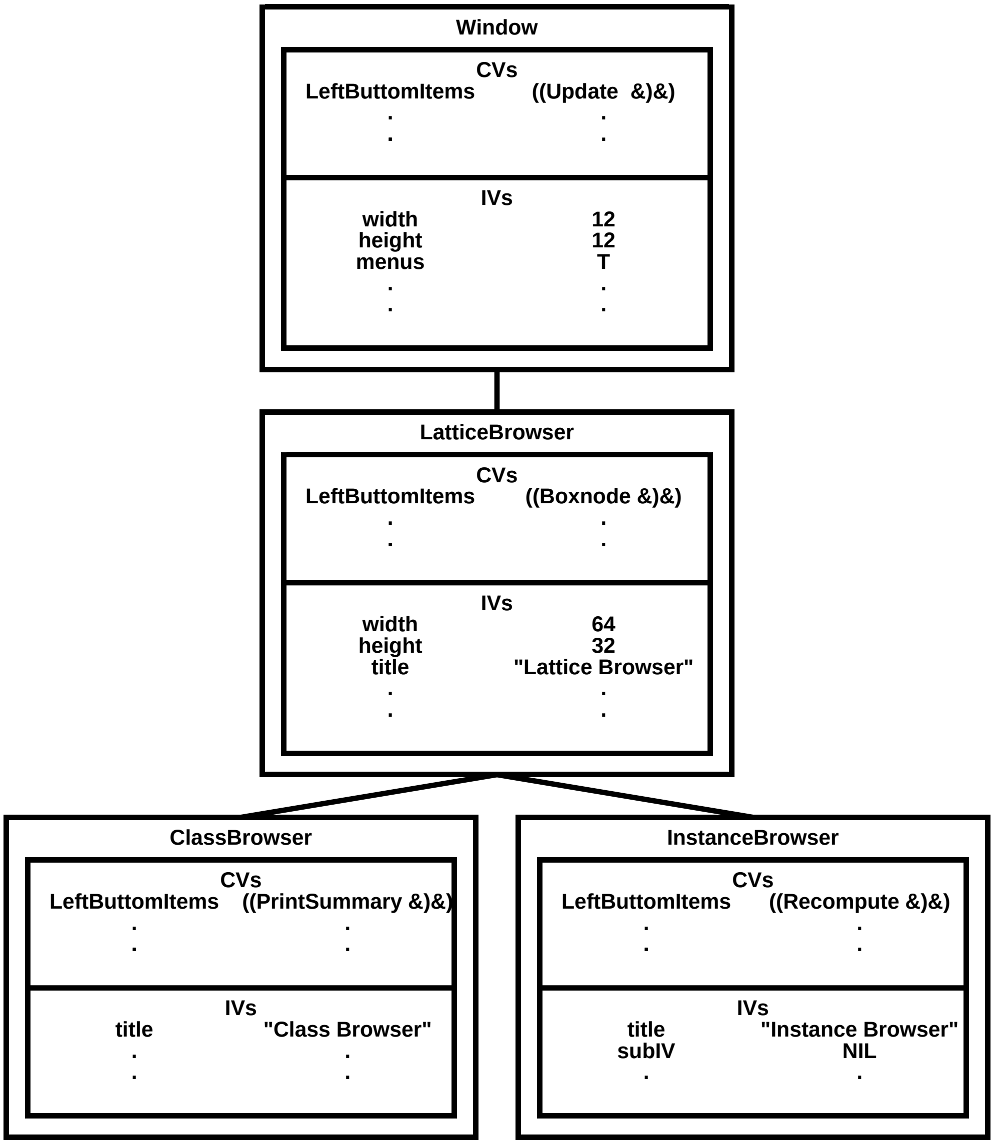
The manual was printed under a Lyric sysout on the Tsunami printer.

Artwork

- The art for the Beta Release was created in Pasadena, and is not in any files here.
- The file {ERIS}<Dc>LoopsBeta>BinderCover.tedit contains the cover page for the binder.
- The file {ERIS}<Doc>LoopsBeta>CommentForm.tedit contains the comment form for the entire documentation set. The prepaid mailer (see department file) must be added to complete this form.

- The details for packaging are in the following files:
  {ERIS}<Doc>loops>ProductionSpecs>onfiguration-request.tedit

  {ERIS}<Doc>loops>ProductionSpecs>Packaging.tedit


Window
CVs
LeftButtomItems ((Update &)&)
IVs
width 12
height 12
menus T
LatticeBrowser
CVs
LeftButtomItems ((Boxnode &)&)
IVs
width 64
height 32
title "Lattice Browser"
ClassBrowser
CVs
LeftButtomItems ((PrintSummary &)&)
IVs
title "Class Browser"
InstanceBrowser
CVs
LeftButtomItems ((Recompute &)&)
IVs
title "Instance Browser"
subIV NIL

Replace this page with
Table of Contents
tab

Replace this page with
1. Introduction
tab

Replace this page with
2. Instances
tab

Replace this page with
3. Classes
tab

Replace this page with
4. Metaclasses
tab

Replace this page with
5. Accessing Data
tab

Replace this page with
6. Methods
tab

Replace this page with
7. Message Sending Forms
tab

Replace this page with
8. Active Values
tab

Replace this page with
9. Data Type Predicates
tab

Replace this page with
10. Browsers
tab

Replace this page with
11. Errors and Breaks
tab

Replace this page with
12. Breaking and Tracing
tab

Replace this page with
13. Editing
tab

Replace this page with
14. File Package
tab

Replace this page with
15. Performance Issues
tab

Replace this page with
16. Processes
tab

Replace this page with
17. Reading and Printing
tab

Replace this page with
18. User Input/Output Modules
tab

Replace this page with
19. Windows
tab

Replace this page with
20. System Variables and
Functions
tab

Replace this page with
A. Active Values in Buttress
LOOPS
tab

Replace this page with
Glossary
tab

Replace this page with
Index
tab

GLOSSARY

abstract class	A class which cannot be instantiated, for example, **ActiveValue**.
active value	The mechanism that carries out access-oriented programming for variables in Xerox LOOPS. Active values send messages as a side effect of having an object's variable referenced.
activeValue	The previous implementation of the active value concept.
ActiveValue	An abstract class that defines the general protocol followed by all active value objects.
annotatedValue	A special Interlisp-D data type that wraps each **ActiveValue** instance.
AnnotatedValue	An abstract class that allows an annotatedValue to be treated as an object.
browser	A window that allows you to examine and change items in a data structure.
class	A description of one or more similar objects; that is, objects containing the sames types of data fields and responding to the same messages.
class inheritance	The means by which a class inherits variables, values, and methods from its super class(es).
class lattice	A network showing the inheritance relationship among classes.
class variable (CV)	A variable that contains information shared by all instances of the class. A class variable is typically used for information about a class taken as a whole.
inheritance	The means by which you can organize information in objects, create objects that are similar to other objects, and update objects in a simplified way.
Inspector	A Xerox Lisp display program that has been modified to allow you to view classes, objects, and active values.
instance	An object described by a particular class. Every object within Xerox LOOPS is an instance of exactly one class.
instance variable (IV)	A variable that contains information specific to an instance.
instantiate	To make a new instance of a class.
lattice	An arrangement of nodes in a hierarchical network, which allows for multiple parents of each node.
Masterscope	A Xerox Lisp Library Module program analysis tool that has been modified to allow analysis of Xerox LOOPS files.
message	A command sent to an object that activates a method defined in the object's class. The object responds by computing a value that is returned to the sender of the message.
metaclass	Classes whose instances are classes or abstract classes.
method	What an object applies to the arguments of a message it receives. This is similar to a procedure in procedure-oriented programming, except that here, you determine the message to send and the object receiving the message

determines the method to apply, instead of the calling routine determining which procedure to apply.

mixin A class that is used in conjunction with another class to create a subclass. Mixins never have instances, and hence have **AbstractClass** as their metaclass.

object A data structure that contains data and a pointer to functionality that can manipulate the data.

property list A place for storing additional information on classes, their variables, and their methods.

selector Part of a message that is sent to an object. The object uses the selector to determine which method is appropriate to apply to the message arguments.

self A method argument that represents the receiver of the message.

specialization The process of creating a subclass from a class, or the result of that process.

subclass A class that is a specialization of another class.

super class A class from which a given class inherits variables, values, and methods.

Tofu An acronym for Top of the universe, which is the highest class in the Xerox LOOPS hierarchy.

Unique Identifier (UID) An alphanumeric identifier that Xerox LOOPS uses to store and retrieve objects. Objects do not have UIDs unless they are named, are instances of indexed objects, or are instances printed to a file.

wrap Objects have fields that can contain data. Some **ActiveValue** can be added so this data is stored within it. When this occurs, the **ActiveValue** wraps the data.

Venue

LOOPS Release Notes

November, 1991

Address comments to:
Venue
User Documentation
1549 Industrial Road
San Carlos, CA 94070
415-508-9672

LOOPS RELEASE NOTES

November 1991

Text was written and produced with Venue text formatting tools; Xerox printers were used to produce text masters.

1. RELEASE OVERVIEW

LOOPS is a multiple-paradigm programming system designed to be used in a variety of artificial intelligence applications. LOOPS requires the Medley Release of Lisp.

1.1 System Configuration

You need the following minimum configuration to run Xerox LOOPS:

1186 Workstation	1108 Workstation
40 megabyte disk drive	40-42 megabyte disk drive
2 megabyte memory	2 megabyte memory

1.2 What's Included in This Release

Xerox LOOPS includes the following items:

- Software on floppy disks.

  See Chapter 2, Changes from Koto LOOPS, for how the Lyric/Medley Release of LOOPS is different from the Koto Release.

- Documentation in binders.

  See Chapter 3, Release Documentation, for details of release documentation.

[This page intentionally left blank]

2. CHANGES FROM KOTO LOOPS

This chapter describes how the current release of LOOPS, Lyric/Medley LOOPS, differs from the Koto Release. With the addition of a patch file, the current Loops release will also run under the Medley release. The inclusion of the patch file is automatically handled by the LOOPS installation process. To convert Koto LOOPS files into Lyric/Medley LOOPS files, see the LOOPS Users' Modules CONVERT-LOOPS-FILES.

A number of significant fixes to both the documentation and software have been made. In addition, LOOPS behaves differently due to new features of Lisp found beginning with the Lyric release:

- LOOPS vs. Packages

  Lyric has Common Lisp functionality, including packages and new case-insensitive readtables. All LOOPS symbols are in the INTERLISP: or IL: package, and LOOPS is case-sensitive, so it is much easier to type LOOPS expressions into an Interlisp Exec (see the *Lisp Release Notes, Lyric Release,* for more on different Exec types).

- Editing

  Lyric has a new default structure editor, SEdit. DEdit is still available as a Lyric Library Module, but most people find SEdit to be much faster and easier to use. SEdit has no specific LOOPS support features yet, but the features it has support LOOPS rather well. As an example, all Lyric LOOPS development was done using SEdit. See the *Lisp Release Notes, Lyric and Medley Releases,* for more information on SEdit.

- Source File Management

  LOOPS source forms are no longer LAMBDATRAN forms; instead they use the definer system which also handles Common Lisp forms in Lyric and Medley (see the *Common Lisp Implementation Notes* for more on definers). This has several consequences:

  - The source file format for Lyric/Medley LOOPS is different. A conversion utility is provided which takes Koto LOOPS source files and converts them to Lyric/Medley LOOPS format. The converter will not work correctly on source files from the prototype Buttress version of LOOPS. These files must first be converted using Koto LOOPS.

  - LOOPS source forms are no longer of filetype FNS; they are instead METHOD-FNS.

  - Comments may appear anywhere in a definer form, since the definer system removes them before making the form executable. This also means that various comments that the LOOPS system uses as help information (doc properties and first comments in method code) now need to be strings rather than comments, so the definer system will not remove them. The conversion utility handles this for most important comments; new code should use strings in these places.

  - Method bodies can now be CL:LAMBDAs or IL:LAMBDAs, by using the **Method** function and specifying a keyword when creating the method. A

LOOPS CL:LAMBDA method can have Common Lisp features like &rest and :keywords.

- Masterscope

  Masterscope is now a LOOPS Library Module, to match Masterscope becoming a Lisp Library Module in the Lyric release of XAIE, primarily because Masterscope does not support Common Lisp yet. Masterscope still supports LOOPS, however, and many bugs in that support from Buttress and Koto LOOPS have been fixed.

  When using LOOPS Masterscope, quote all Method Names that you use. The Masterscope parser currently will not recognize Method Names unless they are quoted.

- ICONW

  ICONW is now a part of Lisp and no longer needs to be loaded to run LOOPS.

- New Compiler

  Lyric/Medley LOOPS uses the new Lyric compiler, which handles Common Lisp. This has several consequences:

  - **_Super** and the other similar functions are now lexically scoped; that is, it is now illegal to call **_Super** anywhere but within a method body, and any selector given must be the same as the selector for that method.

  - Files compiled by the new compiler have no FILECOMS. Use

    ```
    (LOADFROM <FILE> NIL 'ALLPROP)
    ```

    to load Source files so that LOOPS browsers can find them.

  - The .DFASL output of the new compiler loads much faster than .DCOMs or the .LCOMs of Lyric.

  The ByteCompiler is no longer supported for compilation of LOOPS files. With the new compiler and its macrolet facilities, a cleanup of LOOPS files requires that *DEFAULT-CLEANUP-COMPILER* be set to 'CL:COMPILE-FILE. The *Lisp Release Notes, Lyric Release,* contain more information on the new compiler and the cleanup flag. The ByteCompiler is no longer supported for compilation of LOOPS files.

- LOOPS Library Modules

  SSDigimeter will be removed from Gauges in the next release of LOOPS. DigiMeters are inherently self scaling because Meters are. Therefore, **SSDigiMeter** is redundant . Note that SSDigiMeters are also generally slower than DigiMeters.

[This page intentionally left blank]

PREFACE

Overview of the Manual

These Release Notes describe the Lyric/Medley Release of software for Xerox's Lisp Object-Oriented Programming System, LOOPS (TM). This document is directed to the people responsible for installing and testing LOOPS.

This manual describes the Lyric/Medley Release of LOOPS, which runs under the Lyric and Medley (with a small patch) Releases of Lisp.

Organization of the Manual and How to Use It

These Release Notes contain important information about the Lyric/Medley Release of LOOPS. The following chapters outline the major features of LOOPS, and highlight the principal differences between this and previous versions of LOOPS.

All readers should carefully read Chapter 1, Release Overview, Chapter 5, Reporting Procedure, and Chapter 6, Known Problems. Reading Chapter 2, Changes from Koto LOOPS, and Chapter 3, Release Documentation, is recommended for all readers. People responsible for installing LOOPS should read Chapter 4, Installation Procedures.

Conventions

These Release Notes use the following conventions:

- Case is significant in LOOPS and Lisp. All selectors, methods, arguments, etc., must be typed as shown.
- Arguments appear in italic type.
- Selectors, methods, functions, objects, classes, and instances appear in bold type.

  For example, a method appears as follows:

  (← *self* **Selector** *Arg1 Arg2*)
- Examples appear in the following typeface:

```
89←(←LOGIN)
```

- All examples are typed into an Interlisp Exec. This is the recommended Exec for all LOOPS expressions.
- Cautions describe possible dangers to hardware or software.
- Notes describe related text.

References

The following books and manuals augment this manual.

LOOPS Reference Manual

LOOPS Library Modules Manual

LOOPS Users' Modules Manual

Interlisp-D Reference Manual

Common Lisp: the Language by Guy Steele

Common Lisp Implementation Notes, Lyric Release

Lisp Release Notes, Lyric and Medley Releases

*Lisp Library Modules Manual, Lyric and Medley Release*s

XEROX LOOPS
RELEASE NOTES

XEROX

Lyric/Medley Release
July 1988

XEROX LOOPS RELEASE NOTES

Lyric/Medley Release

July 1988

TABLE OF CONTENTS

[This page intentionally left blank]

PREFACE

Overview of the Manual

These Release Notes describe the Lyric/Medley Release of software for Xerox's Lisp Object-Oriented Programming System, Xerox LOOPS (TM). This document is directed to the people responsible for installing and testing Xerox LOOPS.

This manual describes the Lyric/Medley Release of Xerox LOOPS, which runs under the Lyric and Medley (with a small patch) Releases of Xerox Lisp.

Organization of the Manual and How to Use It

These Release Notes contain important information about the Lyric/Medley Release of Xerox LOOPS. The following chapters outline the major features of Xerox LOOPS, and highlight the principal differences between this and previous versions of LOOPS.

All readers should carefully read Chapter 1, Release Overview, Chapter 5, Reporting Procedure, and Chapter 6, Known Problems. Reading Chapter 2, Changes form Koto LOOPS, and Chapter 3, Release Documentation, is recommended for all readers. People responsible for installing Xerox LOOPS should read Chapter 4, Installation Procedures.

Conventions

These Release Notes use the following conventions:

- Case is significant in Xerox LOOPS and Lisp. All selectors, methods, arguments, etc., must be typed as shown.
- Arguments appear in italic type.
- Selectors, methods, functions, objects, classes, and instances appear in bold type.

  For example, a method appears as follows:

  (_ *self* **Selector** *Arg1 Arg2*)
- Examples appear in the following typeface:

  ```
  89_(_LOGIN)
  ```
- All examples are typed into an Interlisp Exec. This is the recommended Exec for all Xerox LOOPS expressions.
- Cautions describe possible dangers to hardware or software.
- Notes describe related text.

References

The following books and manuals augment this manual.

Xerox LOOPS Reference Manual

Xerox LOOPS Library Modules Manual

Xerox LOOPS Users' Modules Manual

Interlisp-D Reference Manual

Common Lisp: the Language by Guy Steele

Xerox Common Lisp Implementation Notes, Lyric Release

Xerox Lisp Release Notes, Lyric and Medley Releases

*Xerox Lisp Library Modules Manual, Lyric and Medley Release*s

1. RELEASE OVERVIEW

Xerox LOOPS is a multiple-paradigm programming system designed to be used in a variety of artificial intelligence applications. Xerox LOOPS runs on the Xerox 1100 series of Artificial Intelligence Workstations and requires the Lyric or Medley Release of Lisp running on the Xerox Artificial Intelligence Environment (XAIE).

1.1 System Configuration

You need the following minimum configuration to run Xerox LOOPS:

1186 Workstation	1108 Workstation
40 megabyte disk drive	40-42 megabyte disk drive
2 megabyte memory	2 megabyte memory

1.2 What's Included in This Release

Xerox LOOPS includes the following items:

- Software on floppy disks.

  See Chapter 2, Changes from Koto LOOPS, for how the Lyric/Medley Release of Xerox LOOPS is different from the Koto Release.

- Documentation in binders.

  See Chapter 3, Release Documentation, for details of release documentation.

[This page intentionally left blank]

2. CHANGES FROM KOTO LOOPS

This chapter describes how the current release of Xerox LOOPS, Lyric/Medley LOOPS, differs from the Koto Release. The current Xerox LOOPS release runs under the Lyric release of Xerox Lisp. With the addition of a patch file, the current Xerox Loops release will also run under the Xerox Lisp Medley release. The inclusion of the patch file is automatically handled by the LOOPS installation process. To convert Koto LOOPS files into Lyric/Medley LOOPS files, see the Xerox LOOPS Users' Modules CONVERT-LOOPS-FILES.

A number of significant fixes to both the documentation and software have been made. In addition, Xerox LOOPS behaves differently due to new features of Xerox Lisp found beginning with the Lyric release:

- Xerox LOOPS vs. Packages

  Lyric has Common Lisp functionality, including packages and new case-insensitive readtables. All Xerox LOOPS symbols are in the INTERLISP: or IL: package, and Xerox LOOPS is case-sensitive, so it is much easier to type Xerox LOOPS expressions into an Interlisp Exec (see the *Xerox Lisp Release Notes, Lyric Release,* for more on different Exec types).

- Editing

  Lyric has a new default structure editor, SEdit. DEdit is still available as a Lyric Library Module, but most people find SEdit to be much faster and easier to use. SEdit has no specific Xerox LOOPS support features yet, but the features it has support Xerox LOOPS rather well. As an example, all Lyric LOOPS development was done using SEdit. See the *Xerox Lisp Release Notes, Lyric and Medley Releases,* for more information on SEdit.

- Source File Management

  Xerox LOOPS source forms are no longer LAMBDATRAN forms; instead they use the definer system which also handles Common Lisp forms in Lyric and Medley (see the *Xerox Common Lisp Implementation Notes* for more on definers). This has several consequences:

  - The source file format for Lyric/Medley LOOPS is different. A conversion utility is provided which takes Koto LOOPS source files and converts them to Lyric/Medley LOOPS format. The converter will not work correctly on source files from the prototype Buttress version of LOOPS. These files must first be converted using Koto LOOPS.

  - Xerox LOOPS source forms are no longer of filetype FNS; they are instead METHOD-FNS.

  - Comments may appear anywhere in a definer form, since the definer system removes them before making the form executable. This also means that various comments that the Xerox LOOPS system uses as help information (doc properties and first comments in method code) now need to be strings rather than comments, so the definer system will not remove them. The conversion utility handles this for most important comments; new code should use strings in these places.

  - Method bodies can now be CL:LAMBDAs or IL:LAMBDAs, by using the **Method** function and specifying a keyword when creating the method. A

Xerox LOOPS CL:LAMBDA method can have Common Lisp features like &rest and :keywords.

- Masterscope

  Masterscope is now a Xerox LOOPS Library Module, to match Masterscope becoming a Xerox Lisp Library Module in the Lyric release of XAIE, primarily because Masterscope does not support Common Lisp yet. Masterscope still supports Xerox LOOPS, however, and many bugs in that support from Buttress and Koto LOOPS have been fixed.

  When using LOOPS Masterscope, quote all Method Names that you use. The Masterscope parser currently will not recognize Method Names unless they are quoted.

- ICONW

  ICONW is now a part of Xerox Lisp and no longer needs to be loaded to run LOOPS.

- New Compiler

  Lyric/Medley LOOPS uses the new Lyric compiler, which handles Common Lisp. This has several consequences:

  - **_Super** and the other similar functions are now lexically scoped; that is, it is now illegal to call **_Super** anywhere but within a method body, and any selector given must be the same as the selector for that method.

  - Files compiled by the new compiler have no FILECOMS. Use

    ```
    (LOADFROM <FILE> NIL 'ALLPROP)
    ```

    to load Source files so that Xerox LOOPS browsers can find them.

  - The .DFASL output of the new compiler loads much faster than .DCOMs or the .LCOMs of Lyric.

  The ByteCompiler is no longer supported for compilation of LOOPS files. With the new compiler and its macrolet facilities, a cleanup of LOOPS files requires that *DEFAULT-CLEANUP-COMPILER* be set to 'CL:COMPILE-FILE. The *Xerox Lisp Release Notes, Lyric Release,* contain more information on the new compiler and the cleanup flag. The ByteCompiler is no longer supported for compilation of LOOPS files.

- LOOPS Library Modules

  SSDigimeter will be removed from Gauges in the next release of LOOPS. DigiMeters are inherently self scaling because Meters are. Therefore, **SSDigiMeter** is redundant . Note that SSDigiMeters are also generally slower than DigiMeters.

[This page intentionally left blank]

3. RELEASE DOCUMENTATION

This chapter describes the documentation that is part of Xerox LOOPS.

3.1 Xerox LOOPS Reference Manual

This manual provides a detailed description of all the methods, functions, classes, and other items available in Xerox LOOPS. This manual is for people who are already familiar with Xerox LOOPS programming principles.

3.2 Xerox LOOPS Library Modules Manual

This manual describes the Xerox LOOPS Library Modules: Gauges, Masterscope, and Virtual Copies. This manual is for people who want to use the additional features that these Library Modules provide.

3.3 Xerox LOOPS Users' Modules Manual

This manual describes the various Xerox LOOPS Users' Modules. This manual is for people who want to use the additional features that these Users' Modules provide.

[This page intentionally left blank]

4. INSTALLATION PROCEDURES

This chapter describes how to install the Xerox LOOPS program files, Library Modules program files, and Users' Modules program files on both the 1186 and 1108 workstations. The installation procedure for Xerox LOOPS is the same for both the Lyric and Medley releases.

4.1 Overview of the Distribution Kits

The distribution kits for Xerox LOOPS are different for the 1186 and the 1108.

4.1.1 1186 Distribution Kit

The distribution kit for Xerox LOOPS on the 1186 consists of four 5-1/4" diskettes:

- Lyric/Medley LOOPS System #1, which contains Xerox LOOPS program files.
- Lyric/Medley LOOPS System #2, which contains Xerox LOOPS program files.
- Lyric/Medley LOOPS Library, which contains the program files for the Xerox LOOPS Library Modules.
- Lyric/Medley LOOPS Users, which contains the program files for Xerox LOOPS Users' Modules.

4.1.2 1108 Distribution Kit

The distribution kit for Xerox LOOPS on the 1108 consists of three 8" diskettes:

- Lyric/Medley LOOPS System, which contains Xerox LOOPS program files.
- Lyric/Medley LOOPS Library, which contains the program files for the Xerox LOOPS Library Modules.
- Lyric/Medley LOOPS Users, which contains the program files for Xerox LOOPS Users' Modules.

4.2 Preparation

Before installing Xerox LOOPS, make sure that you have performed the following steps:

- Load the Lyric or Medley version of Xerox Lisp. Have your Lyric or Medley Release Kit handy if you are not on a network, as you may need to load parts of Lyric or Medley Xerox Lisp not previously loaded.

  A standard partition of 8 MB is acceptable. A LOOPS sysout with an average number of LISP Library and LOOPS Users' Modules will require about 11,000 pages or 5 MB.

- Boot Xerox Lisp
- Open an Interlisp Executive Window.
- Make certain the time is set.

  Machines on a Xerox network automatically get the current time of day from the net when they boot. If you are not on a network, make certain that the function (DATE) returns the current date and time. If it does not, then use the function (SETTIME) to correct it.

CAUTION

Rebooting a VMEM.PURE.STATE sysout without an Ethernet connection and without the time being set will erroneously allow you to create objects without informing you that the time is not set.

4.3 Installation

The Lyric/Medley LOOPS installation tool makes the installation of Xerox LOOPS, its Library Modules, and Users' Modules almost identical for 1186 and 1108 workstations. The differences are in the names of floppies. 1186 floppies are smaller, so there are more of them to hold the same data. The installation tool will determine what workstation you are using and prompt you with the appropriate floppy names.

1. Have your Lyric/Medley Release Kit handy, or, if you are connected to a network, set the **DIRECTORIES** and **DISPLAYFONTDIRECTORIES** variables appropriately so the sysout can find your Lyric library and font files.

2. Make the floppy drive your connected directory:

   ```
   CONN {FLOPPY}
   ```

   ```
   DIR {FLOPPY}
   ```

3. If you are using an 1108 workstation, insert the floppy labeled *Lyric LOOPS System*. If you are using an 1186 workstation, insert the floppy labeled *Lyric LOOPS System #1*. Enter the following into your Exec:

   ```
   LOAD(LOOPS)
   ```

A menu will appear that looks like this:

```
Loops System
Install from floppies
Load into sysout
```

4. Select the menu option **Install from floppies**.

   The following menu will appear:

```
Loops directories                          Click here when done
LOOPSDIRECTORY             {DSK}<LISPFILES>LOOPS>
LOOPSLIBRARYDIRECTORY      {DSK}<LISPFILES>LOOPS>LIBRARY>
LOOPSUSERSDIRECTORY        {DSK}<LISPFILES>LOOPS>USERS>
LOOPSUSERSRULESDIRECTORY   {DSK}<LISPFILES>LOOPS>RULES>
```

   This menu shows the current (or default, if unset) values of the variables LOOPS examines when it loads things.

   • If you want LOOPS installed on your local disk under {DSK}<LISPFILES>LOOPS>, just select **Click here when done**. LOOPS requires approximately 2200 pages on the local disk.

   • If you want LOOPS installed somewhere else, select the directory names using the left mouse button. Change the directory names by backspacing over them and typing new locations; select **Click here when done** when you are finished.

5. The installation tool prompts you for floppies with this menu:

```
Please insert floppy Lyric LOOPS System
           Click here when done
```

   It copies the LOOPS files from the floppies to the directories you just specified in the menu.

   When the installation tool asks you for a floppy, insert the floppy in the drive, then select **Click here when done**.

   As the last installation step, the installation tool modifies the file LOOPS, writes it out to LOOPSDIRECTORY, and compiles it.

   When installation is finished, the first menu reappears:

```
Loops System
Install from floppies
Load into sysout
```

6. Select the menu option **Load into sysout** to load LOOPS into your system. The following menu appears:

```
Load Which?
Loops
Loops Masterscope
Gauges
LoopsBackwards
VirtualCopy
```

7. Select **LOOPS** from the menu to load LOOPS from the location where you installed it.

   Once LOOPS is loaded the LOOPS System menu reappears. To load one of the other LOOPS library or User modules, select the appropriate name in the Load Which? menu.

8. Position your mouse cursor anywhere on the screen except for the Load Which? menu, then press the left mouse button to exit the installation procedure.

To load LOOPS in the future, after the installation procedure is finished, load the LOOPS.DFASL file which the installation procedure created:

```
(CNDIR LOOPS DIRECTORY)
```

```
(LOAD 'LOOPS.DFASL)
```

This will load the rest of LOOPS.

When you perform a `CNDIR`, `LOOPS DIRECTORY` in the example above is the directory you defined in Step 4; e.g.,

```
(CNDIR '{DSK}<LISPFILES>LOOPS>)
```

LISPUSERSDIRECTORIES should point to a directory containing GRAPHER.LCOM, and DISPLAYFONTDIRECTORIES should point to a directory containing the Helvetica display font files from your Lyric or Medley XAIE distribution floppies.

CAUTION

LOOPS uses the new XAIE compiler and its macrolet facilities. When LOOPS is loaded, it sets your ***DEFAULT-CLEANUP-COMPILER*** to **'CL:COMPILE-FILE**. More information on this cleanup flag and the new compiler is available in the *Xerox Lisp Release Notes, Lyric Release*.

[This page intentionally left blank]

5. REPORTING PROCEDURE

If you have problems with this release, file an Action Request (AR) Bug Report.

[This page intentionally left blank]

6. KNOWN PROBLEMS

This chapter is a compilation of known problems in the Lyric/Medley Release of Xerox LOOPS. These problems are in the form of Action Requests (ARs) from the Xerox AR data base, followed by a brief description.

The ARs are divided into the following categories:

- Installation
- Instances
- Classes
- MetaClasses
- Active values
- Browsers
- Breaking and tracing
- File Manager
- Masterscope
- Extensions to ?=
- Windows
- System Variables and Functions
- Xerox LOOPS Library Packages
- Conversion to newer releases

6.1 Installation

AR	Description
10169	During the installation process, if fonts need to be loaded the LOOPS installer will try to write them to the directory on the front (CAR) of the list **DISPLAYFONTDIRECTORIES**. **Workaround:** Be sure that this is a directory you can write to, e.g. by pushing the name of a local directory onto the list like "{dsk}<lispfiles>fonts>." Grapher is loaded into the current sysout without being copied to a local directory; you may wish to do this and put that directory on your **DIRECTORIES** list.

6.2 Instances

AR	Description
9225	The implementation of multiple names for instances has NOT been changed in the Lyric/Medley release of LOOPS. However, some conditions which previously caused instances to appear twice on a file have been fixed.

6.3 Classes

AR Description

8881 The Supers **MACRO** is not documented. It retrieves the supers of the given class. It accepts a class object as an argument and returns a list of class objects.

9134 Caution: **ListAttribute** (page 3-27) finds non-local **IVPROPS** on classes.

9881 To use the method **CVMissing**, it is necessary to define a class called **MyClass** which is a specialization of the class **Class**. Then the **CVMissing** method is specialized at this level (in **MyClass**). Finally, a class **Foo** is defined whose metaclass is **MyClass**. At this point the expression

```
(GetClassValue (_ ($ Foo) New)  'NewCV)
```

can be evaluated and a new class variable will be created.

9884 To create a class / instance variable without a value the variable should be set to the value of **NotSetValue**.

6.4 MetaClasses

AR Description

10049 A new low level accessor function has been defined for programmers who wish to implement their own inheritance schemes.

(**FetchMethodLocally** *classobj selector*)

If *classobj* has a method for *selector* returns its name, otherwise NIL.

6.5 Active Values

AR Description

9158 If an **IndirectVariable** points at an **ActiveValue** in a remote object then the remote active value's **Get** or **PutWrappedValue** method is triggered with the **containingObj** argument holding the object containing the **IndirectVariable**.

6.6 Browsers

AR	Description

9244	The behavior of **LatticeBrowser** and **ClassBrowser** are different for **AddRoot**.
	
	Workaround: The example for **AddRoot** in section 10.5.3 should be changed to:
	
	The following creates an instance of **LatticeBrowser** and adds Tofu as a root:
	
	`54_ (_ ($ LatticeBrowser) New 'LB1)`
	`55_ (_ ($ LB1) AddRoot 'Tofu)`

9851	Section 10.3.2.7 "Extending Functionality with the Left Mouse Button:" should be modified as follows. On the 1186 the **COPY** key is similar to **SHIFT**, the **MOVE** and **CTRL** keys behave similarly, and **META** and **SAME** are similar as well.
9859	The specialization of **LatticeBrowser** methods into **ClassBrowser** methods is not enumerated. In section 10.5 "Programmer's Interface to Lattice Browsers," some methods are described which take different arguments when invoked on Class Browsers, e.g., **BoxNode** sent to a **ClassBrowser** does not allow the **unboxPrevious** flag.
10010	The documentation for **NewItem** in section 10.5.3 has the following additional information: Purpose: Gets an object, prompting the user if necessary. Behavior: ...to be added. **NewItem** accepts only names and maps them to objects using **$!**. If no name is entered at the prompt (by pressing return), **NewItem** returns **NIL**.
10367	When specializing a method from the class browser, if there are no methods to inherit, other than generic methods from **Object,** the menu lists methods already defined in the current class. Be careful about specializing methods defined directly under **Object**.

6.7 Breaking and Tracing

AR	Description
10356	**BT** will not show the send frames for broken methods. **Workaround:** You can do **BT!** and grab the method name from the ***APPLY*** frame. Inspect the **METHOD-FNS** definition of it.

6.8 File Manager

AR	Description
9159	Renaming a method does not smash the associated old symbol's function definition.
9166	Renaming a method does not remove the associated old method-fns definition from the file manager's change list.
10484	**CLASS** coms compiled by Lyric/Medley LOOPS are combined into a single large form that is read all at once. This causes occasional "Class x not defined, defining one now." messages to appear. Aside from the inconvenience of not seeing the names of classes during compilation, this should not cause problems.

6.9 MasterScope

AR	Description
9911	When LOOPS MasterScope analyzes forms like

```
(LET* ((A B)
       (C (F A)))
     --)
```

dwim asks about what it thinks is an unbound use of A. The analysis is correct, however.

6.10 Extensions to ?=

AR	Description
9828	**?=** documentation for the **ClassBrowser** methods is not available.

6.11 Windows

AR	Description
9243	**PutSavedValue** and **SavedValue** have **not** been removed from this release of Xerox LOOPS. We no longer plan to remove them.
10040	There are two new methods on the **Window** class:

```
(_ window Open?)
```

returns non-NIL if window is a LOOPS **Window** and is open.

```
(_ window Shade shade)
```

shades a LOOPS **Window** if it's open. The *shade* argument defaults to the value of **GRAYSHADE1**.

6.12 System Variables and Functions

AR	Description
9222	The variable **LispUserFilesForLoops** actually names Lisp Library modules.

6.13 Xerox LOOPS Library Modules

AR	Description
9797	After **VirtualCopyMixin** classes are "destroyed," instances of them can still be created without an error occurring.
9868	If a gauge is **Attach**'ed to one value, and then **Attach**'ed to another value the title of the gauge will not change even though the gauge will display the new value.
9871	Sending a **Shape** message to a **Meter** with the **ExtraSpaceFlag** set to T has no effect.

6.14 Conversion to Newer Releases

AR	Description
10404	There is currently no really convenient way of converting instance files from Buttress into Koto and Lyric/Medley. However, one can load **LOOPSBACKWARDS**, which modifies the **OLD-INTERLISP-FILE** readtable to handle the older style instances. Then one can read instances individually from the file and write them into a new file. In the new file they will be correctly formatted in the new manner.

[This page intentionally left blank]

INDEX

[This page intentionally left blank]

APPENDIX A. SUN INSTALLATION PROCEDURE

This appendix describes how to install the LOOPS System files, Library Modules files, and Users' Modules files on the Sun Workstation.

Overview of the Distribution Kit

The distribution kit for LOOPS on the Sun consists of a single ¼-inch tape cartridge. It contains the complete release in "tar" format and creates appropriate directories when its contents are extracted.

Preparation

Preparing to install LOOPS requires that the Medley release of Lisp is already installed and that adequate file space is available.

Before installing LOOPS, remember that

- the Medley 1.0-S release of Lisp must already be installed on your Sun Workstation;
- the complete LOOPS distribution requires about 1.2 MBytes of file space.

Installation

The software installation procedure shows the steps required for installing the Lyric/Medley LOOPS software on a Sun Workstation with Medley 1.0-S already installed. Examples are given where appropriate. Only those users who are system administrators and have **root** privileges can install the LOOPS, Lyric/Medley release.

Before starting software installation, remember that the LOOPS software requires about 1.2 MBytes of file space.

1. Log in under your username.

   `login` **`yourname`**

   `prompt%`

   where **`yourname`** is replaced by your username.

2. Check for available space with the **df** command:

```
prompt% df
```

Filesystem	kbytes	used	avail	capacity	Mounted on
/dev/xy0a	7437	5470	1223	82%	/
/dev/xy0h	148455	4900	128709	96%	/usr/misc

3. Determine if you need to run **su** to make a directory for the distribution. If so, type in **su**:

```
prompt% su
```

4. Make a directory for the distribution. This directory should be named /usr/local/lde/loops. If you have enough space on the file system containing /usr/local/lde, then

```
prompt# mkdir /usr/local/lde/loops/
```

   If you don't have enough space on /usr/local/lde, go to step 6.

5. Make yourself owner of this directory:

```
prompt# /etc/chown yourname /usr/local/lde/loops/
```

   where **yourname** is your username.

6. If you don't have space on the file system which contains /usr/local/lde, but do have space somewhere else, for instance on /usr1, then make the directory there and link /usr/local/lde/loops to it:

```
prompt# mkdir /usr1/loops
prompt# /etc/chown yourname /usr/usr1/loops
prompt# ln -s /usr1/loops /usr/local/lde/loops
```

7. If you ran **su**, leave the privileged shell by typing:

```
prompt% exit
```

8. Insert the ¼-inch cartridge tape, containing the LOOPS software, in the drive.

9. Connect to /usr/local/lde/loops:

```
prompt# cd /usr/local/lde/loops
```

10. Load the Medley software from tape. Indicate the appropriate device abbreviation for your tape by replacing **xx** in the example below with

    ar for the Archive drive,

    st for a SCSI tape drive.

    This example shows the command entry sequence:

```
prompt# tar xvpf /dev/rxx0
```

    As the software is loaded (a process that takes some time) the system prints a series of lines in the following form:

```
x ./system/LOOPS., 28552 bytes, 56 tape blocks
```

The x at the beginning of the line indicates that the file is being extracted from the tape.

This creates directories named:

/usr/local/lde/loops/system/

/usr/local/lde/loops/library/

/usr/local/lde/loops/users/

This is a good time to set the protection of the extracted directories and files so that the work group using LOOPS has at least read access to them.

11. Boot Medley Lisp.

12. Open an Interlisp Executive Window.

13. Make certain the time is set correctly.

14. Set the **DIRECTORIES** and **DISPLAYFONTDIRECTORIES** variables appropriately so the sysout can find your Lyric/Medley library and font files.

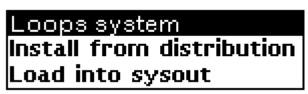
15. Make the LOOPS System directory your connected directory:

    ```
    CONN {DSK}/usr/local/lde/loops/system/
    ```

16. Enter the following into your Exec:

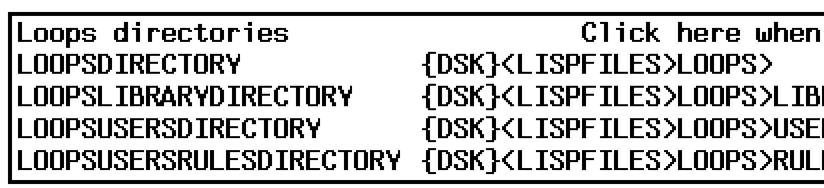
    ```
    LOAD(LOOPS)
    ```

    A menu appears that looks like this:



17. Select the menu option **Install from distribution**.

    The following menu appears:



This menu shows the current (or default, if unset) values of the variables LOOPS examines when it loads things.

If you have installed LOOPS under /usr/local/lde/loops/ click the mouse on the menu items to set these directories to point where the tape was unloaded:

LOOPSDIRECTORY {dsk}/usr/local/lde/loops/system/
LOOPSLIBRARYDIRECTORY {dsk}/usr/local/lde/loops/library/
LOOPSUSERSDIRECTORY {dsk}/usr/local/lde/loops/users/
LOOPSUSERSRULESDIRECTORY {dsk}/usr/local/lde/loops/users/

As the last installation step, the installation tool automatically modifies the file LOOPSSITE, writes it out to **LOOPSDIRECTORY**, and compiles it.

When this setp is finished, the first menu reappears:

Loops system
Install from distribution
Load into sysout

18. Select the menu option **Load into sysout** to load LOOPS into your system. The following menu appears:

Load Which?
Loops
Loops Masterscope
Gauges
LoopsBackwards
VirtualCopy

19. Select **LOOPS** from the menu to load LOOPS from the location where you installed it.

    Once LOOPS is loaded the LOOPS System menu reappears. To load one of the other LOOPS library or Users' modules, select the appropriate name in the Load Which? menu.

20. Position your mouse cursor anywhere on the screen except for the Load Which? menu, then press the left mouse button to exit the installation procedure.

Lyric/Medley LOOPS is now installed on your Sun Workstation.

Loading After Installation

This section describes how to reload LOOPS into a newly started Lisp sysout after LOOPS has been previously installed.

1. Start up Medley on your Sun Workstation.
2. Open an INTERLISP Exec window.
3. Make sure **DIRECTORIES** points to a directory containing GRAPHER.LCOM, and **DISPLAYFONTDIRECTORIES** points to a directory containing the Helvetica display font files from your Lisp distribution kit.
4. Connect to the directory containing the LOOPS system files:

```
(CNDIR '{DSK}/USR/LOCAL/LDE/LOOPS/SYSTEM/)
```

5. Load LOOPS loader program:

```
(FILESLOAD LOADLOOPS)
```

6. Run the LOOPS loader program:

```
(LOADLOOPS)
```

This procedure loads only the LOOPS system files. Please see the manuals describing the LOOPS Library and Users' Modules for their loading procedures.

CAUTION

LOOPS uses the new compiler and its macrolet facilities. When LOOPS is loaded, it sets your ***DEFAULT-CLEANUP-COMPILER*** to **'CL:COMPILE-**

FILE. More information on this cleanup flag and the new compiler is available in the *Lisp Release Notes*, in your Lyric or Medley Lisp kit.

[This page intentionally left blank]

DOCUMENT UPDATE SHEET

Document Name: ***LOOPS Manual***

Document Number: ***310000***

DOC. VERSION	RELEASE DATE	REPLACE PAGES	INSERT PAGES	INSTRUCTIONS/ NOTES
Lyric/Medley	Oct., 1988	NA	NA	Please read the Errata Sheet, accompanying this release material, for last minute release notes.
Lyric/Medley	Oct., 1988	NA	NA	The Lyric/Medley LOOPS documentation contains numerous references to Xerox LOOPS. Xerox LOOPS is now known as Envos LOOPS.
Lyric/Medley	Oct., 1988	NA	A-1-A-4	Add *Appendix A, Sun Installation Procedure,* to your *LOOPS Release Notes.*

Venue

LOOPS

Reference Manual

Library Modules Manual

Users' Modules Manual

November, 1991

Address comments to:
Venue
User Documentation
1549 Industrial Road
San Carlos, CA 94070
415-508-9672

LOOPS

REFERENCE MANUAL
LIBRARY MODULES MANUAL
USERS' MODULES MANUAL

November, 1991

Text was written and produced with Venue text formatting tools; Xerox printers were used to produce text masters. The typeface is Classic.

Venue

LOOPS Users' Modules Manual

November 1991

Address comments to:
Venue
User Documentation
1549 Industrial Road
San Carlos, CA 94070
415-508-9672

LOOPS USERS' MODULES MANUAL

November, 1991

Text was written and produced with Venue text formatting tools; Xerox printers were used to produce text masters. The typeface is Classic.

1. 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 specified 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.

1.1 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 satisfied. Several rule languages 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 specified 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-effect 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 rule language provides a concise syntax for the most common operations.

(9) There is a fast and efficient compiler for translating RuleSets into Interlisp functions.

(10) LOOPS provides facilities for debugging rule-oriented programs.

The following sections are organized as follows: Section 1.2, "Basic Concepts," outlines the basic concepts of rule-oriented programming in LOOPS. It contains many examples that illustrate techniques of rule-oriented programming. Section 1.3, "Organizing a Rule-Oriented Program," describes the rule syntax, and the remaining sections in this chapter discuss the facilities for creating, editing, and debugging RuleSets in LOOPS.

1.2 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 effort 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.

1.3 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 differently, 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.

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 finished. *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. For example, Figure 1 shows the 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 of some rule is applied. Some rules, called one-shot rules, are executed at most once. These rules are indicated by preceding them with a one in braces ({1}).

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

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

  IF .Operational THEN (STOP T);

  IF load>1.0 THEN .ReduceLoad;

  If ~pluggedInTo THEN .PlugIn;

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

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

{1}  THEN dealer.RequestService;

{1}  THEN manufacturer.Complain;

{1}  THEN $ConsumerBoard.Complain;

{1}  THEN (STOP T);
```

Figure 1. Basic RuleSet

1.4 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 satisfied for it to be executed. Execution is also controlled by data in that different values in the data allow different rules to be satisfied. Criteria for iteration and rule selection are specified by a RuleSet control structure. There are two primitive control structures for RuleSets in LOOPS which operate as follows:

Do1 [RuleSet Control Structure]

The first rule in the RuleSet whose conditions are satisfied 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 specific and general situations, the specific 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 satisfied. 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 satisfied. In a single invocation of the RuleSet, all of the applicable rules are invoked.

Figure 2 illustrates the use of a **Do1** control structure to select one of 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 .FillSoak1 .Agitate .Soak4 .Agitate
    .Soak1 .SpinAndDrain .SprayAndRinse
    .SpinAndDrain .Fill .DeepRinse .Pause .DampDry;
```

Figure 2. RuleSet showing Do1

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 satisfied, the first rule is executed whose conditions are satisfied. This is repeated as long as the while condition is satisfied or until a **Stop** statement or transfer call is executed (see Section 2.14, "Stop Statements"). 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 satisfied, every rule is executed whose conditions are satisfied. This is repeated as long as the while condition is satisfied 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 specified 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. Figure 3 illustrates a simple use of the **WhileAll** control structure to specify a sensing/acting feedback loop for controlling the filling of a washing machine tub with water.

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

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

{1!} IF loadSetting = 'Small THEN waterLimit_10;
{1!} IF loadSetting = 'Meduim 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 termperatureSetting = 'Hot
  THEN HotWaterValve.Open ColdWaterValve.Close;

  IF termperatureSetting = 'Warm
  THEN HotWaterValve.Open ColdWaterValve.Open;

  IF termperatureSetting = 'Cold
  THEN HotWaterValve.Close ColdWaterValve.Open;

(* Stop when the water reaches its limit.)

  IF waterLevelSensor.Test >= waterLimit
  THEN HotWaterValve.Close ColdWaterValve.Close
    (Stop T);
```

Figure 3. RuleSet with WhileAll

There are two control structures in LOOPS that specify iteration over a set of elements 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.

FOR1 [RuleSet Control Structure]

This is a cyclic version of **Do1**. If the iteration-condition (or while-condition) is satisfied, the first rule is executed whose conditions are satisfied or until a **Stop** statement is executed. This is repeated as long as the iteration condition is satisfied. The value of the RuleSet is the value of the last rule that was executed, or **NIL** if no rule was executed.

FORALL [RuleSet Control Structure]

This is a cyclic version of **DoAll**. If the iteration-condition is satisfied, every rule is executed whose conditions are satisfied. This is repeated as long as the iteration condition is satisfied 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 "iteration-condition" is specified in terms of the variables and constants accessible from the RuleSet. The simplest condition is

(FOR <iterVar> IN <setExpr> DO ruleSet) ;

The **setExpr** will be parsed with the RuleSet parser. The symbol **ruleSet** is a reserved word, and must be spelled as shown (no changes in capitalization).

Here is an example of iteration:

Control Structure: FORALL;
Iteration Condition: (FOR buyer IN (RoadStops ($ Consumer)) DO ruleSet) ;

For each buyer in the list produced by RoadStops, the ruleSet will be run. In a **FOR1**, the iteration will go on to the next buyer as soon as one rule executes. In a **FORALL**, all rules in the RuleSet will be tried.

For nested iteration one can use a slightly more complicated form, as illustrated by the following example:

Iteration Condition: (FOR buyer IN (RoadStops ($ Consumer)) DO
(FOR seller in (RoadStops ($ Producer)) DO ruleSet)) ;

An experienced Lisp user can see that this resembles the CLISP iteration construct. In fact, except that you can (must) use the RuleSet syntax in the construct, it is the CLISP construct, and any such construct can be used. A DO1 or DOALL ruleSet will be substituted for the occurrence of the atom ruleSet, depending on whether the Control Structure is a FOR1 or FORALL.

As an abbreviation, if the construct does not contain the atom ruleSet, then (DO ruleSet) is appended to the Iteration Condition for a **FOR1** or **FORALL**. Thus one could write the first example as:

Iteration Condition: (FOR buyer IN (RoadStops ($ Consumer)))

1.5 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 in which some of the rules should be executed only once. This was illustrated in the Washing Machine example in Figure 1 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 3.

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 *condition condition* **THEN triedRule3_T** *action* **;**

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** *condition condition* **THEN** *action* **;**

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 *condition condition* **THEN** *action* **;**

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

{1!} **IF** *condition* *condition* **THEN** *action* **;**

These rules are called "try-once" rules.

The two kinds of one-shot rules are our first examples of the use of meta-descriptions preceding the rule body in braces. See Section 1.7, "Saving an Audit Trail of Rule Invocation," for information on using meta-descriptions for describing the creation of audit trails.

1.6 First/Last Rules

It is sometimes useful to have rules which fire before or after the ordinary part of the RuleSet is invoked, independent of the form of the control structure. For example, in a DO1, such "FIRST " rules could be used for initialization. These now exist, and are notated by putting a {F} for a first rule in the MetaDescription field, and a {L} for a last rule. If a RuleSet has L rules which execute, the value of the RuleSet is the value of the last rule which executed.

1.7 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.

1.7.1 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 efficiently 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-defined calculus of belief revision, at whatever level of abstraction is appropriate for an application.

1.7.2 Overview of Audit Trail Implementation

When *audit mode* is specified 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-effect of the assignment of values to instance variables. Audit records are LOOPS objects, whose class is specified 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 specific rules. Rule-specific audit information is specified 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-specific information overrides RuleSet information.

During rule execution in audit mode, the audit information is evaluated after the rule's LHS has been satisfied 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 specified in the audit record classes in order to define a uniform set of side-effects for rules of the same category. In the following example, such an active value is used to carry out a "certainty factor" calculation.

1.7.3 An Example of Using Audit Trails

The following example illustrates one way to use the audit trail facilities. Figure 4 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 difficulty of incomplete information about the product. For example in this RuleSet, the reliability of the washing machine is estimated to be 0.5 in the absence of specific information from *Consumer Reports*. The meta-descriptions for the rules, which appear in braces, categorize them in terms of the *basis of belief* (the category *basis* is either a fact or estimate) and a *certainty factor (cf)* that is supposed to measure the "implication power" of the rule. 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-defined audit trails and experimental methods of belief revision. (Realistic belief revision systems are usually more sophisticated than this example.)

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. Figure 5 illustrates the evaluation report for one machine and one of its audit records. In this example, each of the entries in the report has a reason and a cumulative certainty (cc) property in addition to the value. The value of the reason properties are audit records created as a side effect 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 effects. For example, the active in the **cf** variable or the audit record performs a computation to maintain a calculated cumulative certainty in the reliability variable of the evaluation report.

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.)

```
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.)

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

  {(basics_Fact cf_.8)}
  reliability_(_($ ConsumerReports) GetFacts machine);

  {(basics_Estimate cf_.4)}
  IF 'reliability THEN reliability_.5;
  .
  .
  .
```

Figure 4. RuleSet Showing Evaluation

```
EvaluationReport "uid1"
expense:  510
```

```
suitability:  Poor cc 1 reason ...
reliability:  .5 cc .6 reason "uid2"
.
.
.
AuditRec "uid2"
rule:  "uid3"
basis:  Estimate;
cf:  #(.4 NIL PutCumulativeCertainity)
```

Figure 5. Example of an Audit Trail

1.8 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 find ways to factor control information out of the rules.

1.8.1 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 Figure 1:

{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 specified in braces in front of the LHS of a rule. For another example, the following rule from Figure 4:

{(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 specified 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 first option, intelligibility is degraded. In the second option, the transparency of the system is degraded because the knowledge is hidden.

1.8.2 The Rationale for RuleSet Hierarchy

Some advocates of production systems have praised the flatness of traditional production systems, and have resisted the imposition of any organization to the rules. The flat 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 flat 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 flat 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 flow 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 different parts of a program. Although context clauses admit the creation of "subroutine contexts", they require you 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.

1.8.3 The Rationale for RuleSet Control Structures

Production languages are sometimes described as having a *recognize-act cycle*, which specifies how rules are selected for execution. An important part of this cycle is the *conflict resolution strategy*, which specifies how to choose a production rule when several rules have conditions that are satisfied. For example, the **OPS5** production language has a conflict 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 specific 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 conflict 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 different control relationships among the rules. For example, the **DoAll** control structure is useful for rules whose effects 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 Figure 1.

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 *a b c* **THEN** *d e* **;**
IF *a b c* **THEN** *d e* **;**
IF *a b c* **THEN** *d e* **;**

could be written alternatively as

Control Structure: DoAll;
Task Vars: firedSomeRule;

IF *a b c* **THEN firedSomeRule_T** *d e* **;**
IF ~firedSomeRule *a b c* **THEN firedSomeRule_T** *d e* **;**
IF ~firedSomeRule *a b c* **THEN firedSomeRule_T** *d e* **;**

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 reflect the assumption that earlier rules in the RuleSet were not satisfied.

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 *a b c* **THEN** *d e* **;**
IF ~*a b c* **THEN** *d e* **;**
IF ~*a* ~*b c* **THEN** *d e* **;**

can be written as

Control Structure: Do1;

IF *a b c* **THEN** *d e* **;**
IF *b c* **THEN** *d e* **;**
IF *c* **THEN** *d e* **;**

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**.

1.8.4 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 benefits. It facilitates the construction of programs which don't entirely fit 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. Figure 6 illustrates the installation of the RuleSet **SimulateWashingMachineRules** to carry out the **Simulate** method for instances of the class **WashingMachine**. This definition of the class **WashingMachine** specifies 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 message to invoke the RuleSet may be sent by any LOOPS program, including other RuleSets.

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 (* "rtk: 12-Jun-87 07:57")
    doc (* Home appliance for wachine cloothes.))
  (Supers ElectricalDevice PlumbedDevice CleaningDevice)
  (ClassVariables)
  (InstanceVariables
    (controlSetting Meduim
      doc (* One of Small, Medium, Large, ExtraLarge))...)
  (Methods
    (Fill WashingMachine.Fill doc (* Fill the tub with water.))
    (Wash WashingMachine.Wash doc (* Perofrm the wash cycle.))
    (Simulate UseRuleSet RuleSet SimulateWashingMachineRules)
    .
    .
    .
]
```

Figure 6. RuleSet Invoked as a Method

Using the data-oriented paradigm, RuleSets can be installed in active values so that they are triggered by side-effect 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 figure. 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.

TABLE OF CONTENTS

[This page intentionally left blank]

2. THE RULE LANGUAGE

This chapter describes the syntax and semantics of the rule language.

2.1 Language Introduction

A rule in LOOPS describes actions to be taken when specified conditions are satisfied. 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-specific control information, rule descriptions, audit instructions, and debugging instructions. For example, the syntax for one-shot rules shown in Section 1.5, "One-Shot Rules:"

{1} IF *condition* *condition* **THEN** *action* **;**

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 in Section 1.7, "Saving an Audit Trail of Rule Invocation."

LHS Syntax: The clauses on the LHS of a rule are evaluated in order from left to right to determine whether the LHS is satisfied. If they are all satisfied, then the rule is satisfied. 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 satisfied 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 satisfied. 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 first character as follows:

(* This is a comment)

2.2 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 defined 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 first 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 specific 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.

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-effect 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.

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

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 first 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.

2.3 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 satisfied 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 satisfied 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**.

2.4 Infix Operators and Brackets

To enhance the readability of rules, a few infix operators are provided. The following are infix binary operators in the rule syntax:

+ [Rule Infix Operator]

Addition.

++ [Rule Infix Operator]

Addition modulo 4.

- [Rule Infix Operator]

Subtraction.

-- [Rule Infix Operator]

Subtraction modulo 4.

***** [Rule Infix Operator]

Multiplication.

/ [Rule Infix Operator]

Division.

> [Rule Infix Operator]

Greater than.

< [Rule Infix Operator]

Less than.

>= [Rule Infix Operator]

Greater than or equal.

<= [Rule Infix Operator]

Less than or equal.

= [Rule Infix Operator]

EQ -- simple form of equals. Works for atoms, objects, and small integers.

~= [Rule Infix Operator]

NEQ. (Not **EQ**.)

== [Rule Infix Operator]

EQUAL -- long form of equals.

<< [Rule Infix 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 first and second minus signs are both treated as binary subtraction statements. That is, the first 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.

2.5 Interlisp Functions and Message Sending

Calls to Interlisp functions are parenthesized with the function name as the first 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 specified 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;

2.6 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 first 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 first 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.

2.7 Computing Selectors and Variable Names

The short notations for instance variables, properties, and unary messages all show the selector and variable names *as they actually appear* in the object.

*object***.***selector*
*object***:***ivName*
*object***::***cvName*
*object***:***varname***:,***propName*

(_ *object selector arg   arg* **)**

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 selector arg   arg* **)**

2.8 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 finally (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.

2.9 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, first the object that is the value of **y** is computed, then the value of its instance variable **b** is set to **0**.

Properties: Assignment statements can also be used to set property values as in:

box:x:,origin_47   fact:,reason_currentSupport;

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.

2.10 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 specifications. Each meta-assignment statement specifies information to be saved in audit records when a rule is applied. In the following example from Figure 4, 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 satisfied. In addition, if the RuleSet was compiled in *audit* mode, then during RuleSet execution an audit record is created as a side-effect 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 satisfied, 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.

2.11 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 infix operator **<<** for membership testing.

2.12 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.

2.13 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 first 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.

2.14 Stop Statements

To provide premature terminations in the execution of a RuleSet, the Stop statement is provided.

(Stop *value***)** [RuleSet Statement]

value is the value to be returned by the RuleSet.

[This page intentionally left blank]

PREFACE

Overview of the Manual

This manual describes the Users' Modules for Xerox's Lisp Object-Oriented Programming System, LOOPS (TM), to developers.

Note: Venue does not support LOOPS Users' Modules. However, each Users' Module contains the name and network mailing address of the person who wrote or last modified that module, and the date it was written or last modified.

This manual describes the Lyric/Medley Release of the LOOPS Users' Modules, which run under the Lyric and Medley Releases of Lisp.

Organization of the Manual and How to Use It

This manual is divided into chapters, with each chapter describing a separate Users' Module.

To use the manual, read the chapter that corresponds to the Users' Module you want to use. A general Table of Contents is provided to help you locate specific information.

Conventions

This manual uses the following conventions:

- Case is significant in LOOPS and Lisp. All selectors, methods, arguments, etc., must be typed as shown. Typically, this means that method names are capitalized and variables are not.
- Arguments appear in italic type.
- Selectors, methods, functions, objects, classes, and instances appear in bold type.

  For example, a method appears as follows:

  (← *self* **Selector** *Arg1 Arg2*)

- Examples appear in the following typeface:

  `89←(←LOGIN)`

- All examples are typed into an Interlisp Exec. This is the recommended Exec for all LOOPS expressions.
- Methods with an exclamation mark (!) suffix usually perform operations deeply into class structure instead of only on a given object.

- Methods with a question mark (?) suffix usually are predicates; that is, truth functions.
- Methods oftn appear in the form **ClassName.SelectorName**.
- Cautions describe possible dangers to hardware or software.
- Notes describe related text.

References

The following books and manuals augment this manual.

LOOPS Reference Manual

LOOPS Release Notes

LOOPS Library Modules Manual

Interlisp-D Reference Manual

Common Lisp: the Language by Guy Steele

Common Lisp Implementation Notes, Lyric Release

Lisp Release Notes, Lyric Release and *Medley Release*

Lisp Library Modules Manual, Lyric Release and *Medley Release*

3. 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.

3.1 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***)**

3.2 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 **EditMethod** option from a browser containing the class. Alternatively, the **EditMethod** message can be used:

(_ *ClassName* **EditMethod** *selector***)** [Message]

Both approaches to editing retrieve the source of the RuleSet and put the user into the TTYIN or TEdit editor, treating the rule source as text.

Initially, the source is a template for RuleSets as shown in Figure 7. The rules are entered after the comment at the bottom. The declarations at the beginning are filled in as needed and superfluous declarations can be discarded.

```
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 7. Initial Template for a RuleSet

You 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 you choose 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 you leave 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 you are given another opportunity to edit the RuleSet.

3.3 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.

3.4 Saving RuleSets on Lisp Files

RuleSets can be saved on Lisp files 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 file, it is necessary to add two statements to the file commands for the file 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.

If RuleSets are methods associated with a class, and they are saved by using (FILES?), then the file package saves the appropriate entries. The user does not have to be concerned with editing the filecoms of the file being made.

3.5 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 files will be printed on the **DEFAULTPRINTINGHOST** as is standard in Interlisp-D. **ListRuleSets** can be given four 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 file will be printed.

3.6 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 arg2 ... argN***)** [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-effect as a result of accessing the variable on which they are installed.

3.7 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.

DefRSM can be invoked with the browser as follows:

- Position the cursor over a class in a browser.
- Press the middle mouse button. A menu pops up.
- Select the Add option in this menu, and drag the mouse to the right to display the submenu that includes the "DefRSM" option. You are prompted to enter a selector name.

After a RuleSet has been installed as a method by using **DefRSM**, you can then edit that RuleSet by selecting the "EditMethod" option from the browser editing menu.

3.8 Installing RuleSets in Active Values

Note: The following section and any other references to active values within the rule documentation refer to active values as they were implemented in the Buttress release. The functionality of triggering rules from active values has not been tested using the current implementation of active values. It should work to use the **ExplicitFnActiveValue** class to implement this behavior.

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.

3.9 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. The following 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.

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 Section 3.10, "The Rule Exec") 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.

3.10 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, you 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**.

3.11 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 a meta-descriptor.

Meta Assignments: cf_.5 support_'GroundWff;

3.12 Loading Rules

Set the variable LOOPSUSERSDIRECTORIES to include the directory where the Rules files are stored.

Load the file LOOPSRULES-ROOT.LCOM, which will load the following files from LOOPSUSERSDIRECTORIES:

- LOOPSBACKWARDS.LCOM
- LOOPSMIXIN
- LOOPSRULES.LCOM
- LOOPSRULESP.LCOM
- LOOPSRULESC.LCOM
- LOOPSRULESD.LCOM, which will load the file TTY.LCOM from LISPUSERSDIRECTORIES.

Editing rules will be easier if TEdit is loaded. Loading the Rules does not automatically load TEdit.

3.13 Known Problems

In a rule, the expression $pipe.ri..$p compiles to (RunRS (QUOTE ($ pipe)) ($ p)), which fails.

Meta-assignment statements cannot handle expressions. This means that statements like {cf _ .5} work fine, but {validity _ 'fact} fails.

A value of 1 in a meta-descriptor statement is always taken to be a one-shot designator. You cannot have a meta-descriptor statement like {cf_1}. However, the number 1.0 can be used; the meta-descriptor statement, {cf_1.0}, works.

Rules have not been tested without loading TEdit in order to edit RuleSets.

[This page intentionally left blank.]

CONVERT-LOOPS-FILES

By: Bob Bane (Bane.pa@Xerox.com)

5-Feb-88

INTRODUCTION

CONVERT-LOOPS-FILES allows you to convert files from the Koto release of LOOPS to the Medley release. The changes from the Koto release are described in detail in the *LOOPS Release Notes*.

PROCEDURE

To convert Koto LOOPS Files to Lyric/Medley Loops:

1. Your files must be from Koto LOOPS. Pre-Koto LOOPS files must first be run through the Buttress->Koto converter in a Koto LOOPS sysout.

2. Install and load your Lyric LOOPS sysout. You need to have TEdit loaded, as the converter uses it.

3. Load the file CONVERSION-AIDS.DFASL, which defines the function

   ```
   (CONVERT-LOOPS-FILES <list-of-files> <dump-files-p>)
   ```

   where:

   `<list-of-files>` A filename or a list of filenames to be converted.

   `<dump-files-p>` Determines disposition of files.

   - If non-NIL, converted files will be dumped back out immediately.
   - If :COMPILE, they will be compiled.
   - If :COMPILE/LOAD, they will be compiled and the compiled code loaded.

4. Call CONVERT-LOOPS-FILES with the names of the files you want converted and an appropriate option.

Note the following:

- CONVERT-LOOPS-FILES makes more than one pass over the files being converted; the first pass is a TEdit textual change to make the files loadable into Medley LOOPS. If you specify version numbers in your <list-of-files>, these changes will be made in place on your original files.
- The converter doesn't work completely automatically on systems consisting of several files that automatically load themselves with FILES coms; the converter may try to load subfiles before they are converted. It may be necessary to RETURN NIL from the LOAD calls in the break windows that will occur when this problem comes up. Aside from that, the converter works reasonably well, and has been used to convert large LOOPS systems with almost no source code changes from Koto Loops.

[This page intentionally left blank.]

[This page intentionally left blank]

LOOPSBACKWARDS

By: Bob Bane (Bane.pa@Xerox.com)

15-Dec-87

INTRODUCTION

LOOPSBACKWARDS allows you to run files which were previously converted from the Buttress release of LOOPS to the Koto release. Unlike the Koto version of LOOPSBACKWARDS conversion methods for moving files from Buttress LOOPS to Koto LOOPS are included but **not supported**. *We strong recommend that you convert LOOPS source code from Buttress to Koto using the Koto release of LOOPS.* Conversion of Koto LOOPS source code to Lyric LOOPS is done using the CONVERSION-AIDS users' module.

The changes between the Buttress and Koto releases are described in detail in the *LOOPS Release Notes* for the **Koto** release. Old features that are included in LOOPSBACKWARDS are summarized here:

- Many functions that were removed from the Buttress release are defined.
- The messages **List** and **List!** are available.
- The operation of old style active values, from the Buttress release, are provided.
- Support for reading old style macros, such as #(localState getFn putFn) or #$Mumble, is available.

LOOPSBACKWARDS is an unsupported LOOPS users module. It is strongly recommended that it only be used as part of an effort to upgrade very old LOOPS code to newer releases. It will allow very old LOOPS code to run well enough that it can be rewritten for a newer relese.

INSTALLATION

LOOPSVCOPY will be automatically loaded by LOOPSBACKWARDS.

FUNCTIONS

LOOPSBACKWARDS includes **ExplicitFnActiveValue** and **DefAVP**. **ExplicitFnActiveValue** allows the user code triggered by Get- or Put- accesses to be stored within functions which are pointed to by instance variables rather than requiring the redefinition of **GetWrappedValue** or **PutWrappedValue**. These functions must have the form specified in the **DefAVP** function.

ExplicitFnActiveValue [Class]

Purpose: Mimics the behavior of the Buttress style of active values.

Behavior: Get- accesses to the wrapped variable cause the **getFn** to be called, Put- accesses cause **putFn** to be called. Enables the old style activeValue to look like the new style without changing any functionality.

The **getFn** is called by the **ExplicitFnActiveValue GetWrappedValue** method. This method passes to the **getFn** the arguments defined by **DefAVP** as described in the *LOOPS Users' Modules*.

The **putFn** is called by the **ExplicitFnActiveValue PutWrappedValue** method. This method passes to the **putFn** the arguments defined by **DefAVP** as described in the *LOOPS Users' Modules*.

Instance Variables:

localState A place for data storage.

getFn The name of a function applied when the active variable is read.

putFn The name of a function applied when the active variable is changed.

Example:

```
32← (← ($ Bin) New 'bin4)
#,($& Bin (|DAW0.1Y:.H53.]99| . 521))

33← (← ($ ExplicitFnActiveValue) New 'EFAV1)
#,($& ExplicitFnActiveValue (|DAW0.1Y:.H53.]99| . 522))

34←(DEFINEQ (PrintOnGet
          (self varName localSt propName activeVal type)
          (PRINTOUT T "I am:" , activeVal T) localSt))
(PrintOnGet)

35←(←@ ($ EFAV1) getFn 'PrintOnGet)
PrintOnGet

36←(← ($ EFAV1) AddActiveValue ($ bin4) 'height)
#,($A #,NestedNotSetValue PrintOnGet NIL)

37←(←@ ($ bin4) height 123)
123

38←(@ ($ bin4) height)
```

```
I am: #,($ EFAV1)
123
```

(**DefAVP** *fnName putFlg*) [Function]

Purpose: Creates a template for defining an active value function.

Behavior: Creates a template and leaves you in the Interlisp function display editor.

Arguments: *fnName* Name of the function.

putFlg T indicates function is a **putFn**, NIL indicates a **getFn**.

Returns: The function name on exit from the editor .

Example: In each of the following cases the template only is shown. User code is to be added immediately after the comment by using the display editor.

```
66←(DefAVP 'AGetFn)
AGetFn

67←PP* AGetFn
(AGetFn
  [LAMBDA (self varName localSt propName activeVal type)
         (* This is a getFn. The value of this getFn is
returned as the value of the enclosing GetValue.)
    localSt])
(AGetFn)

68←(DefAVP 'APutFn T)
APutFn

69←PP* APutFn
(APutFn
  [LAMBDA (self varName newValue propName activeVal type)
          (* This is a putFn. ***NOTE*** The value of this
function will be returned as the value of any enclosing
PutValue. This usually means that you want to return the value
returned by PutLocalState.)
    (PutLocalState activeVal newValue self varName propName
type])
(APutFn)
```

[This page intentionally left blank.]

LOOPSMIXIN

By: Bob Bane (Bane.pa@Xerox.com)

15-Dec-87

INTRODUCTION

LOOPSMIXIN defines several small classes that can be mixed into your application classes. It also defines the class **Perspective** and its support classes, which are used in the implementation of LOOPS Rules. Perspectives allow you to view one object as having more than one class at a time; they have not been tested extensively outside of Rules and are known to have major bugs which don't affect Rules, so they are not documented here.

LOOPSMIXIN Classes

DatedObject - Defines the instance variables **created** and **creator** which are set at object creation time to the values of (DATE) and (USERNAME).

NamedObject - Defines the instance variable name and initializes it to an ActiveValue that insures that the LOOPS system name for the containing object is uniquely name; i.e. storing a name for the object in name causes the previous name for the object to be removed with **DeleteObjectName** and the new name for the object to be asserted with **NameEntity**.

GlobalNamedObject - A subclass of **NamedObject** that works the same way as **NamedObject**.

ListMetaClass - Specializes the **New** and **DestroyInstance** methods to keep a list of all instances of that class in the class property **AllInstances** for that class.

StrucMeta - A MetaClass useful for creating new classes. Specializes the **New** method to create a Class object by copying the instance variable and class variable descriptions of the current class. Class variables with a non-NIL Local property will not be copied.

TempClass - Specializes the **New** method to always create objects of this class using the **NewTemp** method, insuring that the objects will be temporary objects.

Perspective, Node, Template, TextItem - These classes are used in LOOPS Rules.

[This page intentionally left blank.]

[This page intentionally left blank.]

[This page intentionally left blank.]

RULES

Modified by: Rick Martin (Martin.pasa@Xerox.com)

14-Apr-86

[This page intentionally left blank.]

TABLE OF CONTENTS

LIST OF FIGURES

[This page intentionally left blank.]

TABLE OF CONTENTS

[This page intentionally left blank]

PREFACE

Overview of the Manual

This manual describes the Users' Modules for Xerox's Lisp Object-Oriented Programming System, Xerox LOOPS (TM), to developers.

Note: Xerox does not support Xerox LOOPS Users' Modules. However, each Users' Module contains the name and network mailing address of the person who wrote or last modified that module, and the date it was written or last modified.

This manual describes the Lyric/Medley Release of the Xerox LOOPS Users' Modules, which run under the Lyric and Medley Releases of Xerox Lisp.

Organization of the Manual and How to Use It

This manual is divided into chapters, with each chapter describing a separate Users' Module.

To use the manual, read the chapter that corresponds to the Users' Module you want to use. A general Table of Contents is provided to help you locate specific information.

Conventions

This manual uses the following conventions:

- Case is significant in Xerox LOOPS and Lisp. All selectors, methods, arguments, etc., must be typed as shown. Typically, this means that method names are capitalized and variables are not.

- Arguments appear in italic type.

- Selectors, methods, functions, objects, classes, and instances appear in bold type.

  For example, a method appears as follows:

  (_ *self* **Selector** *Arg1 Arg2*)

- Examples appear in the following typeface:

  ```
  89_(_LOGIN)
  ```

- All examples are typed into an Interlisp Exec. This is the recommended Exec for all Xerox LOOPS expressions.

- Methods with an exclamation mark (!) suffix usually perform operations deeply into class structure instead of only on a given object.

- Methods with a question mark (?) suffix usually are predicates; that is, truth functions.
- Methods often appear in the form **ClassName.SelectorName**.
- Cautions describe possible dangers to hardware or software.
- Notes describe related text.

References

The following books and manuals augment this manual.

Xerox LOOPS Reference Manual

Xerox LOOPS Release Notes

Xerox LOOPS Library Modules Manual

Interlisp-D Reference Manual

Common Lisp: the Language by Guy Steele

Xerox Common Lisp Implementation Notes, Lyric Release

Xerox Lisp Release Notes, Lyric Release and *Medley Release*

Xerox Lisp Library Modules Manual, Lyric Release and *Medley Release*

Writer's Notes -- Conventions

This file includes notes on conventions for *Xerox LOOPS Users' Modules Manual*, Lyric Beta Release. This manual is packaged in one binder.

Writer: Raven Kontur Brewster

Printing Date: 22 February 1988

Directories and Files

The directory {ERIS}<Doc>Loops>Lyric>Beta>UserMods> contains the files for the manual. This directory has the following subdirectories:

- {ERIS}<Doc>Loops>Lyric>Beta>UserMods>Z-ReleaseInfo> contains this file on writing conventions and a file on production details.

Filenames describe the contents of the file. For example, the filename

{ERIS}<Doc>Loops>Lyric>Beta>UserMods>LoopsMixin.tedit

contains the chapter on LoopsMixin.

Assemble the files in the following order for the manual:

{ERIS}<Doc>Loops>Lyric>Beta>UserMods>A1-TitlePage.tedit
{ERIS}<Doc>Loops>Lyric>Beta>UserMods>A2-TOC.tedit
{ERIS}<Doc>Loops>Lyric>Beta>UserMods>A3-Preface.tedit
{ERIS}<Doc>Loops>Lyric>Beta>UserMods>Converter.tedit
{ERIS}<Doc>Loops>Lyric>Beta>UserMods>LoopsBackwards.tedit
{ERIS}<Doc>Loops>Lyric>Beta>UserMods>LoopsMixin.tedit
{ERIS}<Doc>Loops>Lyric>Beta>UserMods>Rules-A1-TitlePage.tedit
{ERIS}<Doc>Loops>Lyric>Beta>UserMods>Rules-A2-TOC.tedit
{ERIS}<Doc>Loops>Lyric>Beta>UserMods>Rules-A3-LOF.tedit
{ERIS}<Doc>Loops>Lyric>Beta>UserMods>Rules1-Intro.tedit
{ERIS}<Doc>Loops>Lyric>Beta>UserMods>Rules2-Language.tedit
{ERIS}<Doc>Loops>Lyric>Beta>UserMods>Rules3-Use.tedit
{ERIS}<Doc>Loops>Lyric>Beta>UserMods>RulesA-Convert.tedit

Conventions

This manual uses the following conventions:

- Case is significant in Xerox LOOPS and Lisp. All selectors, methods, arguments, etc., must be typed as shown. Typically, this means that method names are capitalized and variables are not.
- Arguments appear in italic type.
- Selectors, methods, functions, objects, classes, and instances appear in bold type.

  For example, a method appears as follows:

  (_ *self* **Selector** *Arg1 Arg2*)
- Examples appear in the following typeface:

```
89_(_LOGIN)
```

- Methods with an exclamation mark (!) suffix usually perform operations deeply into class structure instead of only on a given object.
- Methods with a question mark (?) suffix usually are predicates; that is, truth functions.
- Methods often appear in the form **ClassName.SelectorName**.
- Cautions describe possible dangers to hardware or software.
- Notes describe related text.

Style Sheet Addenda

Here are some guidelines I used when writing the LOOPS manuals. Items appear in rather random order.

- Avoid contractions.
- Avoid subscripts. Use WORD1 rather than WORD   to avoid inconsistent line leading.
- Avoid wording that starts "Note that..." or "Notice that...". Either make it a note with correct format or eliminate the "Note that".
- Use semicolons rather than m-dashes.
- Each item in the template starts with an initial capital letter; e.g., "Describes..."
- The arguments are identical in the call and in the argument description.
- Parenthesies appear around expressions and square brackets appear around the name of the functionality.
- The arrow in the expression is the NS character ←, not _. These characters appear similarly when printed, but differently on the screen. See the section, "Special Notes and Cautions," for details.
- A period appears after the word None, after argument descriptions, and Returns: item.
- Items are set to or return T (instead of true).
- Menus contain options, not items or selections.
- You drag (not roll) the mouse to the right of a menu option to see its submenu.
- Use "above" and "below" when referrering to things in the same section, section numbers and names when referrering to things in the same chapter, and chapter numbers and names when referrering to things in another chapter.
- Please study the following stle sheet carefully before you start to edit. The various appearances of active value and annotated values are especially crazy making.

  These things appear in **bold**:
  class variables
  functions
  instance variables
  messages
  methods
  variables

ActiveValue - specific class/instance
active value - general information
activeValue - previous implementation of **ActiveValue**

annotatedValue - data type
AnnotatedValue - specific class
annotated values - general information

bitmap

data type

file package
filecoms

inspector

Lisp Library package
localState - instance variable

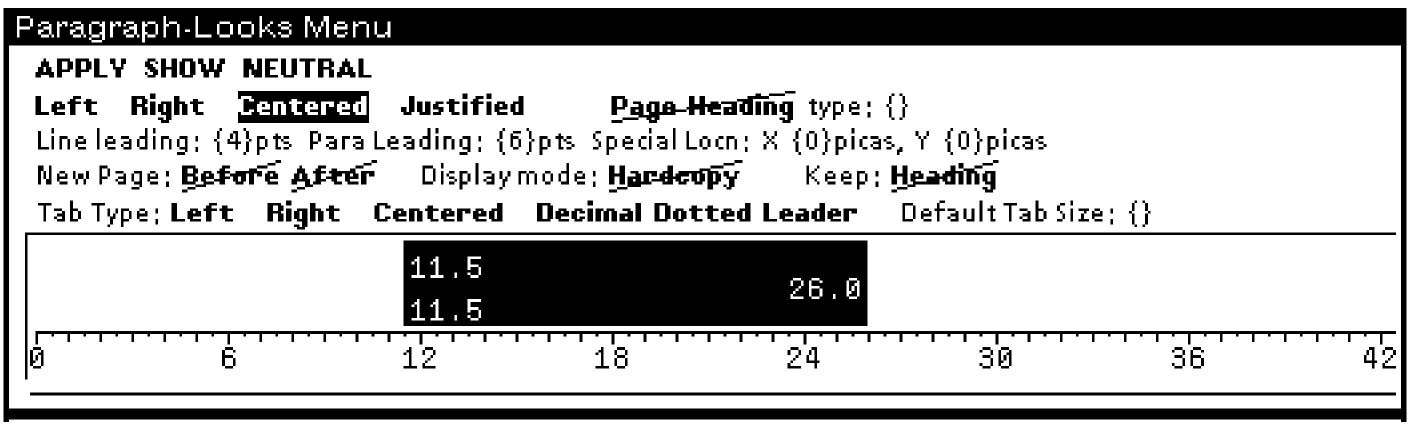
non-NIL

prettyprints

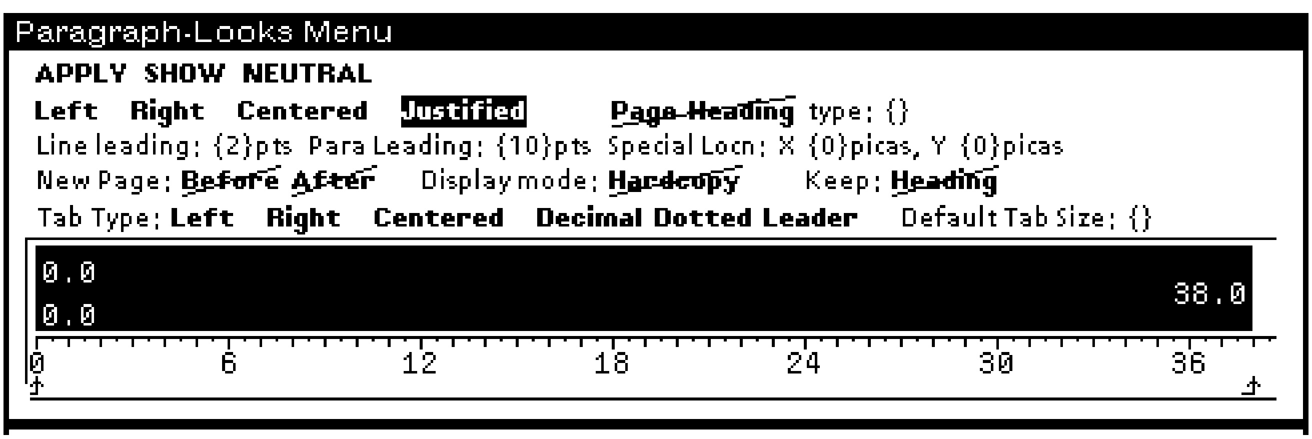
supers list

Paragraph Formatting

The heading has the following format:

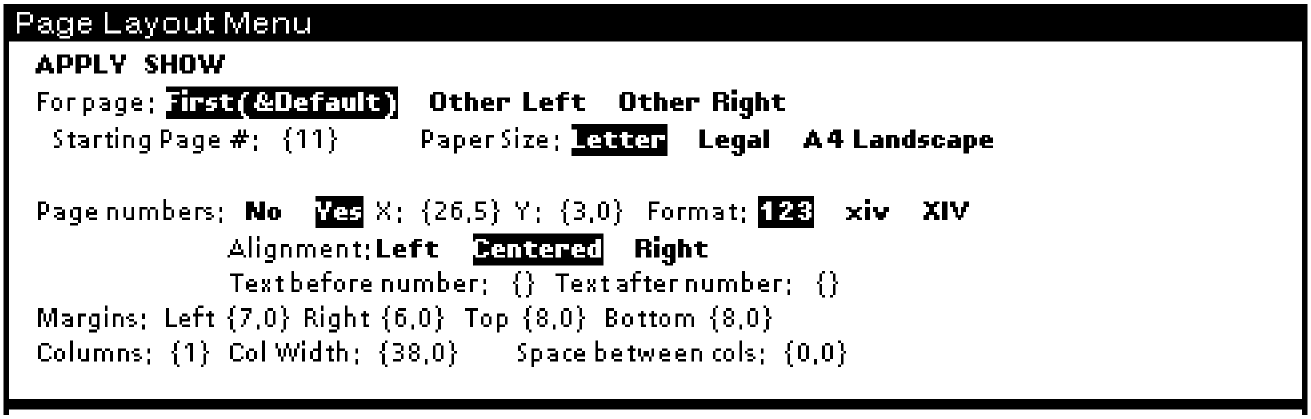


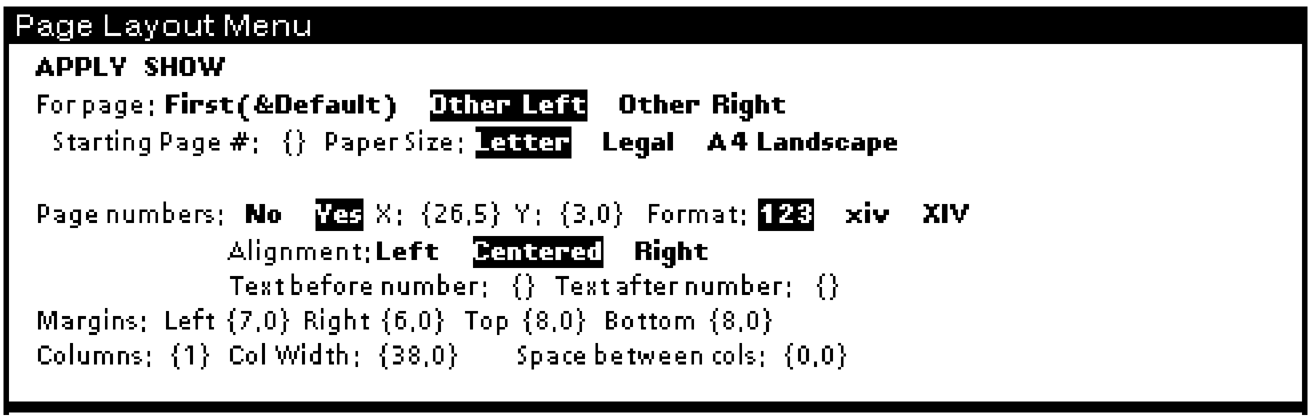
The text the following format:

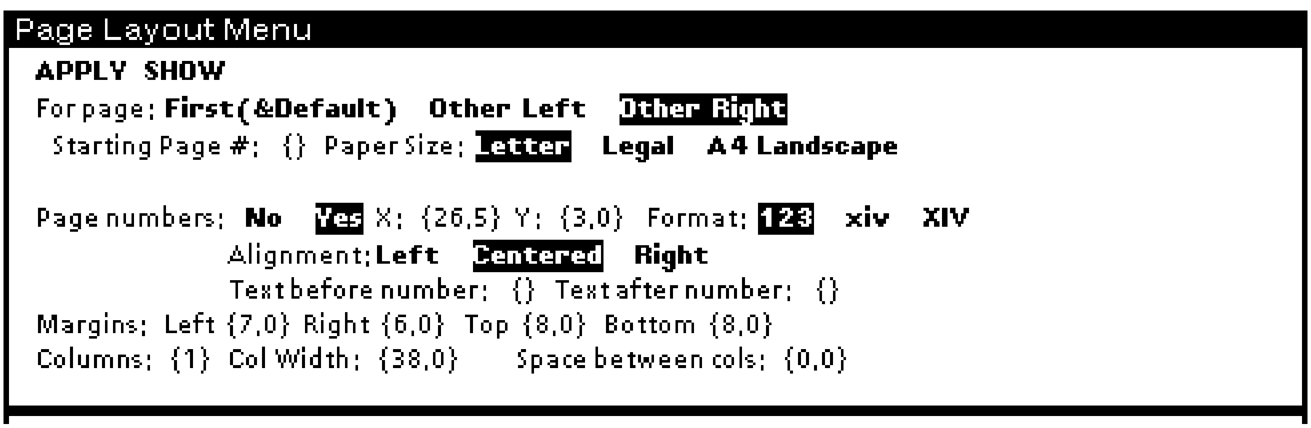


Page Layout

The starting page number varies with the package.







Bitmaps, Graphs, and Sketches

Scale for bitmaps is 0.75.

Special Notes and Cautions

Make sure you have changed the underscore to be a left arrow before loading and printing any files. To do this,

- Enter the following commands into your Executive:

```
(GETCHARBITMAP (CHARCODE _) '(MODERN 10 MRR))
(EDITBM IT)
```

- When the bitmap editor apears, delete the underscore and insert the following left arrow:

```
..........
..........
..........
..........
..........
....X.....
...XX.....
..XXXXXX..
...XX.....
....X.....
..........
..........
..........
..........
```

- Finally, enter the following commands into your Executive to store the pattern:

```
(PUTCHARBITMAP (CHARCODE _) '(MODERN 10 MRR) IT)
(PUTCHARBITMAP (CHARCODE _) '(MODERN 10 BRR) IT)
(PUTCHARBITMAP (CHARCODE _) '(TERMINAL 10 MRR) IT)
(PUTCHARBITMAP (CHARCODE _) '(TERMINAL 10 BRR) IT)
(PUTCHARBITMAP (CHARCODE _) '(TERMINAL 12 BRR) IT)
```

CONVERT-LOOPS-FILES

By: Bob Bane (Bane.pa@Xerox.com)

5-Feb-88

INTRODUCTION

CONVERT-LOOPS-FILES allows you to convert files from the Koto release of Xerox LOOPS to the Lyric/Medley release. The changes from the Koto release are described in detail in the *Xerox LOOPS Release Notes*.

PROCEDURE

To Convert Koto LOOPS Files to Lyric/Medley Loops:

1. Your files must be from Koto LOOPS. Pre-Koto LOOPS files must first be run through the Buttress->Koto converter in a Koto LOOPS sysout.

2. Install and load your Lyric LOOPS sysout. You need to have TEdit loaded, as the converter uses it.

3. Load the file CONVERSION-AIDS.DFASL, which defines the function

   ```
   (CONVERT-LOOPS-FILES <list-of-files> <dump-files-p>)
   ```

   where:

   `<list-of-files>` A filename or a list of filenames to be converted.

   `<dump-files-p>` Determines disposition of files.

   - If non-NIL, converted files will be dumped back out immediately.
   - If :COMPILE, they will be compiled.
   - If :COMPILE/LOAD, they will be compiled and the compiled code loaded.

4. Call CONVERT-LOOPS-FILES with the names of the files you want converted and an appropriate option.

Note the following:

- CONVERT-LOOPS-FILES makes more than one pass over the files being converted; the first pass is a TEdit textual change to make the files loadable into Lyric/Medley LOOPS. If you specify version numbers in your <list-of-files>, these changes will be made in place on your original files.

- The converter doesn't work completely automatically on systems consisting of several files that automatically load themselves with FILES coms; the converter may try to load subfiles before they are converted. It may be necessary to RETURN NIL from the LOAD calls in the break windows that will occur when this problem comes up. Aside from that, the converter works reasonably well, and has been used to convert large LOOPS systems with almost no source code changes from Koto Loops.

[This page intentionally left blank.]

LOOPSBACKWARDS

By: Bob Bane (Bane.pa@Xerox.com)

15-Dec-87

INTRODUCTION

LOOPSBACKWARDS allows you to run files which were previously converted from the Buttress release of Xerox LOOPS to the Koto release. Unlike the Koto version of LOOPSBACKWARDS conversion methods for moving files from Buttress LOOPS to Koto LOOPS are included but **not supported**. *We strong recommend that you convert LOOPS source code from Buttress to Koto using the Koto release of LOOPS.* Conversion of Koto LOOPS source code to Lyric LOOPS is done using the CONVERSION-AIDS users' module.

The changes between the Buttress and Koto releases are described in detail in the *Xerox LOOPS Release Notes* for the **Koto** release. Old features that are included in LOOPSBACKWARDS are summarized here:

- Many functions that were removed from the Buttress release are defined.
- The messages **List** and **List!** are available.
- The operation of old style active values, from the Buttress release, are provided.
- Support for reading old style macros, such as #(localState getFn putFn) or #$Mumble, is available.

LOOPSBACKWARDS is an unsupported LOOPS users module. It is strongly recommended that it only be used as part of an effort to upgrade very old LOOPS code to newer releases. It will allow very old LOOPS code to run well enough that it can be rewritten for a newer relese.

INSTALLATION

LOOPSVCOPY will be automatically loaded by LOOPSBACKWARDS.

FUNCTIONS

LOOPSBACKWARDS includes **ExplicitFnActiveValue** and **DefAVP**. **ExplicitFnActiveValue** allows the user code triggered by Get- or Put- accesses to be stored within functions which are pointed to by instance variables rather than requiring the redefinition of **GetWrappedValue** or **PutWrappedValue**. These functions must have the form specified in the **DefAVP** function.

ExplicitFnActiveValue [Class]

Purpose: Mimics the behavior of the Buttress style of active values.

Behavior: Get- accesses to the wrapped variable cause the **getFn** to be called, Put- accesses cause **putFn** to be called. Enables the old style activeValue to look like the new style without changing any functionality.

The **getFn** is called by the **ExplicitFnActiveValue GetWrappedValue** method. This method passes to the **getFn** the arguments defined by **DefAVP** as described in the *Xerox LOOPS Users' Modules*.

The **putFn** is called by the **ExplicitFnActiveValue PutWrappedValue** method. This method passes to the **putFn** the arguments defined by **DefAVP** as described in the *Xerox LOOPS Users' Modules*.

Instance Variables:

localState A place for data storage.

getFn The name of a function applied when the active variable is read.

putFn The name of a function applied when the active variable is changed.

Example:

```
32_ (_ ($ Bin) New 'bin4)
#,($& Bin (|DAW0.1Y:.H53.]99| . 521))

33_ (_ ($ ExplicitFnActiveValue) New 'EFAV1)
#,($& ExplicitFnActiveValue (|DAW0.1Y:.H53.]99| . 522))

34_(DEFINEQ (PrintOnGet
          (self varName localSt propName activeVal type)
          (PRINTOUT T "I am:" , activeVal T) localSt))
(PrintOnGet)

35_(_@ ($ EFAV1) getFn 'PrintOnGet)
PrintOnGet

36_(_ ($ EFAV1) AddActiveValue ($ bin4) 'height)
#,($A #,NestedNotSetValue PrintOnGet NIL)

37_(_@ ($ bin4) height 123)
123

38_(@ ($ bin4) height)
```

```
I am: #,($ EFAV1)
123
```

(**DefAVP** *fnName putFlg*) [Function]

Purpose: Creates a template for defining an active value function.

Behavior: Creates a template and leaves you in the Interlisp function display editor.

Arguments: *fnName* Name of the function.

putFlg T indicates function is a **putFn**, NIL indicates a **getFn**.

Returns: The function name on exit from the editor .

Example: In each of the following cases the template only is shown. User code is to be added immediately after the comment by using the display editor.

```
66_(DefAVP 'AGetFn)
AGetFn

67_PP* AGetFn
(AGetFn
  [LAMBDA (self varName localSt propName activeVal type)
         (* This is a getFn. The value of this getFn is
returned as the value of the enclosing GetValue.)
    localSt])
(AGetFn)

68_(DefAVP 'APutFn T)
APutFn

69_PP* APutFn
(APutFn
  [LAMBDA (self varName newValue propName activeVal type)
          (* This is a putFn. ***NOTE*** The value of this
function will be returned as the value of any enclosing
PutValue. This usually means that you want to return the value
returned by PutLocalState.)
    (PutLocalState activeVal newValue self varName propName
type])
(APutFn)
```

[This page intentionally left blank.]

LOOPSMIXIN

By: Bob Bane (Bane.pa@Xerox.com)

15-Dec-87

INTRODUCTION

LOOPSMIXIN defines several small classes that can be mixed into your application classes. It also defines the class **Perspective** and its support classes, which are used in the implementation of Xerox LOOPS Rules. Perspectives allow you to view one object as having more than one class at a time; they have not been tested extensively outside of Rules and are known to have major bugs which don't affect Rules, so they are not documented here.

LOOPSMIXIN Classes

DatedObject - Defines the instance variables **created** and **creator** which are set at object creation time to the values of (DATE) and (USERNAME).

NamedObject - Defines the instance variable name and initializes it to an ActiveValue that insures that the Xerox LOOPS system name for the containing object is uniquely name; i.e. storing a name for the object in name causes the previous name for the object to be removed with **DeleteObjectName** and the new name for the object to be asserted with **NameEntity**.

GlobalNamedObject - A subclass of **NamedObject** that works the same way as **NamedObject**.

ListMetaClass - Specializes the **New** and **DestroyInstance** methods to keep a list of all instances of that class in the class property **AllInstances** for that class.

StrucMeta - A MetaClass useful for creating new classes. Specializes the **New** method to create a Class object by copying the instance variable and class variable descriptions of the current class. Class variables with a non-NIL Local property will not be copied.

TempClass - Specializes the **New** method to always create objects of this class using the **NewTemp** method, insuring that the objects will be temporary objects.

Perspective, Node, Template, TextItem - These classes are used in Xerox LOOPS Rules.

[This page intentionally left blank.]

[This page intentionally left blank.]

[This page intentionally left blank.]

Writer's Notes -- Production Details

This file includes notes on the production of *Xerox LOOPS Users' Modules Manual*, Lyric Beta Release. This manual is packaged in one binder.

Writer: Raven Kontur Brewster

Printing Date: 22 February 1988

Files Needed

To edit or print the manual, make sure you have the following files loaded:

IMTOOLS
SKETCH
GRAPHER

Fonts Used

{ERIS}<LISP>FONTS>

Modern font
- 18-point bold
- 14-point bold
- 12-point bold
- 10-point regular
- 10-point italic
- 10-point bold

Terminal font
- 10-point regular
- 10-point italic
- 10-point bold

Printing Information

The manual was printed under a Lyric sysout on the Tsunami printer.

1. INTRODUCTION TO RULE-ORIENTED PROGRAMMING IN XEROX 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 Xerox LOOPS for representing rules, and for organizing knowledge-based systems with rule-oriented programming. The Xerox 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 Xerox LOOPS.

Rules in Xerox LOOPS are organized into production systems (called RuleSets) with specified control structures for selecting and executing the rules. The work space for RuleSets is an arbitrary Xerox 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.

1.1 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 satisfied. Several rule languages 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 Xerox LOOPS.

Xerox LOOPS has the following major features for rule-oriented programming:

(1) Rules in Xerox LOOPS are organized into ordered sets of rules (called RuleSets) with specified 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 Xerox LOOPS is an arbitrary Xerox 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 Xerox 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-effect 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 rule language provides a concise syntax for the most common operations.

(9) There is a fast and efficient compiler for translating RuleSets into Interlisp functions.

(10) Xerox LOOPS provides facilities for debugging rule-oriented programs.

The following sections are organized as follows: Section 1.2, "Basic Concepts," outlines the basic concepts of rule-oriented programming in Xerox LOOPS. It contains many examples that illustrate techniques of rule-oriented programming. Section 1.3, "Organizing a Rule-Oriented Program," describes the rule syntax, and the remaining sections in this chapter discuss the facilities for creating, editing, and debugging RuleSets in Xerox LOOPS.

1.2 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 Xerox LOOPS is intended for applications to expert and knowledge-based systems.

The following sections outline some of the main concepts of rule-oriented programming. Xerox 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, Xerox 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 effort 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.

1.3 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 differently, 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 Xerox LOOPS program that uses more than one programming paradigm is factored across several of these dimensions.

There are three approaches to organizing the invocation of RuleSets in Xerox 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 finished. *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 Xerox LOOPS object. As with active values with Lisp functions for the *getFn* or *putFn*, these RuleSet active values can be triggered by any Xerox 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. For example, Figure 1 shows the RuleSet of consumer instructions for testing a washing machine. The work space for the ruleSet is a Xerox 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 of some rule is applied. Some rules, called one-shot rules, are executed at most once. These rules are indicated by preceding them with a one in braces ({1}).

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

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

  IF .Operational THEN (STOP T);

  IF load>1.0 THEN .ReduceLoad;

  If ~pluggedInTo THEN .PlugIn;

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

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

{1}  THEN dealer.RequestService;

{1}  THEN manufacturer.Complain;

{1}  THEN $ConsumerBoard.Complain;

{1}  THEN (STOP T);
```

Figure 1. Basic RuleSet

1.4 Control Structures for Selecting Rules

RuleSets in Xerox 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 satisfied for it to be executed. Execution is also controlled by data in that different values in the data allow different rules to be satisfied. Criteria for iteration and rule selection are specified by a RuleSet control structure. There are two primitive control structures for RuleSets in Xerox LOOPS which operate as follows:

Do1 [RuleSet Control Structure]

The first rule in the RuleSet whose conditions are satisfied 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 specific and general situations, the specific 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 satisfied. 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 satisfied. In a single invocation of the RuleSet, all of the applicable rules are invoked.

Figure 2 illustrates the use of a **Do1** control structure to select one of 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 .FillSoak1 .Agitate .Soak4 .Agitate
    .Soak1 .SpinAndDrain .SprayAndRinse
    .SpinAndDrain .Fill .DeepRinse .Pause .DampDry;
```

Figure 2. RuleSet showing Do1

There are two control structures in Xerox 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 satisfied, the first rule is executed whose conditions are satisfied. This is repeated as long as the while condition is satisfied or until a **Stop** statement or transfer call is executed (see Section 2.14, "Stop Statements"). 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 satisfied, every rule is executed whose conditions are satisfied. This is repeated as long as the while condition is satisfied 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 specified 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. Figure 3 illustrates a simple use of the **WhileAll** control structure to specify a sensing/acting feedback loop for controlling the filling of a washing machine tub with water.

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

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

{1!} IF loadSetting = 'Small THEN waterLimit_10;
{1!} IF loadSetting = 'Meduim 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 termperatureSetting = 'Hot
  THEN HotWaterValve.Open ColdWaterValve.Close;

  IF termperatureSetting = 'Warm
  THEN HotWaterValve.Open ColdWaterValve.Open;

  IF termperatureSetting = 'Cold
  THEN HotWaterValve.Close ColdWaterValve.Open;

(* Stop when the water reaches its limit.)

  IF waterLevelSensor.Test >= waterLimit
  THEN HotWaterValve.Close ColdWaterValve.Close
    (Stop T);
```

Figure 3. RuleSet with WhileAll

There are two control structures in Xerox LOOPS that specify iteration over a set of elements 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.

FOR1 [RuleSet Control Structure]

This is a cyclic version of **Do1**. If the iteration-condition (or while-condition) is satisfied, the first rule is executed whose conditions are satisfied or until a **Stop** statement is executed. This is repeated as long as the iteration condition is satisfied. The value of the RuleSet is the value of the last rule that was executed, or **NIL** if no rule was executed.

FORALL [RuleSet Control Structure]

This is a cyclic version of **DoAll**. If the iteration-condition is satisfied, every rule is executed whose conditions are satisfied. This is repeated as long as the iteration condition is satisfied 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 "iteration-condition" is specified in terms of the variables and constants accessible from the RuleSet. The simplest condition is

(FOR <iterVar> IN <setExpr> DO ruleSet) ;

The **setExpr** will be parsed with the RuleSet parser. The symbol **ruleSet** is a reserved word, and must be spelled as shown (no changes in capitalization).

Here is an example of iteration:

Control Structure: FORALL;
Iteration Condition: (FOR buyer IN (RoadStops ($ Consumer)) DO ruleSet) ;

For each buyer in the list produced by RoadStops, the ruleSet will be run. In a **FOR1**, the iteration will go on to the next buyer as soon as one rule executes. In a **FORALL**, all rules in the RuleSet will be tried.

For nested iteration one can use a slightly more complicated form, as illustrated by the following example:

Iteration Condition: (FOR buyer IN (RoadStops ($ Consumer)) DO
(FOR seller in (RoadStops ($ Producer)) DO ruleSet)) ;

An experienced Lisp user can see that this resembles the CLISP iteration construct. In fact, except that you can (must) use the RuleSet syntax in the construct, it is the CLISP construct, and any such construct can be used. A DO1 or DOALL ruleSet will be substituted for the occurrence of the atom ruleSet, depending on whether the Control Structure is a FOR1 or FORALL.

As an abbreviation, if the construct does not contain the atom ruleSet, then (DO ruleSet) is appended to the Iteration Condition for a **FOR1** or **FORALL**. Thus one could write the first example as:

Iteration Condition: (FOR buyer IN (RoadStops ($ Consumer)))

1.5 One-Shot Rules

One of the design objectives of Xerox 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 in which some of the rules should be executed only once. This was illustrated in the Washing Machine example in Figure 1 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 3.

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 *condition condition* **THEN triedRule3_T** *action* **;**

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, Xerox LOOPS provides a shorthand notation as follows:

{1} **IF** *condition condition* **THEN** *action* **;**

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 *condition condition* **THEN** *action* **;**

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

{1!} **IF** *condition* *condition* **THEN** *action* **;**

These rules are called "try-once" rules.

The two kinds of one-shot rules are our first examples of the use of meta-descriptions preceding the rule body in braces. See Section 1.7, "Saving an Audit Trail of Rule Invocation," for information on using meta-descriptions for describing the creation of audit trails.

1.6 First/Last Rules

It is sometimes useful to have rules which fire before or after the ordinary part of the RuleSet is invoked, independent of the form of the control structure. For example, in a DO1, such "FIRST " rules could be used for initialization. These now exist, and are notated by putting a {F} for a first rule in the MetaDescription field, and a {L} for a last rule. If a RuleSet has L rules which execute, the value of the RuleSet is the value of the last rule which executed.

1.7 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 Xerox LOOPS support experimentation with systems that can not only use rules to make inferences, but also keep records of the inferential process itself.

1.7.1 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 efficiently 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-defined calculus of belief revision, at whatever level of abstraction is appropriate for an application.

1.7.2 Overview of Audit Trail Implementation

When *audit mode* is specified 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-effect of the assignment of values to instance variables. Audit records are Xerox LOOPS objects, whose class is specified 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 specific rules. Rule-specific audit information is specified 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-specific information overrides RuleSet information.

During rule execution in audit mode, the audit information is evaluated after the rule’s LHS has been satisfied 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 specified in the audit record classes in order to define a uniform set of side-effects for rules of the same category. In the following example, such an active value is used to carry out a "certainty factor" calculation.

1.7.3 An Example of Using Audit Trails

The following example illustrates one way to use the audit trail facilities. Figure 4 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 difficulty of incomplete information about the product. For example in this RuleSet, the reliability of the washing machine is estimated to be 0.5 in the absence of specific information from *Consumer Reports*. The meta-descriptions for the rules, which appear in braces, categorize them in terms of the *basis of belief* (the category *basis* is either a fact or estimate) and a *certainty factor (cf)* that is supposed to measure the "implication power" of the rule. 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-defined audit trails and experimental methods of belief revision. (Realistic belief revision systems are usually more sophisticated than this example.)

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. Figure 5 illustrates the evaluation report for one machine and one of its audit records. In this example, each of the entries in the report has a reason and a cumulative certainty (cc) property in addition to the value. The value of the reason properties are audit records created as a side effect 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 effects. For example, the active in the **cf** variable or the audit record performs a computation to maintain a calculated cumulative certainty in the reliability variable of the evaluation report.

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.)

```
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.)

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

  {(basics_Fact cf_.8)}
  reliability_(_($ ConsumerReports) GetFacts machine);

  {(basics_Estimate cf_.4)}
  IF 'reliability THEN reliability_.5;
  .
  .
  .
```

Figure 4. RuleSet Showing Evaluation

```
EvaluationReport "uid1"
expense:  510
```

```
suitability:  Poor cc 1 reason ...
reliability:  .5 cc .6 reason "uid2"
.
.
.
AuditRec "uid2"
rule:  "uid3"
basis:  Estimate;
cf:  #(.4 NIL PutCumulativeCertainity)
```

Figure 5. Example of an Audit Trail

1.8 Comparison with Other Rule Languages

This section considers the rationale behind the design of the Xerox 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 find ways to factor control information out of the rules.

1.8.1 The Rationale for Factoring Meta-Level Syntax

One of the most striking features of the syntax of the Xerox 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 Figure 1:

{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 Xerox LOOPS rule language, a meta-description (MD) is specified in braces in front of the LHS of a rule. For another example, the following rule from Figure 4:

{(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 specified 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 first option, intelligibility is degraded. In the second option, the transparency of the system is degraded because the knowledge is hidden.

1.8.2 The Rationale for RuleSet Hierarchy

Some advocates of production systems have praised the flatness of traditional production systems, and have resisted the imposition of any organization to the rules. The flat 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 Xerox 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 flat 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 flat 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 flow 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 different parts of a program. Although context clauses admit the creation of "subroutine contexts", they require you 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 Xerox LOOPS is another example of the our desire to simplify the rules by factoring out control information.

1.8.3 The Rationale for RuleSet Control Structures

Production languages are sometimes described as having a *recognize-act cycle*, which specifies how rules are selected for execution. An important part of this cycle is the *conflict resolution strategy*, which specifies how to choose a production rule when several rules have conditions that are satisfied. For example, the **OPS5** production language has a conflict 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 specific conditions.

In designing the rule language for Xerox LOOPS, we have favored the use of a small number of specialized control structures to the use of a single complex conflict 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 different control relationships among the rules. For example, the **DoAll** control structure is useful for rules whose effects 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 Figure 1.

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 *a b c* **THEN** *d e* **;**
IF *a b c* **THEN** *d e* **;**
IF *a b c* **THEN** *d e* **;**

could be written alternatively as

Control Structure: DoAll;
Task Vars: firedSomeRule;

IF *a b c* **THEN firedSomeRule_T** *d e* **;**
IF ~firedSomeRule *a b c* **THEN firedSomeRule_T** *d e* **;**
IF ~firedSomeRule *a b c* **THEN firedSomeRule_T** *d e* **;**

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 reflect the assumption that earlier rules in the RuleSet were not satisfied.

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 *a b c* **THEN** *d e* **;**
IF ~*a b c* **THEN** *d e* **;**
IF ~*a* *~b c* **THEN** *d e* **;**

can be written as

Control Structure: Do1;

IF *a b c* **THEN** *d e* **;**
IF *b c* **THEN** *d e* **;**
IF *c* **THEN** *d e* **;**

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**.

1.8.4 The Rationale for an Integrated Programming Environment

RuleSets in Xerox 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 benefits. It facilitates the construction of programs which don't entirely fit 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 Xerox LOOPS objects. Figure 6 illustrates the installation of the RuleSet **SimulateWashingMachineRules** to carry out the **Simulate** method for instances of the class **WashingMachine**. This definition of the class **WashingMachine** specifies 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 message to invoke the RuleSet may be sent by any Xerox LOOPS program, including other RuleSets.

The use of object-oriented paradigm is facilitated by special RuleSet syntax for sending messages to objects, and for manipulating the data in Xerox LOOPS objects. In addition, RuleSets, work spaces, and tasks are implemented as Xerox LOOPS objects.

```
[DEFCLASS WashingMachine
  (MetaClass Class Edited (* "rtk: 12-Jun-87 07:57")
    doc (* Home appliance for wachine cloothes.))
  (Supers ElectricalDevice PlumbedDevice CleaningDevice)
  (ClassVariables)
  (InstanceVariables
    (controlSetting Meduim
      doc (* One of Small, Medium, Large, ExtraLarge))...)
  (Methods
    (Fill WashingMachine.Fill doc (* Fill the tub with water.))
    (Wash WashingMachine.Wash doc (* Perofrm the wash cycle.))
    (Simulate UseRuleSet RuleSet SimulateWashingMachineRules)
    .
    .
    .
]
```

Figure 6. RuleSet Invoked as a Method

Using the data-oriented paradigm, RuleSets can be installed in active values so that they are triggered by side-effect when Xerox 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 figure. This data-oriented triggering can be caused by any Xerox LOOPS program when it changes the variable, whether or not that program is written in the rules language.

2. THE RULE LANGUAGE

This chapter describes the syntax and semantics of the rule language.

2.1 Language Introduction

A rule in Xerox LOOPS describes actions to be taken when specified conditions are satisfied. 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-specific control information, rule descriptions, audit instructions, and debugging instructions. For example, the syntax for one-shot rules shown in Section 1.5, "One-Shot Rules:"

{1} IF *condition* *condition* **THEN** *action* **;**

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 in Section 1.7, "Saving an Audit Trail of Rule Invocation."

LHS Syntax: The clauses on the LHS of a rule are evaluated in order from left to right to determine whether the LHS is satisfied. If they are all satisfied, then the rule is satisfied. 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 satisfied 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 satisfied. These actions are evaluated in order from left to right. Actions can be the invocation of RuleSets, the sending of Xerox 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 first character as follows:

(* This is a comment)

2.2 Kinds of Variables

Xerox 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 defined for RuleSets. These are declared in an **Args:** declaration.

Instance variables: All RuleSets use a Xerox LOOPS object for their workSpace. In the LHS and RHS of a rule, the first 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 Xerox 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 specific 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.

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-effect 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.

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

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, Xerox 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 first 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 Xerox LOOPS objects. This notation uses *compound literals*. A compound literal is a literal composed of multiple parts separated by a periods, colons, and commas.

2.3 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 satisfied 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 satisfied 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**.

2.4 Infix Operators and Brackets

To enhance the readability of rules, a few infix operators are provided. The following are infix binary operators in the rule syntax:

+ [Rule Infix Operator]

Addition.

++ [Rule Infix Operator]

Addition modulo 4.

- [Rule Infix Operator]

Subtraction.

-- [Rule Infix Operator]

Subtraction modulo 4.

***** [Rule Infix Operator]

Multiplication.

/ [Rule Infix Operator]

Division.

> [Rule Infix Operator]

Greater than.

< [Rule Infix Operator]

Less than.

>= [Rule Infix Operator]

Greater than or equal.

<= [Rule Infix Operator]

Less than or equal.

= [Rule Infix Operator]

EQ -- simple form of equals. Works for atoms, objects, and small integers.

~= [Rule Infix Operator]

NEQ. (Not **EQ**.)

== [Rule Infix Operator]

EQUAL -- long form of equals.

<< [Rule Infix 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 first and second minus signs are both treated as binary subtraction statements. That is, the first 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.

2.5 Interlisp Functions and Message Sending

Calls to Interlisp functions are parenthesized with the function name as the first 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 -> ... ;

Xerox LOOPS Objects and Message Sending: Xerox LOOPS classes and other named objects can be referenced by using the dollar notation. The sending of Xerox 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 specified 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 Xerox 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 Xerox LOOPS objects. For example,

$Tile.Type='BlueGreenCross ...

In this example, a unary **Type** message is sent to the Xerox 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;

2.6 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 Xerox 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 Xerox 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 Xerox LOOPS object. For example,

$TopTile:type='BlueGreenCross ...

refers to the **type** variable of the object whose Xerox 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 first 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 first 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.

2.7 Computing Selectors and Variable Names

The short notations for instance variables, properties, and unary messages all show the selector and variable names *as they actually appear* in the object.

*object***.***selector*
*object***:***ivName*
*object***::***cvName*
*object***:***varname***:,***propName*

(_ *object selector arg   arg* **)**

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 Xerox 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 selector arg   arg* **)**

2.8 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 finally (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.

2.9 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, first the object that is the value of **y** is computed, then the value of its instance variable **b** is set to **0**.

Properties: Assignment statements can also be used to set property values as in:

box:x:,origin_47 fact:,reason_currentSupport;

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.

2.10 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 specifications. Each meta-assignment statement specifies information to be saved in audit records when a rule is applied. In the following example from Figure 4, 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 satisfied. In addition, if the RuleSet was compiled in *audit* mode, then during RuleSet execution an audit record is created as a side-effect 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 satisfied, 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.

2.11 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 infix operator **<<** for membership testing.

2.12 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 Xerox LOOPS name, as in

$MyRules..ws1

which invokes the RuleSet object that has the Xerox 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.

2.13 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 first 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 Xerox 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.

2.14 Stop Statements

To provide premature terminations in the execution of a RuleSet, the Stop statement is provided.

(Stop *value***)** [RuleSet Statement]

value is the value to be returned by the RuleSet.

[This page intentionally left blank]

3. USING RULES IN LOOPS

The Xerox 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.

3.1 Creating RuleSets

RuleSets are named Xerox 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 Xerox LOOPS name as

RuleSet name: *RuleSetName*

Afterwards, the RuleSet can be referenced using Xerox 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***)**

3.2 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 **EditMethod** option from a browser containing the class. Alternatively, the **EditMethod** message can be used:

(_ *ClassName* **EditMethod** *selector***)** [Message]

Both approaches to editing retrieve the source of the RuleSet and put the user into the TTYIN or TEdit editor, treating the rule source as text.

Initially, the source is a template for RuleSets as shown in Figure 7. The rules are entered after the comment at the bottom. The declarations at the beginning are filled in as needed and superfluous declarations can be discarded.

```
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 7. Initial Template for a RuleSet

You 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 you choose 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 you leave 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 you are given another opportunity to edit the RuleSet.

3.3 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.

3.4 Saving RuleSets on Lisp Files

RuleSets can be saved on Lisp files just like other Xerox 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 file, it is necessary to add two statements to the file commands for the file 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.

If RuleSets are methods associated with a class, and they are saved by using (FILES?), then the file package saves the appropriate entries. The user does not have to be concerned with editing the filecoms of the file being made.

3.5 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 files will be printed on the **DEFAULTPRINTINGHOST** as is standard in Interlisp-D. **ListRuleSets** can be given four 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 file will be printed.

3.6 Running RuleSets from Xerox LOOPS

RuleSets can be invoked from Xerox LOOPS using any of the usual protocols.

Procedure-oriented Protocol: The way to invoke a RuleSet from Xerox LOOPS is to use the **RunRS** function:

(RunRS *RuleSet workSpace arg2 ... argN***)** [Function]

workSpace is the Xerox 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 Xerox 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-effect as a result of accessing the variable on which they are installed.

3.7 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.

DefRSM can be invoked with the browser as follows:

- Position the cursor over a class in a browser.
- Press the middle mouse button. A menu pops up.
- Select the Add option in this menu, and drag the mouse to the right to display the submenu that includes the "DefRSM" option. You are prompted to enter a selector name.

After a RuleSet has been installed as a method by using **DefRSM**, you can then edit that RuleSet by selecting the "EditMethod" option from the browser editing menu.

3.8 Installing RuleSets in Active Values

Note: The following section and any other references to active values within the rule documentation refer to active values as they were implemented in the Buttress release. The functionality of triggering rules from active values has not been tested using the current implementation of active values. It should work to use the **ExplicitFnActiveValue** class to implement this behavior.

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 Xerox LOOPS system function that can be used in an active value to invoke a RuleSet in response to a Xerox 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 Xerox 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.

3.9 Tracing and Breaking RuleSets

Xerox 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. The following 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.

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 Xerox 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 Xerox LOOPS is to enter the rule executive (see Section 3.10, "The Rule Exec") after the LHS is satisfied and before the RHS is executed. The rule-specific form:

{B} IF *cond* **THEN** *action***;**

indicates that Xerox 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.

3.10 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, you 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**.

3.11 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 a meta-descriptor.

Meta Assignments: cf_.5 support_'GroundWff;

3.12 Loading Rules

Set the variable LOOPSUSERSDIRECTORIES to include the directory where the Rules files are stored.

Load the file LOOPSRULES-ROOT.LCOM, which will load the following files from LOOPSUSERSDIRECTORIES:

- LOOPSBACKWARDS.LCOM
- LOOPSMIXIN
- LOOPSRULES.LCOM
- LOOPSRULESP.LCOM
- LOOPSRULESC.LCOM
- LOOPSRULESD.LCOM, which will load the file TTY.LCOM from LISPUSERSDIRECTORIES.

Editing rules will be easier if TEdit is loaded. Loading the Rules does not automatically load TEdit.

3.13 Known Problems

In a rule, the expression $pipe.ri..$p compiles to (RunRS (QUOTE ($ pipe)) ($ p)), which fails.

Meta-assignment statements cannot handle expressions. This means that statements like {cf _ .5} work fine, but {validity _ 'fact} fails.

A value of 1 in a meta-descriptor statement is always taken to be a one-shot designator. You cannot have a meta-descriptor statement like {cf_1}. However, the number 1.0 can be used; the meta-descriptor statement, {cf_1.0}, works.

Rules have not been tested without loading TEdit in order to edit RuleSets.

[This page intentionally left blank.]

RULES

Modified by: Rick Martin (Martin.pasa@Xerox.com)

14-Apr-86

[This page intentionally left blank.]

TABLE OF CONTENTS

LIST OF FIGURES

[This page intentionally left blank.]

A. CONVERTING FROM BUTTRESS RULES

The primary change to rules was the removal of the **VarLength** mixin as a super class of **RuleSetSource**. Rules are now stored on the IV **ruleList** of **RuleSetSource**. If any user had added properties to a rule with a construct similar to the following expression, these will need to be reimplemented in some fashion:

```
(PutNthValue ($ some-Instance-Of-A-RuleSetSource) ruleNumber prop)
```

A number of other changes were made to account for changes from Buttress LOOPS to the product release. These should not be noticeable to the general user.

A utility was built to allow rules that were saved in Buttress to be loaded into the product release of LOOPS. In order to convert from Buttress, follow this procedure.

- Load the necessary files for Rules as described in the Section 3.12, "Loading Rules."
- Load the file LOOPSRULESBACKWARDS. This redefines `ConvertLoopsFiles` to add functionality for converting rules.
- Do not load the Buttress files that need to be converted. Instead call:

  `(ConvertLoopsFiles` *files-to-convert* `T T)`

  This will load, convert, remake, and recompile the files specified by *files-to-convert*.

Any rules that were saved with auditing turned on will need to run through the rule compiler. The old rule compiler output a form that was (_ ($ StandardAuditRecord) NewTemp). The message **NewTemp** no longer exists. The rule compiler now puts out the form (_ ($ StandardAuditRecord) New).

If rules were saved with auditing turned on, then call `(ConvertLoopsFiles` *files-to-convert* `T)`. Deleting the final `"T"`, will not enable the cleanup. After recompiling the rules to eliminate the message **NewTemp**, remake your files.

[This page intentionally left blank]

SOURCES>

* LOOPS 10183 10346 10456
* LOADLOOPS 10346
* MEDLEY-PATCH 10346
* SEDIT-PATCH 10189
* LOOPS-FILEPKG 10495
* LOOPSBROWSE 10235
* INSPECT-PATCH 10357
* LOOPSUTILITY 10370

LIBRARY>

* LOOPSMS 10281
* MASTERSCOPE 10346 10436
* MSPARSE 10346
* MSANALYZE 10346
* MSCOMMON 10346

USERS>

* LOOPSBACKWARDS
* CONVERSION-AIDS

USERS>RULES>

* LOOPSRULES
* LOOPSRULES-ROOT
* LOOPSRULESC
* LOOPSRULESD
* LOOPSRULESP
* LOOPSRULESTTY

Ron:
10464 Comment removal AR
10436 MASTERSCOPE fix to MSANALYZE for CLASSES, into Medley

CACHEOBJECT

By: sML (Lanning.pa@Xerox.com)

4-Sep-86

INTRODUCTION

The file CACHEOBJECT defines a Loops mixin that defines a protocol for instances that cache computed values.

CLASSES

ObjectWithCache [Class]

Instance of the class ObjectWithCache follow a standard protocol for manipulating a dynamic cache. The cache is not stored when an ObjectWithCache is saved on a file.

(← *self* ClearCache) [Method of ObjectWithCache]

Clears the entire cache of *self*.

(← *self* ClearCacheEntry *name*) [Method of ObjectWithCache]

Removes the cache entry for *name* on *self*.

(← *self* GetCache *name*) [Function]

Returns the value of the cache entry for *name* on *self*. If there is no cached value, returns NIL.

(← *self* PutCache *name datum*) [Function]

Stores *datum* as the value of the cache entry for *name* on *self*. Returns *datum*.

LOOPS-FB

By: sML (Lanning.pa@Xerox.com)

INTRODUCTION

LOOPS-FB adds a command to the Lisp File Browsers for opening Loops browsers on files.

Loading the file will automatically add a "Browse" command to all new Lisp FIle Browsers. Selecting the "Browse" item will open a Loops FileBrowser on each of the currently selected files. The files will be loaded first if they are not currently loaded.

LOOPSIDLE

By: smL (Lanning.pa@Xerox.com)

4-Sep-86

INTRODUCTION

LOOPSIDLE make IDLE "bouncing box" function bounce a Loops icon about about the screen.

VARIABLES

BouncingLoopsIcon [Variable]

The (value of the) variable BouncingLoopsIcon is an expanded copy of the Loops icon. LOOPSIDLE sets IDLE.BOUNDING.BOX to (the value of) BouncingLoopsIcon.

Loops Image Objects

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn **Unknown IMAGEOBJ**
GETFN: LoopsImageObjectGetFn

Interlisp `IMAGEOBJ`s are "objects" that know how to display themselves on `IMAGESTREAM`s. `IMAGEOBJ`s are most often used to insert non-character items into a TEdit document. Standard Interlisp `IMAGEOBJ`s are available for displaying bitmaps, graphs, and horizontal bars.

Interlisp `IMAGEOBJ`s are not objects in the sense of Loops: there is no provision for specialization of existing `IMAGEOBJ`s and no default behavior is provided. Creation of a new type of `IMAGEOBJ`s requires some effort. Functions must be specified for printing, reading, displaying, and handling button events for the new `IMAGEOBJ` type.

Loops image objects are Loops objects that can be used as Interlisp `IMAGEOBJ`s. They let the programer use the full power of the Loops environment in the creation of new `IMAGEOBJ`s.

The LoopsImageObjects file defines a number of classes that can be used to create Interlisp `IMAGEOBJ`s, and an interface that makes it easy to insert these Loops image objects into a TEdit document. Section **Unknown IMAGEOBJ GETFN: LoopsImageObjectGetFn** tells how to insert Loops image objects into a TEdit document; section **Unknown IMAGEOBJ GETFN: LoopsImageObjectGetFn** describes some predefined Loops image object classes; sections **Unknown IMAGEOBJ GETFN: LoopsImageObjectGetFn** **Unknown IMAGEOBJ GETFN: LoopsImageObjectGetFn** describe the Loops image object protocol, for people wishing to define their own Loops image objects.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn Inserting Loops image objects into a TEdit document is easy.

`(LIO)` [Function]

The standard way to insert IMAGEOBJs into a TEdit file is via the CONTROL-O character. Hitting CONTROL-O lets you type in a LISP form, and the value of this form is an IMAGEOBJ that will be inserted into the document at the current location. (LIO) will present you with a menu of all known Loops image object classes. If you select a class from this menu, (LIO) will return the IMAGEOBJ that points to a new Loops image object of the specified class. If the instance has IVs that can be edited, you will be given a chance to edit the instance first.

LIOInsertCharCodes [Variable]

When the Loops image object package is loaded, it redefines the interpretation in TEdit of the characters specified by the variable LIOInsertCharCodes. Hitting one of these characters is equivilent to hitting CONTROL-O and then typing in the form (LIO). The default value of LIOInsertCharCodes is (the value of) (LIST (CHARCODE #W) (CHARCODE #w)). This is an INITVAR, so you override the default before you load the package.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Many of these objects display themselves as text. This can be confusing for the user, who can easily mistake the object for a sequence of characters. To avoid this possibility for confusion, these objects appear boxed when displayed in a TEdit window. The box will not appear in a printed document.

The character looks of these objects can be set from TEdit in the same way that normal characters can have their looks set: select the object and use TEdit's Character Looks Menu, or the Looks item on the title bar pop-up menu. Note that the object can have only a single character looks--all the characters in the object will be displayed with the same looks. There is no way to change the looks of single characters displayed by a Loops image object.

These objects often violate the principle of WYSIWYG (What You See Is What You Get) in that they display more text in a TEdit window then they do when printed. They let you know what text will appear in print by displaying all non-printing text inverted. Thus, the object that displays in a TEdit window as Index: IndexEntry will show up as the string "IndexEntry" when printed. (Note the interaction of this inversion with the boxing mentioned in the previous paragraph: the inversion is performed after the boxing, so that the entire object will still appear boxed if displayed inverted (as in TEdit's highlighting to indicate pending-delete).)

These objects respond to a button event by presenting a menu of options to the user. The items in the menu depend on the class of the object. All objects include a menu item for storing the Loops instance in `(SavedValue)`, and for inspecting the instance. If the instance has IVs that can be set by the user, the menu will include an "Edit" item. The "Edit" item will bring up the standard Loops instance editor on the object. If the instance has a textual IV, the menu will include an option to edit the value of this IV in a new TEdit window.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Many people include in a TEdit file the time the file was last edited, or the name of the file that holds the document. The following classes can be used to automate this process.

`WhenLastSaved` [Class]

A `WhenLastSaved` image object will display a time stamp indidating the time the TEdit document was last saved. For example, a `WhenLastSaved` instance in this document might produce the image 3-Jan-86 18:21:15 by arnl when displayed in a TEdit window (but would appear without the box when hardcopied).

If the document has not been saved since the `WhenLastSaved` object was inserted, the `WhenLastSaved` object will display the string "NotYetSaved" instead of the time stamp.

`WhereLastSaved` [Class]

A `WhereLastSaved` image object is much like a `WhenLastSaved` object, except that it displays the name of the file that the TEdit document was last saved in, instead of the time last saved.

`WhenLastSaved` and `WhereLastSaved` objects are used in the running footers in this document.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Using image objects, it is easy to build an index table for a TEdit document.

`IndexEntry` [Class]

An `IndexEntry` object associates a string with a page number for inclusion in an index. The IV `text` of an `IndexEntry` will be added to the accumulated index, together with the number of the page containing the object. If the IV `displayText?` is non-`NIL` (the default), then the text string will also appear in the document, otherwise the object will be invisible in the final hardcopy. For example, an `IndexEntry` that adds a reference to the string "IndexEntry" appears in a TEdit window as

Index: IndexEntry . If the displayText? IV were NIL, it would appear as Index: IndexEntry it window, but would be invisible when printed.

InitIndex [Class]

An InitIndex object must appear in a TEdit document before any IndexEntry object. The InitIndex object initializes the collection of the index table. It will be invisible when printed.

CollectIndex [Class]

A CollectIndex object should appear after all the IndexEntry objects that appear in a TEdit document. It will sort the index and print the index information into a new TEdit window (you will be prompted for the window). You can then format the text, save it away, print it, or discard it. The IVs looks, paraLooks, firstPageFormat, rectoPageFormat, and versoPageFormat determine the initial formatting of the index. See the TEdit documentation for a discussion of the definition of these fields. The CollectIndex object will not be visible in the printed document.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Automatic chapter and section numbering, and generation of a table of contents, is provided by the following classes.

SectionHeading [Class]

SectionHeading objects have three IVs: level, text, and displayText?. The level IV determines the subsection nesting of the object. Each SectionHeading object in the document increments the current section number at the given level. Smaller level numbers will not be affected. Larger level numbers are reset to zero. For example, a sequence of SectionHeading objects with level IVs of 1, 2, 1, 2, 2, 3, 2, and 1 will produce final section numbers of 1., 1.1., 2., 2.1., 2.2., 2.2.1., 2.3., and 3.

The text IV determines the section title for inclusion in the table of contents. If the displayText? is non-NIL (the default), the text will also be displayed in the document.

As an example, when first displayed in a TEdit window the SectionHeading for this subsection looked like n.n. Building a Table of Contents . If the displayText? IV were NIL, the text would have been inverted to indicate that it will not be present in the final hardcopy: n.n. Building a Table of Contents . Once the containing document had been hardcopied, the object displayed the section number used at hardcopy time instead of the placeholder "n.n.". (Due to a "bug", you have to redisplay the window to see the updated section numbers.)

`InitTOC` [Class]

An `InitTOC` object must appear in a TEdit document before any `SectionHeading` object. The `InitTOC` object initializes the collection of the table of contents. The IV `initialSectionNumbers` can be used to initialize the first section numbers to values other than `1`. For example, if the `initialSectionNumbers` IV is `(2 1 3)`, and the first `SectionHeading` object has a `level` IV of `3`, it will be given the subsection number "2.1.4.". The `InitTOC` object will be invisible when printed.

`CollectTOC` [Class]

A `CollectTOC` object should be placed at the end of a document, after all `SectionHeading` objects, to collect the table of contents information into a new TEdit stream. When the document is hardcopied, a new TEdit stream will be opened containing the accumulated table of contents. You can then format this TEdit document as you wish. `CollectTOC` uses the same IVs as does the `CollectIndex` to determine the initial formatting of the table of contents. The `CollectTOC` object not be visible in the final hardcopy.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

The following classes provide a somewhat clumsy way of having page and section references included an a document. Because there is no lookahead in the TEdit page formatting, documents containing these objects must be hardcopied twice. The first time will compute the correct page and section numbers, the second time will incorporate them into the document.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

`PageNote` [Class]

At hardcopy time, a `PageNote` object remembers the current page number, associating it with the value of its `tag` IV. See the `PageReference` class, below.

`PageReference` [Class]

At hardcopy time, a `PageReference` object looks up the value of its `tag` IV to see if a `PageNote` object has noted a page number for that tag. If there is page number stored, the `PageReference` object will display the number, otherwise it will display the string "nn". If the `PageNote` occurs before the PageReference object in the document, this reference will be found the first time the document is printed. If the PageNote occurs after the PageReference object, a second hardcopy must be generated.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

SectionNote [Class]

At hardcopy time, a SectionNote object remembers the current section number, associating it with the value of its tag IV. See the SectionReference class, below.

SectionReference [Class]

At hardcopy time, a SectionReference object looks up the value of its tag IV to see if a SectionNote object has noted a section number for that tag. If there is section number stored, the SectionReference object will display the number, otherwise it will display the string "n.". If the SectionNote occurs before the SectionReference object in the document, this reference will be found the first time the document is printed. If the SectionNote occurs after the SectionReference object, a second hardcopy must be generated.

SectionReferences are used in this document in the last paragraph of the first section.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

The following class illustrates the posiblity of nesting image objects inside of other image objects.

BoxedImageObject [Class]

IVs: boxShade (the shade of the box: either a shade or a form that will be evaluated to a shade; e.g. GRAYSHADE), boxWhiteSpace (blank space between the boxed object and the box), boxWidth (the width of the box), boxedObject (a LoopsImageObject).

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

A common document form consists of a table of text entries. These are easy to build in TEdit when each entry fits within its row without having to occupy multiple lines.

**Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn** **Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn** **Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn**

Example: BoxedImageObjects and TableTextObjects

The following class takes care of the case where the table entries are too large to fit on a single line.

`TableTextObject` [Class]

describes a (possibly multiple line) table entry. The class `TableTextObject` defines five important IVs.

`text` [IV of TableTextObject]

contains the text that will be displayed in the table entry.

`widthInWs` [IV of TableTextObject]

defines the width of the table entry. This width is measured in "W"s, in the current font. If the `text` is too long, it will be broken at word boundaries and displayed in multiple lines. The default value is `10`.

`nLines` [IV of TableTextObject]

defines the maximum number of lines to be used to print the text. If `nLines` is `NIL` (the default), the object will be as tall as it needs to be to display the entire `text` subject to the width constraint specified by the `widthInWs` IV.

`justify` [IV of TableTextObject]

determines the horizontal justification of the individual lines within the table entry. Possible values are `left` (for left–justify), `center` (for centered text), and `right` (for right–justify). The defualt is `left`.

`verticalJustify` [IV of TableTextObject]

determines the vertical justification of the lines within the table entry. Possible values are `top` (the first line will be flush with the top of the table entry), `center` (the text will be centered vertically), and `bottom` (the last line of text will be at the bottom of the entry). The defualt is `bottom`. Note that this IV only has an effect if the IV `nLines` is specified.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Image objects can be used to include prettyprinted forms in a document.

`PPImageObject` [Class]

prettyprints a form in the image stream. The relevant IVs of `PPImageObject` are:

`form` [IV of PPImageObject]

~mumble~

`minWidth` [IV of PPImageObject]

~mumble~

`maxWidth` [IV of PPImageObject]

~mumble~

Due to an unfortunate bug in Interlisp, `PPImageObject` work correctly on display devices .

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Example: PPImageObject

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Loops image objects are Loops objects that are wrapped around an Interlisp `IMAGEOBJ`. The `IMAGEOBJ` in turn points back to the Lops image object. When Interlisp "sends a message" to the underlying `IMAGEOBJ` (more correctly, applies one of the `IMAGEOBJ`'s `IMAGEFNS` to the `IMAGEOBJ`), the `IMAGEOBJ` forwards this message on to the Loops image object. The translation from Interlisp `IMAGEOBJ` protocol to Loops image object protocol is accomplished by a special set of `IMAGEFNS` and the `LoopsImageObject` class.

`LoopsImageObject` [AbstractClass]

~mumble~

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

The following methods implement the `IMAGEFNS` for the Loops image object. These message are not intended to be sent by the user; they are sent automatically by TEdit and other packages.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

(_ *self* ImageBox *imageStream currentX rightMargin*) [Method of LoopsImageObject]

returns an instance of the record `IMAGEBOX` with fields `XSIZE`, `YSIZE`, `XKERN`, and `YDESC`. This is used by TEdit to determine the size of the object for formatting purposes. Specializations of `LoopsImageObject` should override this method.

(_ *self* Display *imageStream*) [Method of LoopsImageObject]

should actually display the object on the *imageStream*. Specializations of `LoopsImageObject` should override this method.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

(_ *self* ButtonEventIn *windowStream selection relX relY window textStream button*) [Method of LoopsImageObject]

~mumble~

(_ *self* CopyButtonEventIn *windowStream*) [Method of LoopsImageObject]

~mumble~

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

(_ *self* Copy) [Method of LoopsImageObject]

~mumble~

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Due to restrictions imposed by IMAGEOBJects, LoopsImageObjects cannot duplicate the full generality of saving and retoring Loops objects. Specialized subclasses of LoopsImageObjects cannot change how they are stored or read back in. However, the following methods do give the user some control over what extra information is dumped out when a LoopsImageObjects is stored.

(_ *self* BeforePutToFile *stream*) [Method of LoopsImageObject]

~mumble~

(_ *self* AfterPutToFile *fileStream*) [Method of LoopsImageObject]

~mumble~

(_ *self* AfterGetFromFile *textStream*) [Method of LoopsImageObject]

~mumble~

(_ *self* PrePrint) [Method of LoopsImageObject]

~mumble~

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

(_ *self* WhenCopied *targetWindowStream* *sourceTextStream* *targetTextStream*) [Method of LoopsImageObject]

~mumble~

(_ *self* WhenDeleted *targetWindowStream* *sourceTextStream* *targetTextStream*) [Method of LoopsImageObject]

~mumble~

(_ *self* WhenInserted *targetWindowStream* *sourceTextStream* *targetTextStream*) [Method of LoopsImageObject]

~mumble~

(_ *self* WhenMoved *targetWindowStream* *sourceTextStream* *targetTextStream*) [Method of LoopsImageObject]

~mumble~

(_ *self* WhenOperatedOn *windowStream howOperatedOn selection textStream*) [Method of LoopsImageObject]

~mumble~

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

The class `LoopsImageObject` provides a number of other methods that can be used by specializations of the above methods.

(_ *self* CachedImageBox *imageStream*) [Method of LoopsImageObject]

returns the image box computed by the last `ImageBox` method. This is often used inside the `Display` method to avoid resending the `ImageBox` message.

(_ *self* DisplayImageStream? *imageStream*) [Method of LoopsImageObject]

returns `NIL` unless the image stream is a display image stream.

(_ *self* PrintText *imageStream text font*) [Method of LoopsImageObject]

prints the string *text* in the font *font*, centered in the object's `CachedImageBox`.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

Certain subclasses of `LoopsImageObject` capture some commonly desired functionality. These may be useful to anyone interested in building their own Loops image objects.

`TEditImageObject` [AbstractClass]

The class `TEditImageObject` contains a few methods that are only applicable when the object is being displayed in a TEdit stream.

(_ *self* CurrentFont *imageStream*) [Method of TEditImageObject]

returns the font of the image object in the current TEdit stream.

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

`(_ self TextStream imageStream)` [Method of TEditImageObject]

returns the text stream that is being displayed in *imageStream*.

`(_ self TEditIV ivName textStream)` [Method of TEditImageObject]

edits the text stored in the IV *ivName* of *self* in a new TEdit window. The image object should be part of the TEdit stream *textStream*. The textStream argument is used to update the display when the IV has been edited.

`(_ self AllObjects textStream)` [Method of TEditImageObject]

returns a list of all imageobjects contained in *textStream*, in order.

`WhenSavedImageObject` [AbstractClass]

~mumble~

`HardcopySideEffectObject` [AbstractClass]

~mumble~

`LabelImageMixin` [AbstractClass]

~mumble~

`EditableImageObjectMixin` [AbstractClass]

~mumble~

`TEditableImageObjectMixin` [AbstractClass]

~mumble~

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

On occasion, it may be useful to wrap a Loops image object around an existing Interlisp `IMAGEOBJ`, say to wrap a BoxedImageObject around it.

`ImageObjectWrapper` [Class]

~mumble~

Unknown IMAGEOBJ
GETFN: LoopsImageObjectGetFn

LOOPSINDEX

By: sML (Lanning.pa@Xerox.com)

4-Sep-86

INTRODUCTION

LOOPSINDEX sets up the SINGLEFILEINDEX package to dump out classes, methods, and instances in an intelligible way. LOOPSINDEX can be loaded into any Interlisp sysout: you do not need to have Loops loaded to load LOOPSINDEX.

LOOPSINDEX adds appropriate entries to the lists SINGLEFILEINDEX.TYPES and SINGLEFILEINDEX.FILTERS to cause SINGLEFILEINDEX to index all classes, methods, and instances defined on a file. See the documentation of SINGLEFILEINDEX for a detailed description of these variables.

LOOPSINDEX will load SINGLEFILEINDEX if it is not already loaded.

HOW TO SET UP FOR TESTING LOOPS

Start with a fresh LISP.SYSOUT from {erinyes}<lisp>lyric>basics>, dated 27-Apr-87.

Give no INIT file

Log in.

Type into the exec:
(SETQ IL:DISPLAYFONTDIRECTORIES '("{ERINYES}<lisp>LYRIC>FONTS>"))
CONN {ERINYES}<lisp>Lyric>library>
(il:load 'TEDIT.lcom)
(il:tedit '{erinyes}<cate3>loops>LOOPS-setup.tedit)

This should bring up this document.
Shift selected all lines below here into the EXEC:

(il:load 'FILEBROWSER.lcom)
CONN {erinyes}<lisp>lyric>lispusers>
(il:load 'WHO-LINE.dfasl)
(il:load 'filewatch.lcom)

CONN {erinyes}<lispusers>lyric>
(il:load 'CROCK.lcom)

CONN {erinyes}<test>lisp>lyric>internal>library>
(il:load 'do-test.dfasl)

(setq il:*DEFAULT-CLEANUP-COMPILER* 'cl:COMPILE-FILE)

(setq il:ch.default.domain "AISNORTH")
(SETQ IL:CH.DEFAULT.ORGANIZATION "XEROX")
CONN |{PELE:AISNorth:Xerox}<CATE3>|

(il:closew il:logow)
(IL:CHANGEBACKGROUND 42405)
(il:crock)

Follow the instructions for loading loops from the release notes manual.
As long as the workstation can get the font files from the network, steps 1-3 can be ignored in the Installation of Loops (4.3 of Release Notes)
For step 4:
The modified network version:

CONN {ERINYES}<lisp>Lyric>library>
(il:load 'grapher.lcom)

For steps 5-6:
Insert Lyric LOOPS System #1 floppy, then type:
(il:fb '{floppy})
and copy all files to {dsk}<lispfiles>loops>
Do the same for the Lyric LOOPS System #2 floppy
Then type or shift select in an Interlisp window:
CONN "{DSK}<lispfiles>loops>"
(LOAD 'LOOPS)

Stuff the following lines into a temp file in core then shift select out the sysout command.
(il:sysout '{erinyes}<cate3>loops>test-loops.sysout)
(il:login)
(il:fb '{erinyes}<cate3>loops>)
; Make sure the Loops stuff is copied to <lispfiles>loops>
load the right software, and change to loops stuff

The basic loop is (TIMEALL *form* 10000).

[[Timing old SmallLoops, on an 1132]]

Test name	**Time (secs)**	**Time in test (less var lookup)**
Global variable lookup	0.05	0.00
Field fetch	0.07	0.02
Function call	0.11	0.06
Send	0.48	0.43 (= x 7 a function call)
Inherited GetValue	1.95	1.90 (= x 32 a function call)
Local GetValue	0.58	0.53 (= x 9 a function call)
Local PutValue	0.69	0.64 (= x 11 a function call)
Inherited GetIVProp	1.97	1.92 (= x 32 a function call)
Local PutIVProp	1.20	1.05 (= x 18 a function call)
Instance creation, 2IVs	???	??? (= x ??? a function call)

[[Timing new Loops, on an 1132]]

Test name	**Time (secs)**	**Time in test (less var lookup)**
Global variable lookup	0.05	0.00
Field fetch	0.07	0.02
Function call	0.11	0.06
Send (local cache)	0.22	0.17 (= x 3 a function call)
Send (global cache)	0.??	0.?? (= x 7 a function call)
Send (no cache)	0.??	0.?? (= x 7 a function call)
Inherited GetValue	1.77	1.72 (= x 29 a function call)
Local GetValue	0.37	0.32 (= x 5 a function call)
Local PutValue	0.48	0.43 (= x 7 a function call)
Inherited GetIVProp	3.36	3.31 (= x 55 a function call)
Local PutIVProp	1.11	1.06 (= x 18 a function call)
Instance creation, 2IVs	7.85	7.85 (= x 131 a function call)