other objects that have been created. The Layer for this is marked as associated with *KBName*. Finally, the environmental objects for the knowledge base are written out to a boot layer.

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

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

## 9.6 Freezing and Thawing References to Knowledge Bases

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

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

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

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

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

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

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

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

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

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

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

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

## 9.7 Using Several Knowledge Bases in an Environment

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

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

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

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

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

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

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

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

The next step is to open an environment:

```
(← $environmentName Open)
```

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

(← $*environmentName* AssocKB '*otherKBName*$_1$)

This message first checks that there is a knowledge base named *otherKBName*$_1$ in the environment. It does not cause the changes to be written to the other knowledge bases. Rather, it causes a specially marked layer to be created in the user's personal knowledge base which can be accessed later by the community knowledge base manager.

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

(← $*environmentName* AssocKB '*KBName*)

As before, the user can type

(← $*environmentName* Close)

when he is done with the session.

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

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

## 9.8 Changing the Associations of Objects

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

(← $*objectName* AssocKB '*newKBName*)

## 9.9 Switching Among Environments

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

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

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

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

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

```
(← $environmentName1 Close)
```

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

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

```
(← $environmentName2 MakeCurrent)
```

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

```
(← $environmentName3 MakeCurrent)
```

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

```
(← $testedEnvironment IsCurrent).
```

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

```
(← CurrentEnvironment MakeNotCurrent)
```

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

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

```
(← $Environment Cleanup)
(← $Environment Close)
```

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

```
(← $toEnvironment CopyObjects objectsList)
```

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

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

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

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

## 9.10 Saving Parts of a Session

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

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

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

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

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

```
(← $environmentName Cancel)
```

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

## 9.11 Copying Layers from one Knowledge Base to Another

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

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

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

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

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

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

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

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

For example:

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

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

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

```
(← $toKBName CopyFileLayers layerDescription)
```

## 9.12 Summarizing and Combining Knowledge Bases

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

(← $KB Summarize *fromKBName toKBName assocKBNames*)

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

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

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

(← $KB Summarize *fromKBNames toKBName assocKBNames*)

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

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

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

(← *environment* DumpToKB *toKBName assocKBNames*)

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

## 9.13 Subdividing a Knowledge Base

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

(← $*environmentName* MapObjectNames (FUNCTION *UserFn*) *AssocKBs NoUIDs*)

where

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

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

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

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

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

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

The next step is to move the objects as follows:

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

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

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

The final step is to write out the changes:

```
(← $environmentName Cleanup)
```

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

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

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

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

```
(← $KBName Copy)
```

## 9.15 Affecting what is Saved

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

### 9.15.1 Temporary Objects

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

(← *class* NewTemp)

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

### 9.15.2 Not Saving some IV values

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

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

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

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

### 9.15.3 Ignoring changes on an IV

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

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

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

### 9.15.4 Getting rid of objects explicitly

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

(← *obj* `Destroy`)

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

## 9.16 Examining Environmental Objects

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

(← $*Layer1* `MapObjectNames` *mapFn noUIDs*)

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

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

(← $KB `Old` '*KBName*)

without necessarily opening an environment, then one can send:

(← $*KBName* `PP`)

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

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

(← $E1 ChangedKBs)

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

## 9.17 The Class KBState

`KBState` [Class]

IVs:

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

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

Methods:

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

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

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

(← *self* `CurrentState`) [Method of KBState]
Create a KB state which reflects the current state of this KB.

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

(← *self* `Files` *fileList*) [Method of KBState]
*fileList* is a `TCONC` list of files already found. Add any new ones found. Very similar in structure to `KBState.Connect`.

(← *self* `MyKB`) [Method of KBState]
Return the KB object corresponding to this KBState.

(← *self* `ReadBoot`) [Method of KBState]
Read the boot file for this KB.

(← *self* `SetContents` *lst*) [Method of KBState]
Make KB have new contents. Check types of elements.

## 9.18 The Class KB

`KB` [Class]

IVs:

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

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

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

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

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

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

Methods:

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

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

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

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

(← *self* `Disconnect`) [Method of KB]
Disconnect this KB and close its file if open.

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

(← *self* `PrintContents` *file*) [Method of KB]
Fn to Print out a formatted description of the contents of a knowledge base.

(← *self* SetContents *lst*) [Method of KB]

Make KB have new contents. Check types of elements.

(← *self* ThawKB *name*) [Method of KB]

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

(← *self* WriteBoot) [Method of KB]

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

(← *self* WriteEntityFile *changedEntities namedEntities assockbName*) [Method of KB]

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

(← *self* WriteFileLayer *kbName nameTable*) [Method of KB]

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

## 9.19 The Class Environment

Environment [Class]

IVs:

status [IV of Environment]

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

nameTable [IV of Environment]

nameTable for looking up UIDs and names.

outputKB [IV of Environment]

KB to which changes will be filed, and which specifies contents.

assocKB [IV of Environment]

Name of the KB associated with new objects created.

Methods:

(← *self* AssocKB *akb*) [Method of Environment]

Make *akb* be the assocKB of this KB.

(← *self* Cancel) [Method of Environment]

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

(← *self* ChangedKBs) [Method of Environment]

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

(← *self* `Cleanup` *KBNames noBootLayerFlg*) [Method of Environment]

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

(← *self* `ClearObjectMemory`) [Method of Environment]

Write out boot layer if needed and clear nameTable.

(← *self* `Close` *assocKBs*) [Method of Environment]

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

(← *self* `ConnectOutput` *KB*) [Method of Environment]

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

(← *self* `CopyObjects` *objList*) [Method of Environment]

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

(← *self* `DumpToKB` *kbName assocKBNames*) [Method of Environment]

???

(← *self* `Files` *fileLst*) [Method of Environment]

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

(← *self* `IsCurrent`) [Method of Environment]

Test if current.

(← *self* `MakeCurrent`) [Method of Environment]

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

(← *self* `MakeNotCurrent` *bitchIfNotCurrent*) [Method of Environment]

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

(← *self* `MapObjectNames` *mapFn assocKBs noUIDs*) [Method of Environment]

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

(← *self* `MarkDeleted` *objToBeDeleted*) [Method of Environment]

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

(← *self* `Open`) [Method of Environment]

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

(← *self* `WriteBoot`) [Method of Environment]

Make outputKB write it's boot file.

(← *self* WriteUpdate *kbName*) [Method of Environment]

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

## 9.20 The Class Layer

Layer [Class]

IVs:

file [IV of Layer]

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

kbName [IV of Layer]

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

position [IV of Layer]

Index on file where FileLayer is found.

assocKB [IV of Layer]

Name of KB with which this Layer is associated conceptually.

Methods:

(← *self* AddEntities *entityList*) [Method of Layer]

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

(← *self* Connect *nameTable*) [Method of Layer]

Open layer file and read in index.

(← *self* Files *fileLst*) [Method of Layer]

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

(← *self* MapObjectNames *mapFn noUIDs*) [Method of Layer]

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

## 9.21 The Class KBMeta

KBMeta [Class]

Methods:

(← *self* New *kbName envName newVersionFlg*) [Method of KBMeta]

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

(← *self* Old *kbName envName*) [Method of KBMeta]

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

(← *self* `ReadBoot`) [Method of KBMeta]

Read in index of existing KB given kbName.

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

Read in index of already existing KB.

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

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

## 9.22 The Class EnvironmentMeta

`EnvironmentMeta` [Class]

Methods:

(← *self* `Cleanup`) [Method of EnvironmentMeta]

Write updates for all open environments.

(← *self* `Close` *leaveKBattachedFlg*) [Method of EnvironmentMeta]

Close all the open environments.

(← *self* `OpenFiles`) [Method of EnvironmentMeta]

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

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.

## 10.1 Introduction

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

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

(1) Rules in Loops are organized into ordered sets of rules (called RuleSets) with 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 invocation of RuleSets can also be organized in terms of tasks, that can be executed, suspended, and restarted. Using task primitives it is convenient to specify many varieties of agenda-based control mechanisms.

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

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

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

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

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

## 10.2 Basic Concepts

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

The following sections outline some of the main concepts of rule-oriented programming. Loops provides a special language for rules because of their central role, and because special facilities can be associated with rules that are impractical for procedural programming languages. For example, Loops can save specialized audit trails of rule execution. Audit trails are important in knowledge systems that need to explain their conclusions in terms of the knowledge used in solving a problem. This capability is essential in the development of large knowledge-intensive systems, where a long and sustained 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.

## 10.3 Organizing a Rule-Oriented Program

In any programming paradigm, it is important to have an organizational scheme for composing large systems from smaller ones. Stated 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.

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

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

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

        IF load>1.0 THEN .ReduceLoad;

        IF ~pluggedInTo THEN .PlugIn;

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

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

{1}     THEN dealer.RequestService;

{1}     THEN manufacturer.Complain;

{1}     THEN $ConsumerBoard.Complain;

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

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

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

*Procedure-oriented Approach.* This approach is analogous to the use of subroutines in procedure-oriented programming. Programs are decomposed into RuleSets that call each other and return values when they are 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.

## 10.4 Control Structures for Selecting Rules

RuleSets in Loops consist of an ordered list of rules and a control structure. Together with the contents of the rules and the data, a RuleSet control structure determines which rules are executed. Execution is determined by the contents of rules in that the conditions of a rule must be 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 14 illustrates the use of a `Do1` control structure to specify three mutually exclusive actions.

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

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

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

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

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

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

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

`While1` [RuleSet Control Structure]

This is a cyclic version of `Do1`. If the while-condition is 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 page 93). The value of the RuleSet is the value of the last rule that was executed, or `NIL` if no rule was executed.

`WhileAll` [RuleSet Control Structure]

This is a cyclic version of `DoAll`. If the while-condition is 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 15 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;
While Cond: T;

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

{1!} IF loadSetting='Small THEN waterLimit←10;
{1!} IF loadSetting='Medium THEN waterLimit←13.5;
{1!} IF loadSetting='Large THEN waterLimit←17;
```

```
{1!} IF loadSetting='ExtraLarge THEN waterLimit←20;

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

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

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

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

   (* Stop when the water reaches its limit.)

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

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

## 10.5 One-Shot Rules

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

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

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

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

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

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

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

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

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

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

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

These rules are called "try-once" rules.

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

## 10.6 Task-Based Control for RuleSets

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

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

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

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

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

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

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

```
RepairTask5:

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

Figure 16. An example of a Task object. This Task could have been created for an invocation of the RuleSet in figure_17. The Task records the RuleSet, its data, and its execution status. The instance variable `ruleNumber` is used only for the control structures `DoNext` and `CycleNext` as described in the next section. The instance variable priority was created in response to the Task Vars declaration in the RuleSet.

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

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

(* Rules for washing machine repair.)

{1}   priority←300;
...
{1}   IF ~(replacementPart←motor.FindBrokenPart)
   THEN (STOP T 'Suspended 'NoDiagnosis);

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

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

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

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

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

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

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

   IF currentTask:reason='UnavailablePart
      substitutePart←expert.AskForSubstitutePart
   THEN currentTask:self:replacementPart←substitutePart
        (Start currentTask);

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

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

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

## 10.7 Control Structures for Generators

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

`DoNext` [RuleSet Control Structure]

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

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

`WhileNext` [RuleSet Control Structure]

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

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

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

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

## 10.8 Saving an Audit Trail of Rule Invocation

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

### 10.8.1 Motivations and Applications

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

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

*Belief Revision.* Another active research area is the development of systems that can "change their minds". This characteristic is critical for systems that must reason from incomplete or errorful information. Such systems get leverage from their ability to make assumptions, and then to recover from bad assumptions by 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.

### 10.8.2 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.

### 10.8.3 An Example of Using Audit Trails

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

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

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


   ...

   {(basis←'Fact cf←1)}
   IF buyer:familySize>2  machine:capacity<20
   THEN suitability←'Poor;

   {(basis←'Fact cf←.8)}
   reliability←(← $ConsumerReports GetFacts machine);

   {(basis←'Estimate cf←.4)}
   IF ~reliability THEN reliability←.5;
   ...
```

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

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

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


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

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

RuleSet in figure 19. In this example, each of the entries in the report has a `reason` and a `cc` (for cumulative certainty) property in addition to the value. The value of the `reason` properties are *audit records* created as a side 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 value in the `cf` variable of the audit record performs a computation to maintain a calculated cumulative certainty in the `reliability` variable of the evaluation report.

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

## 10.9 Comparison with other Rule Languages

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

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

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

### 10.9.1 The Rationale for Factoring Meta-Level Syntax

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

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

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

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

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

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

```
{(basis←'Fact cf←.8)}
```

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

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

In a large knowledge-based system, a substantial amount of control information must be 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, perspecuity is degraded. In the second option, the transparency of the system is degraded because the knowledge is hidden.

### 10.9.2 The Rationale for RuleSet Hierarchy

Some advocates of production systems have praised the 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 a user to explicitly program a stack of return locations in cases where contexts are invoked from more than one place. The decision to use an implicit calling-stack for RuleSet invocation in Loops is another example of the our desire to simplify the rules by factoring out control information.

### 10.9.3 The Rationale for RuleSet Control Structures

Production languages are sometimes described as having a *recognize-act cycle*, which 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 [Forgy81] 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 15.

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

```
Control Structure: Do1;

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

could be written alternatively as

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

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

However, the Do1 control structure admits a much more concise expression of mutually exclusive actions. In the example above, the Do1 control structure makes it possible to abbreviate the rule conditions to 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  a1  b1 c1 THEN d1 e1;
IF ~a1  b1 c1 THEN d2 e2;
IF ~a1 ~b1 c1 THEN d3 e3;
```

can be written as

```
Control Structure: Do1;

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

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

### 10.9.4 The Rationale for an Integrated Programming Environment

RuleSets in Loops are integrated with procedure-oriented, object-oriented, and data-oriented programming paradigms. In contrast to single-paradigm rule systems, this integration has two major 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 21 illustrates the installation of the RuleSet `SimulateWashingMachineRules` to carry out the `Simulate` method for instances of the class `WashingMachine`. The use of object-oriented paradigm is facilitated by special RuleSet syntax for sending messages to objects, and for manipulating the data in Loops objects. In addition, RuleSets, work spaces, and tasks are implemented as Loops objects.

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

Figure 21. Example of using a RuleSet as a method for object-oriented invocation. This 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` messages to invoke the RuleSet may be sent by any Loops program, including other RuleSets.

Using the data-oriented paradigm, RuleSets can be installed in active values so that they are triggered by side-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.

# 11 THE RULE LANGUAGE

## 11.1 Rule Forms

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 on page 68:

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

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

*LHS Syntax:* The clauses on the LHS of a rule are evaluated in order from left to right to determine whether the LHS is 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)
```

## 11.2 Kinds of Variables

Loops distinguishes the following kinds of variables:

*RuleSet arguments:* All RuleSets have the variable `self` as their workspace. References to `self` can often be elided in the RuleSet syntax. For example, the expression `self.Print` means to send a `Print` message to `self`. This expression can be shortened to `.Print` . Other arguments can be 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.

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

*Audit record variables:* Literals can also be used to refer to instance variables of audit records created by rules. These literals are used only in *meta-assignment* statements in the MD part of a rule. They are used to describe the information saved in audit records, which can be created as a side-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.

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

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

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

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

`self` [Variable]

The current work space.

`rs` [Variable]

The current RuleSet.

`task` [Variable]

The Task representing the current invocation of this RuleSet.

`caller` [Variable]

The RuleSet that invoked the current RuleSet, or `NIL` if invoked otherwise.

`ruleApplied` [Variable]

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

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

`ruleObject` [Variable]

Variable bound to the object representing the rule itself.

`ruleNumber` [Variable]

Variable bound to the sequence number of the rule in a RuleSet.

`ruleLabel` [Variable]

Variable bound to the label of a rule or `NIL`.

`reasons` [Variable]

Variable bound a list of audit records supporting the instance variables mentioned on the LHS of the rule. (Computed at run time.)

`auditObject` [Variable]

Variable bound to the object to which the reason record will be attached. (Computed at run time.)

`auditVarName` [Variable]

Variable bound to the name of the variable on which the reason will be attached as a property.

*Other Literals:* As described later, literals can also refer to Interlisp functions, Loops objects, and message selectors. They can also be used in strings and quoted constants.

The determination of the meaning of a literal is done at compile time using the declarations and syntax of RuleSets. The characters used in literals are limited to alphabetic characters and numbers. The 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.

## 11.3 Rule Forms

*Quoted Constants:* The quote sign is used to indicate constant literals:

```
a b=3 c='open d=f  e='(This is a quoted expression) -> ...
```

In this example, the LHS is 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.

## 11.4 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.

## 11.5 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;
```

## 11.6 Variables and Properties

When a single colon is used in a literal, it indicates access to an instance variable of an object. For example:

```
tile:type='BlueGreenCross command:type=Slide4 -> ... ;
```

In this example, access to the Loops object is indirect in that it is referenced through an instance variable of the work space. The left literal is the variable `tile`, and its value must be a Loops object when the rule is executed. The right literal `type` must be the name of an instance variable of that object. The compound literal `tile:type` refers to the value of the `type` instance variable of the object in the instance variable `tile`.

The colon notation can be combined with the dollar notation to access a variable in a named Loops object. For example,

```
$TopTile:type='BlueGreenCross ...
```

refers to the `type` variable of the object whose Loops name is `TopTile`.

A double colon notation is provided for accessing class variables. For example

```
truck::MaxGas<45 ::ValueAdded>600 -> ... ;
```

In this example, `MaxGas` is a class variable of the object bound to `truck`. `ValueAdded` is a class variable of `self`.

A colon-comma notation is provided for accessing property values of class and instance variables. For example

```
wire:,capacitance>5  wire:voltage:,support='simulation -> ...
```

In the 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.

## 11.7 Perspectives

* * * Not implemented yet in the rule language * * *

In many cases it is useful to organize information in terms of multiple points of view. For example, information about a man might be organized in terms of his role as a *father*, as an *employee*, and as a *traveler*. Each point of view, called a *perspective*, contains information for a different purpose. The perspectives are related to each other in the sense that they collectively provide information about the same object. As described in the Loops manual, Loops supports this organizational metaphor by providing special mixin classes called `perspectives` and `nodes`.

Loops perspectives can be accessed in the rule language by using a comma notation. In the following rule, the variable `washingMachine` is bound to an object with three perspectives: `commodity`, `electrical`, and `cleaning`. The rule accesses the `voltage` variable of the object that is the `electrical` perspective.

```
IF washingMachine,electrical:voltage<100 THEN ....
```

In this syntax, the term before the comma names a variable, and the term after the comma is the name of the perspective.

## 11.8 Computing Selectors and Variable Names

The short notations for instance variables, properties, perspectives, and unary messages all show the selector, variable, and perspective names *as they actually appear* in the object.

*object.selector*
*object:ivName*
*object::cvName*
*object:varname:,propName*
*object,perspName*

(← *object selector* $arg_1$ $arg_2$)

For example,

```
apple:flavor
```

refers to the `flavor` instance variable of the object bound to the variable `apple`. In Interlisp terminology, this implies implicit quoting of the name of the instance variable (`flavor`).

In some applications it is desired to be able to compute the names. For this, the Loops rule language provides analogous notations with an added exclamation sign. After the exclamation sign, the interpretation of the variable being evaluated starts over again. For example

```
apple:!\x
```

refers to the same thing as `apple:flavor` if the Interlisp variable `x` is bound to `flavor`. The fact that `x` is a Lisp variable is indicated by the backSlash. If `x` is an instance variable of `self` or a temporary variable, we could use the notation:

```
apple:!x
```

If `x` is a class variable of `self`, we could use the notation:

```
apple:!::x
```

All combinations are possible, including:

*object*.!*selector*
*object*.!*selector*
*object*.!::*selector*
*object*:!*ivName*
*object*::!*cvName*
*object*:!*varname*:,*propName*
*object*,!*perspName*

(←! *object selector* $arg_1$ $arg_2$)

## 11.9 Recursive Compound Literals

Multiple colons or periods can be used in a literal. For example:

```
a:b:c
```

means to (1) get the object that is the value of `a`, (2) get the object that is the value of the `b` instance variable of `a`, and 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.

## 11.10 Assignment Statements

An assignment statement using a left arrow can be used for setting all kinds of variables. For example,

```
x←a;
```

sets the value of the variable x to the value of a. The same notation works if x is a task variable, rule variable, class variable, temporary variable, or work space variable. The right side of an assignment statement can be an expression as in:

```
x←a*b + 17*(LOG d);
```

The assignment statement can also be used with the colon notation to set values of instance variables of objects. For example:

```
y:b←0 ;
```

In this example, first the object that is the value of yis computed, then the value of its instance variable b is set to 0.

*Properties and perspectives:* Assignment statements can also be used to set property values as in:

```
box:x:,origin←47    fact:,reason←currentSupport;
```

or variables of perspectives as in:.

```
washingMachine,electrical:voltage←110;
```

*Nesting:* Assignment statements can be nested as in

```
a←b←c:d←3;
```

This statement sets the values of a, b, and the d instance variable of c to 3. The value of an assignment statement itself is the new assigned value.

## 11.11 Meta-Assignment Statements

Meta-assignment statements are assignment statements used for specifying rule descriptions and audit trails. These statements appear in the MD part of rules.

*Audit Trails:* The default interpretation of meta-assignment statements for undeclared variables is as audit trail specifications. Each meta-assignment statement specifies information to be saved in audit records when a rule is applied. In the following example from figure 19, the audit record must have variables named basis and cf:

```
{(basis←'Fact cf←1)}
IF buyer:familySize>2  machine:capacity<20
THEN suitability←'Poor;
```

In this example, the RHS of the rule assigns the value of the work space instance variable suitability to 'Poor if the conditions of the rule are 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.

*Rule Descriptions:* Meta-assignment statements can also be used to set variables in the objects that represent individual rules. This interpretation of meta-assignment statements is indicated when the assignment variable of the meta-assignment statement has been declared to be a rule variable. For example, if the variable `cf` in the previous example was declared to be a rule variable, then the meta-assignment statement would set the `cf` instance variable of the rule object to `.5` at compilation time, instead of saving a `cf` in every audit record for every rule application at execution time. The value on the right hand side of the meta-assignment statement for a rule variable must be known at compile time.

## 11.12 Push and Pop Statements

A compact notation is provided for pushing and popping values from lists. To push a new value onto a list, the notation ←+ is used:

```
myList←+newItem;
```

```
focus:goals←+newGoal;
```

To pop an item from a list, the ←- notation is used:

```
item←-myList;
```

```
nextGoal←-focus:goals;
```

As with the assignment operator, the push and pop notation works for all kinds of variables and properties. They can be used in conjunction with infix operator << for membership testing.

## 11.13 Invoking RuleSets

One of the ways to cause RuleSets to be executed is to invoke them from rules. This is used on the LHS of rules to express predicates in terms of RuleSets, and on the RHS of rules to express actions in terms of RuleSets. A short double-dot syntax for this is provided that invokes a RuleSet on a work space:

```
Rs1..ws1
```

In this example, the RuleSet bound to the variable `Rs1` is invoked with the value of the variable `ws1` as its work space. The value of the invocation expression is the value returned by the RuleSet. The double-dot syntax can be combined with the dollar notation to invoke a RuleSet by its Loops name, as in

```
$MyRules..ws1
```

which invokes the RuleSet object that has the Loops name `MyRules`.

This form of RuleSet invocation is like subroutine calling, in that it creates an implicit stack of arguments and return addresses. This feature can be used as a mechanism for *meta-control* of RuleSets as in:

```
IF breaker:status='Open
THEN source←$OverLoadRules..washingMachine;

IF source='NotFound
THEN $ShortCircuitRules..washingMachine;
```

In this example, two "meta-rules" are used to control the invocation of specialized RuleSets for diagnosing overloads or short circuits.

## 11.14 Transfer Calls

An important optimization in many recursive programs is the elimination of tail recursion. For example, suppose that the RuleSet A calls B, B calls C, and C calls A recursively. If the 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.

## 11.15 Task Operations

Tasks in the Loops rule language represent the invocation of RuleSets. They provide a mechanism for specifying and controlling processes in terms of tasks that can be created, started, suspended, and restarted. They also provide a handle for specifying concurrent processing.

A Task records the work space of a RuleSet (`ws`), the value returned (`value`), and two special variables called the `status` and `reason`. A Task can also have RuleSet-specific instance variables called task variables for saving process information.

*Creating Tasks:* A Task is represented as a Loops object and can be created and associated with a work space as follows:

Task6←(← $Task New *RuleSet workSpace*)

The *workSpace* argument is optional. Specialized versions of `Task` will eventually be available, such as `RemoteTask`. Information about a Task is stored in its instance variables, and can be accessed like other Loops variables:

```
Task6:status
Task6:reason
Task6:ws
Task6:value
```

*Starting Tasks:* The primary operations on Tasks are starting them and waiting for them to finish execution. These operations have been designed to work when Loops is extended for concurrent processing. The operations for starting tasks are as follows:

(Start1 *taskList*) [Function]
(StartAll *taskList*) [Function]
(StartAll *taskList*) [Function]

Each of the start operations takes an argument *taskList* which is either a Task object, or a list of Task objects. A Task cannot be started if it is already running, as indicated by its `status` variable. `Start1` iterates through its *taskList* and starts the first Task that is not already running. The value of *Start1* is the Task that was started. `StartAll` starts all of the tasks, and does not return control until all of the tasks have been started. `StartTogether` is like `StartAll` except that none of the tasks are started until all of them are ready. The synchronization aspect of `StartTogether` is important for avoiding Task deadlock situations in programs that share Tasks as resources. (It avoids the difficulties associated with partial allocation of Tasks when a complete set of Tasks is needed.)

*Waiting for Tasks:* The following operations are provided for waiting for Tasks:

(Wait1 *taskList*) [Function]
(WaitAll *taskList*) [Function]

`Wait1` iterates through its *taskList* and returns as its value the first Task that is not running. `WaitAll` returns when all of its Tasks have finished running The value returned by the RuleSet that ran in a Task can be obtained from the Task object, as in:

```
task6:value.
```

*Running Tasks:* In many cases, the specification of Task control can be simplified by using a *run* operation that combines the start and wait operations. The run operations are as follows:

(Run1 *taskList*) [Function]
(RunAll *taskList*) [Function]
(RunTogether *taskList*) [Function]

`Run1` goes through its arguments left to right and selects the first Task that is not running. It starts that Task and then waits for it to complete. The value of `Run1` is the Task that was executed. `RunAll` starts all of the Tasks running and then waits for them all to complete. `RunTogether` waits for all of the Tasks to become available, runs them all, and then waits for them all to complete.

## 11.16 Stop Statements

At invocation, the `status` in the Task is set to `Running`. If a RuleSet ends normally, the `status` in the Task is set to `Done` and the `reason` saved in the RuleStep is `Success`. Other terminations can be specified in a Stop statement as follows:

`(Stop` *value status reason*`)` [RuleSet Statement]

*value* is the value to be returned by the RuleSet. *status* characterizes the termination of the Task, and *reason* is a symbolic reason for the status. Typical examples of the use of Stop are:

```
(Stop value 'Aborted reason)
(Stop value 'Suspended reason)
```

where `Aborted` means that the RuleSet has failed, and `Suspended` means that the RuleSet has stopped but may be re-invoked. Particular applications will probably develop standardized notations for status and reason. Values for these can be Interlisp atoms or Loops objects. The arguments *status* and *reason* are optional in a `Stop` statement.

## 12 USING RULES IN LOOPS

The Loops rules language is supported by an integrated programming environment for creating, editing, compiling, and debugging RuleSets. This section describes how to use that environment.

### 12.1 Creating RuleSets

RuleSets are named Loops objects and are created by sending the class `RuleSet` a `New` message as follows:

```
(← $RuleSet New)
```

After entering this form, the user will be prompted for a Loops name as

`RuleSet name:` *RuleSetName*

Afterwards, the RuleSet can be referenced using Loops dollar sign notation as usual. It is also possible to include the RuleSet name in the `New` message as follows:

`(← $RuleSet New NIL` *RuleSetName*`)`

### 12.2 Editing RuleSets

A RuleSet is created empty of rules. The RuleSet editor is used to enter and modify rules. The editor can be invoked with an `EditRules` message (or `ER` shorthand message) as follows:

`(←` *RuleSet* `EditRules)`
`(←` *RuleSet* `ER)`

If a RuleSet is installed as a method of a class, it can be edited conveniently by selecting the `EM` option from a browser containing the class. Alternatively, the `EM` function or `EditMethod` message can be used:

`(←` *ClassName* `EditMethod` *selector*`)` [Message]

`(EM` *ClassName selector*`)` [Function]

Both approaches to editing retrieve the source of the RuleSet and put the user into the TTYIN editor, treating the rule source as text.

Initially, the source is a template for RuleSets as follows:

```
RuleSet Name:          RuleSetName;
WorkSpace Class:       ClassName;
Control Structure:       doAll;
While Condition: ;
Audit Class:          StandardAuditRecord;
Rule Class:              Rule;
```

```
Task Class:         ;
Meta Assignments:       ;
Temporary Vars:;
Lisp Vars:          ;
Debug Vars:       ;
Compiler Options:   ;


   (* Rules for whatever.  Comment goes here.)
```

Figure 22. Initial template for a RuleSet. The rules are entered after the comment at the bottom. The declarations at the beginning are filled in as needed and superfluous declarations can be discarded.

The user can then edit this template to enter rules and set the declarations at the beginning. In the current version of the rule editor, most of these declarations are left out. If the user chooses the `EditAllDecls` option in the RuleSet editor menu, the declarations and default values will be printed in full.

The template is only a guide. Declarations that are not needed can be deleted. For example, if there are no temporary variables for this RuleSet, the `Temporary Vars` declaration can be deleted. If the control structure is not one of the while control structures, then the `While Condition` declaration can be deleted. If the compiler option A is not chosen, then the `Audit Class` declaration can be deleted.

When the user leaves the editor, the RuleSet is compiled automatically into a LISP function.

If a syntax error is detected during compilation, an error message is printed and the user is given another opportunity to edit the RuleSet.

## 12.3 Copying RuleSets

Sometimes it is convenient to create new RuleSets by editing a copy of an existing RuleSet. For this purpose, the method `CopyRules` is provided as follows:

(← *oldRuleSet* `CopyRules` *newRuleSetName*) [Message]

This creates a new RuleSet by some of the information from the pespectives of the old RuleSet. It also updates the source text of the new RuleSet to contain the new name.

## 12.4 Saving RuleSets on LISP Files

RuleSets can be saved on LISP 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.

## 12.5 Printing RuleSets

To print a RuleSet without editing it, one can send a `PPRules` or `PPR` message as follows:

(← *RuleSet* `PPRules`) [Message]
(← *RuleSet* `PPR`) [Message]

A convenient way to make hardcopy listings of RuleSets is to use the function `ListRuleSets`. The files will be printed on the `DEFAULTPRINTINGHOST` as is standard in Interlisp-D. `ListRuleSets` can be given three kinds of arguments as follows:

(`ListRuleSets` *RuleSetName*)
(`ListRuleSets` *ListOfRuleSetNames*)
(`ListRuleSets` *ClassName*)
(`ListRuleSets` *FileName*)

In the *ClassName* case, all of the RuleSets that have been installed as methods of the class will be printed. In the last case, all of the RuleSets stored in the file will be printed.

## 12.6 Running RuleSets from Loops

RuleSets can be invoked from Loops using any of the usual protocols.

*Procedure-oriented Protocol:* The way to invoke a RuleSet from Loops is to use the `RunRS` function:

(`RunRS` *RuleSet* *workSpace* $arg_2 \cdots arg_N$) [Function]

*workSpace* is the Loops object to be used as the work space. This is "procedural" in the sense that the RuleSet is invoked by its name. *RuleSet* can be either a RuleSet object or its name.

*Object-oriented Protocol:* When RuleSets are installed as methods in Loops classes, they can be invoked in the usual way by sending a message to an instance of the class. For example, if `WashingMachine` is a class with a RuleSet installed for its `Simulate` method, the RuleSet is invoked as follows:

```
(← washingMachineInstance Simulate)
```

*Data-oriented Protocol:* When RuleSets are installed in active values, they are invoked by side-effect as a result of accessing the variable on which they are installed.

## 12.7 Installing RuleSets as Methods

RuleSets can also be used as methods for classes. This is done by installing automatically-generated invocation functions that invoke the RuleSets. For example:

```
[DEFCLASS WashingMachine
   (MetaClass Class doc (* comment) ...)
```