Reference Manual and Tutorial for the LispController (lisp-controller) Class

Version 1.0 April 2010

Paul Krueger, Ph.D.

Introduction

Displaying Lisp objects using Cocoa user interface views and functionality can be a big challenge. It can require an understanding of many Objective-C classes and methods. Even Objective-C developers face a similar challenge. To make life easier for them, Cocoa provides "controller" classes which make the process easier. For example, an NSArrayController hides much of the complexity of displaying an NSArray object in an NSTableView. It is only necessary for the developer to include an NSArray controller in their interface design within Interface Builder. They then configure it to do things like displaying elements from the NSArray in appropriate table cells. They can also configure an NSArrayController to modify the NSArray by adding or deleting elements of some specified class; typically in response to the pushing of some user-defined button.

Other controllers such as the NSTreeController can navigate hierarchical collections of NSArrays in views such as an NSOutlineView. This view allows the user to expand a displayed item to see its children and perhaps expand further if the children have children of their own.

All of this is very nice for Objective-C programmers, but can be a pain for Lisp programmers. We don't want to keep things in NSArrays and most of our classes are not derived from Objective-C classes. So I set out to create helper functionality that is similar to that which Objective-C programmers can get from various existing controller classes, but which is geared entirely to Lisp programming. I call this class LispController (the Objective-C name) or lisp-controller (the Lisp name). Generally I will try to use the former name when discussing it within the context of Interface Builder (IB) and the latter name when discussing Lisp interactions.

The lisp-controller class and methods discussed in this document provide something of an analog to an Objective-C controller class, but make it easy for lisp programmers to display rather arbitrary types of lisp objects. It is possible to configure a lisp-controller to add objects to Lisp collections in response to user interface actions in much the same manner that various Objective-C controllers do. A lisp-controller can also be used to *bind* user interface text fields to ordinary instance slots or other Lisp objects using Key-Value Coding (KVC) just as Objective-C developers do. For example, this can be used to display a single Lisp instance using multiple text fields that each bind to a different slot within the instance.

To make the LispController class easy to use within Interface Builder (IB), a LispControllerPlugin module has been provided. This permits the developer to select, configure, and link a LispController within IB just as Objective-C controller classes can be selected, configured, and linked. But make no mistake, the runtime functionality of the lisp-controller is all implemented in lisp, not Objective-C. Only the functionality needed specifically within IB is implemented in Objective-C. The configuration of the lisp-controller class and views that it interfaces with will look very familiar to Lisp developers. Lisp symbols and forms are used as necessary to specify things like class names, accessor functions, initialization forms, etc. Package qualifiers are permitted in names as needed. The examples shown at the end of this document will make all of this much more clear. There is quite a bit of flexible functionality available here, but it is quite easy to do simple interfaces. For example, you can hand a list to the controller (or let it create one for you if you want), specify the type of objects you want in that list, and away you go.

This document is organized as follows:

1. Plugin Build and Install

Build and Install the LispControllerPlugin module

In the finder, double-click on .../ccl/contrib/krueger/InterfaceProjects/Lisp IB Plugins/LispControllerPlugin.xcodeproj to open up the Xcode application. Hopefully this will be the only time that you will ever need to use Xcode. If you haven't used it before, don't worry too much; I'll walk you through the few things you'll need to do to make this work. When you open it up, the upper left part of the window should look similar to Figure 1 below.

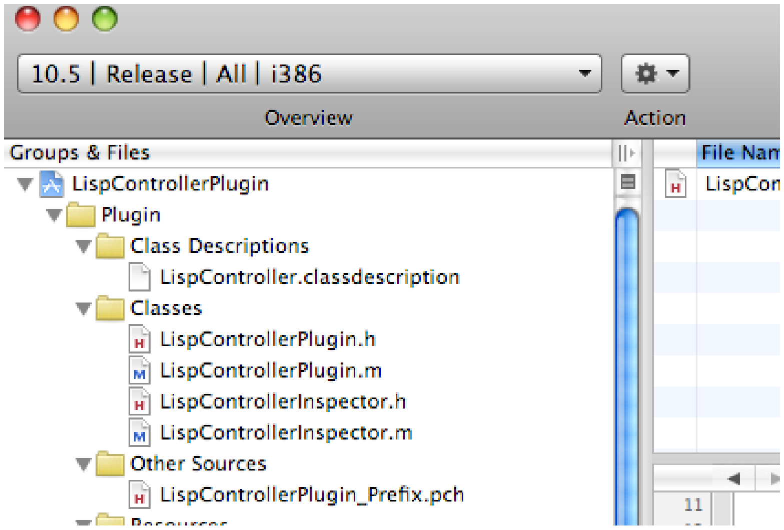


Figure 1

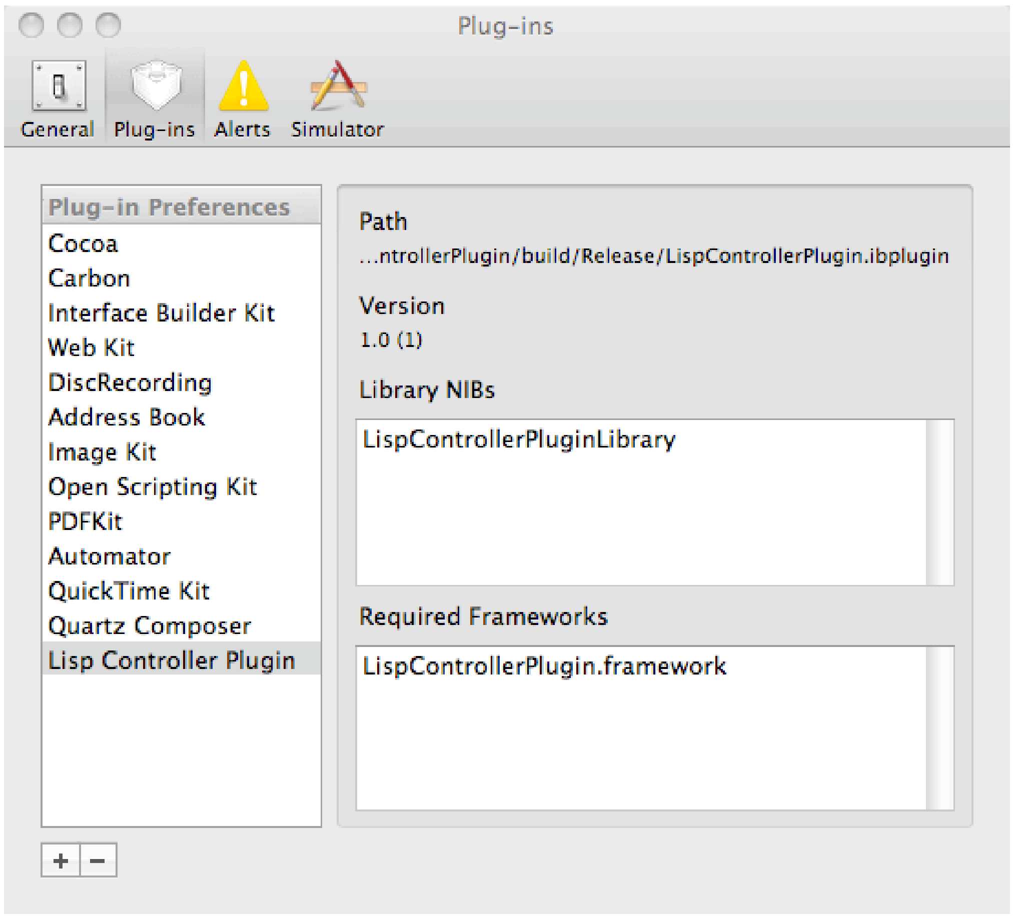
In your version of the pull down menu labeled "10.5 l Release l All l i386" in Figure 1 you should select choices that are appropriate for your environment. Just make sure to select the "release" option. Then in the "Build" menu select "Clean all targets" to make sure that everything starts from scratch. Then select the "Build" option from the "Build" menu to re-build the framework and IB plugin. If everything goes ok you should see no errors for the build. If so, you can quit Xcode.

Install the LispControllerPlugin Framework

Next we have to install the newly created framework in a place where IB can find it. I often duplicate the framework and move the duplicate rather than the original and then rename it back to the original name, but you can always build a new one as well. Using either the finder or the Terminal application move
.../ccl/contrib/krueger/InterfaceProjects/Lisp IB Plugins/LispControllerPlugin/build/Release/LispControllerPlugin.framework
inside /Library/Frameworks.

Install the LispControllerPlugin in Interface Builder

The next step is to make IB aware of this plugin. Start IB and select preferences from the "Interface Builder" menu. In the window that pops up select "Plug-ins" at the top and you will see a window that looks much like Figure 2, except that yours will not show the Lisp Controller Plugin just yet.



Path
...ntrollerPlugin/build/Release/LispControllerPlugin.ibplugin

Version
1.0 (1)

Library NIBs
LispControllerPluginLibrary

Required Frameworks
LispControllerPlugin.framework

Figure 2

There are two ways that you can now install the plugin. Either click the "+" button and navigate to
.../ccl/contrib/krueger/InterfaceProjects/Lisp IB Plugins/LispControllerPlugin/build/Release/LispControllerPlugin.ibplugin
and select it. Or locate that file in the finder and just drag it into the Plug-in Preferences pane in the window.
In either case, it should install and your window should look just like Figure 8.2. From this point on, IB will remember about this plugin and you can directly use the Lisp controller object that we will now discuss.

At this point the reader can take one of two paths. Sections 2, 3, and 4 provide a reference for all lisp-controller functionality and section 5 provides examples. Some may prefer to see examples first and some may prefer to see what is possible before seeing examples of use. In either case it may be necessary to refer back and forth to really understand it all.

2. Configuring the lisp-controller in IB

We are now ready to actually use a lisp-controller within some project. This section will provide a description of each field that can be configured within the IB inspector for a lisp-controller object. Several examples of these choices are provided in section 5. As we discuss each option, we will reference an appropriate example that demonstrates that choice.

2.1 lisp-controller Access Functions

All accessor functions to lisp-controller objects that are intended for external use are interned and exported from the package "lisp-controller" (nickname "lc"). They are defined in lisp-controller.lisp as follows:

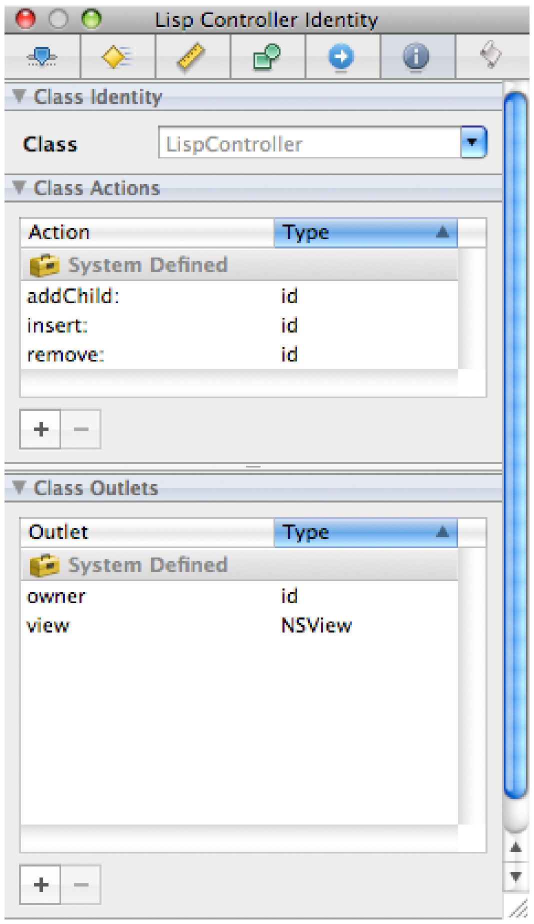
```
(defpackage :lisp-controller
  (:nicknames :lc)
  (:use :ccl :common-lisp :iu)
  (:export
   added-func
   add-child-func
   children-func
   content-class
   count-func
   delete-func
   edited-func
   gen-root
   lisp-controller
   lisp-controller-changed
   objects
   reader-func
   removed-func
   root
   root-type
   select-func
   writer-func))
```

These functions can be used to configure a lisp-controller at runtime rather than using the preferred method of configuration through Interface Builder. Runtime modification has not been heavily tested, so if you encounter bugs, please let me know (plkrueger <AT> comcast.net).

The use of the root function has been tested and should work as expected. Most of the other functions do the same things that configuration in IB does and will be discussed within those contexts below. However the function "lisp-controller-changed" is exclusively used at runtime. This function informs the lisp-controller that you have changed something within the root object and would like that change reflected in the displayed table. Changes that are made as a consequence of user actions such as adding a new child, modifying a column value, deleting a row, etc. do not require any outside call to lisp-controller-changed. Even if you specify override functions of your own as described later, you do not have to call lisp-controller-changed. This is strictly reserved for informing the lisp-controller about asynchronous modifications to the root structure being displayed.

2.2 Adding a lisp-controller to an Interface Design in IB

If you have installed the lisp controller plugin into IB you can now select a lisp-controller object from the "Lisp Controller Plugin" folder in the Library window and drag it to any document window that you create. If you examine it using the identity inspector you should see something like Figure 3 below.



Action	Type
System Defined	
addChild:	id
insert:	id
remove:	id

Outlet	Type
System Defined	
owner	id
view	NSView

Figure 3: Lisp Controller Identity Inspector

The first thing to note here is the "view" outlet. This should be linked to an NSTableView or NSOutlineView that is part of your interface design. Simply ctrl-click and drag from the lisp-controller object in the document window to the view in your design window and select the view field (which should be the only option).

The other outlet is the "owner" outlet. This is only used as an argument to user-specified override functions. This might be needed if, for example, several windows of the same type were open and all using the same override functions. Typically a developer might link this back to the "File's Owner" object in the nib file and then at runtime the override function could determine exactly which window was being operated on by using this argument. If no override functions are specified, then this outlet need never be linked to anything. See the section below that defines override functions for more details.

There are three actions associated with a lisp-controller that can be triggered via controls that you add to your interface. The "addChild:" action will cause the lisp controller to add a new child to the item currently selected in the table. This is only appropriate when the view is an NSOutlineView and you are trying to add another object at the next level.

The "insert:" action will cause the lisp controller to add a new child to the root object. The root object is the lisp object that is represented by the whole table. So a call to "insert:" will add a new row to the table (or a new top-level row in the case of an NSOutlineView).

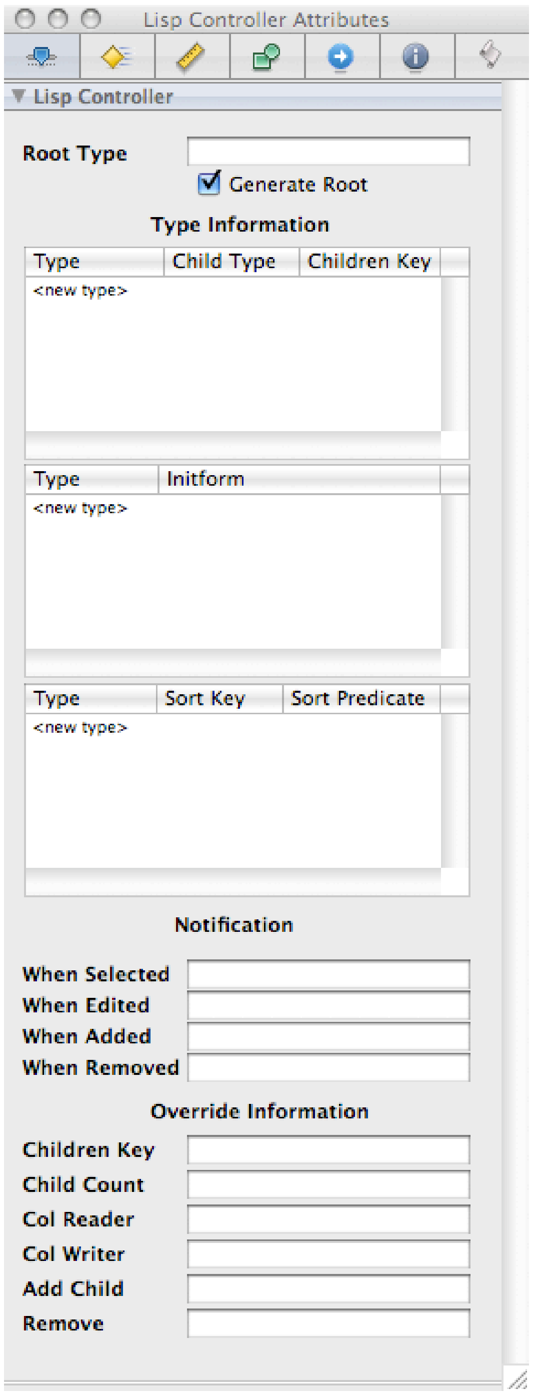
The "remove:" action will cause the lisp-controller to remove the object (row) currently selected from its parent. If it is a top-level row, then the object represented by the row will be removed from the root object. The details of this vary somewhat depending on the type of the root object, but in all cases, external references to the root object remain valid. For example, if the root object is a list to which you have an external pointer and you delete the first item from the list, then the external list pointer will now point to the new first item. The developer should be aware of this as it could cause unwanted side-effects (like having the deleted first item be garbage collected) if the only external reference to the deleted first object is via a pointer to the list that was set as the root object.

The advantage of the behavior is that an outside lisp object can cache a pointer to the root object of the lisp controller and be assured that it will remain valid through any additions, deletions, or reordering of the list. There is one unfortunate exception to all this. If the root object is a list with a single element and that element is deleted, then the result is of course nil.

Subsequent additions of children will do what you might expect, but the root object is now a different list than it was previously and an outside user would have to once again retrieve the root object from the lisp controller in order to get the latest value. Note however, that if any of the various notification methods (discussed below) are specified, then the root object is provided as part of the notification, so if a list is used as the root object and it is possible that the user will delete the last item in it, then the new root can be obtained as an argument via a notification. If the "When Removed" notification is used, then the object removed is also available and can be saved elsewhere if desired. Subsequently if the "When Added" notification is called, then the new root list is provided. Any other type of persistent root object (e.g. vector, hash-table, CLOS instance) will not have this problem.

2.3 Configuring Data Type, Access, and Initialization

The IB inspector pane for a lisp-controller object is shown in Figure 4 below:



Root Type

[✓ Generate Root]

Type Information

Type	Child Type	Children Key
<new type>		

Type	Initform
<new type>	

Type	Sort Key	Sort Predicate
<new type>		

Notification

When Selected

When Edited

When Added

When Removed

Override Information

Children Key

Child Count

Col Reader

Col Writer

Add Child

Remove

Figure 4: Lisp Controller Attributes Inspector

We will next discuss each of the fields in the Attributes Inspector window.

Root Information

Root Type

The first configurable field is "Root Type". The root object is the object being used to access all data that is displayed within the table. This can be an arbitrary Lisp object, but there are a few types that the lisp-controller handles automatically for you. Specifically, lists, vectors, and hash-tables can be easily used as root objects without much further configuration needed. Examples <example> demonstrate this. But not much more effort is required to use almost any type of lisp object as a root object. The root object type can be any type acceptable to Lisp as shown in examples 2 and 3. Package specifiers are acceptable, just as they are in Lisp (all examples). Capitalization is ignored, just as it is within Lisp.

At runtime the root type specified is used to validate that an object set as the root object is of the correct type. This ensures that you don't set up accessors for table columns assuming one type of object and then try to use them for another. It is also used to find the correct initform if you choose to have the lisp-controller generate a root object for you.

Each *child* of the root object is treated as a row that is to be displayed within an NSTableView or or a top-level row to be displayed within an NSOutlineView. There are a few default ways of locating the children of a root object or you can provide a "children key" as specified later. If the root object is a list or vector, then by default its children are simply the elements of that sequence; i.e. each row displays one element of the root sequence. If the root object is a hash-table, then each row represents a key/value pair from that hash-table. These pairs are encapsulated in an ht-entry object described later.

Generate Root

If you wish the lisp-controller to automatically generate the root object for you, check the "Generate Root" box (see examples 1 and 2). The root object will be generated using either an initform that you define for the root type or a default that is appropriate for the root type (as described in the discussion of initforms below). If you do not check this box, then nothing will be displayed until the root object has been set in the lisp-controller at runtime via a call that is of the form:

```
(setf (lc:root <lisp-controller>) <your-root-object>)
```

The object specified should be a subtype of the type specified in the "root type" field. If it is not, an error will be generated at the time the root is set.

Type Information

There are three tables that let you specify information about the types of objects to be displayed. Each of these tables automatically keeps an extra entry for the next line that you might add. Similarly, if you delete all items from a line, it will be removed from the table and disappear. So it is not necessary to ever explicitly add or remove a line from one of these tables.

The Type Information table allows you to specify how to find the children of a given type and what type to expect for those children. Each child of the root object is displayed within a single row of the table. When the lisp-controller is used in conjunction with an NSTableView the only entry that may be needed in the Type Information table is one to tell the lisp-controller how to find the children of the root object. When used in conjunction with an NSOutlineView additional entries may be needed to tell the lisp-controller how to find more deeply nested levels of children. Of course if all levels represent objects of the same type then a single entry to specify how to find children suffices (see example 2).

Type

The first column of the Type Information table is where you put the type name. This could be the same as the root type or any other type for which you want to specify how to retrieve children or create new ones.

Child Type

The second column of the Type Information table specifies a type that may be used by the lisp-controller to create a new child for an object of the type specified in column 1. If you never want the lisp-controller to create child objects for you, then it is not necessary to specify a child-type for any parent type. No type checking is done to assure that the children returned from an instance of some type (via its children key) are of the same as the child type specified.

Children Key

In the third column of the Type Information table you can specify a key that will be used to retrieve children from an object of the type specified in column 1. A children key can be specified for the root type or any other more deeply nested type. The latter are only used to display indented structures within NSOutlineViews. The children key should specify a function of one argument. It should always return either a list or a vector or a hash-table. If the result is a list or a vector, it must contain all of the argument's children objects. If it is a hash-table, then a list of ht-entry objects is created which represents the set of children. Each ht-entry object encapsulates a single key/value pair. When applied to a root object those children represent the rows of an NSTableView (no example yet) or the top-level rows of an NSOutlineView (example 3). When applied to some other more deeply nested object they represent the children of that more deeply indented object (example 3).

There are a few default children-keys for common types of parent objects. The children object for lists or vectors is just the sequence itself. The children object for a hash-table is a generated list of ht-entry objects representing its key/value pairs. To find the children of an ht-entry object, the value of the key/value pair is treated as if it is the parent object and then normal processing is done to find an appropriate child-key for that type of parent. So if you know that the value of some hash-table entry is of a particular type and want the table to expand that type, you can add a Type Table entry for that type that specifies an appropriate children-function. Note that if a hash-table value is a list or vector the default children-key for those types will apply and those values will be expanded for NSOutlineViews. See Section 5.3: Controller Test 3 below for an example of how to block that behavior if it is not what you want.

Initform

The next table allows you to specify initforms to use when creating new instances of the specified type. Each row of the table lets you specify some type and an initform for creating objects of that type. This is used to create a new root object if the type is the root type and the "generate root" box was checked. Immediate children of the root type (i.e. rows in the table) may be created in response to button presses that invoke the lisp-controller's insert: method. More embedded types can also be created in response to buttons which invoke the lisp-controller's addChild: method.

Sort Key

The sort information table allows you to specify how the rows are to be sorted (either top-level rows or more deeply nested rows of children within NSOutlineViews). The Sort Key should be a function specifier (symbol that names a function or #'function-name). It is applied to every element of the sequence being sorted and the rows are ordered according to the results of applying the sort predicate to the results of the sort key application, just as with an ordinary lisp sort.

However there are some subtle differences from an ordinary sort. All sorts done by the lisp-controller are done "in place". That is, the result of doing the sort results in a reordering of the sequence, but any reference to the beginning of the sequence prior to sorting remains valid. For example, if the sequence is a list, then the first element of the sorted list will physically be the same cons cell that was first prior to the sort, but the car and cdr of that cons cell may both be different after the sort.

If the children-key for an object returns a list that is, for example, the value of some slot in an object, then the elements of that list may be modified as a result of the sort done by the lisp-controller. The slot value itself remains valid as previously described. But suppose that the user cached a pointer to the end of that list for some reason. In the general case that pointer would no longer point to the last element of the list after the sort, although it would point to *some* element of the list.

Sort Predicate

The sort predicate should name an ordering function of two arguments that returns either nil if the order relation does not hold for the two arguments or non-nil if it does.

Notification Functions

If desired, there are four types of notifications that the lisp-controller can provide to the user. As with other function specifiers, these should either name a function or be of the form #'function-name with package specifiers permitted.

When Selected

The NSTableView and NSOutlineView classes permit a user to select either a row or a column. In general they permit the selection of multiple rows or columns, but the lisp-controller does not support reporting that at this time. Note that neither of these Objective-C view classes reports the selection of a single cell, only a row or column. Don't ask me why ...

The function specified should take six arguments:
1) owner object: This is whatever is linked to the "owner" outlet in the lisp-controller
2) controller object: This is the lisp-controller object itself
3) root object: This is the root object displayed in this table
4) row number: The row number if a row was selected or -1 if a column was selected
5) column number: the column number if a row was selected or -1 if a row was selected
6) object selected: The row object if a row was selected or the column title if a column was selected

The owner object might be needed if there are multiple windows of the same type currently open. This can help to disambiguate where the action occurred. The controller object is provided to facilitate any interaction with it that might be desired by the notification function. The root object is also provided as a convenience. The row and column numbers are straightforward, but when the table is an NSOutlineView the object represented by any particular row number may vary as other rows are expanded or collapsed. For such views it is probably better to rely on the sixth argument (the object selected) to determine what was selected.

When Edited

If you have permitted editing of a column and also specified a function to be called "When Edited", then after cell editing is complete (whether or not a change was actually made) that function will be called with eight arguments. The first six are identical to those described above, with the exception that both the row number and column number are guaranteed to be non-negative numbers. In addition, the following two arguments are passed:
7) old value of the edited cell: the lisp object that was previously displayed in the cell
8) new value of the edited cell: the lisp object that results from transforming the value entered by the user

When Added

New objects can be added as children to the root object. For NSTableViews that means adding a new row to the table. For NSOutlineViews that means adding a new top-level row. In addition, it is possible to add children to more deeply embedded objects that are displayed in NSOutlineViews. After that has been done, any user-specified When Added notification function is called with five arguments:
1) owner object: This is whatever is linked to the "owner" outlet in the lisp-controller
2) controller object: This is the lisp-controller object itself
3) root object: This is the root object displayed in this table
4) parent object: This is the object to which a new child was added
5) child object: This is the newly created object that was added as a child to the parent

If the view is an NSTableView, then the root object and the parent object arguments will always be the same. If a list is used as the root object and this is the first object added to it, then the root argument may be saved for future reference, avoiding the need to retrieve it before the window is closed. It will remain valid as long as something remains in it. If all objects of a root list are removed, it will become nil. A subsequent addition will result in a new list.

When Removed

When a child is removed from some parent object, this notification function will be called. Its arguments are identical to those of the When Added function except that the last argument represents the child that was removed rather than the child that was added.

Override Functions

Override functions permit the developer to circumvent most of the default lisp-controller functionality and provide alternative functionality. In all cases, override functions are called in preference to their default lisp-controller counterparts.

Children Key

If provided, the Children Key is used to retrieve all children, both for the root object and for subordinate objects if the view is an NSOutline view. It is called with the following three arguments:
1) owner object: This is whatever is linked to the "owner" outlet in the lisp-controller
2) controller object: This is the lisp-controller object itself
3) parent object: This is the object for which children are needed

The result of this call should be a list, vector, or hash-table that "contains" the children of the parent.

Child Count

This function is called to return the count of rows to be displayed in an NSTableViews. Note that it is not called for NSOutlineViews. Instead, this value is determined retrieving the children and counting them. This function takes three arguments:
1) owner object: This is whatever is linked to the "owner" outlet in the lisp-controller
2) controller object: This is the lisp-controller object itself
3) root object: This is the root object displayed in this table

The result of this function must be an integer.

Col Reader

This function is called to provide the lisp value for a specified row and column. It is called with two arguments:
1) row-object: This is one of the children of the root object for NSTableViews or some displayed row object for NSOutlineViews.
2) column identifier: this is a Lisp object derived by doing a read-from-string of the Identifier specified for the table column

within Interface Builder. It could be a simple number or any other type of Lisp object that can be read from a string.

It should return a Lisp value that is then converted as needed to an NSObject for display in the table.

Col Writer

This function is called to set a new value for a specified row and column. It is called with three arguments:
1) new value: This is the new Lisp object which should be made the value for the specified column and row.
2) row-object: This is one of the children of the root object for NSTableViews or some displayed row object for NSOutlineViews.
3) column identifier: this is a Lisp object derived by doing a read-from-string of the Identifier specified for the table column within Interface Builder. It could be a simple number or any other type of Lisp object that can be read from a string.

The value returned from this function is ignored.

Add Child

This function is called when a new child must be added to some object. For NSTableViews this will always be the root object, but for NSOutlineViews it could be any displayed object. It is called with a single argument:
1) parent object: This is the root object for NSTableViews or some displayed row object for NSOutlineViews.

The Add Child function must return two values. The first value is an object that represents the new set of children. It should be a list, vector, or hash-table. The second value must be an object representing the single new child that was created. It can be of any type. The second value is only used as an argument to the When Added notification function, so if you do not use that notification or don't care that the value of the child argument is nil, then it is not necessary to return the second value.

Remove

This function is called to delete a child from some parent. For NSTableViews the parent will always be the root object, but for NSOutlineViews it could be any displayed object. It is called with two arguments:
1) parent object: This is the root object for NSTableViews or some displayed row object for NSOutlineViews.
2) child object: This is the child that should be removed from the parent

The Remove function should return an object that represents the new set of children. It should be a list, vector, or hash-table.

2.4 lisp-controller actions

In Figure 3 you probably noticed three LispController actions that can be triggered by user interface operations of some sort. Typically this would be done by adding a button to the interface and configuring it to trigger one of these actions.

insert:

When triggered, this action results in a new child being added to the lisp-controller's root object.

addChild:

When triggered, this action results in a new child being added to the lisp object represented in the currently selected row.

remove:

When triggered, this action results in removing the lisp object represented in the currently selected row from its parent.

2.4 Enabling buttons using LispController bindings

There are three LispController fields that can be used to enable or disable buttons to ensure that they are only active when appropriate. Those fields are:

- canRemove
  - indicates it is appropriate to remove an object
- canInsert
  - indicates it is appropriate to add a child to the root object
- canAddChild
  - indicates it is appropriate to add a child to the selected object

These are used by binding a button's enabled field to the LispController's canRemove, canInsert, or canAddChild path respectively. See example 1 for more detailed instruction.

2.5 Key-Value Coding (KVC) through the LispController

KVC is a mechanism that Cocoa provides which allows object values to be *bound* together. When the value of one of the bound objects is modified, then the other is changed to reflect that new value. This ability was used in several of the projects in the tutorial to bind interface values to the values of foreign-slots in classes that we defined to be sub-classes of NSObject. The lisp-controller class provides a way to bind interface elements through one or more standard Lisp objects or structures and to a Lisp slot value. Data conversion is automatically provided. When the Lisp slot is changed in some manner, it will be necessary to call the Lisp method will-change-value-for just prior to making the change and did-change-value-for just after making it. These are direct analogs to the Objective-C methods #/willChangeValueFor: and #/didChangeValueFor:. The good news for Lisp developers is that it is easy to create slot accessors that include these calls so that no other code need ever be aware of them.

All bindings to Lisp slots must be made through a lisp-controller. Within IB, the interface element to be bound must be selected and the bindings pane of the inspector window should be opened. Then a "Model Key Path" must be specified, just as it would be if this was being bound to an Objective-C object. A Model Key Path consists of a series of dot separated path elements. For example: "root.slotA.mySlotB" or "selection._pkg_slot".

The initial element of that path must be either "root" or "selection". This determines whether the initial object in the path is the Lisp object that has been made the root object or the Lisp object that is currently selected. From that point on the path elements specify Lisp accessors.

Unfortunately arbitrary strings are not permitted as paths, so we are constrained by the syntax allowed. This means that we must accept the same name conversion that is needed for Class Names and Action method names elsewhere within IB with some additional conventions added to support package specification. Recall that names are translated from Objective-C to Lisp by inserting a hyphen wherever a capital letter or number appears and then making everything upper-case. In addition, we add the convention that if the path element starts with _<some-string>_, then <some-string> is assumed to be a package name. With these conventions in mind the examples above would be translated as follows:

    root.slotA.mySlotB would effectively become (MY-SLOT-B (SLOT-A (ROOT <controller>)))
    selection._pkg_slot would effectively become (PKG::SLOT (SELECTION <controller>)).

Note that the combined forms shown above are NOT actually created and evaluated. To support notification when intermediate path elements are changed the accessors are applied one at a time through intermediate proxy objects that the lisp-controller creates.

A package's full name or nickname can be used for any path element. The form used is cached and used when a reverse translation is needed (perhaps to tell KVC that the slot value was changed so that some interface element can retrieve it for display). Different forms of a package's name can be used for different path elements, but if it is necessary to bind to the same element within different bindings, then the same form should be used for all of them. Failure to do this may result in updates to Lisp not being detected by Objective-C interface objects.

Elements of a path can be either Lisp or Objective-C objects, but Objective-C objects must be KVC-compliant for the following path elements. Once an Objective-C object is reached in a path, then necessarily all following path elements will also reference Objective-C objects and each must be compliant in the same way that they would have to be for normal Objective-C binding paths. This includes any Objective-C subclasses defined within Lisp. Suppose that you have such an object and wished to bind to an interface object to one of its Lisp slots. To do this you would have to define the necessary Objective-C methods to make your class compliant for whatever name used in the binding. Suppose that the slot was named my-slot. You could, for example, define Objective-C methods #/mySlot and #/setMySlot that did the necessary translation for the contents of my-slot. This is just what we did in some of the projects discussed in the tutorial (See discussion in Project 6).

Failure to make an Objective-C object in the path KVC-compliant for the next path element will result in an Objective-C runtime exception that will be displayed in the AltConsole window. That will look something like the following:

```
> Error: Objective-C runtime exception:
>        Error setting value for key path root.bnd_input1 of object <LispController: 0x1410d8b0>
(from bound object <NSTextField: 0x11f3e360>(null)): [<BindingWindowOwner 0x1410b2e0>
setValue:forUndefinedKey:]: this class is not key value coding-compliant for the key bnd_input1.
> While executing: (:INTERNAL GUI::|-[LispApplication sendEvent:]|), in process Initial(0).
> Type :POP to abort, :R for a list of available restarts.
> Type :? for other options.
```

When Objective-C slots are modified in a compliant way, then the object that they are bound to is notified that a change was made. That compliance can be achieved by using appropriately named Objective-C methods or by calling the Objective-C function #/willChangeValueFor: just prior to making the slot change and calling #/didChangeValueFor just after making the change. The lisp-controller implementation provides two functions that are analogs of these that can be used by Lisp developers to assure that when Lisp slots are modified a KVC notification is sent to any Objective-C object that might have been bound to it. As you might expect, those corresponding Lisp functions are named #'will-change-value-for and #'did-change-value-for. These are declared in ns-binding-utils.lisp and required by lisp-controller.lisp. A convenient way to make sure that these functions are always called for a slot that you know will be used as a binding target is to do something like the following:

```
(require :ns-binding-utils)

(defclass my-object ()
  ((my-slot :reader my-slot)))

(defmethod (setf my-slot) (new-value (self my-object))
  ;; we set up the slot writer to note a change in value to make it KVC compliant
  (will-change-value-for-key self 'my-slot)
  (setf (slot-value self 'my-slot) new-value)
  (did-change-value-for-key self 'my-slot))
```

Now when any other Lisp code does something like:

```
      (setf (my-slot <my-object instance>) value)
```

the appropriate methods will be called. The Lisp object is now KVC-compliant for my-slot. Note that the Lisp symbol 'my-slot was used in the will-change... and did-change... calls. This was done rather than requiring the use of the Objective-C name "mySlot" or "_pkgName_mySlot" that would have been used as the binding element specified in IB. The Lisp functions will translate as needed. This saves the developer from having to know exactly how the slot was referenced from Objective-C.

See the example in *Section 5.4: Binding Test* which demonstrates the use of bindings to Lisp objects.

3. Data Conversion

In the previous section you learned how to configure a lisp-controller to find and/or generate appropriate lisp objects for each row in the table you are using. In the next few sections you will learn how to display exactly what you want within each column of a table for each of those rows. Before discussing how to configure a column to display the desired lisp object, it helps to understand how the conversion back and forth will be done. The lisp-controller will automatically convert lisp values to appropriate Objective-C instances for display. If editing has been permitted for a column, it will also convert Objective-C instances into appropriate lisp objects. The functions for doing this are contained primarily in the source file "ip:Utilities;ns-object-utils.lisp". (Note that if you haven't set up the ip logical directory in your lisp, see InterfaceBuilderWithCCLTutorial2.0.pdf for information about how to do that. Basically it is a reference to the Interface Projects directory where all of the work described here resides.)

Conversion from Lisp objects to NSObjects: lisp-to-ns-object

When converting Lisp objects to appropriate NSObjects for display, the lisp-controller takes some hints both from the lisp objects themselves and from the way that that formatters have been attached to the columns in the table. If you are not familiar with how to use Cocoa formatter objects within IB, then you may want to look at their use in previous projects described in the tutorial referenced in the last paragraph. Basically there are a few kinds of formatters that enforce the look of things like dates and numbers. They may also define the type of NSObject that is used to pass data to the application. These can be useful indicators of what sort of lisp object will be displayed in that column and how it should be converted.

First, if the object being displayed is itself an instance of a subclass of NSObject, then it is passed straight through to the table for display and no other conversion is done.

If a date formatter was used for the column that will receive the lisp value, then the lisp-controller will assume that the lisp value represents a date (as for example the result of executing (get-universal-time) in lisp). It will convert appropriately.

If a column formatter was used that specifies the use of NSDecimal objects to pass data back and forth and the lisp object is an integer, then the lisp-controller assumes that a format defined in "ip:Utilities;decimal.lisp" is being used. This format is more thoroughly described for Project 6 in InterfaceBuilderWithCCLTutorial2.0.pdf. It is primarily used to represent things like monetary amounts using integers rather than float values to avoid rounding and truncation difficulties.

Other numeric types are converted as required.

Finally, any other type of lisp object (e.g. symbols, class instances, etc.) are simply converted to an NSString identical to the way such objects would be displayed in a Lisp listener window (i.e. their printed form) with the exception that a string will be shown without the "" around it.

Conversion from NSObjects to Lisp objects: ns-to-lisp-object

Conversion from NSObjects to Lisp objects occurs when a user edits some value in a table. In this case the lisp-controller also has the advantage of knowing what type of Lisp object was the source of the previously displayed value. As much as possible, the lisp-controller will try to maintain the same type of object as was there previously.

If the previous object was itself an NSObject, then the value is left unconverted and passed through as the resulting Lisp object.

If the object is an NSDecimal, then the lisp-controller will either convert it to a float or to a Lisp integer that represents a scaled decimal value depending on the type of the previous value. Let's suppose that you desire to use float values in Lisp, but want to require the use of NSDecimal values to pass data back and forth for some reason. To make sure that the lisp controller doesn't misconstrue your intent, you should assure that all the original values displayed are actually float values and not fixnums. Otherwise it may interpret your fixnum as a scaled integer and you will not get the desired result. If you ARE using scaled integers, then you should assure that the number of decimals used to do the scaling is the same as the number specified for the "Minimum fraction digits" in the number formatter for that column in IB because that is what the lisp-controller will assume.

Other numeric NS classes are converted appropriately.

NSDate objects are converted to a corresponding Lisp date.

If the previous object was a string, then the NSObject will be a string and will just be converted to a Lisp string.

Anything else is converted by reading from the string to construct a corresponding lisp object. Any error in reading will result in a nil value being returned.

4. Configuring Column Accessors

Each column has an associated text field called its "identity". Objective-C developers can use this to specify some identifier string that can be used at runtime to decide what to put in the column. That ends up being something like

```
     if (colId == @"column 1") {
          return column1Accessor(rowObject);
     } else if (colId == @"column 2") { ...
```

We could have used this in much the same way within Lisp, but because Lisp permits dynamic evaluation of forms, we can shortcut the process and directly place accessor forms or function names in the column identifiers and apply those to a row to acquire the values that will be displayed in corresponding columns. The lisp-controller class defines a number of different ways to define these accessors that we will discuss below.

To set a column identity in IB you must click on the table (which will result in the selection of the surrounding scroll view), click again to select the NSTableView or NSOutlineView object, click again over one of the columns to select the NSTableColumn object that you want to edit. Then in the attributes inspector modify the "Identifier" field to be as described below.

Indexical accessors

If the lisp object that is being displayed in a table row is a sequence of some kind (e.g. a list or a vector) and we want to display some particular element of that sequence in a column, then we can simply put the index in the column identifier and the lisp-controller will use it to access that element within the row sequence to retrieve the value to display in the column (example 1).

Functional accessors

If the value desired for a particular column can be accessed by applying a function to the row-object, then you can directly specify the name of that function as the column identifier. That can be a simple symbol that names a function or a function specifier (e.g. #'function-name). Package identifiers can be included to specify a function that is defined in a package that is not used by the "common-lisp-user" package (which is where the name that you specify will be evaluated). For example you could set the column identifier to something like: my-package::my-function or equivalently #'my-package::my-function (examples 1, 2, and 3). The function specified should take a single argument that will be bound to the value of the lisp object being displayed within that row. The table requests data for each column and row and the lisp-controller will funcall the function you specified with the row object as the function's argument.

Accessor Forms

If the value desired for a column can be specified as a simple form, then you you may choose to enter it directly as the identifier for that column. There is a relatively limited amount of space in that field, so forms that are not fairly short should probably be turned into functions and the function name put into the column identifier instead. When to do that is up to the developer. It is likely that somewhere within the form you will want to use the lisp object being displayed within that row. To do that use the keyword :row wherever you would wish to use the row object. If you like, you can think of the :row keyword as being bound to the value of the row object. Of course you cannot bind to keywords so this isn't what really happens. As an example you could specify something like:

```
     (second (some-slot :row))
```

if the row object is an object with an accessor named "some-slot" that returns a list and you wanted to display the second item of that list in the column.

Note that you could also just use the single keyword :row as the column identifier to indicate that the row-object itself should be displayed in this column. This is the default if no other mechanism is used to define what should be displayed within a column.

Hash-table Accessors

If the root object is a hash-table, then each row object is effectively a key/value pair from within the hash-table. In some columns you may want to directly display either the key or the value or you may want to apply some function to either the key or the value. To do this you will use the keywords :key and :value in much the same way that the keyword :row was discussed above. For example, if you want to display the key from a key/value pair in a column, you would simply put :key into the column identifier for that column. If the value was known to be an object that has a slot named "my-slot" and you wanted to display its value in a column, then you might set the column identifier for that column to

```
(my-slot :value).
```

This idiom of applying a function to the value is common enough that if you simply specify a function name, e.g.

```
my-slot
```

and the row-object is an ht-entry object representing a key/value pair from a hash-table, then the lisp-controller will assume that you want to apply the named function to the value. This will have the same effect as the preceding form. Similarly, if you specify a number as the identifier of a column and the row object is a key/value pair, then the value is assumed to be a sequence and the number will be used as an index into that sequence to retrieve the value to be displayed. Example 3 illustrates the use of hash-table roots.

Setf accessors

If a column permits editing of its values, then the lisp controller will attempt to construct an appropriate function to "setf" the location of the original value with the new value input by the user for any cell in that column. If the accessor is an indexical, i, then the setf form will be equivalent to (setf (elt :row i) new-value). If the accessor is a functional accessor, f, then the setf form will be equivalent to (setf (f :row) new-value). If the accessor is a form, F, then the setf form will be equivalent to (setf F new-value). In all cases, the lisp-controller will first assure that the resulting setf form is valid before trying to use it. This mechanism seems to work well for a broad class of accessors, but if some other behavior is desired, then it is probably necessary to use the "Col Reader" and/or "Col Writer" override functions to achieve the desired behavior.

5. Example Code

Introduction

All example code for the lisp-controller will be found in the ...ccl/contrib/krueger/InterfaceProjects directory. Each example is in its own subdirectory titled "Controller Test N" where N is the example number.

The instructions that follow assume some familiarity with IB procedures. If you're not already familiar with how to use IB, you may want to look at early projects described in InterfaceBuilderWithCCLTutorial2.0.pdf which describes the use of Interface Builder (in conjunction with Lisp of course) in much more detail. Each example assumes that you have mastered the techniques described in previous examples.

5.1 Controller Test 1: Auto generated list displayed in an NSTableView

This example shows how to configure a lisp-controller to generate new objects that are displayed in an NSTableView. It demonstrates the use of indices, function names, and forms as column accessors. It shows how to configure interface buttons to add and remove objects from the table. It shows how to use number formatters for table columns both to control how data is displayed and to provide hints that tell the lisp-controller how data conversion should be done. Finally, it contains example notification functions for all types of notification. This example doesn't do anything too useful, but it nicely illustrates a number of different lisp-controller features.

The Lisp code and NIB file for this example are in the directory "ip:Controller Test 1". To follow along you can double-click lc-test1.nib so that it opens up in IB. Or you can start with a new Cocoa window nib and follow along with the instructions below to see how everything gets set up. If you start up IB without giving it a file or if you have IB started and select "New" from the file menu, IB will ask you to select a template to be used as a basis for your new nib. From the Cocoa category select "window".

Select the File's Owner object and in the Object Identity browser pane set the class field to "LispControllerTest". This will be the name of the class that we will create that will load this nib at runtime. Add an outlet field for this class call "lispCtrl". When you get done with that, the Identity pane should look much like Figure 5 below.

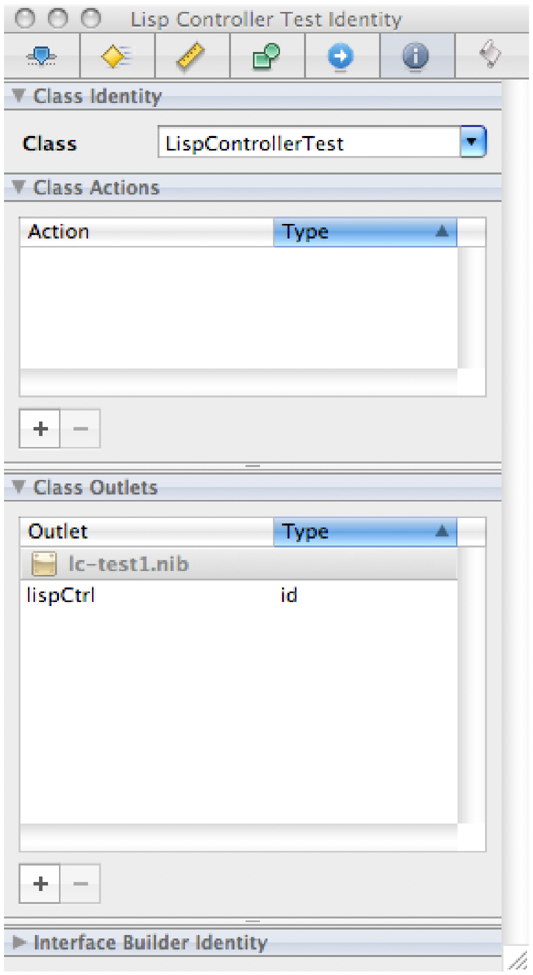


Figure 5: LispControllerTest Identity

Next locate and drag a Table View object from the Library to your new window. Configure it to have four columns. Label the columns and window as you see fit. Add a couple of buttons below the table and label them "+" and "-" or whatever you like to indicate that they result in adding and deleting a row, respectively. When completed, this window may look something like Figure 6 below.

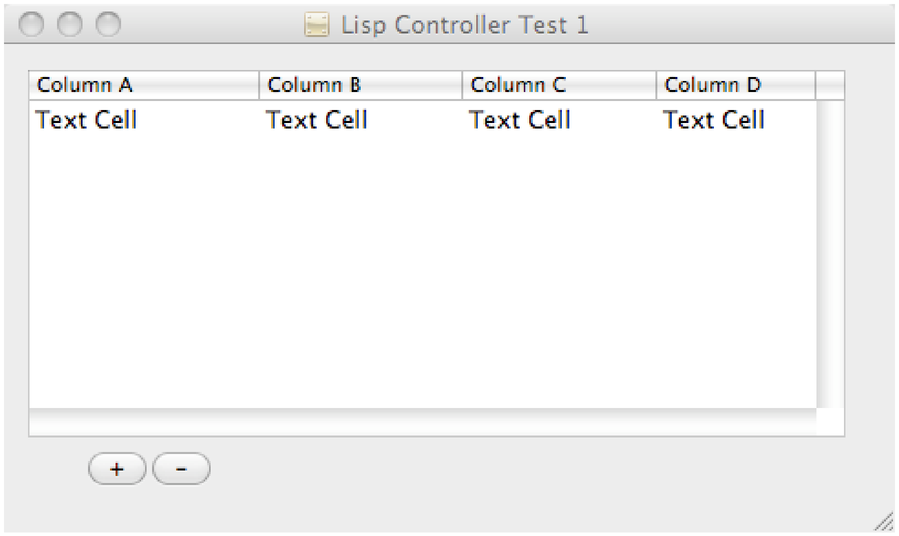


Figure 6: Lisp Controller Test 1 window

Now we will add a lisp-controller to the mix. From the "Lisp Controller Plugin" folder in the Library window, drag a Lisp Controller object to the document window (where the File's Owner and other objects are already shown). Control-click on the lisp-controller and a pop-up window similar to the one in Figure 7 will be shown.

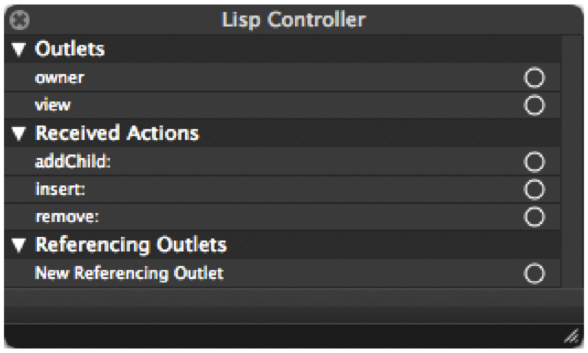
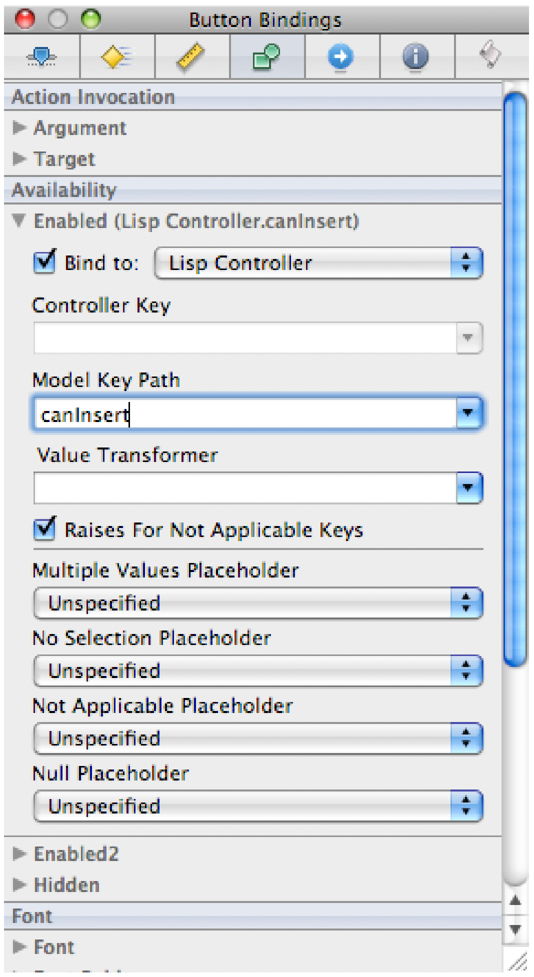


Figure 7: Lisp Controller bindings

Click in the circle to the right of the "owner" outlet and drag that to the File's Owner object. Similarly connect the view outlet to the Table View in your display window. Control-click on the Table View and connect both the delegate and dataSource outlets to the Lisp Controller object. This will cause the Table View to ask the Lisp Controller for data when it needs it and to tell the LIsp Controller about events that occur that might be of interest (such as changing what is selected).

Control-click on the "+" button (or your equivalent) and drag to the Lisp Controller object. Link it to the "insert:" action. Similarly control-click and drag from the "-" button to the Lisp Controller and link it to the "remove:" action. As you might expect, this will cause the buttons to send the action specified to the Lisp Controller when they are pressed. We want to make sure that those buttons will only be enabled when it is appropriate. For example, we would want to disable the "-" button if nothing was currently selected or there wasn't anything in the table. To do that we'll bind the enabled state of the button to a slot value in our Lisp Controller. If you've worked through the tutorial you're already familiar with bindings and how they work. If not, well, trust me and follow the directions and everything will work out. Click on the "+" button and select the bindings pane in the inspector window. Click on the small arrow by "Enabled" to show the fields for that section. *Before* you click in the checkbox before "Bind to:", select "Lisp Controller" from the pull-down. Then click in the check-box. If you don't do it in this order, then IB may add whatever object is the default in the pull-down to your document window. If that happens to you, just delete the object. In the "Model Key Path" field put "canInsert". This is the name of a foreign slot in the Lisp Controller that at runtime will be contain a logical value that says whether it is currently appropriate to enable insertion into the table. The bindings pane for the "+" button should look like Figure 8 below.


Button Bindings

Action Invocation
- Argument
- Target

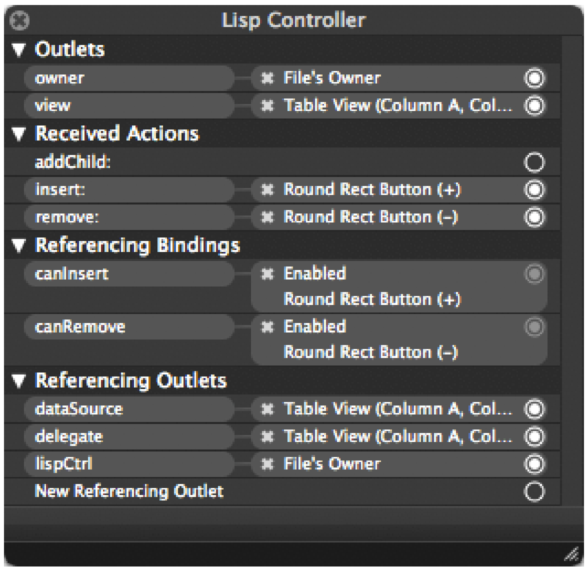
Availability
- Enabled (Lisp Controller.canInsert)
  - [✓] Bind to: Lisp Controller
  - Controller Key:
  - Model Key Path: canInsert
  - Value Transformer:
  - [✓] Raises For Not Applicable Keys
  - Multiple Values Placeholder: Unspecified
  - No Selection Placeholder: Unspecified
  - Not Applicable Placeholder: Unspecified
  - Null Placeholder: Unspecified
- Enabled2
- Hidden

Font
- Font

Figure 8: "+" button bindings

Similarly bind the enable state of the "-" button to the Lisp Controller Model Key Path "canRemove".

After you've done all that, control-click on the Lisp Controller object and the pop-up should look much like Figure 9 below.


Lisp Controller

Outlets
- owner — File's Owner
- view — Table View (Column A, Col...

Received Actions
- addChild:
- insert: — Round Rect Button (+)
- remove: — Round Rect Button (-)

Referencing Bindings
- canInsert — Enabled, Round Rect Button (+)
- canRemove — Enabled, Round Rect Button (-)

Referencing Outlets
- dataSource — Table View (Column A, Col...
- delegate — Table View (Column A, Col...
- lispCtrl — File's Owner
- New Referencing Outlet

Figure 9: Lisp Controller after linking

Next we'll configure the Lisp Controller for our application. Click on the Lisp Controller and select the attributes pane in the inspector window. Initially this will look like Figure 4 above. For this example we'll display a list of lists. Each sublist will represent one row in the table and will have four elements, one for each column. We'll make the first column be a date and we'll sort the rows in the table by the numeric value in the second column. The third and fourth elements in the sublist will be arbitrary lisp objects. Simple!

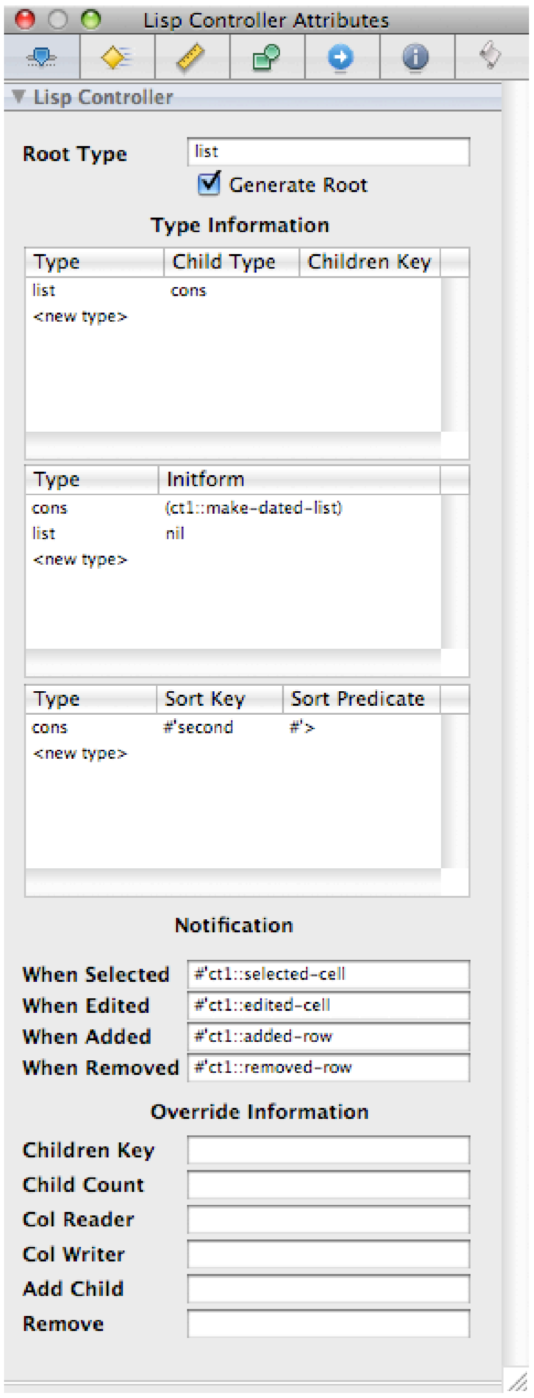
The first thing to do is to set the Root Type field to "list" and check the "Generate Root" box. We'll see in a bit how this happens, but effectively what we've told the Lisp Controller is that to create a new root object it needs to create a "list". Next, in the Type Information table specify that the child type of the list type is "cons". This takes a little advantage of the Lisp type hierarchy to avoid having to define our own types in order to tell the Lisp Controller what to do. We've now told the Lisp Controller that in order to insert a new object into the root (of type List) that it needs to create a new object of type "cons".

The initform table tells the Lisp Controller exactly how to create new objects of specified types. For the "cons" type enter the initform: (ctl::make-dated-list). When the Lisp Controller needs a new object of type "cons" it will evaluate this form. For the type "list" enter the initform: nil. This will be evaluated to create a new object of the type "list". This will be done to generate a new root, just as we requested when we checked the "Generate Root" box earlier.

We decided that we wanted to sort the rows based on the descending value of the second element in each row. To make that happen we make an entry into the sort table. For the cons type we enter a Sort Key of #'second and a Sort Predicate of #'>.

Next we configure the Lisp Controller to call specified functions to notify us when various events occur. We'll look more closely at those functions when we get to the Lisp code, but for now enter #'ct1::selected-cell as the "When Selected" function, #"ct1::edited-cell as the "When Edited" function, #'ct1::added-row as the "When Added" function, and #'ct1::removed-row as the "When Removed" function.

Once we're done with this configuration, the Lisp Controller attributes pane should look like Figure 10.



Lisp Controller Attributes

Lisp Controller

Root Type: list

[✓] Generate Root

Type Information

Type	Child Type	Children Key
list	cons	
<new type>		

Type	Initform
cons	(ct1::make-dated-list)
list	nil
<new type>	

Type	Sort Key	Sort Predicate
cons	#'second	#'>
<new type>		

Notification

When Selected	#'ct1::selected-cell
When Edited	#'ct1::edited-cell
When Added	#'ct1::added-row
When Removed	#'ct1::removed-row

Override Information

Children Key	
Child Count	
Col Reader	
Col Writer	
Add Child	
Remove	

Figure 10: Completed Lisp Controller attributes

The final thing to do in IB is to configure each column of the table so that it displays the values that we want in an appropriate manner. We'll attach a date formatter for the first column and a number formatter for the second. Let's start with the date formatter for column 1. Drag a Date Formatter object from the Library window into your document window. Next we need to connect that formatter to the column. Attaching a formatter to a column can be a bit tricky because it must actually be connected to the Text Field Cell object for the column. Recall that in IB you get to more deeply embedded objects within a complex arrangement of cooperating objects like a Table View by clicking repeatedly. Each click selecting a more deeply embedded object. The trick to getting to the Text Field Cell is (assuming nothing in the Table is already selected) to click 4 consecutive times over the "Text Cell" displayed in the window. Each click selects a more deeply embedded object until you get to the Text Cell. Now control-click and drag from the Text Cell in the first column to the Date Formatter object. You can then configure the Date Formatter to display dates in whatever form you wish: long, short, or something in between.

Similarly, add a Number Formatter to your document window and connect it to the Text Cell for the second column. Why do we do this? Recall that we decided to sort on this value and we have also left all columns editable. If a user decided to enter a non-numeric value into a value for the second column in some row, then our sorting would try to compare it using the #'< predicate that we specified, resulting in a run-time error. We could write our own predicate that did type checking and did something appropriate with anything that might be there, but in this case it is trivially easy to just enforce the number constraint so that we don't have to worry about it.

Ok, so we have now constrained how things look a bit, but we still need to configure each column so that the Lisp Controller knows how to retrieve the value that will be displayed in each column from the list that represents each row. For this example we will do something fairly simple, namely just use the corresponding list element for each column. But we could just as well use any arbitrary function on that list for each column. Select the Table Column object for column 1 (click three times over the column if nothing is already selected). If you have not already done so, you can change the column's title here. In the "Identifier" field type the number 0. This is an indexical accessor that tells the Lisp Controller that if the row is represented by a sequence it should retrieve the value for this column from element 0 of that sequence. Similarly, set the Identifier for the second Table Column to the number 1. Just to be different and to demonstrate that other forms of accessor work, set the Identifier for the third Table Column to #'third. And demonstrating that a form also works, set the Identifier field for the fourth Table Column to (nth 3 :row). At runtime the keyword :row is effectively replaced by the actual row object (in this case a list). Arbitrary forms and functions are permitted. And the object that represents a row could be an arbitrary object as well.

We are now done with IB. If you have created your own NIB rather than use mine, save it in some convenient location and make sure to save it as a NIB rather than as an XIB. See the tutorial for more information about this difference.

The lisp code for this example is "ip;Controller Test 1;controller-test1.lisp" and is also shown immediately below:

```
;; controller-test1.lisp

;; Test window that displays lisp lists using an NSTableView

(eval-when (:compile-toplevel :load-toplevel :execute)
  (require :lisp-controller)
  (require :ns-string-utils)
  (require :nslog-utils)
  (require :date))

(defpackage :controller-test1
  (:nicknames :ct1)
  (:use :ccl :common-lisp :iu :lc)
  (:export test-controller get-root))

(in-package :ct1)

(defclass lisp-controller-test (ns:ns-window-controller)
   ((lisp-ctrl :foreign-type :id :accessor lisp-ctrl))
  (:metaclass ns:+ns-object))

(defmethod get-root ((self lisp-controller-test))
  (when (lisp-ctrl self)
    (root (lisp-ctrl self))))

(objc:defmethod (#/initWithNibPath: :id)
                ((self lisp-controller-test) (nib-path :id))
  (let* ((init-self (#/initWithWindowNibPath:owner: self nib-path self)))
    init-self))

(defun make-dated-list ()
  (list (now) 0 (random 20) (random 30)))

(defun selected-cell (window controller root row-num col-num obj)
  (declare (ignore window controller root))
  (cond ((and (minusp row-num) (minusp col-num))
         (ns-log "Nothing selected"))
        ((minusp row-num)
         (ns-log (format nil "Selected column ~s with title ~s" col-num obj)))
        ((minusp col-num)
         (ns-log (format nil "Selected row ~s: ~s" row-num obj)))
        (t
         (ns-log (format nil "Selected ~s in row ~s,  col ~s" obj row-num col-num)))))
```

```
(defun edited-cell (window controller root row-num col-num obj old-val new-val)
  (declare (ignore window controller root))
  (ns-log (format nil "Changed ~s in row ~s, col ~s: ~s to ~s"
                  old-val row-num col-num obj new-val)))

(defun added-row (window controller root parent new-row)
  (declare (ignore window controller root))
  (ns-log (format nil "Added ~s to ~s" new-row parent)))

(defun removed-row (window controller root parent old-row)
  (declare (ignore window controller root))
  (ns-log (format nil "Removed row ~s from ~s " old-row parent)))

(defun test-controller ()
  (let* ((nib-name (lisp-to-temp-nsstring
                    (namestring (truename "ip:Controller Test 1;lc-test1.nib"))))
         (wc (make-instance 'lisp-controller-test
               :with-nib-path nib-name)))
    (#/window wc)
    wc))

(provide :controller-test1)
```

As with all examples, this one is put into its own package so you can safely experiment with it without worrying about interfering with your own code. First we define the lisp-controller-test class, which you will recall was specified as the File's Owner object for our NIB. If you're now thinking that the names are different, then go back to the longer tutorial to learn about name conversions between Objective-C and Lisp. A lisp-controller-test object is an NSWindowController subclass so it knows how to manage windows and load NIB files. It has a single slot that is linked to the lisp-controller object that is created when the NIB is loaded.

The get-root method retrieves the root object from that attached lisp-controller. Since we let the lisp-controller generate the root, we may want to find out what was created at some later point.

The #/initWithNibPath: method will look familiar to anyone who has worked through other tutorials. It is there simply to facilitate loading NIB files from arbitrary locations without having to include the NIB file in the application bundle where NSWindowControllers like to find them.

The make-dated-list function was specified in the initform for the cons type when we configured the Lisp Controller in IB. It creates a list of four elements. The function "now" is defined in "ip:Utilities;date.lisp" and just returns the current time/date in the normal Lisp internal format. Note that we don't have to worry about doing any sort of conversion for display by Objective-C view objects because that is done by the lisp-controller for us.

The next four functions are the notification functions that we specified for the Lisp Controller in IB. These functions only log an appropriate message to the console log. Use the Console application to see these when the application is running. Perhaps what is significant here is understanding the arguments provided by the lisp-controller. For a complete discussion of those parameters see the *Notification Functions* section in part 2 of this document.

Finally there is a test-controller function that can be called from the REPL to get things started.

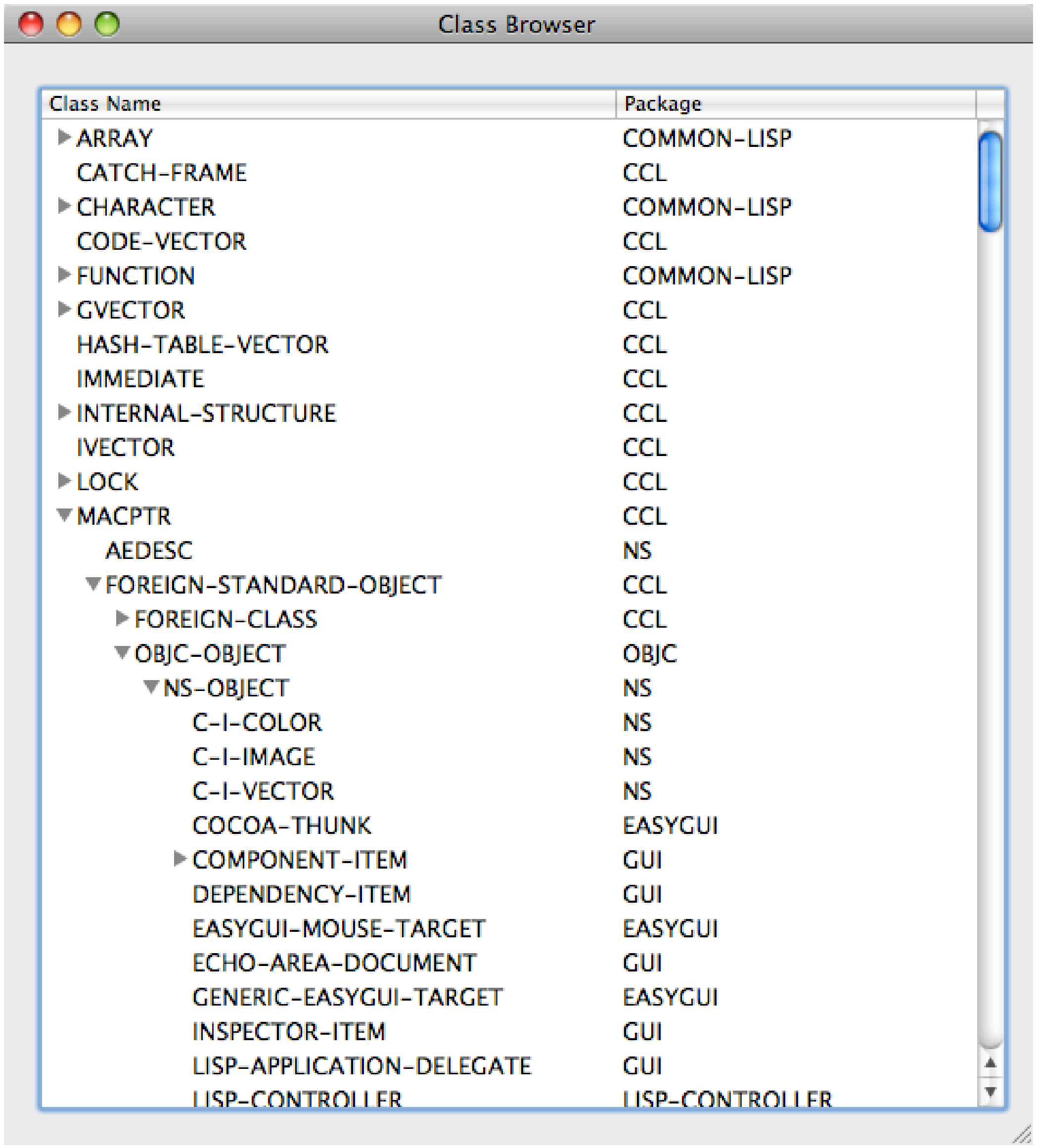
And that's it for the configuration and code needed. If everything is installed properly and your initialization functions have been set up as specified in the longer tutorial, then in the listener you can do the following:

```
Welcome to Clozure Common Lisp Version 1.5-dev-r13523M-trunk  (DarwinX8664)!
? (require :controller-test1)
:CONTROLLER-TEST1
("NS-STRING-UTILS" "DATE" "DECIMAL" "NS-OBJECT-UTILS" "NSLOG-UTILS" "ASSOC-ARRAY" "LIST-UTILS"
"LISP-CONTROLLER" "CONTROLLER-TEST1")
? (ct1:test-controller)
#<LISP-CONTROLLER-TEST <LispControllerTest: 0x129d5e60> (#x129D5E60)>
?
```

That will open up a window and you can add, edit, and remove to your heart's content.

5.2 Controller Test 2: Class Browser

In the second example we will create a somewhat more practical application. It illustrates the interaction between a lisp-controller and an NSOutlineView. It also illustrates how both rows and their children can be objects; in this case they are instances of Lisp Class objects. It's actually a very simple application that illustrates just how easy it is to display hierarchical arrangements of objects. The general idea is that the top-level rows will be all the immediate sub-classes of the class T. Each row will display one of these classes and when that row is expanded the indented rows immediately beneath it will display its immediate sub-classes. Any class with subclasses can be expanded in subsequent rows that are even more indented. The second column will provide the name of the package where the class name is interned. We'll sort all classes at any level alphabetically. In operation the window will look like Figure 11 below.



Class Browser

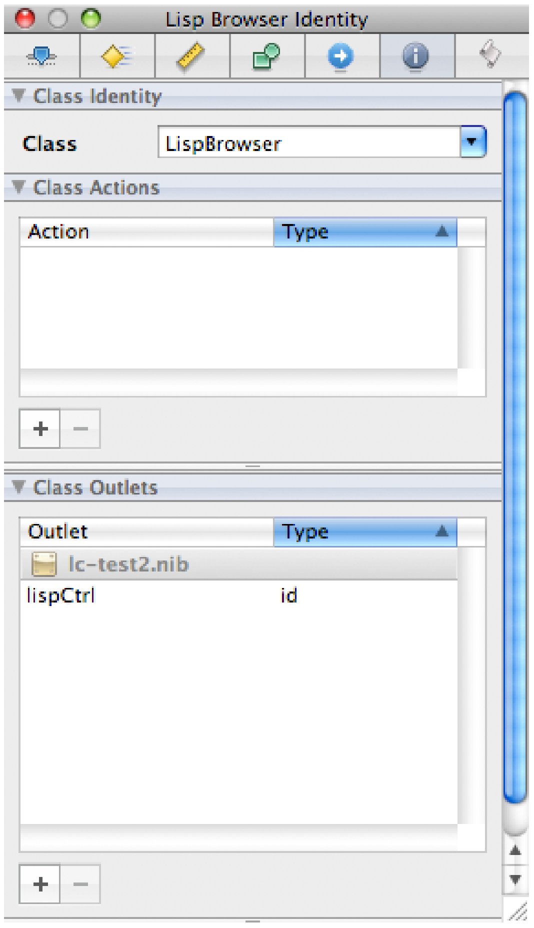
Class Name	Package
▶ ARRAY	COMMON-LISP
CATCH-FRAME	CCL
▶ CHARACTER	COMMON-LISP
CODE-VECTOR	CCL
▶ FUNCTION	COMMON-LISP
▶ GVECTOR	CCL
HASH-TABLE-VECTOR	CCL
IMMEDIATE	CCL
▶ INTERNAL-STRUCTURE	CCL
IVECTOR	CCL
▶ LOCK	CCL
▼ MACPTR	CCL
AEDESC	NS
▼ FOREIGN-STANDARD-OBJECT	CCL
▶ FOREIGN-CLASS	CCL
▼ OBJC-OBJECT	OBJC
▼ NS-OBJECT	NS
C-I-COLOR	NS
C-I-IMAGE	NS
C-I-VECTOR	NS
COCOA-THUNK	EASYGUI
▶ COMPONENT-ITEM	GUI
DEPENDENCY-ITEM	GUI
EASYGUI-MOUSE-TARGET	EASYGUI
ECHO-AREA-DOCUMENT	GUI
GENERIC-EASYGUI-TARGET	EASYGUI
INSPECTOR-ITEM	GUI
LISP-APPLICATION-DELEGATE	GUI
LISP-CONTROLLER	LISP-CONTROLLER

Figure 11: The Class Browser Window

As in the first example, you can start with the example NIB file "ip:Controller Test 2;lc-test2.nib" or start with your own and build it as described below.

Open up IB and start with a new Cocoa window template. Change the title for the window to "Class Browser" or something else that you prefer. Find and drag an Outline View from the Library window to the new window. Configure it to have two columns labeled "Class" and "Package". Click once on the new Outline View to select the Scroll View object and then select the size pane of the inspector window. Click on both of the red arrows inside the box shown to cause the object to be resized when the window is resized. By default all View objects will automatically resize subviews, which is what we want here, so we'll leave that alone. The resizing behavior that we want is for the last column to remain the same size, even when the window is resized and the first column to get any additional width that comes with being in a bigger window. To get that, select the Outline View object and in the attributes pane of the inspector window select the "First Column Only" option for the "Col. Sizing" value using the pull-down menu.

Next we'll configure the File's Owner object. Click on it and in the Identity Inspector window change the name of its class to LispBrowser. As in previous projects, we'll define a corresponding class in Lisp and create an instance of it at runtime. Add an outlet to this class called lispCtrl of type id. This should look like Figure 12 below.

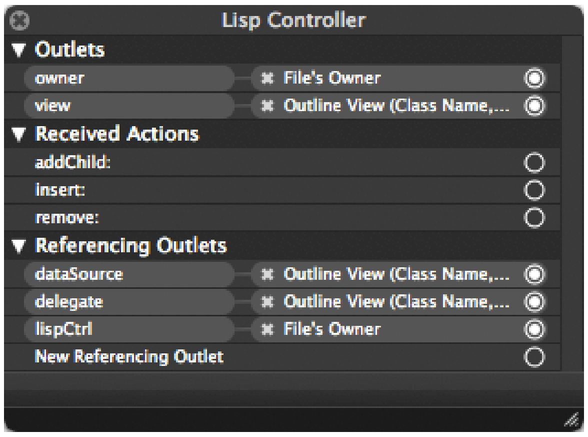


Class: LispBrowser

Outlet	Type
lispCtrl	id

Figure 12: LispBrowser Identity Inspector

Next we'll add and configure a Lisp Controller for our application. Find the Lisp Controller Plugin folder in the Library window and drag a Lisp Controller object to your document window. First we will link the lisp-controller to other objects in our interface design. Start by ctrl-clicking on the Lisp Controller and drag to the Outline view. Make the Outline View the value for the view outlet. Then ctrl-click and drag from the Lisp Controller to the File's Owner and make it the value of the owner outlet. Next ctrl-click on the outline view and drag from both the dataSource and delegate outlets to the Lisp Controller object. Finally ctrl-click and drag from the File's Owner object to the Lisp Controller and set the lispCtrl outlet. When you have completed all of those actions you can ctrl-click on the Lisp Controller and the window that pops up should be as shown in Figure 14 below.


Lisp Controller

▼ Outlets

owner	File's Owner	●
view	Outline View (Class Name,...	●

▼ Received Actions

addChild:	○
insert:	○
remove:	○

▼ Referencing Outlets

dataSource	Outline View (Class Name,...	●
delegate	Outline View (Class Name,...	●
lispCtrl	File's Owner	●
New Referencing Outlet		○

Figure 14: Lisp Controller links

Next we will configure the lisp-controller attributes. Click on the Lisp Controller object and select the attributes pane in the inspector window. Initially this will look like Figure 4 above. For this example we'll display the names and packages for standard Lisp class objects (i.e. objects of class #<STANDARD-CLASS CLASS>). Set the Root Type field to: class. Select the Generate Root checkbox. In the Type Information table we will tell the lisp-controller how to find the "children" of a class and what type to expect for each child. This will require a single entry in the Type Information table. That entry will specify

Type: class
Child Type: class
Children Key: #'ccl::class-direct-subclasses

So what we have told the lisp-controller is that to find the children of a class object it should funcall #'ccl::class-direct-subclasses on the parent class.

We selected the Generate Root checkbox, so it is necessary to tell the lisp-controller how to construct the root object. Since we told it that the root object was a class, we want to now specify an initform for the type "class". Since we do not allow the addition of any children for this application, constructing the root object is the only time that this initform will be used. So create an entry in the initform table:

Type: class
Initform: (find-class t)

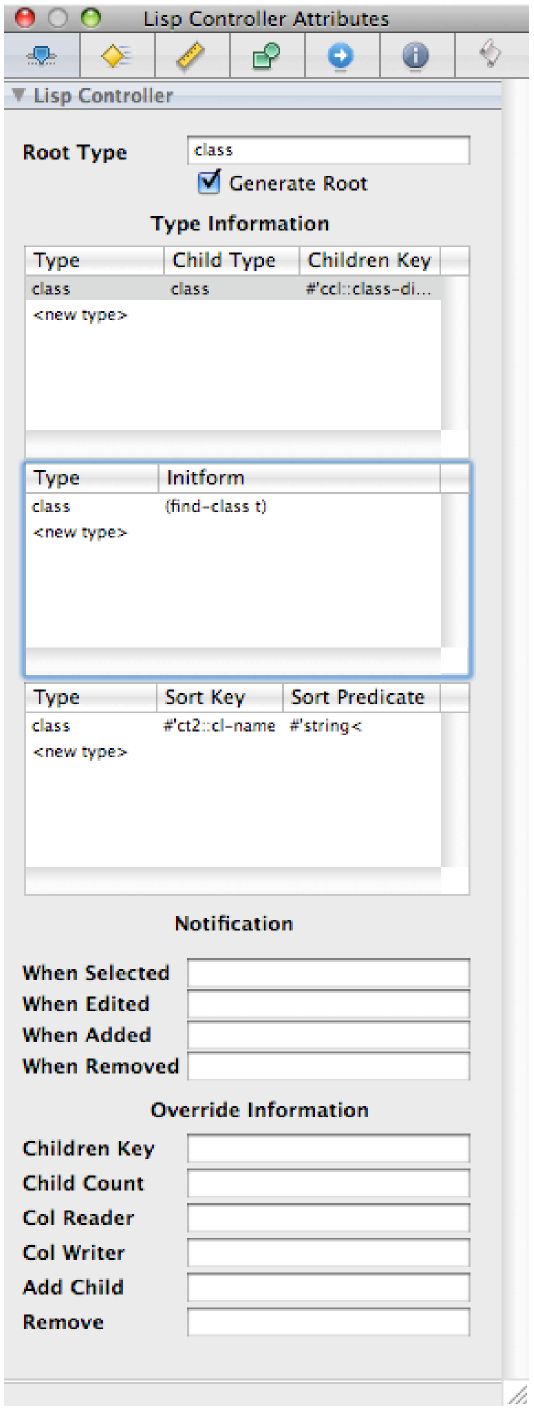
This initform will return the root class which is the super-type of all other classes.

Just for fun, we will order the subclasses shown for any class alphabetically. To do that we will put a single entry in the sort table. That entry will be:

Type: class
Sort Key: #'ct2::cl-name
Sort Predicate: @'string<

As we'll see when we look at the lisp code for this example, cl-name is a particularly simple function and string< is of course a standard Lisp function.

When the attribute configuration is complete, the inspector window should look like Figure 13 below.


Lisp Controller Attributes

Lisp Controller

Root Type: class

[✓] Generate Root

Type Information

Type	Child Type	Children Key
class	class	#'ccl::class-di...
<new type>		

Type	Initform
class	(find-class t)
<new type>	

Type	Sort Key	Sort Predicate
class	#'ct2::cl-name	#'string<
<new type>		

Notification

When Selected

When Edited

When Added

When Removed

Override Information

Children Key

Child Count

Col Reader

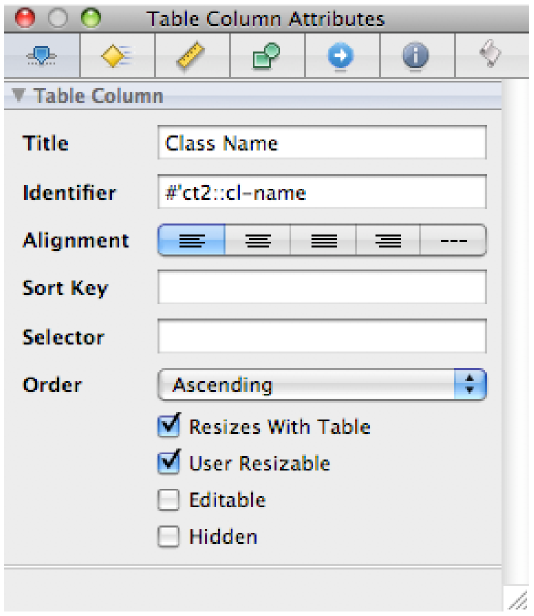
Col Writer

Add Child

Remove

Figure 13: Lisp Controller Attribute Inspector

The last thing to do within IB is to specify accessors for the two table columns. Start by clicking over the first column of the outline view until just the column has been selected. If you have done this correctly, the attribute inspector will show you that you have selected a Table Column. If you have not already set the title for this table column, do so now in the inspector window. I called the first column "Class Name", but you can choose whatever you would like for this. In the Identifier field type in #'ct2::cl-name to indicate that this function should be applied to the row-object (i.e. a class instance) to retrieve the value that should be put into this column. Make sure that this column is not editable by deselecting the Editable check box. If this has been done correctly, the attribute inspector window should look pretty similar to Figure 14 below:



Title: Class Name
Identifier: #'ct2::cl-name
Alignment
Sort Key
Selector
Order: Ascending
[✓] Resizes With Table
[✓] User Resizable
[ ] Editable
[ ] Hidden

Figure 14: Attribute Inspector for the first column

Modify the second column in much the same way. Title it something like "Package" and set the identifier to be #'ct2::cl-package Shortly we'll examine those two Lisp accessor functions.

This completes the interface design using IB. Save the NIB file (not XIB) as lc-test2.nib within the "Controller Test 2" subdirectory of the "Interface Projects" directory.

Now we'll take a look at the Lisp code needed to support this. Open the file "ip:Controller Test 2;controller-test2.lisp" within the CCL IDE.

Everything within this file is done within the package :ct2.

Recall that we specified that the File's Owner is of the class LispBrowser. First we define this class:

```
(defclass lisp-browser (ns:ns-window-controller)
   ((lisp-ctrl :foreign-type :id :accessor lisp-ctrl))
  (:metaclass ns:+ns-object))
```

This is similar to other NIB-managing classes that were used in projects defined in the InterfaceBuilderWithCCLTutorial. We make it a subclass of NSWindowController in order to make loading and managing NIB files a bit easier.

```
(objc:defmethod (#/initWithNibPath: :id)
                ((self lisp-browser) (nib-path :id))
  (let* ((init-self (#/initWithWindowNibPath:owner: self nib-path self)))
    init-self))
```

Similarly, this init method is very similar to other initialization functions done for previous projects.

```
(defun cl-name (cls)
  (symbol-name (class-name cls)))
```

The cl-name function was specified in IB as both the accessor for the first column and the sort key for ordering the children of any object. This simply returns the class-name as a string.

```
(defun cl-package (cls)
  (package-name (symbol-package (class-name cls))))
```

The cl-package function was specified as the accessor for the second column of the table. It simply returns the package where the class name is interned as a Lisp string.

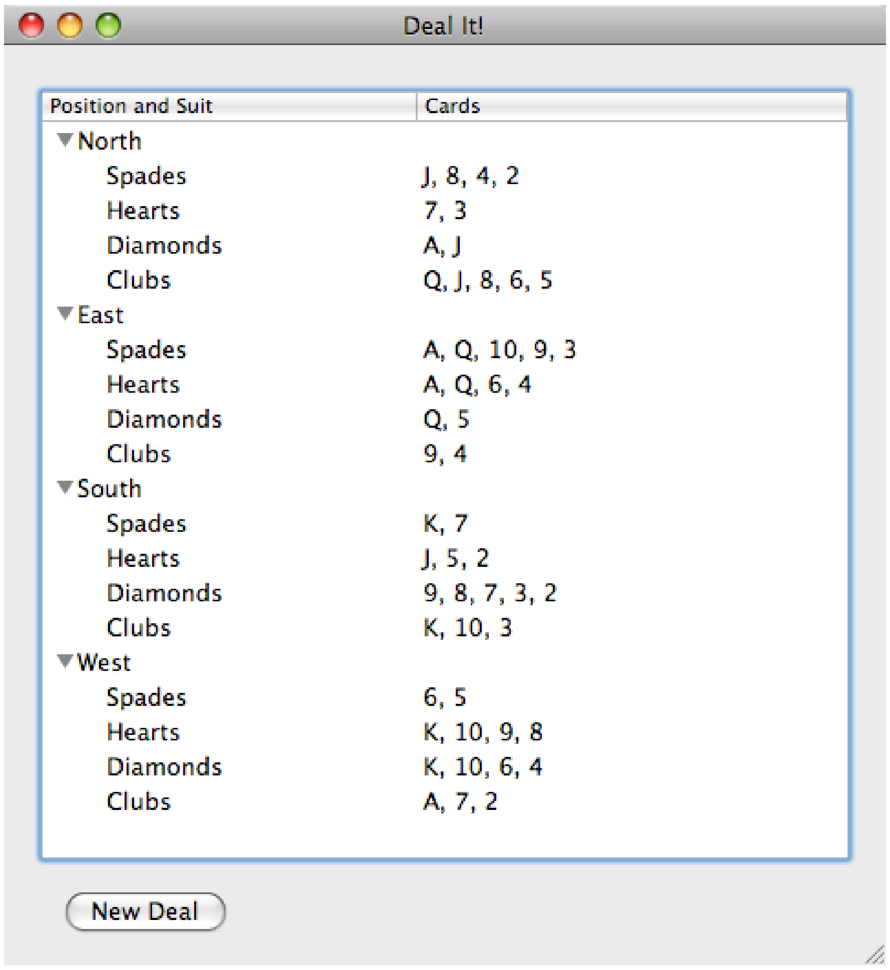
```
(defun test-browser ()
  (let* ((nib-name (lisp-to-temp-nsstring
                   (namestring (truename "ip:Controller Test 2;lc-test2.nib"))))
         (wc (make-instance 'lisp-browser
               :with-nib-path nib-name)))
    (#/window wc)
    wc))
```

This test function makes an instance of lisp-browser and returns it. All of the rest of the work necessary to support the display is done by the lisp-controller instance in coordination with the NSOutlineView display object. You can expand or contract classes to see their subclasses (if any). Note that they are ordered alphabetically, just as we specified in IB.

5.3 Controller Test 3: Card Dealer

This example is intended to show how the LispController works in conjunction with hash-tables. In fact, not only is the root object a type of hash-table, so are its children and grandchildren. The application deals out four hands of cards (North, South, East, and West) as in a bridge game from a conventional 52-card deck. The display shows each hand and each hand can be expanded to show four suits. A button is defined to deal out a new hand. This isn't a particularly useful application, but does nicely demonstrate the use of hash-tables. And I admit that I have spent some time dealing out hands, showing just the North/South cards, and imagine how I would bid and play them in a bridge match. Then I reveal the East/West cards to figure out what I should have done instead. If you are a bridge player, you might find this amusing.

At runtime after complete expansion of displayed rows the window will look something like Figure 15 below.



Position and Suit	Cards
▼North	
Spades	J, 8, 4, 2
Hearts	7, 3
Diamonds	A, J
Clubs	Q, J, 8, 6, 5
▼East	
Spades	A, Q, 10, 9, 3
Hearts	A, Q, 6, 4
Diamonds	Q, 5
Clubs	9, 4
▼South	
Spades	K, 7
Hearts	J, 5, 2
Diamonds	9, 8, 7, 3, 2
Clubs	K, 10, 3
▼West	
Spades	6, 5
Hearts	K, 10, 9, 8
Diamonds	K, 10, 6, 4
Clubs	A, 7, 2

Figure 15: The Card Dealer window

The positions North, East, South, and West are always shown in this order. And the suits are always displayed in the order shown (which will be familiar to bridge players). For this project it may be easier to look at the Lisp data side of the design first and then design the interface. I am a strong proponent of a data-first design methodology, but my experience is that a sort of iterative approach is probably best: first design the core data structures and data manipulation functionality, next design the interface, and finally design any control/integration functionality needed to bring the two together. So let's see if we can follow that path for this project.

Start by opening up the file "ip:Controller Test 3;controller-test3.lisp" within the CCL IDE. For purposes of this project we want to represent a complete deal of the cards; i.e. four hands, each with four suits, and some number of cards in each suit. There are, of course, many possible ways that we might choose to implement this and I'll admit that my choice was perhaps a bit artificial and made primarily because it demonstrates how hash-tables are handled by the lisp-controller. I chose to represent a whole deal as an instance of an assoc-array. What, you may well ask, is that?

An assoc-array is a class that I created that lets me make a multi-dimensional sparse array that can be indexed by arbitrary lisp objects (anything that could be used as a key in a hash table). I have found this to be a useful way of representing lots of miscellaneous data and in fact these are used in several places in the implementation of the lisp-controller class itself.

Assoc-arrays are defined in the file "ip:Utilities;assoc-array.lisp. An assoc-array is basically a hierarchical set of hash-tables. An assoc-array object contains a root hash-table. Each key in that table is some Lisp value that has been used as the first index when storing into the assoc-array. The value corresponding to that key will either be another hash-table if this is not the last dimension of the assoc-array or the value being stored. In an N-dimensional assoc-array then, there would be N-1 levels of embedded hash-tables and the last index would be the key into the most deeply embedded table. Hash tables are only created as needed to store new values. Perhaps it is just easier to see what the code does ...

```
(defclass assoc-array ()
  ((arr-rank :accessor arr-rank :initarg :rank)
   (index1-ht :accessor index1-ht)
   (index-tests :accessor index-tests :initarg :tests)
   (default-value :accessor default-value :initarg :default))
  (:default-initargs
    :rank 2
    :tests nil
    :default nil))
```

The assoc-array instance keeps track of the rank of the array, the first level hash-table, the tests that are used for each dimension to define a matching index and a default value for the table that is returned if no value is found.

```
(defmethod initialize-instance :after ((self assoc-array) &key tests &allow-other-keys)
  (setf (index1-ht self)
        (make-hash-table :test (or (first tests) #'eql))))
```

When created, the first level hash-table is initialized.

```
(defmethod assoc-aref ((self assoc-array) &rest indices)
  (unless (eql (list-length indices) (arr-rank self))
    (error "Access to ~s requires ~s indices" self (arr-rank self)))
  (do* ((res (index1-ht self))
        (index-list indices (rest index-list))
        (indx (first index-list) (first index-list))
        (found-next t))
       ((null index-list) (if found-next
                            (values res t)
                            (values (default-value self) nil)))
    (if found-next
      (multiple-value-setq (res found-next) (gethash indx res))
      (return-from assoc-aref (values (default-value self) nil)))))
```

Like #'aref for normal arrays, this method retrieves an indexed value. And like gethash, if the indices reference an existing entry, then that value is returned and otherwise the default value is returned. A second returned value indicates whether what was returned was found (t) or the default (nil).

```
(defmethod (setf assoc-aref) (new-val (self assoc-array) &rest indices)
  (unless (eql (list-length indices) (arr-rank self))
    (error "Access to ~s requires ~s indices" self (arr-rank self)))
  (let* ((ht (index1-ht self))
         (last-indx (do* ((dim 1 (1+ dim))
                          (index-list indices (rest index-list))
                          (indx (first index-list) (first index-list))
                          (tests (rest (index-tests self)) (rest tests))
                          (test (first tests) (first tests)))
                         ((>= dim (arr-rank self)) indx)
                      (multiple-value-bind (next-ht found-next) (gethash indx ht)
                        (unless found-next
                          (setf next-ht (make-hash-table :test (or test #'eql)))
                          (setf (gethash indx ht) next-ht))
                        (setf ht next-ht)))))
    (setf (gethash last-indx ht) new-val)))
```

The setf function for assoc-aref will create new hash-tables as necessary to represent each dimension of the path defined by the indices provided. At the end of the path the value is put into the final hash-table.

Following are several utility functions that can be used to inspect what is in an assoc-array. They are provided here without additional explanation. Hopefully the code is relatively self-explanatory.

```
(defmethod mapcar-assoc-array ((func function) (self assoc-array) &rest indices)
  ;; collects list of results of applying func to each bound index at
  ;; the next level after the indices provided.
  (unless (<= (list-length indices) (arr-rank self))
    (error "Access to ~s requires ~s or fewer indices" self (arr-rank self)))
  (do* ((res (index1-ht self))
        (index-list indices (rest index-list))
        (indx (first index-list) (first index-list))
        (found-next t))
       ((null index-list) (when found-next
                            ;; apply map function to res
                            (typecase res
                              (hash-table (mapcar-hash-keys func res))
                              (cons (mapcar func res))
                              (sequence (map 'list func res)))))
    (if found-next
      (multiple-value-setq (res found-next) (gethash indx res))
      (return-from mapcar-assoc-array nil))))

(defmethod map-assoc-array ((func function) (self assoc-array) &rest indices)
  ;; collects list of results of applying func of two arguments to
  ;; a bound index at the next level after the indices provided and to
  ;; the value resulting from indexing the array by appending that index
  ;; to those provided as initial arguments. This would typically be used
  ;; to get a list of all keys and values at the lowest level of an
  ;; assoc-array.
  (unless (<= (list-length indices) (arr-rank self))
    (error "Access to ~s requires ~s or fewer indices" self (arr-rank self)))
  (do* ((res (index1-ht self))
        (index-list indices (rest index-list))
        (indx (first index-list) (first index-list))
        (found-next t))
       ((null index-list) (when found-next
                            ;; apply map function to res
                            (typecase res
                              (hash-table (map-hash-keys func res))
                              (cons (mapcar func res nil))
                              (sequence (map 'list func res nil)))))
    (if found-next
      (multiple-value-setq (res found-next) (gethash indx res))
      (return-from map-assoc-array nil))))

(defun print-last-level (key val)
  (format t "~%Key = ~s   Value = ~s" key val))

(defmethod last-level ((self assoc-array) &rest indices)
  (apply #'map-assoc-array #'print-last-level self indices))

(defmethod map-hash-keys ((func function) (self hash-table))
  (let ((res nil))
    (maphash #'(lambda (key val)
                 (push (funcall func key val) res))
             self)
    (nreverse res)))
```

```
(defmethod mapcar-hash-keys ((func function) (self hash-table))
  (let ((res nil))
    (maphash #'(lambda (key val)
                 (declare (ignore val))
                 (push (funcall func key) res))
             self)
    (nreverse res)))
```

OK, so now that you know a little bit more about what an assoc-array is, we'll show how to use one to represent four bridge hands. Later we will display it in an NSOutlineView using our lisp-controller. All the following code is defined in package :ct3 as shown in controller-test3.lisp. At this point we will only look at the parts of that file that define the data structures (i.e. the "model" from Apple's Model/View/Controller paradigm). After that we will define the interface (i.e. the View) and finally we will come back to Lisp to add to the Controller part of the process.

We will represent any combination of cards from a deck as a bit vector that is 52 bits long. Cards included have a corresponding bit value of 1. Cards not included have a corresponding bit of 0. This representation can be used to represent a single hand, a complete deck, a single suit, or any other card combination. For example, we define some useful constants as follows:

```
(defconstant *aces* #*1000000000000100000000000010000000000001000000000000)
(defconstant *kings* #*0100000000000010000000000001000000000000100000000000)
(defconstant *queens* #*0010000000000001000000000000100000000000010000000000)
(defconstant *jacks* #*0001000000000000100000000000010000000000001000000000)
(defconstant *spades* #*1111111111111000000000000000000000000000000000000000)
(defconstant *hearts* #*0000000000000111111111111100000000000000000000000000)
(defconstant *diamonds* #*0000000000000000000000000011111111111110000000000000)
(defconstant *clubs* #*0000000000000000000000000000000000000001111111111111)
```

In addition, we want to define some constants to represent card ranks and suits:

```
(defconstant *card-ranks* '("A" "K" "Q" "J" "10" "9" "8" "7" "6" "5" "4" "3" "2"))
(defconstant *card-suits* '("Spades" "Hearts" "Diamonds" "Clubs"))
(defconstant *hand-suits* '("Spades" "Hearts" "Diamonds" "Clubs" "North" "East" "South" "West"))
```

Using all of these constants we can easily create two utility functions that tell us about any particular card.

```
(defun card-rank (card)
  ;; card is a bit index
  (nth (mod card 13) *card-ranks*))

(defun card-suit (card)
  ;; card is a bit index
  (nth (floor card 13) *card-suits*))
```

To deal cards we will start with a full deck and randomly remove cards from it one by one.

```
(defun deal-cards ()
  ;; randomizes and returns four unique hands
  (let ((deck (full-deck))
        (deal (make-instance 'assoc-array :rank 2))
        (card nil))
    (dotimes (i 13)
      (setf card (pick-random-card deck))
      (add-card deal "West" card)
      (remove-card card deck)
      (setf card (pick-random-card deck))
      (add-card deal "North" card)
      (remove-card card deck)
      (setf card (pick-random-card deck))
      (add-card deal "East" card)
      (remove-card card deck)
      (setf card (pick-random-card deck))
      (add-card deal "South" card)
      (remove-card card deck))
    deal))
```

The deal parameter defined within the let form just creates a two-dimensional assoc-array. The indices of this assoc-array will be the position (i.e. North, East, South, or West) and the suit of the card. The value indexed for any position and suit combination will be a list of the ranks of the cards of the specified suit dealt to the specified position. The function was deliberately designed to represent the way that a dealer would hand out cards; viz. one at a time in rotation. In the following we'll look at the functions called in a little more detail.

```
(defun full-deck ()
  (make-array '(52) :element-type 'bit :initial-element 1))
```

The full-deck function simply makes a bit-vector will all bits set.

```
(defun pick-random-card (deck)
  ;; returns a card index
  (let* ((cnt (count 1 deck))
         (card (1+ (random cnt))))
    (position-if #'(lambda (bit)
                     (when (plusp bit)
                       (decf card))
                     (zerop card))
                 deck)))
```

There are almost certainly more efficient mechanisms for selecting a random card from the set, but in practice the previous function seems to work fast enough. It counts the number of cards left and then picks one of them at random, returning the bit index that refers to that card.

```
(defun add-card (deal hand card)
  (setf (assoc-aref deal hand (card-suit card))
        (cons (card-rank card) (assoc-aref deal hand (card-suit card)))))
```

The add-card function adds the rank of the card to the list of ranks that is the value of the assoc-array for the specified position (as given by the hand parameter) and suit of the card.

```
(defun remove-card (card deck)
  ;; card is a bit index
  (setf (aref deck card) 0))
```

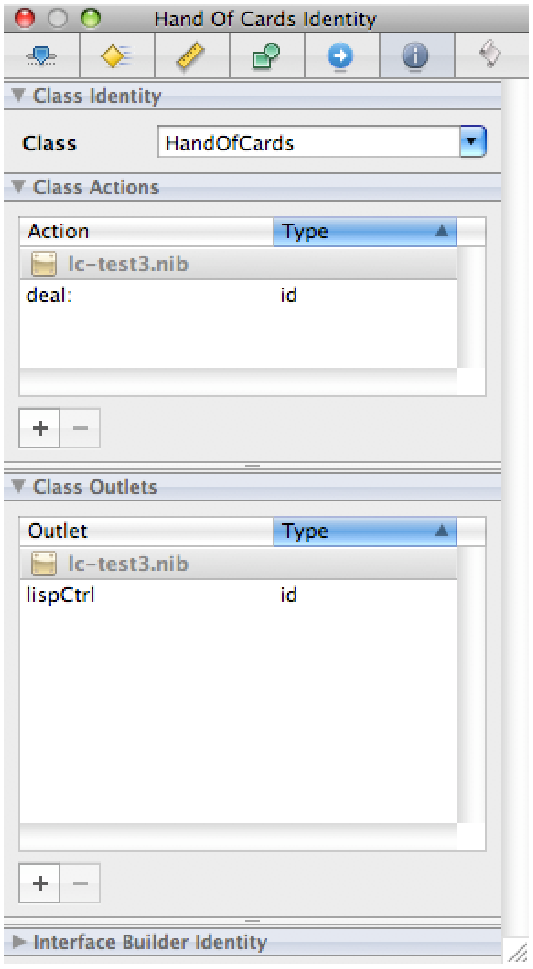
This function removes the card from the deck by setting the corresponding bit to 0.

This completes the definition of the basic data structures. Next we will define an interface to display deals. After that we will come back to Lisp to add the necessary glue functionality to pull everything together. As for all projects you can either open up the nib that I have already created: ."ip:Controller Test 3;lc-test2.nib" or start your own window nib in Interface Builder.

If you are starting from scratch, open up IB and start with a new Cocoa window template. Change the title for the window to "Deal It!" as I did, or to something else that you prefer. Find and drag an Outline View from the Library window to the new window. Configure it to have two columns labeled "Position and Suit" and "Cards" or pick your own titles. Click once on the new Outline View to select the Scroll View object and then select the size pane of the inspector window. Click on both of the red arrows inside the box shown to cause the object to be resized when the window is resized. By default all View objects will automatically resize subviews, which is what we want here, so we'll leave that alone. The resizing behavior that we want is for the last column to remain the same size, even when the window is resized and the first column to get any additional width that comes with being in a bigger window. To get that, select the Outline View object and in the attributes pane of the inspector window select the "First Column Only" option for the "Col. Sizing" value using the pull-down menu.

Next add a button to the interface window and label it "New Deal". This will be used to trigger the generation and display of a new deal. Position and size it as you choose.

Next we'll configure the File's Owner object. Click on it and in the Identity Inspector window change the name of its class to HandOfCards. As in previous projects, we'll define a corresponding subclass of NSWindowController in Lisp and create an instance of it at runtime. Add an outlet to this class called lispCtrl of type id. Also add an Action called "deal:" with a type of id. This should look like Figure 16 below.


Hand Of Cards Identity

Class Identity

Class: HandOfCards

Class Actions

Action	Type
lc-test3.nib	
deal:	id

Class Outlets

Outlet	Type
lc-test3.nib	
lispCtrl	id

Interface Builder Identity

Figure 16: Identity of File's Owner

Now ctrl-click and drag from the New Deal button the the File's Owner Object. Link it to the "deal:" action that you defined for the File's Owner. Later we'll create a function in Lisp that will handle this message.

Next we'll add and configure a Lisp Controller for our application. Find the Lisp Controller Plugin folder in the Library window and drag a Lisp Controller object to your document window. First we will link the lisp-controller to other objects in our interface design. Start by ctrl-clicking on the Lisp Controller and drag to the Outline view. Make the Outline View the value for the view outlet. Then ctrl-click and drag from the Lisp Controller to the File's Owner and make it the value of the owner outlet. Next ctrl-click on the outline view and drag from both the dataSource and delegate outlets to the Lisp Controller object. Finally ctrl-click and drag from the File's Owner object to the Lisp Controller and set the lispCtrl outlet. These are exactly the same actions that we took for the Class Browser project, so at this point if you ctrl-click on the Lisp Controller object you should see something that looks like Figure 14 from the previous project.

Next we will configure the lisp-controller attributes. Click on the Lisp Controller object and select the attributes pane in the inspector window. Initially this will look like Figure 4 above. For this example we'll display the assoc-array that we used to represent a complete deal, so set the Root Type field to: iu::assoc-array. Make sure that the Generate Root checkbox is NOT checked since we will use the New Deal button to trigger the generation of a new root element to be displayed. In the Type Information table we will tell the lisp-controller how to find the "children" of our assoc-array. Not that we do not need to specify the child-type since we will never be creating new instances of them. For our purposes we want there to be two levels displayed under the root object. The first level contain each of the hands and for each hand we will want to display each of the four suits. This will require two entries in the Type Information table. The first tells the lisp-controller how to find the first-level hash-table within the assoc-array:

Type: iu::assoc-array
Children Key: #'ct3::hand-children

What this will do is take advantage of our knowledge about how assoc-arrays are implemented to return the hash-table that is used internally. This is not generally a good practice, but please remember that I wanted to create an example that used hash-tables and this seemed like an easy way to go.

The second entry in the Type information table is needed to *block* the default behavior of the lisp-controller. By default the children of a hash-table entry are found by treating the value of that entry as a parent and then finding its children. So the type of the value determines what function is applied to find its children. In our case, when we get to the point where the value of a hash-table entry is a list of card ranks, if we did nothing else its type would be found to be LIST. The default way of finding the children of a list is just to use all elements of the list as the children. So in our case, each card in the list would be treated as a child and the suit would be shown as being expandable. Expanding it would show as many children as there are cards in the suit. So what we really want to do here is define a Children Key entry that guarantees a null return when applied to the list of card ranks. So to do that we define the following entry:

Type: cons
Children Key: #'null

Since CONS is a subtype of LIST, the lisp-controller will identify a list of card-ranks as being of type CONS and look for a valid children-key for that type. We have defined #'null as being that children-key. Applying #'null to any CONS type will return nil, so we have guaranteed that no children will be found. If you would like to see what happens without this table entry, just eliminate it from the NIB file and run the test function.

Note that in our example we took advantage of the fact that we didn't want to use lists as parent objects anywhere else in the table. It is possible that you may not be so lucky and might want some types of lists to have children and some not or have different sorts of lists have different children-keys. In those cases it would be necessary to create an appropriate type definition and use it to distinguish different types of lists. Generally this should be quite easy to do.

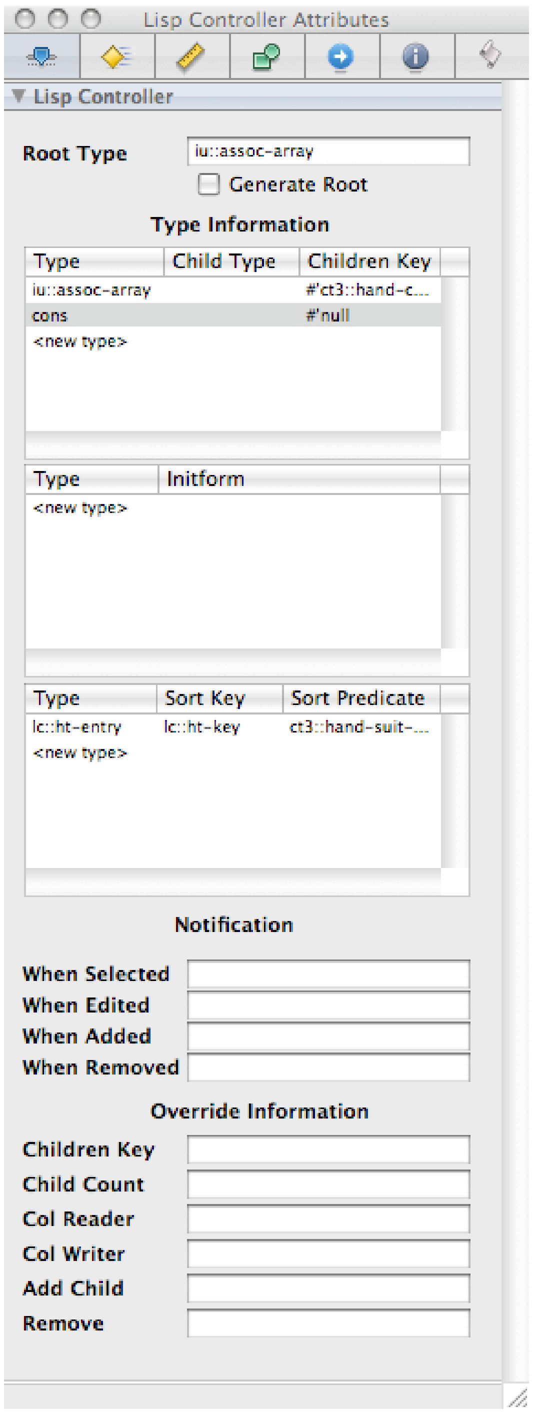
No object is being created for us by the lisp-controller, so no entries into the Initform table are required.

We want to order the position and suit hash-table pairs that are displayed so that we see the same order every time. That is, we effectively want to order the children of hash-tables which are, as discussed above, ht-entry objects. There are a few different ways that we could do this. For example, we could define special types that correspond to the different sub-types of ht-entry and then define a sort-key and predicate for each type. For this project we can rather easily define a single predicate for a list of ht-entry children, so that is what we will do. Create a single Sort Table entry:

Type: lc::ht-entry
Sort Key: lc::ht-key
Sort Predicate: ct3::hand-suit-order

Note that we intentionally specified the sort key and predicate without using the #' notation. It is entirely optional and we left it off here as a means of testing that both forms work correctly. This entry specifies that we should sort a list of ht-entry children using #'ct3::hand-suit-order as a predicate applied to the keys of the ht-entry key-value pairs. I should add parenthetically that I am not entirely happy with the requirement that the developer understand the internals of ht-entry objects in order to make this work and may consider alternative syntax at some future time.

When the attribute configuration is complete, the inspector window should look like Figure 17 below.


Lisp Controller Attributes

Lisp Controller

Root Type: iu::assoc-array

Generate Root [ ]

Type Information

Type	Child Type	Children Key
iu::assoc-array		#'ct3::hand-c...
cons		#'null
<new type>		

Type	Initform
<new type>	

Type	Sort Key	Sort Predicate
lc::ht-entry	lc::ht-key	ct3::hand-suit-...
<new type>		

Notification

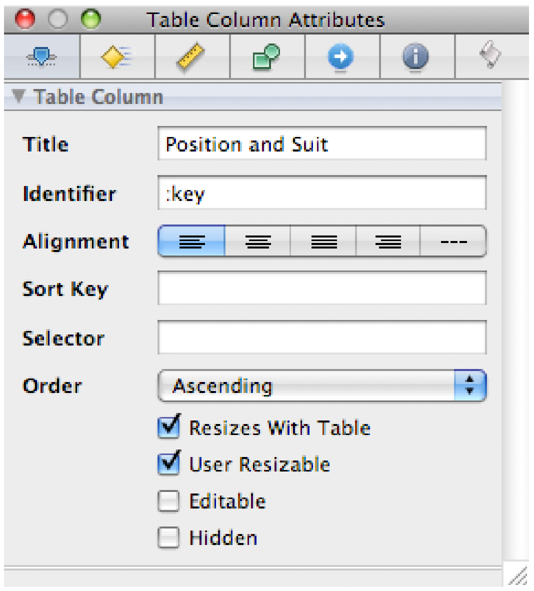
When Selected	
When Edited	
When Added	
When Removed	

Override Information

Children Key	
Child Count	
Col Reader	
Col Writer	
Add Child	
Remove	

Figure 17: Lisp Controller Attribute Inspector

The last thing to do within IB is to specify accessors for the two table columns. Start by clicking over the first column of the outline view until just the column has been selected. If you have done this correctly, the attribute inspector will show you that you have selected a Table Column. If you have not already set the title for this table column, do so now in the inspector window. I called the first column "Position and Suit", but you can choose whatever you would like for this. In the Identifier field type in the keyword :key to indicate that the key of the key-value pair should be put into this column. Make sure that this column is not editable by deselecting the Editable check box. If this has been done correctly, the attribute inspector window should look pretty similar to Figure 18 below:



Title: Position and Suit
Identifier: :key
Alignment
Sort Key
Selector
Order: Ascending
[✓] Resizes With Table
[✓] User Resizable
[ ] Editable
[ ] Hidden

Figure 18: Attribute Inspector for the first column

Modify the second column in much the same way. Title it something like "Cards" and set the identifier to be #'ct3::sorted-by-rank. By default, this function will be applied to the *value* of the ht-entry key-value pair. We could have equivalently specified this identifier as the form:
(ct3::sorted-by-rank :value)
Shortly we'll examine this Lisp accessor function.

This completes the interface design using IB. Save the NIB file (not XIB) as lc-test3.nib within the "Controller Test 3" subdirectory of the "Interface Projects" directory.

Next we will go back to Lisp and define the remaining functionality required for the Controller part of the Model/View/Controller paradigm.

```
(defclass hand-of-cards (ns:ns-window-controller)
   ((lisp-ctrl :foreign-type :id :accessor lisp-ctrl))
  (:metaclass ns:+ns-object))
```

This is our NSWindowController subclass.

```
(objc:defmethod (#/initWithNibPath: :id)
                ((self hand-of-cards) (nib-path :id))
  (let* ((init-self (#/initWithWindowNibPath:owner: self nib-path self)))
    init-self))
```

This is the same init method that we have used previously.

```
(objc:defmethod (#/deal: :void)
                ((self hand-of-cards) (sender :id))
  (declare (ignore sender))
  (unless (eql (lisp-ctrl self) (%null-ptr))
    (setf (root (lisp-ctrl self)) (deal-cards))))
```

The deal method will be invoked when the "New Deal" button is pushed by the user. It sets the root of the associated lisp-controller to the assoc-array that results from calling deal-cards. The lisp-controller then manages all further interaction with the NSOutlineView object.

```
(defun hand-children (hand)
  ;; hand will be an assoc-array
  (iu::index1-ht hand))
```

This function was specified as the children-key for the root assoc-array. This just returns the top-level hash-table from that object. So the root will be a hash-table and its displayed children will be key-value pairs that represent entries within that hash-table. In this case, there will be one key for each of the four positions and the value of each key will be another hash-table.

```
(defun hand-suit-order (a b)
  (< (position a *hand-suits* :test #'string=)
     (position b *hand-suits* :test #'string=)))
```

The hand-suit-order function was specified as the sort-predicate for ht-entry objects. Since those objects might have either positions as keys or suits as keys (depending on how deeply nested we are), the sort predicate must be able to handle either case. So we made a single ordering that works for either that we called *hand-suits*. This is a bit of a kludge and it might have been somewhat clearer to define a separate type for each subtype of ht-entry and a corresponding sort predicate. If you're ambitious, feel free to modify the code to make this happen.

For the second column we specified an accessor called #'sorted-by-rank. We want this to apply only to values that are a list of card ranks. To accomplish that we define a type called rank-list that can be used to identify such lists and our accessor uses this type to determine whether or not to return a value to be displayed. Note that this is a general consideration needed when displaying objects of different types at different levels of an NSOutlineView. The column accessors will be applied whenever it is valid to do so. So you must assure that specified accessor functions can handle any object type that might be given to it by your application. In this case we assure that the value of the key-value pair is in fact a list of card ranks.

```
(deftype rank-list ()
  '(satisfies all-ranks))

(defun all-ranks (rank-list)
  (and (listp rank-list)
       (null (set-difference rank-list *card-ranks* :test #'string=))))

(defun higher-rank (r1 r2)
  (< (position r1 *card-ranks* :test #'string=) (position r2 *card-ranks* :test #'string=)))

(defun sorted-by-rank (rlist)
  (when (typep rlist 'rank-list)
    (format nil "~{~a~^, ~}" (sort-list-in-place rlist #'higher-rank))))
```

The final bit of Lisp code needed is a test function that we can use in the listener to start things off:

```
(defun test-deal ()
  (let* ((nib-name (lisp-to-temp-nsstring
                     (namestring (truename "ip:Controller Test 3;lc-test3.nib"))))
         (wc (make-instance 'hand-of-cards
               :with-nib-path nib-name)))
    (#/window wc)
    wc))
```

In the listener type (ct3:test-deal) to open up the window.

5.4 Binding Test

This example is intended to show how the LispController can be used to bind IB interface elements to normal Lisp slots in normal (non-Objective-C) Lisp class instances. The window is simple and will be familiar to those who have worked through the tutorial. We will re-implement the Simple Sum example, but will bind the interface elements to Lisp slots. So the window will look as in Figure 19 below:

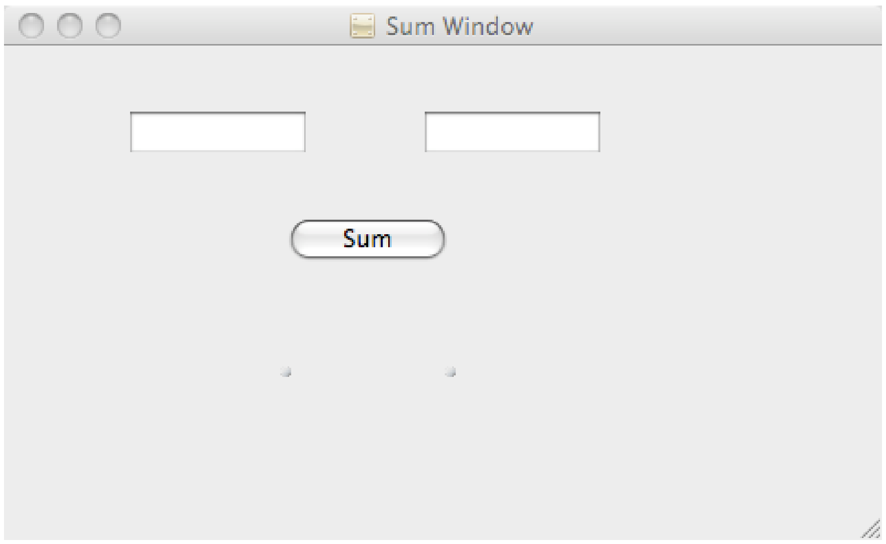
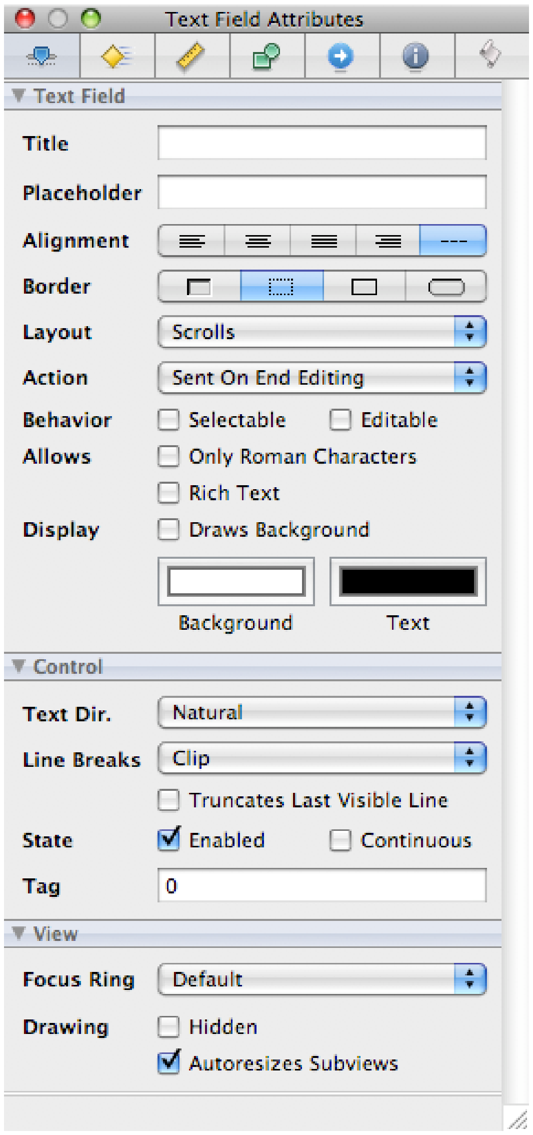


Figure 19: Simple Sum Window

As usual you can follow along with the interface design by opening the NIB file: "ip:BindingTest;binding.nib" in IB or starting from scratch with your own Window. If you are starting from scratch, open up IB and start with a new Cocoa window template. Change the title for the window to "Sum Window" as I did, or to something else that you prefer. Now let's design the interface. As we did before, select your window and give it a title in the inspector window. From the library window select and drag three "Text Fields" and one Button of your choice to your window and arrange them any way that pleases you. Double-click on your button to rename it to "Sum" If you like, select a text field and use the inspector to explore various options you can select for its appearance.

Note that my third text field is barely visible (only visible at all because I clicked on it) because I used the inspector to make it's border invisible and deselected the "Draws Background" box. I also deselected the "Selectable" and "Editable" boxes so that the user cannot click within it to do anything. When you change an attribute of a view object, IB tries to make it look exactly the way that it will at runtime so that you can easily visualize it. That's true even if it makes your object invisible. Sometimes that can make it hard to find when you need it, so you may want to wait until the design is mostly complete before setting attributes that make something invisible.


Text Field Attributes

Text Field

Title

Placeholder

Alignment

Border

Layout: Scrolls

Action: Sent On End Editing

Behavior: ☐ Selectable ☐ Editable

Allows: ☐ Only Roman Characters

☐ Rich Text

Display: ☐ Draws Background

Background Text

Control

Text Dir.: Natural

Line Breaks: Clip

☐ Truncates Last Visible Line

State: ☑ Enabled ☐ Continuous

Tag: 0

View

Focus Ring: Default

Drawing: ☐ Hidden

☑ Autoresizes Subviews

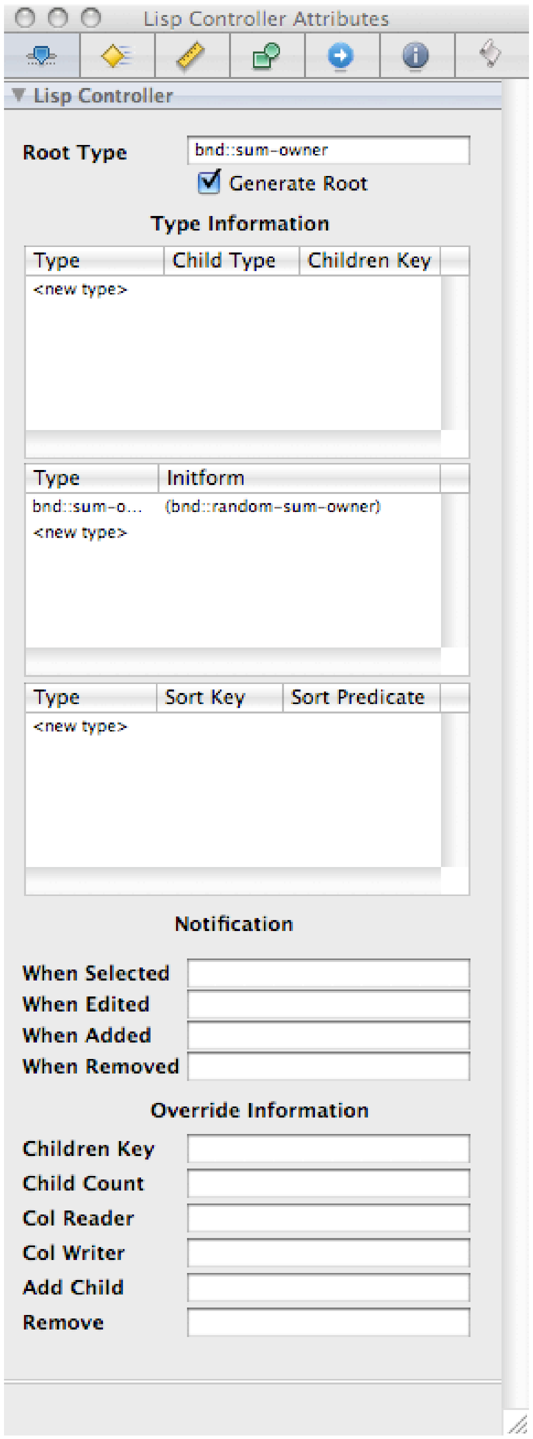
Figure 20: Setting attributes for the output text field

Next we need to add a lisp-controller to the project. Find the Lisp Controller Plugin folder in the Library window and drag a Lisp Controller object to your document window. In the attribute inspector for the Lisp Controller set the root type to bnd::sum-owner. This will be the name of the Lisp class that we will define. As for all Lisp Controller values, this is the Lisp name of the class. Then check the Generate Root checkbox. Since we are only going to use the Lisp Controller as an intermediate target on the way to our Lisp Object it isn't necessary to put any information about its "children" into the type table. But since we configured the Lisp Controller to generate the root object we must tell it how to do so. So make an entry into the Initform table:

Type: bnd::sum-owner
Initform: (bnd::random-sum-owner)

This initform will call a function that we will define in Lisp. It will simply create an instance of the sum-owner class that has the two input values initially set to random values.

This is all of the configuration needed for the Lisp Controller. The only field absolutely required when simply binding through the root object would be the Root Type. When the root object is set, the Lisp Controller will validate that its type is valid. When configuration is complete, the attribute pane for the Lisp Controller should look much like Figure 21 below.


Lisp Controller Attributes

Lisp Controller

Root Type: bnd::sum-owner

[✓ Generate Root]

Type Information

Type	Child Type	Children Key
<new type>		

Type	Initform
bnd::sum-o...	(bnd::random-sum-owner)
<new type>	

Type	Sort Key	Sort Predicate
<new type>		

Notification

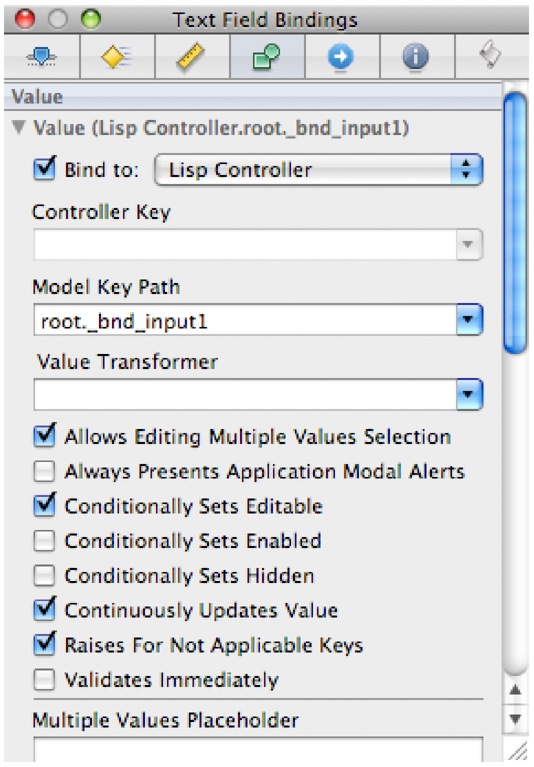
When Selected	
When Edited	
When Added	
When Removed	

Override Information

Children Key	
Child Count	
Col Reader	
Col Writer	
Add Child	
Remove	

Figure 21: Lisp Controller attribute inspector

Finally we need to set up the binding paths for our three TextFields. Click on the first input TextField and then open up the binding pane of the inspector window. Click on the arrow next to "Value" to reveal the binding configuration fields needed to create a binding between the TextField's value and some other object. Before clicking in the "Bind to:" checkbox, first select "Lisp Controller" from the pull-down menu. Then click in the "Bind to:" checkbox. Doing these two things in reverse order will result in IB adding a SharedObjectsController to your document window. You can just select and delete it if this happens to you. Additionally click in the "Continuously Updates Value" checkbox. This makes sure that the value is updated in the bound slot even if you don't leave the Text Field after editing it. Make sure the Controller Key field is blank and enter "root._bnd_input1" into the Model Key Path field. This tells the Text Field to get its value from the bnd::input-1 slot of the Lisp object that is the value of the root of the Lisp Controller. When you get done this should look much like Figure 22 below.


Text Field Bindings

Value

Value (Lisp Controller.root._bnd_input1)

[✓] Bind to: Lisp Controller

Controller Key

Model Key Path

root._bnd_input1

Value Transformer

[✓] Allows Editing Multiple Values Selection
[ ] Always Presents Application Modal Alerts
[✓] Conditionally Sets Editable
[ ] Conditionally Sets Enabled
[ ] Conditionally Sets Hidden
[✓] Continuously Updates Value
[✓] Raises For Not Applicable Keys
[ ] Validates Immediately

Multiple Values Placeholder

Figure 22: Binding Pane for first TextField

Now do the same for the second Text Field, but bind its value to root._bnd_input2. Finally bind the sum Text Edit field to "root._bnd_sum".

And that's all of the IB work needed to make this application work. Let's take a look at the Lisp code needed to support it.

The code is contained in the file "ip:BndingTest;binding.lisp". All code is defined within a package named :binding (nickname :bnd). The first thing done is to define an object class to hold the two values and their sum. We will set the root of the lisp-controller to be one of these objects. Then the slots of this object will be referenced from the TextFields by virtue of the bindings that we created in IB. The class is defined as follows:

```
(defclass sum-owner ()
  ((input-1 :reader input-1 :initarg :input-1)
   (input-2 :reader input-2  :initarg :input-2)
   (sum :reader sum :initform 0))
  (:default-initargs
     :input-1 0
     :input-2 0))
```

The slots were defined to have automatically generated reader methods, but not writers. We will define these explicitly so that we can add code to make them KVC-compliant as follows.

```
(defmethod (setf input-1) (new-value (self sum-owner))
  ;; we set up the slot writer to note a change in value to make it KVC compliant
  (will-change-value-for-key self 'input-1)
  (setf (slot-value self 'input-1) new-value)
  (did-change-value-for-key self 'input-1))

(defmethod (setf input-2) (new-value (self sum-owner))
  ;; we set up the slot writer to note a change in value to make it KVC compliant
  (will-change-value-for-key self 'input-2)
  (setf (slot-value self 'input-2) new-value)
  (did-change-value-for-key self 'input-2))
```

```
(defmethod (setf sum) (new-value (self sum-owner))
  ;; we set up the slot writer to note a change in value to make it KVC compliant
  (will-change-value-for-key self 'sum)
  (setf (slot-value self 'sum) new-value)
  (did-change-value-for-key self 'sum))
```

Then we add an initialize-instance method just to assure that the sum value initially equals the real sum of the two input value slots:

```
(defmethod  initialize-instance :after ((self sum-owner)
                                       &key &allow-other-keys)
  (setf (sum self) (+ (input-1 self) (input-2 self))))
```

Next we create the functions that we specified in IB for creating the lisp-controller's root:

```
(defun random-sum-owner ()
  ;; return a sum-owner instance with random initial values
  (make-instance 'sum-owner
    :input-1 (random 100)
    :input-2 (random 100)))
```

Next we create a class and methods for the File's Owner class, just as we have done for many other projects. Nothing too special here except to make it the target of the "Sum" button and provide a method that will be invoked when the button is pushed.

```
(defclass binding-window-owner (ns:ns-object)
  ((controller :foreign-type :id
               :accessor controller)
   (nib-objects :accessor nib-objects :initform nil))
  (:metaclass ns:+ns-object))

(defmethod initialize-instance :after ((self binding-window-owner)
                                       &key &allow-other-keys)
  (setf (nib-objects self)
        (load-nibfile
         (truename "ip:BindingTest;binding.nib")
         :nib-owner self
         :retain-top-objs t))
  ;; we do the following so that ccl:terminate will be called before we are garbage
  ;; collected and we can release the top-level objects from the NIB that we retained
  ;; when loaded
  (ccl:terminate-when-unreachable self))

(defmethod ccl:terminate ((self binding-window-owner))
  (dolist (top-obj (nib-objects self))
    (unless (eql top-obj (%null-ptr))
      (#/release top-obj))))
```

Below is the #/doSum method that we specified for the button in IB. It simply sets the sum. I suppose the one thing to note here is that although we put most of the method within a "with-slots" form, we don't include the sum slot itself, but rather access it via the more complete (sum (root (controller self))) form. Why is that? The answer is that when the with-slots form is used, the accessor methods defined for the slot are bypasses and the method "slot-value" (or some compiler-generated equivalent) is used. If we were to include the sum slot in the variables declared for the with-slots form then the extra methods that we added to call will-change-value-for and did-change-value-for would never get called. That's just a little something to be aware of if you implement the writer for a slot that you want to bind to in the way that I did.

```
(objc:defmethod (#/doSum: :void)
                ((self binding-window-owner) (s :id))
  (declare (ignore s))
  (with-slots (input-1 input-2) (root (controller self))
    (setf (sum (root (controller self)))
          (+ input-1 input-2))))
```

Then we create a simple test function that returns this window.

```
(defun test-binding ()
  (make-instance 'binding-window-owner))
```

In addition, you can run additional tests from the listener by changing the root object of the lisp-controller to see what happens. For example you could do:

```
? (setf bwo (bnd:test-binding))
#<BINDING-WINDOW-OWNER <BindingWindowOwner: 0x12c5a690> (#x12C5A690)>
? (setf lc (bnd:controller bwo))
#<LISP-CONTROLLER <LispController: 0x12c52930> (#x12C52930)>
? (setf (lc:root lc) (bnd::random-sum-owner))
;Compiler warnings :
;   In an anonymous lambda form at position 15: Undeclared free variable LC
#<BINDING:SUM-OWNER #x30200105B2DD>
? (setf (lc:root lc) (bnd::random-sum-owner))
;Compiler warnings :
;   In an anonymous lambda form at position 15: Undeclared free variable LC
#<BINDING:SUM-OWNER #x30200105117D>
?
```

and watch what happens in your window. You could also set the individual slots input-1 and input-2 in the current root object. Note that these changes will be immediately reflected in the TextFields that are bound to these slots. But of course the sum field will not be updated until you click on the sum button.

6.0 Implementation

This section goes through the implementation of the lisp-controller in some detail. This may help your understanding of what it does or provide a template for making similar classes of your own.

In the Interface Builder with CCL Tutorial: Project 5 where we built the Package viewing window, it was necessary to implement Objective-C methods for supplying the proper elements from our lisp arrays as demanded by the NSTextView interface objects. As you might expect, doing this sort of thing is fairly common, both in the Objective-C and Lisp worlds. To help developers, Cocoa provides a number of "data controller" classes. The idea is that you can, for example, hand an NSArray to an NSArrayController and it will provide all of the relevant interface methods that make it act as a data source for an NSTableView. These controllers also have "add" and "remove" methods that can be invoked by interface buttons in order to create and add or remove an Objective-C object from the collection. This is all well and good if you happen to have an NSArray or other Objective-C container class and if every sort of thing that you want inside your collection is an Objective-C object, but that's not so useful for Lisp programmers.

We could, of course, just use those Objective-C container classes directly if wanted to go to a lot of trouble. But in my opinion you might as well just use Objective-C directly if you want to go that route. Instead, in this project we'll create an analog to an Objective-C controller that knows how to use Lisp arrays, lists and hash-tables as container objects and which can instantiate any sort of Lisp object and add it to the collection in response to a user-interface control action. In other words, it will act very similarly to a standard Objective-C controller, but be much more useful to Lisp programmers.

In addition, we would like our Lisp controller to be easily accessible from within IB, so I have provided an Xcode project that you can build. It will create a framework that you will need to install and an IB Plugin module that you will need to load into Interface Builder. That will allow you to directly select and configure a lisp-controller object as part of your interface design.

I'm not going to document the construction of the Xcode proejct for the IB plugin module here. If I get enough requests for that material I will add it. Otherwise you can use the project I provided as an example for your own work or feel free to modify what I've done for your own purposes. If you do either of those you will want to consult:
http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/IBPlugInGuide/IBPlugInGuide.pdf
I will, of course, discuss the Lisp code for those controller objects below.

Some developers might prefer to just use my lisp-controller object and not care about understanding what goes on behind the scenes. To accommodate those people I have duplicated the examples provided in section 5 above within the Interface Builder with CCL Tutorial:

...ccl/contrib/krueger/Interface Projects/Documentation/InterfaceBuilderWithCCLTutorial2.1.

Before describing the Lisp code that underlies the lisp-controller object it is best of the reader experiments with it first in order to understand its capabilities. Use the instructions provide in this document to install and run some of the examples to get a better feel for how this works. Go ahead, I'll wait here while you go do that.

Right. Now that you have a better feel for what the controller can do we'll take a look at how all that is accomplished. I will say just a little bit about the Objective-C plugin module, so you understand the approach that I took. A "normal" plugin would be defined by specifying all of the controller object's behavior in Objective-C. That would be turned into a Framework that would then be loaded into any program that desired to use the plugin objects. But in our case we want to specify the plugin's functionality using Lisp and not via Objective-C. So what I actually did was to provide just enough functionality in Objective-C so that IB could create instances of our lisp-controller plugin object. Luckily for us, what actually happens is that when we select a lisp controller object, add it to the interface, and then save it away in a NIB file, IB will "encode" the object into an archive that is part of the NIB file and then reconstitute it by "decoding" the archive at runtime. So at runtime we can specify a Lisp definition for the "LispController" class and as long as it knows how to create itself by decoding the archive, everything will work out fine.

If you look at the Xcode project you'll see a minimal implementation of the LispController class that has slots for the various parameters that we want to set within IB and knows how to archive itself and recreate itself from an archive and that's about it. All of the real functionality is implemented in the Lisp version of that class and that's what we will look at next.

Lisp Controller Class

Open up the file "ip:Utilities;lisp-controller.lisp" and follow along with the description below. I'll warn you that the functionality described here is somewhat complex. Consequently we will jump around within the Lisp code quite a bit in order to hopefully make the explanation comprehensible. We'll first discuss the basic data structures, then look at how they are initialized, and finally look at how the Lisp controller responds to request made by the interface object.

Let's start with the lisp-controller class itself:

```
(defclass lisp-controller (ns:ns-object)
  ((root :accessor root)
   (root-type :accessor root-type)
   (gen-root :accessor gen-root)
   (objects :accessor objects)
   (types :reader types)
   (reader-func :accessor reader-func)
   (writer-func :accessor writer-func)
   (count-func :accessor count-func)
   (select-func :accessor select-func)
   (edited-func :accessor edited-func)
   (added-func :accessor added-func)
   (removed-func :accessor removed-func)
   (delete-func :accessor delete-func)
   (add-child-func :accessor add-child-func)
   (children-func :accessor children-func)
   (type-info :accessor type-info)
   (obj-wrappers :accessor obj-wrappers)
   (column-info :accessor column-info)
   (nib-initialized :accessor nib-initialized)
   (view-class :accessor view-class)
   (can-remove :foreign-type #>BOOL :accessor can-remove)
   (can-insert :foreign-type #>BOOL :accessor can-insert)
   (can-add-child :foreign-type #>BOOL :accessor can-add-child)
   (owner :foreign-type :id :accessor owner)
   (view :foreign-type :id :accessor view))
  (:metaclass ns:+ns-object))
```

If you looked at the Objective-C version of this class that was used to build the IB plugin, you will see that although there are some similarities in naming, they are very different objects. Thats not a problem because the Objective-C version is used when objects are instantiated during an IB session and this Lisp version is used at application runtime. As long as they both understand a common archived format, there is no problem.

The root slot will contain the main Lisp object to be displayed. Typically this will be an array or list or hash-table, but it can be an object of some sort as well. If you worked through the examples, the meaning of this value should be clear to you.

The root-type is the type of the root object. If the lisp-controller is configured to generate a root object, then it will use this root-type to find the appropriate initialization form (as specified in IB).

The gen-root slot is a boolean value that specifies whether or not the lisp-controller should generate the root object or whether one will be provided by the implementor in some way. If the latter was specified in IB, the lisp-controller will wait until the root slot has been set before displaying anything.

The objects slot contains a cache of the immediate children of the root object. This exists strictly to minimize the time required to respond to a view object's request for information that should be put into the table.

The types slot contains an ordered list of all the types that were specified within IB plus some default types like list and array that are always acceptable. The order is determined by sub-type relationships. This is important when the lisp-controller needs to determine what type of object it has been given so that it can use an appropriate function to find its children or initialize an instance of it.

The reader-func, writer-func, count-func, delete-func, add-child-func, and children-func slots may each contain an override function that is used if the Lisp programmer wants to replace lisp-controller functionality with some alternative. Although the default lisp-controller behavior is fairly comprehensive, there may well be situations where the developer will want to provide a more customized alternative. These are all specified in IB when configuring the lisp-controller object.

The select-func, edited-func, added-func, and removed-func slots may each contain a notification function that is invoked whenever the lisp controller does the corresponding action. These could be used for whatever purposes the developer might have.

The type-info slot will contain an instance of an "assoc-array" and it's probably worth a small digression at this point to talk about what that is because it plays a fairly prominent role in the lisp-controller code.

Assoc-arrays are defined in the file "ip:Utilities;assoc-array.lisp. It is an object that implements a sparse multi-dimensional associative array that can be indexed by arbitrary lisp-objects. I've found this to be a fairly useful tool to use when organizing data. Without something similar to this, it would be necessary to define several additional classes and accessors to maneuver between them. I think their utility will become more clear as we go through the rest of the lisp-controller code.

Assoc-arrays were discussed more in the LispController Reference document, so I'm not going to describe them in any detail here. The reader is invited to read through the code to understand it. I will only say that the implementation is done using nested hash-tables and is fairly straight-forward.

So back to the lisp-controller object ...

The obj-wrappers slot is used when displaying things in an NSOutlineView. An NSOutlineView keeps track of objects at each level of the hierarchy and as each one is expanded it will request its children so that they can be displayed as well. Unfortunately, those must be Objective-C objects. The lisp-controller arranges to wrap each lisp object inside an Objective-C object just so that it can be delivered to the NSOutlineView. If a displayed object is then contracted and re-expanded, the same Objective-C objects must be used for its children. So in the obj-wrappers slot we keep an assoc-array that contains the association between an arbitrary lisp object and a corresponding Objective-C "wrapper" object. This could be a simple hash-table, but an assoc-array was used to make the code look similar to other usage.

The column-info slot contains another assoc-array object that maintains information about each column in the associated NSTableView or NSOutlineView object. This is initialized after the NIB has been loaded by looking at the table and extracting information about each column. As you saw when you went through the examples, some of this information is in the form of lisp constructs that are interpreted in different ways, depending on the type of Lisp object being displayed. We will expand on this idea later when we examine how this slot is initialized.

The nib-initialized slot is a boolean that tells us that the NIB file has been completely initialized. This is necessary to avoid problems during the brief period after the lisp-controller object has been created and before it has been completely initialized by the NIB loading mechanism.

The view-class slot contains the class of the associated NSTableView or NSOutlineView object. Slightly different functionality is required for the two, so the value is saved here and referenced as needed.

The can-add, can-remove, and can-add-child slots are foreign slots that may be bound to by interface elements to indicate whether they should be enabled. For example, if the lisp-controller is in a state where the interface should permit the addition of a child to the root object, then the value of can-add will contain #$YES. If some interface button is conditionally enabled via a link to the canAdd path of the lisp-controller object, then it will be enabled. If the state is such that addition should not be allowed (as for example when no parent is currently selected), then the slot will contain #$NO and the button would be disabled. Similarly, the can-remove and can-add-child slots indicate whether it is currently valid to remove the currently selected object and add a child to the currently selected object, respectively.

The owner slot will typically contain a reference to the FileOwner object that was used to initialize the NIB file. This must be set in IB by ctrl-dragging from the lisp-controller object to the FileOwner object and setting the owner outlet. The only time this is used is when calling notification functions. This value of the owner slot is passed as one of the arguments. This may be useful if a developer defines a single notification function that is to be used by several different windows. Having the value of the owner can be used to disambiguate where the call originates.

Finally, the view slot contains a reference to the NSTableView or NSOutlineView object that is associated with this controller. This is set in IB by ctrl-dragging from the lisp-controller to the view object and setting the view outlet.

Lisp Controller Initialization

These are the fundamental data structures of the lisp-controller. Next we will look at how all of their values are initialized. As previously mentioned, many of these values are specified in IB, some for the lisp-controller object and some for the view object. We'll first look at how those specified for the lisp-controller object are extracted from the archived object that was part of the NIB.

When the NIB file is loaded, space is allocated for a lisp-controller object and the Objective-C function #/initWithCoder is called to initialize it. We'll get to the details of our lisp implementation of that method in just a bit, but the first thing it does is call another Objective-C method called #/init. this is mostly done so that if a lisp-controller object is created by calling (make-instance 'lisp-controller) from some Lisp context that a meaningful object is constructed. Normally that will not be the case, but it was useful to have around while I was debugging the code. That #/init method is shown below:

```
(objc:defmethod (#/init :id)
                ((self lisp-controller))
  ;; need to do this to initialize default values that are needed when
  ;; this object is instantiated from Objective-C runtime
  (unless (slot-boundp self 'nib-initialized)
    (setf (nib-initialized self) nil))
  (unless (slot-boundp self 'select-func)
    (setf (select-func self) nil))
  (unless (slot-boundp self 'edited-func)
    (setf (edited-func self) nil))
  (unless (slot-boundp self 'added-func)
    (setf (added-func self) nil))
  (unless (slot-boundp self 'removed-func)
    (setf (removed-func self) nil))
  (unless (slot-boundp self 'add-child-func)
    (setf (add-child-func self) nil))
  (unless (slot-boundp self 'delete-func)
    (setf (delete-func self) nil))
  (unless (slot-boundp self 'reader-func)
    (setf (reader-func self) nil))
  (unless (slot-boundp self 'writer-func)
    (setf (writer-func self) nil))
  (unless (slot-boundp self 'count-func)
    (setf (count-func self) nil))
  (unless (slot-boundp self 'objects)
    (setf (objects self) nil))
  (unless (slot-boundp self 'root)
    ;; note that we have to set root slot to avoid
    ;; calling accessor functions. This is a
    ;; special case and the only place where we
    ;; want to set the root to something that
    ;; doesn't match the root-type specified in IB
    (setf (slot-value self 'root) nil))
  (unless (slot-boundp self 'root-type)
    (setf (root-type self) t))
  (unless (slot-boundp self 'types)
    (setf (types self) nil))
  (unless (slot-boundp self 'type-info)
    (setf (type-info self) (make-instance 'assoc-array :rank 2)))
  ;; Now set up some default type info for standard types
  ;; These can be overridden in the lisp-controller setup
  ;; within Interface Builder.
  ;; Typically users would define their own types and
  ;; specify values for them rather than overriding these
  ;; but it is permissable to do so.
  (setf (assoc-aref (type-info self) 'hash-table :child-key) #'children)
  (setf (assoc-aref (type-info self) 'ht-entry :child-key) #'children)
  (setf (assoc-aref (type-info self) 'list :child-key) #'children)
  (setf (assoc-aref (type-info self) 'vector :child-key) #'children)
  (setf (assoc-aref (type-info self) 'hash-table :child-setf-key) #'(setf children))
  (setf (assoc-aref (type-info self) 'ht-entry :child-setf-key) #'(setf children))
  (setf (assoc-aref (type-info self) 'list :child-setf-key) #'(setf children))
  (setf (assoc-aref (type-info self) 'vector :child-setf-key) #'(setf children))
  (setf (assoc-aref (type-info self) 'hash-table :child-type) 'ht-entry)
  (setf (assoc-aref (type-info self) 'list :child-type) 'list)
  (setf (assoc-aref (type-info self) 'vector :child-type) 'vector)
  (setf (assoc-aref (type-info self) 'hash-table :initform)
        '(make-hash-table))
  (setf (assoc-aref (type-info self) 'list :initform)
        nil)
  (setf (assoc-aref (type-info self) 'vector :initform)
        '(make-array '(10) :adjustable t :fill-pointer 0 :initial-element nil))
  (unless (slot-boundp self 'obj-wrappers)
    (setf (obj-wrappers self) (make-instance 'assoc-array :rank 1)))
  self)
```

The first part of the #/init method just sets up default values for simple slots. In most cases that value is nil. The first somewhat interesting thing is when we set the default value for the type-info slot. We set a value that is a two-dimensional assoc-array. The first dimension will be indexed by a Lisp type. The second dimension will be indexed by a keyword value that specifies what type of information is contained in the value. Various default values are then put into the assoc-array for the standard Lisp types list, vector, and hash-table. The type ht-entry is defined specifically to handle hash-table children and will be discussed later. The user is free to override these defaults in IB by specifying other values. Normally the :child-key, :child-setf-key, and :child-type should be pretty good defaults, but the :initforms would be overridden in IB if these types were used.

Note that the column-info slot is not initialized at this point. We should not do that because we do not yet know that the view slot is valid. So obviously we cannot go look at it. That will happen sometime later after we have been notified that the NIB was initialized by a call to #/awakeFromNib.

The normal way that lisp-controller objects will be created is via a call to #/initWithCoder:. As previously mentioned, this first calls the #/init method. Once this basic, default initialization has been completed we begin to extract information from the archive by making calls to the decoder object that is given to us.

```
(objc:defmethod (#/initWithCoder: :id)
                ((self lisp-controller) (decoder :id))
  ;; This method is called when the Nib is loaded and provides values defined
  ;; when the NIB was created
  (#/init self)
  (with-slots (reader-func writer-func count-func select-func edited-func
               add-child-func root-type gen-root added-func removed-func
               children-func type-info delete-func) self
    (let ((type-info-array (#/decodeObjectForKey: decoder #@"typeInfo")))
      (dotimes (i (#/count type-info-array))
        ;; for each type specified in IB by the user
        (let* ((row-array (#/objectAtIndex: type-info-array i))
               (ns-str-type-name (#/objectAtIndex: row-array 0))
               (type-name (nsstring-to-sym ns-str-type-name))
               (child-type (nsstring-to-sym (#/objectAtIndex: row-array 1)))
               (child-func-str (ns-to-lisp-string (#/objectAtIndex: row-array 2)))
               (child-func (find-func child-func-str))
               (reader-sym (and child-func (function-name child-func)))
               (writer-form `(setf (,reader-sym thing) new-val))
               (child-writer-func (and child-func
                                       (valid-setf-for writer-form)
                                       (eval `(function (lambda (new-val thing)
                                                          ,writer-form))))))
          (when child-type
              (setf (assoc-aref type-info type-name :child-type) child-type))
          (when child-func
            (setf (assoc-aref type-info type-name :child-key) child-func))
          (when child-writer-func
              (setf (assoc-aref type-info type-name :child-setf-key) child-writer-func)))))
    (let ((initform-array (#/decodeObjectForKey: decoder #@"initforms")))
      (dotimes (i (#/count initform-array))
        ;; for each initform specified in IB by the user
        (let* ((row-array (#/objectAtIndex: initform-array i))
               (ns-str-type-name (#/objectAtIndex: row-array 0))
               (type-name (nsstring-to-sym ns-str-type-name))
               (initform (ns-to-lisp-object t (#/objectAtIndex: row-array 1))))
          (when initform
              (setf (assoc-aref type-info type-name :initform) initform)))))
    (let ((sort-info-array (#/decodeObjectForKey: decoder #@"sortInfo")))
      (dotimes (i (#/count sort-info-array))
        ;; for each sort predicate and key specified in IB by the user
        (let* ((row-array (#/objectAtIndex: sort-info-array i))
               (ns-str-type-name (#/objectAtIndex: row-array 0))
               (type-name (nsstring-to-sym ns-str-type-name))
               (sort-key (nsstring-to-func (#/objectAtIndex: row-array 1)))
               (sort-pred (nsstring-to-func (#/objectAtIndex: row-array 2))))
          (when sort-pred
            (setf (assoc-aref type-info type-name :sort-pred) sort-pred))
          (when sort-key
            (setf (assoc-aref type-info type-name :sort-key) sort-key)))))
    (setf root-type (nsstring-to-sym (#/decodeObjectForKey: decoder #@"rootType")))
    (setf (types self) (delete-duplicates (list* root-type
                                                  'ht-entry
                                                  (mapcar-assoc-array #'identity type-info))))
    (setf reader-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"readerFunc")))
    (setf writer-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"writerFunc")))
    (setf count-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"countFunc")))
    (setf select-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"selectFunc")))
    (setf edited-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"editedFunc")))
    (setf added-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"addedFunc")))
    (setf removed-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"removedFunc")))
    (setf delete-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"deleteFunc")))
    (setf add-child-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"addChildFunc")))
    (setf children-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"childrenFunc")))
    (setf gen-root (#/decodeBoolForKey: decoder #@"genRoot")))
  self)
```

You can think of an archive as something like a Lisp hash-table. That is, you can specify a key and some data to associate with it. That's what our Xcode-defined LispController Objective-C method "encodeWithCoder:" did when it was called by IB. Here we are just doing the reverse by specifying a key and getting back the Objective-C object that was archived. We then convert that object into something that is more usable within Lisp.

In the IB version of the LispController class I defined three arrays that corresponded to the three tables that you see when you inspect a LispController object in IB. That made it pretty easy to implement. But in Lisp I wanted to pull all of this information into the single assoc-array that is the value of the type-info slot. No problem, we simply extract each of the NSArray objects from the archive and then iterate through them to extract relevant information. This is then put into the type-info assoc-array via calls of the form

(setf (assoc-aref type-info type-name <some keyword>) converted-value)

In many cases the value that we originally put into the NSArray was a simple NSString, but in Lisp we want to treat it as either a lisp form, symbol, string, or the name of a function. To make this easy, several utility functions were defined to convert appropriately. The function nsstring-to-sym creates a Lisp symbol. The function nsstring-to-func returns a Lisp function (if it exists). The function ns-to-lisp-string converts an NSString to a Lisp string. Finally ns-to-lisp-object will take an NSString and effectively do a read-from-string to turn it into an arbitrary lisp object. This is useful for things like initialization forms.

Slots for notification and override slots are similarly populated from archived values. Empty NSStrings are converted to nil values in functions like nsstring-to-func.

Once this function is complete, all of the values specified for the LispController in IB have been transferred to the corresponding runtime instance of lisp-controller.

There is additional initialization that occurs once the NIB has been completely loaded and all the objects are linked together. At that time the lisp-controller object can examine the associated view object to retrieve additional information that the developer provided when the view's column IDs were set. The #/awakeFromNib method is called to inform the lisp-controller that the NIB was completely loaded, so we define a function to manage that additional initialization:

```
(objc:defmethod (#/awakeFromNib :void)
                ((self lisp-controller))
  (setf (nib-initialized self) t)
  (unless (eql (view self) (%null-ptr))
    (setf (view-class self) (#/class (view self)))
    (init-column-info self (view self))
    (when (gen-root self)
      ;; create the root object
      (setf (root self) (new-object-of-type self (root-type self))))
    (when (objects self)
      (setup-accessors self))))
```

This function sets the view-class slot and then initializes the column-info slot by calling init-column-info:

```
(defmethod init-column-info ((self lisp-controller) (view ns:ns-table-view))
  (with-slots (column-info) self
    (let* ((tc-arr (#/tableColumns view))
           (col-obj nil)
           (idc nil)
           (col-count (#/count tc-arr)))
      (unless tc-arr
        (ns-log "#/tableColumns returned nil for view")
        (return-from init-column-info))
      (setf column-info (make-instance 'assoc-array :rank 2))
      (dotimes (i col-count)
        (setf col-obj (#/objectAtIndex: tc-arr i))
        (setf (assoc-aref column-info col-obj :col-indx) i)
        (setf (assoc-aref column-info i :col-obj) col-obj)
        (setf idc (ns-to-lisp-string (#/identifier col-obj)))
        (setf (assoc-aref column-info col-obj :col-string) idc)
        (setf (assoc-aref column-info col-obj :col-val)
              (read-from-string idc nil nil))
        (setf (assoc-aref column-info col-obj :col-title)
              (ns-to-lisp-string (#/title (#/headerCell col-obj))))
        ;; find any formatter attached to the data cell for this column and
        ;; use info from it to help us translate to and from lisp objects
        ;; appropriately
        (let ((formatter-object (#/formatter (#/dataCell col-obj))))
          (unless (eql formatter-object (%null-ptr))
            (cond ((typep formatter-object 'ns:ns-date-formatter)
                   (setf (assoc-aref column-info col-obj :col-format) :date))
                  ((typep formatter-object 'ns:ns-number-formatter)
                   (cond ((#/generatesDecimalNumbers formatter-object)
                          (let ((dec-digits (#/maximumFractionDigits formatter-object)))
                            (setf (assoc-aref column-info col-obj :col-format)
                                  (list :decimal dec-digits))))
                         (t
                          (setf (assoc-aref column-info col-obj :col-format)
                                :number))))))))))
```

First this method gets all the table's columns from the view. This is returned in an NSArray object. That array is traversed and information about that column is put into the assoc-array that is in the column-info slot of the lisp-controller. That assoc-array is a two-dimensional assoc-array that is indexed by the column object pointer and a keyword that indicates the type of column information that is being accessed. The value is some Lisp value that is appropriate for that type of information. The information kept includes the column number (:col-indx keyword as the second index), a reference to the Objective-C column object (:col-obj keyword as the second index), the column's identifier as an NSString (:col-string keyword as the second index), the Lisp object that results from reading the identifier string (:col-val keyword as the second index), and the title of the column that was specified in IB as a Lisp string (:col-title keyword as the second index).

In addition this method will examine any data formatters that were attached to the column to see if they can be used to determine how Lisp data should be packaged when given to the view and how Lisp data should be extracted from objects that are given to the Lisp code. That information is stored using the :col-format keyword as the second index to the assoc-array.

The next thing that the #/awakeFromNib method does is to initialize the root object if that has been specified. It does that by calling the new-object-of-type method. We'll defer discussion of that method until later when we consider how new objects are created.

Finally the #/awakeFromNib method calls setup-accessors to initialize appropriate accessor methods for each column for every possible type that might be contained in any row of the table. The general idea is that if we can find a way to apply the accessor that was specified for the column to the object displayed in any particular row, then we will do that to retrieve the Lisp data to be displayed in that column for that row.

```
(defmethod setup-accessors ((self lisp-controller))
  ;; This method must be called to initialize the column value
  ;; accessor functions for a lisp-controller.
  ;; It is called after NIB loading has been done.
  (with-slots (reader-func column-info type-info types) self
    (unless reader-func
      (dolist (col (mapcar-assoc-array #'identity column-info))
        (let ((col-id (assoc-aref column-info col :col-val)))
          (dolist (typ types)
            (setf (assoc-aref type-info typ col)
                  (reader-writer-pair typ col-id)))))))
```

First setup-accessors iterates across all columns. For each column it iterates across all of the known types (both default and user-specified) and tries to makes sense of the column identity as a read accessor for that type. If it can, then it also tries to construct a corresponding write accessor. Those accessors are saved in the type-info assoc-array indexed by the type and column object reference as a dotted pair. Let's take a look at how those accessors are created:

```
(defun reader-writer-pair (typ col-val)
  (let* ((reader-form nil)
         (writer-form nil))
    (cond ((null col-val)
           ;; reader justs return the object itself
           ;; leave the writer-form null
           (setf reader-form 'row))
          ((and (eq col-val :key) (eq typ 'ht-entry))
           ;; used for the key value in a hash table
           (setf reader-form '(ht-key row))
           (setf writer-form '(setf (ht-key row) new-val)))
          ((and (eq col-val :value) (eq typ 'ht-entry))
           ;; used for the value in a hash table
           (setf reader-form '(ht-value row))
           (setf writer-form '(setf (ht-value row) new-val)))
          ((eq col-val :row)
           (setf reader-form 'row)
           (setf writer-form '(setf row new-val)))
          ((numberp col-val)
           (cond ((subtypep typ 'vector)
                  (setf reader-form `(aref row ,col-val))
                  (setf writer-form `(setf (aref row ,col-val) new-val)))
                 ((subtypep typ 'list)
                  (setf reader-form `(nth ,col-val row))
                  (setf writer-form `(setf (nth ,col-val row) new-val)))
                 ((eq typ 'ht-entry)
                  ;; Index if the value is a sequence
                  (setf reader-form `(when (typep (ht-value row) 'sequence)
                                       (elt (ht-value row) ,col-val)))
                  (setf writer-form `(when (typep (ht-value row) 'sequence)
                                       (setf (elt (ht-value row) ,col-val) new-val))))
                 ((subtypep typ 'hash-table)
                  ;; use the number as a key into the hash table and return the value
                  (setf reader-form `(gethash ,col-val row))
                  (setf writer-form `(setf (gethash ,col-val row) new-val)))
                 (t
                  ;; index if row is any other type of sequence
                  (setf reader-form `(when (typep row 'sequence)
                                       (elt row ,col-val)))
                  (setf writer-form `(when (typep row 'sequence)
                                       (setf (elt row ,col-val) new-val))))))
          ((and (symbolp col-val) (fboundp col-val))
           (cond ((eq typ 'ht-entry)
                  ;; Assume the function applies to the value
                  (setf reader-form `(,col-val (ht-value row)))
                  (when (valid-setf-for reader-form)
                    (setf writer-form `(setf (,col-val (ht-value row)) new-val))))
                 (t
                  (setf reader-form `(,col-val row))
                  (when (valid-setf-for reader-form)
                    (setf writer-form `(setf (,col-val row) new-val))))))
          ((symbolp col-val)
           (cond ((subtypep typ 'hash-table)
                  ;; use the symbol as a key into the hash table and return the value
                  (setf reader-form `(gethash ,col-val row))
                  (setf writer-form `(setf (gethash ,col-val row) new-val)))))
          ((and (consp col-val) (eq (first col-val) 'function))
           (let ((col-val (second col-val)))
             (when (and (symbolp col-val) (fboundp col-val))
               (cond ((eq typ 'ht-entry)
                      ;; Assume the function applies to the value
                      (setf reader-form `(,col-val (ht-value row)))
                      (when (valid-setf-for reader-form)
                        (setf writer-form `(setf (,col-val (ht-value row)) new-val))))
                     (t
                      (setf reader-form `(,col-val row))
                      (when (valid-setf-for reader-form)
                        (setf writer-form `(setf (,col-val row) new-val)))))))
          ((consp col-val)
           ;; accessors are lisp forms possibly using keywords :row, :key, and :value
           ;; which are replaced appropriately
           (setf reader-form (nsubst 'row ':row
                                      (nsubst '(ht-key row) :key
                                              (nsubst '(ht-value row) :value
                                                      col-val))))
           (when (valid-setf-for reader-form)
             (setf writer-form `(setf ,col-val new-val)))))
    (when *lisp-controller-debug*
      (ns-log (format nil "Reader-form: ~s~%Writer-form: ~s" reader-form writer-form)))
    (cons (and reader-form
               (eval-without-errors `(function (lambda (row) ,reader-form))))
          (and writer-form
               (eval-without-errors `(function (lambda (new-val row) ,writer-form)))))))
```

If no column-identifier was specified in IB, then the row-object itself will be displayed in the column. This might be useful for single column tables or when the row-object can itself be used as an identifier of sorts for the row (i.e. printing the object results in some meaningful identifier). No write accessor is created in this case.

If the root object being displayed is a hash-table, then a number of special things are done to represent it correctly. Essentially the hash-table is converted into a list of ht-entry objects, where each such object represents a key-value pair. Each row-object displayed would then be an ht-entry object. The developer doesn't really need to worry about that. The only thing they need to know is that they can use the keyword :key to retrieve the key from the key-value pair and the keyword :value to retrieve the value. If a column identifier is either of these two keywords then the read and write accessors are set to forms that read and write the ht-entry slots respectively. The ht-entry objects have functionality that reflect changed keys and values back to the original hash-table. For example, when the user edits a column value for which the identifier :value has been set, then the ht-entry setf function that is called will also modify the corresponding value in the root hash-table. If you are curious about exactly how this works, examine the methods for the ht-entry class.

If the keyword :row is used, it refers to the row-object itself. If used alone as the column identifier (i.e. not embedded in some other form) then the row object is displayed in that column, just as it would be if no column identifier was specified. But in this case we assume that the developer is doing so knowledgeably, so we allow for the possibility that setting the row object itself to a new value may be possible. So we will create a setf form and if that turns out to be a valid form then it will be used to write a new value.

Next we consider that case where the column identifier is a number. Generally speaking we will use that number as an index into the row-object. For sequences like lists and vectors that has an obvious interpretation. If the row-object is an ht-entry (i.e. it represents a key/value pair from a hash-table), then we will assume that the value is some type of sequence and index into that. It is possible that the row-object could itself be a hash-table. In that case we treat the number as a key into that hash-table and return the result.

Next we consider column identifiers that are Lisp symbols that have a function binding. If the row-object is an ht-entry we will apply that function to the value of the key/value pair. Otherwise we will apply the function to the row-object itself.

If the column identifier is a symbol that does not have a function binding then the only likely use is as a key into a hash-table. So if the row-object is a hash-table we will use it that way. For any other type it will be ignored. Note that this can be a source of error if a symbol is intended to specify a function, but is spelled incorrectly or does not specify a package correctly. In that case it would be silently ignored.

If the column identifier was specified using something of the form #'func or equivalently (function func) then it is used in the same way as a symbol with a function binding.

If the column identifier is any other sort of list, then we assume that it is an accessor form of some sort. The keywords :row, :key, and :value can be used within that form to refer to the row-object or the key of an ht-entry row-object or the value of an ht-entry row-object, respectively. We substitute for those keywords to create read and write accessors.

In all cases we validate that the write form is valid before saving it. This is done by calling the valid-setf-for function shown next.

```
(defun valid-setf-for (read-form)
  (multiple-value-bind (a b c func-form d) (get-setf-expansion read-form)
    (declare (ignore a b c d))
    (or (not (eq (first func-form) 'funcall))
        ;; this must be a built-in function, so assume setf works
        ;; otherwise make sure the function name specified is fboundp
        (let ((func-name (second (second func-form))))
          (and (typep func-name 'function-name) (fboundp func-name))))))
```

This checks to make sure that the function that would be called by expanding the setf form is either a built-in function or is a user-specified function that has a function binding.

The last thing that the reader-writer-pair function does is evaluate the forms. Any errors that occur while evaluating a form are ignored and nil is returned for that evaluation. This hopefully results in accessors that can be funcalled as needed.

Lisp Controller Handling of NSTableView Calls

At this point we have examined all of the initialization functions and can now consider what happens when the end-user interacts with the user interface and objects there subsequently make calls to the lisp-controller. Some calls are made by virtue of the lisp-controller being the data-source for an NSTableView or NSOutlineView. Others are made because the lisp-controller is the delegate object for the view. Both links should always be made between the view and the lisp-controller. We will examine the calls made by each of these types of views.

An NSTableView calls the methods
    #/numberOfRowsInTableView:
    #/tableView:objectValueForTableColumn:row:
    #/tableView:setObjectValue:forTableColumn:row:
on the lisp-controller as its data source. It calls the methods
    #/tableView:shouldEditTableColumn:row:
    #/tableViewSelectionDidChange:
on the lisp-controller as the view's delegate.

The #/numberOfRowsInTableView: method calls the specified override method if it exists. Otherwise it simply returns the length of the objects slot. You will recall that this contains the children of the root object. This tells the view how many rows should be displayed.

The #/tableView:objectValueForTableColumn:row: method returns an appropriate Objective-C object to be displayed in the column and row specified. Here is the method:

```
(objc:defmethod (#/tableView:objectValueForTableColumn:row: :id)
                ((self lisp-controller)
                 (tab :id)
                 (col :id)
                 (row #>NSInteger))
  (declare (ignore tab))
  (let ((ns-format (assoc-aref (column-info self) col :col-format)))
    (lisp-to-ns-object (col-value self (elt (objects self) row) col) ns-format)))
```

This is fairly simple, but calls two helper functions, lisp-to-ns-object and col-value, that provide additional functionality. We will look at those next. The col-value method finds the appropriate lisp value to be displayed:

```
(defmethod col-value ((self lisp-controller) obj col-obj)
  ;; Get the lisp value for some column for an object
  ;; return "" if there isn't one so the display doesn't
  ;; have "nil" for columns without values.
  (let* ((obj-type (controller-type-of self obj))
         (reader-func (or (reader-func self)
                          (car (assoc-aref (type-info self) obj-type col-obj)))))
    (if reader-func
      (funcall reader-func obj)
      "")))
```

The obj parameter is the object being displayed in the selected row and the col-obj parameter is the Objective-C column object for the selected column. If a reader override function has been specified, then it is used. Otherwise the reader functions that were constructed as part of the initialization are used. The choice of which function to use depends on the type of the row object and the column where it will be displayed.

After the lisp object has been retrieved, it must be converted to an appropriate Objective-C object for display. That is the role of the lisp-to-ns-object function which is defined in ip:Utilities;ns-object-utils.lisp:

```
(defun lisp-to-ns-object (lisp-obj &optional (ns-format nil))
  ;; convert an arbitrary lisp object to an appropriate NSObject so
  ;; that it can be displayed someplace
  (cond ((ccl::objc-object-p lisp-obj)
         ;; it's already an NSObject so just return it
         lisp-obj)
        ((eq ns-format :date)
         ;; assume lisp-obj is an integer representing a lisp date
         (lisp-to-ns-date lisp-obj))
        ((and (consp ns-format) (eq (first ns-format) :decimal))
         (cond ((typep lisp-obj 'fixnum)
                (lisp-to-ns-decimal lisp-obj :decimals (second ns-format)))
               ((typep lisp-obj 'number)
                (lisp-to-ns-decimal (round (* (expt 10 (second ns-format)) lisp-obj))
                                    :decimals (second ns-format)))
               (t
                (lisp-to-ns-decimal 0 :decimals (second ns-format)))))
        ((integerp lisp-obj)
         (#/numberWithInt: ns:ns-number lisp-obj))
        ((typep lisp-obj 'double-float)
         (#/numberWithDouble: ns:ns-number lisp-obj))
        ((floatp lisp-obj)
         (#/numberWithFloat: ns:ns-number lisp-obj))
        ((null lisp-obj)
         #@"")
        (t
         (lisp-to-temp-nsstring (if (stringp lisp-obj)
                                    lisp-obj
                                    (format nil "~s" lisp-obj))))))
```

This function uses the ns-format argument, if provided, to guide the decision. If the Lisp object is already an Objective-C object, then it is just returned. If we know from the ns-format argument that it is a date, then an NSDate object is created (see the lisp-to-ns-date function defined in ip:Utilities;date.lisp). If we learned from the formatter attached to a column that the use of NSDecimalNumber objects is desired and if the Lisp object is a fixnum, then we assume that the Lisp object uses the format defined in decimal.lisp which was discussed for Project 6. If the Lisp object is some other kind of number, then we round it appropriately and convert it to an NSDecimalNumber. In the absence of an ns-format value Lisp numbers are converted to an appropriate Objective-C numeric object. In all other cases we just print the Lisp object to a string that is converted to an NSString for display.

The next method needed to support NSTableViews is #/tableView:setObjectValue:forTableColumn:row:. This is called when a user modifies the value displayed within some row/column. The implementation of this is:

```
(objc:defmethod (#/tableView:setObjectValue:forTableColumn:row: :void)
                ((self lisp-controller)
                 (tab :id)
                 (val :id)
                 (col :id)
                 (row #>NSInteger))
  ;; We let the user edit the table and something was changed
  ;; try to convert it to the same type as what is already in that
  ;; position in the objects.
  (declare (ignore tab))
  (let* ((row-obj (elt (objects self) row))
         (old-obj (col-value self row-obj col))
         (ns-format (assoc-aref (column-info self) col :col-format))
         (new-val (ns-to-lisp-object old-obj val ns-format)))
    (if (writer-func self)
      (funcall (writer-func self)
               new-val
               (root self)
               row
```

```
              (assoc-aref (column-info self) col :col-val))
      (set-col-value self row-obj col new-val))
    (when (edited-func self)
      (let* ((row-obj (object-at-row self row))
             (edited-obj (if (typep row-obj 'ht-entry)
                           (list (ht-key row-obj) (ht-value row-obj))
                           row-obj)))
        (funcall (edited-func self)
                 (owner self)
                 self
                 (root self)
                 row
                 (assoc-aref (column-info self) col :col-indx)
                 edited-obj
                 old-obj
                 new-val)))
    ;; re-sort and reload the table
    ;; unfortunately we probably have to do this for every change since
    ;; we don't know what affects the sort order
    (sort-sequence self (objects self))
    (#/reloadData (view self))))
```

This function must convert the new value, represented as an Objective-C object, into a usable Lisp object. The tricky decision is what sort of object that should be. This is done by the ns-to-lisp-object function which makes use of whatever information is available. That includes information from any formatter that has been attached to the field (we retrieved that information as part of lisp-controller initialization described earlier) and the type of Lisp object that was displayed in the field to begin with. In general we try to convert back to the same type as was originally there, but if the user really wants to replace a number with a symbol or string, we allow that to happen. Note that the use of a formatter within IB can prevent the user from changing to an invalid type and that is the mechanism that should be used to force the user to adhere to a particular type when editing. Let's take a closer look at the ns-to-lisp-object function:

```
(defun ns-to-lisp-object (old-lisp-obj ns-obj &optional (ns-format nil))
  ;; convert an arbitrary NSObject object to an appropriate lisp object.
  ;; Often done so that it can replace the old-lisp-obj when edited
  ;; An empty string @"" returns nil if old-lisp-obj is not a string
  (cond ((ccl::objc-object-p old-lisp-obj)
         ;; the old value was an NSObject so just return the new value
         ns-obj)
        ((typep ns-obj 'lisp-ptr-wrapper)
         ;; just strip the wrapper and return the original object
         (lpw-lisp-ptr ns-obj))
        ((typep ns-obj 'ns:ns-decimal)
         (if (floatp old-lisp-obj)
           ;; convert the decimal to a float
           (#/doubleValue ns-obj)
           ;; otherwise convert it to an appropriate lisp integer with assumed
           ;; decimals (see ip;Utilities;decimal.lisp)
           (if (eq (first ns-format) :decimal)
             (lisp-from-ns-decimal ns-obj :decimals (second ns-format))
             (lisp-from-ns-decimal ns-obj))))
        ((typep ns-obj 'ns:ns-number)
         (read-from-string (ns-to-lisp-string (#/descriptionWithLocale: ns-obj (%null-ptr)))
                           nil nil))
        ((typep ns-obj 'ns:ns-date)
         (ns-to-lisp-date ns-obj))
        (t
         (let ((str (ns-to-lisp-string ns-obj)))
           (if (stringp old-lisp-obj)
             str
             (read-from-string str nil nil))))))
```

If the old lisp object, i.e. the one that was retrieved to be displayed initially, was already an Objective-C object, then we will simply return the new Objective-C object. This allows Lisp developers to display and store Objective-C instances if they desire. When we discuss the functionality needed to support NSOutlineViews we will see that it sometimes we just want to package up Lisp objects in an Objective-C wrapper and unwrap them when then are sent back to us. That wrapper is a lisp-ptr-wrapper instance. If we get one of those, the Lisp object that it encapsulates is returned.

If the value is an NSDecimal, then we convert it to either a floating point value (if the old value was a float) or to the internal scaled decimal fixnum format implemented in ip;Utilities;decimal.lisp and described previously in this document.

Any other type of NSNumber is just converted by reading from its string representation. NSDate objects are converted to Lisp date integers. Anything else that we get is converted to a Lisp string. If the previous value was a string it is just left as a Lisp string. Otherwise we read from that string and return whatever was read. In this way we can retrieve any Lisp form that could be typed into a Lisp listener window. No evaluation of that form occurs of course. Such forms could include numbers, symbols, lists, vectors, etc.

After the new value has been set, the lisp-controller will call any notification function that was specified when the lisp-controller was configured in IB.

The lisp-controller implements two additional methods that are called by virtue of it being the delegate of an NSTableView: #/tableView:shouldEditTableColumn:row: and #/tableViewSelectionDidChange:. The first is called when a user clicks in a particular row/column and the lisp-controller must either deny or grant permission to edit the field. If no function has been found to write that column for the type of the current row-object, then we will deny permission to edit the field. If the developer finds that it is not possible to edit a field that should be editable, this may be the cause. For some reason the lisp-controller has been unable to construct a function to write the value. The function to do all this is below.

```
(objc:defmethod (#/tableView:shouldEditTableColumn:row: #>BOOL)
                ((self lisp-controller)
                 (tab :id)
                 (col :id)
                 (row #>NSInteger))
  (declare (ignore tab))
  ;; allow editing if there is a function available to setf a new value
  (if (or (writer-func self)
          (let ((obj-type (controller-type-of self (elt (objects self) row))))
            (cdr (assoc-aref (type-info self) obj-type col))))
    #$YES
    #$NO))
```

The #/tableViewSelectionDidChange:. method tells us that the user changed the previous selection. This may result in a new field being selected or no field being selected. The function to deal with this is:

```
(objc:defmethod (#/tableViewSelectionDidChange: :void)
                ((self lisp-controller) (notif :id))
  (let* ((tab (#/object notif))
         (row-indx (#/selectedRow tab))
         (col-indx (#/selectedColumn tab)))
    ;; enable/disable buttons that remove current selection
    (#/willChangeValueForKey: self #@"canRemove")
    (if (minusp row-indx)
      (setf (can-remove self) #$NO)
      (setf (can-remove self) #$YES))
    (#/didChangeValueForKey: self #@"canRemove")
    ;; enable/disable buttons that want to add a child to
    ;; the current selection
    (set-can-add-child self row-indx)
    ;; User code to do something when a cell is selected
    (when (select-func self)
      (let* ((row-obj (and (not (eql row-indx -1)) (object-at-row self row-indx)))
             (col (assoc-aref (column-info self) col-indx :col-obj))
             (selected-obj (cond ((and (minusp col-indx) (minusp row-indx))
                                  nil)
                                 ((minusp col-indx)
                                  row-obj)
                                 ((minusp row-indx)
                                  (assoc-aref (column-info self) col :col-title))
                                 (t
                                  (col-value self row-obj col)))))
        (funcall (select-func self)
                 (owner self)
                 self
                 (root self)
                 row-indx
                 col-indx
                selected-obj)))))
```

Recall that we allow buttons that add or remove new objects to be selectively enabled depending on what is currently selected. Here we set the lisp-controller slots that control that functionality. Then, if the developer specified a notification function to be called when the selection is changed, we gather up the necessary arguments and call it.

Lisp Controller Handling of NSOutlineView Calls

In a manner similar to what NSTableView objects do, NSOutlineView objects call the methods
    #/outlineView:numberOfChildrenOfItem:
    #/outlineView:child:ofItem:
    #/outlineView:isItemExpandable:
    #/outlineView:objectValueForTableColumn:byItem:
    #/outlineView:setObjectValue:forTableColumn:byItem:
on the lisp-controller as its data source. It calls the methods
    #/outlineView:shouldEditTableColumn:item:
    #/tableViewSelectionDidChange:
on the lisp-controller as the view's delegate.

NSOutlineView objects present a table view, but each object in the table can potentially be expanded into a subordinate set of objects that are displayed indented from its parent. In turn, each of these may also be expandable. The user controls whether or not the subordinate list is expanded and shown by clicking on a small arrow next to the item. One example of this, the Lisp Class browser, was given in the List-Controller reference document.

To implement this functionality the NSOutlineView asks its data source for an initial set of objects (the children of nil) and then for each of those top-level objects it asks whether it is expandable. If so, it asks for the number of children and then asks for each child object by number. In general the Lisp developer would like those objects to be Lisp objects rather than Objective-C objects, so the lisp-controller arranges to encapsulate Lisp objects within an Objective-C object of type lisp-ptr-wrapper. Putting Lisp objects into lisp-ptr-wrapper objects and extracting them is automatic and is hidden from the developer who is using the lisp-controller object. We will first look at how the lisp-ptr-wrapper functionality is implemented. The class definition from ns-object-utils.lisp is:

```
(defclass lisp-ptr-wrapper (ns:ns-object)
  ((lpw-lisp-ptr :accessor lpw-lisp-ptr)
   (lpw-depth :accessor lpw-depth)
   (lpw-parent :accessor lpw-parent))
  (:metaclass ns:+ns-object))
```

The lisp-ptr-wrapper object keeps track of both the object and its parent. The lpw-depth slot was originally intended as a way to permit users to specify a maximum depth of expansion for objects, but I later reconsidered this because I couldn't find a good use for it. Maybe sometime in the future it will come back if the need arises.

```
(defun make-ptr-wrapper (ptr &key (depth 1) (parent nil))
  (let ((lpw (make-instance 'lisp-ptr-wrapper)))
    (setf (lpw-lisp-ptr lpw) ptr)
    (setf (lpw-depth lpw) depth)
    (setf (lpw-parent lpw) parent)
    lpw))
```

This function will look a bit odd to Lisp developers. Why not just provide initargs for all those slots and use make-instance with appropriate keywords? The answer is that the lisp-ptr-wrapper class is an Objective-C class. So calling make-instance with keywords will be translated into a call on a function whose name is determined in part by the names of the keyword arguments. This is a nice convention that makes it easy for Lisp developers to use existing Objective-C classes with specialized init functions, but is not really what we want or need here. We certainly could have created a C init function with the proper name, but all calls would have to provide all of the arguments. So we simply chose to create the make-ptr-wrapper function instead to allow keyword arguments to be optionally provided.

We already saw one use of lisp-ptr-wrapper objects in the ns-to-lisp-object function. It merely extracted the lpw-lisp-ptr and returned it. We'll see other uses as we go through other functions needed to support NSOutlineViews. The first method we'll discuss is #/outlineView:numberOfChildrenOfItem:.

```
(objc:defmethod (#/outlineView:numberOfChildrenOfItem: #>NSInteger)
                ((self lisp-controller)
                 (olview :id)
                 (item :id))
  (declare (ignore olview))
  (cond ((typep item 'lisp-ptr-wrapper)
         (length (children-of-object self (lpw-lisp-ptr item))))
        ((eql item (%null-ptr))
         (length (objects self)))
        (t
         0)))
```

This method makes use of the function children-of-object to retrieve a sequence that has a length and returns that object. If the specified item is nil, then what is being requested is the count of the top-level objects. Since they are cached in the objects slot, we can use it directly. Let's take a closer look at children-of-object:

```
(defmethod children-of-object ((self lisp-controller) obj)
  ;; Get the children of an instance of some type
  (let* ((obj-type (controller-type-of self obj))
         (child-key (assoc-aref (type-info self) obj-type :child-key))
         (children-object nil))
    (if (children-func self)
      (setf children-object (funcall (children-func self) (owner self) self obj))
      (if child-key
        (setf children-object (funcall child-key obj))))
    ;; if the children object is a hash-table, expand it into an ht-entry list
    (when (typep children-object 'hash-table)
      (setf children-object (children children-object)))
    (sort-sequence self children-object)))
```

In IB, the developer might have specified an override function to be called to get the children of an arbitrary object. If so we call it to determine the child object. Alternatively, the developer might have specified a child key to be used to retrieve the children of a particular type of object. If so we use that key. The final possibility is that neither of these was specified in IB and some default is used. Defaults are available for lists, vectors, and hash-tables. The children of a list or a vector is simply the object itself. That is, all elements of the sequence are presumed to be children.

The child object returned must be a list, a vector, or a hash-table. If it is a hash-table, then it is re-represented as a list of ht-entries. This conversion is handled automatically without the user ever needing to know that it has happened. See the *Hast-table Representation* section below for a more complete description of how hash-tables are represented.

The next method needed to support NSOutlineViews is #/outlineView:child:ofItem:.

```
(objc:defmethod (#/outlineView:child:ofItem: :id)
                ((self lisp-controller)
                 (olview :id)
                 (child #>NSInteger)
                 (item :id))
  (declare (ignore olview))
  (with-slots (obj-wrappers objects) self
    (cond ((typep item 'lisp-ptr-wrapper)
           (let* ((parent (lpw-lisp-ptr item))
                  (parent-depth (lpw-depth item))
                  (children (children-of-object self parent))
                  (child-ptr (elt children child)))
             (or (assoc-aref obj-wrappers child-ptr)
                 (setf (assoc-aref obj-wrappers child-ptr)
                       (make-ptr-wrapper child-ptr
                                         :depth (1+ parent-depth)
                                         :parent parent)))))
          ((eql item (%null-ptr))
           (let ((child-ptr (elt objects child)))
             (or (assoc-aref obj-wrappers child-ptr)
                 (setf (assoc-aref obj-wrappers child-ptr)
                       (make-ptr-wrapper child-ptr :depth 1 :parent nil)))))
          (t
           (%null-ptr)))))
```

This method requests the return of an Objective-C object that represents the Nth child, where N is and integer specified by the child parameter. If the parent object is a lisp-ptr-wrapper, we first extract the Lisp parent from the lisp-ptr-wrapper object that we are given. Then we find its children and extract the Nth child from that sequence. If the parent is nil, then we are being asked for a top-level child. We already have these cached in the objects slot, we we can directly find the Nth child.

In either case, we next want to return the child within its own lisp-ptr-wrapper. We either find such an object that we previously created or construct a new one and add it to the assoc-array that is in the obj-wrappers slot of the lisp-controller.

Outline views display a small arrow next to objects that can be expanded. To find out whether a displayed object can be expanded it calls the #/outlineView:isItemExpandable: method:

```
(objc:defmethod (#/outlineView:isItemExpandable: #>BOOL)
                ((self lisp-controller)
                 (olview :id)
                 (item :id))
  (declare (ignore olview))
  (cond ((eql item (%null-ptr))
         ;; root object
         #$YES)
        ((typep item 'lisp-ptr-wrapper)
         (if (children-of-object self (lpw-lisp-ptr item))
           #$YES
           #$NO))
         (t
          #$NO)))
```

The root object is always expandable, so if we are handed a null pointer the answer is always yes. Otherwise, if the object has children we say it is expandable and if it doesn't it is not expandable. Note that this may change if the developer provides a way to add children to an object. Not being expandable now does not mean that an object is never expandable.

NSOutlineViews also call methods to retrieve and set selected values. These are very similar to the NSTableView calls described earler. The former is implemented via the #/outlineView:objectValueForTableColumn:byItem: method and the latter by the #/outlineView:setObjectValue:forTableColumn:byItem: method.

```
(objc:defmethod (#/outlineView:objectValueForTableColumn:byItem: :id)
                ((self lisp-controller)
                 (olview :id)
                 (col :id)
                 (item :id))
  (declare (ignore olview))
  (let ((ns-format (assoc-aref (column-info self) col :col-format)))
    (lisp-to-ns-object (col-value self (lpw-lisp-ptr item) col) ns-format)))
```

The item parameter provides the row object and the col parameter tells us which column we want to display. When the lisp-controller was initialized we gathered and/or constructed everything necessary to display each possible type of row-object in each possible table column. We earlier examined the col-value function and saw how it finds the right Lisp value and also examined how the lisp-to-ns-object function creates an appropriate Objective-C object for display.

```
(objc:defmethod (#/outlineView:setObjectValue:forTableColumn:byItem: :void)
                ((self lisp-controller)
                 (olview :id)
                 (val :id)
                 (col :id)
                 (item :id))
  (let* ((row-obj (lpw-lisp-ptr item))
         (old-obj (col-value self row-obj col))
         (ns-format (assoc-aref (column-info self) col :col-format))
         (new-val (ns-to-lisp-object old-obj val ns-format)))
    (if (writer-func self)
      (funcall (writer-func self)
               new-val
               (root self)
               row-obj
               (assoc-aref (column-info self) col :col-val))
      (set-col-value self row-obj col new-val))
    (when (edited-func self)
      (let* ((row (#/rowForItem: olview item))
             (edited-obj (if (typep row-obj 'ht-entry)
                           (list (ht-key row-obj) (ht-value row-obj))
                           row-obj)))
        (funcall (edited-func self)
                 (owner self)
                 self
                 (root self)
                 row
                 (assoc-aref (column-info self) col :col-val)
                 edited-obj
                 old-obj
                 new-val)))))
```

If the developer specified an override function to retrieve a column value, then we also may have constructed a corresponding function to set a new value. If so, we call it. Otherwise, we may have found a value setf form for a specified column and row object type. If so, we call set-col-value to invoke it. After the value has been modified, we then invoke a notification function that the Lisp developer may have provided in IB. This can take any additional action that is desired.

There are two lisp-controller methods that are called by virtue of its being a delegate for an NSOutlineView: #/outlineView:shouldEditTableColumn:item: and #/tableViewSelectionDidChange:. The second we already examined for NSTableViews. The first is:

```
(objc:defmethod (#/outlineView:shouldEditTableColumn:item: #>BOOL)
                ((self lisp-controller)
                 (olview :id)
                 (col :id)
                 (item :id))
  (declare (ignore olview))
  ;; allow editing if there is a function available to setf a new value
  (if (or (writer-func self)
           (let ((obj-type (controller-type-of self (lpw-lisp-ptr item))))
             (cdr (assoc-aref (type-info self) obj-type col))))
    #$YES
    #$NO))
```

This is called by the NSOutlineView to determine whether editing of the selected row/column should be permitted. This method simply checks to see whether some method exists for writing a new value and if so, permits editing.

Hash-table Representation

Hash-tables are handled in a special way by the lisp-controller. They are effectively treated as a sequence of key/value pairs. Each pair is encapsulated as an ht-entry object. The class definition for this is:

```
(defclass ht-entry ()
  ((ht-key :reader ht-key :initarg :key)
   (ht-value :reader ht-value :initarg :value)
   (ht :accessor ht :initarg :ht))
  (:default-initargs
    :key (gensym "KEY")
    :value nil
    :ht nil))
```

In addition to the key and value, an ht-entry keeps a pointer back to the original hash-table from which it was derived. In this way, if the user modifies a key or value using the user interface we can translate that into an appropriate action in the original hash-table.

One of the things that we want to do is keep track of which hash-tables we have already turned into lists of ht-entry objects. Since this is both time and space consuming we wouldn't want to do it over and over again whenever the user interface requested the children of an object that happened to be a hash-table. To keep track of this we will use, appropriately enough, a hash-table which contains keys that are hash-table references and associated values which are a list of ht-entry objects. We encapsulate the function definitions for children, (setf children), add-to-child-seq, and delete-from-child-seq within a let that creates this hash-table. In that way each of those functions can modify the list of ht-entry objects and/or the hash-table they are derived from.

```
(let ((ht-hash (make-hash-table)))
  ;; in order to treat hash-tables as containers like lists and vectors we
  ;; need to define a few functions which use a cache of the "children" of
  ;; a hash-table so that we don't need to recreate the whole list every time
  ;; a new child is added

  (defmethod children ((parent hash-table))
    (or (gethash parent ht-hash)
        (setf (gethash parent ht-hash)
              (let ((ht-list nil))
                (maphash #'(lambda (key val)
                             (push (make-instance 'ht-entry
                                     :key key
                                     :value val
                                     :ht parent)
                                   ht-list))
                         parent)
                ht-list))))

  (defmethod (setf children) (new-value (parent hash-table))
    (setf (gethash parent ht-hash) new-value))
```

The new-value parameter will be a list of ht-entry objects. So we just set the value corresponding to the parent hash-table in ht-hash to this list.

```
  (defmethod add-to-child-seq (parent (seq list) (thing ht-entry))
    (with-slots (ht ht-key ht-value) thing
      (setf (gethash ht-hash parent) (cons thing seq))
      (setf ht parent)
      (setf (gethash parent ht-key) ht-value)))
```

To add a child (which will be an ht-entry instance) we first just add it to the value corresponding to the parent hash-table used as a key in ht-hash. Then we set the ht field in the ht-entry so that it now points to the correct parent hash-table. Finally we actually add the new entry to the parent hash-table.

```
  (defmethod delete-from-child-seq ((seq list) (thing ht-entry))
    (with-slots (ht ht-key) thing
      (remhash ht-key ht)
      (delete-from-list seq thing)))
```

To delete a child (ht-entry) we remove it from the list and also remove the corresponding entry from the parent hash-table.

```
) ;; end of hash-table functions within let
```

The methods for making changes to existing ht-entry objects are shown below.

```
(defmethod (setf ht-key) (new-key (self ht-entry))
  (block set-it
    (let ((new-key-exists (not (eq :not-found (gethash new-key (ht self) :not-found)))))
      (when new-key-exists
        ;; They are redefining a key to be one that already exists in the hash-table
        ;; first verify they want to do this
        (let* ((alert-str  (lisp-to-temp-nsstring
                            (format nil
                                    "Continuing will reset value for existing key: ~s"
                                    new-key)))
               (res (#_NSRunAlertPanel #@"ALERT"
                                       alert-str
                                       #@"Cancel key change"
                                       #@"Continue and change key"
                                       (%null-ptr))))
          (unless (eql res #$NSAlertAlternateReturn)
            ;; they don't want to continue
            (return-from set-it))))
      ;; change the value for the existing key
      (setf (gethash new-key (ht self))
            (gethash (ht-key self) (ht self)))
      ;; and then remove the old key that was changed both from the hash table
      ;; and from the list of keys
      ;; new keys are always put at the end of the list unless a sort predicate
      ;; has been specified.
      (remhash (ht-key self) (ht self))
      (setf (slot-value self 'ht-key) new-key))))
```

This method is called if the user has modified a displayed field that corresponds to the key of a hash-table entry. This is perhaps an unusual thing to want to do, so the function displays an interactive dialog that requests the user to verify that this

is actually what they want to do. If so, then the effect is the same as:
1. adding a new hash-table entry with the the new key and an associated value that is the same as that held by the old key
2. deleting the hash-table entry the corresponds to the old value of the key.
These changes are reflected both in the list of ht-entry objects and in the hash-table from which they are derived.

```
(defmethod (setf ht-value) (new-val (self ht-entry))
  (setf (gethash (ht-key self) (ht self)) new-val)
  (setf (slot-value self 'ht-value) new-val))
```

Setting the new value of an ht-entry is more straightforward. It sets the slot in the ht-entry as you might expect and has the side-effect of changing the corresponding value in the hash-table from which this entry was derived.

Support for Insertion and Deletion of Child Objects

Three additional methods exist to support:
1) The insertion of a new child into the root object - #/insert:
2) The insertion of a new child into the currently selected object - #/addChild:
3) The deletion of a selected object - #/remove:

```
(objc:defmethod (#/insert: :void)
                ((self lisp-controller) (button :id))
  (declare (ignore button))
  ;; insert a new object into the root object
  (unless (root self)
    ;; need to create a root object
    ;; may still be null if root type is 'list
    (setf (root self)
          (new-object-of-type self (root-type self))))
  (let ((new-children (add-child-to self (root self))))
    (when (null (root self))
      ;; special hack for root objects that are lists and may be null
      (setf (root self) new-children)
      (setf (objects self) new-children)))
  (#/reloadData (view self)))
```

The real work of this method is accomplished with a call to add-child-to. There are some special considerations needed both before and after this call. Prior to calling it we want to make sure that the root object has been created. So if it is currently nil (which may be a valid value and may not be), we go ahead and create a new root object. The function add-child-to will return a list of children that includes a new child object. Typically the new child is spliced into the existing list of children for the root, so there will not be any additional processing required. However there is one special case that needs additional handling. If the root object started out as nil, then it is necessary to set both it and the objects slot to have the value of the new list of children (which will have only a single object). Finally this function tells the table to reload itself so that the new value will be reflected correctly.

Now let's examine the add-child-to method:

```
(defmethod add-child-to ((self lisp-controller) parent)
  (let* ((parent-type (controller-type-of self parent))
         (child-type (assoc-aref (type-info self) parent-type :child-type))
         (child-key (assoc-aref (type-info self) parent-type :child-key))
         (child-initform (and child-type (assoc-aref (type-info self) child-type :initform)))
         (set-child-func (assoc-aref (type-info self) parent-type :child-setf-key))
         (new-children nil)
         (new-child nil))
    (if (and child-type child-key child-initform set-child-func)
      ;; we've got everything we need to set the child ourselves
      (let ((children (funcall child-key parent)))
        (setf new-child (eval child-initform))
        (setf new-children
              (funcall set-child-func
                       (add-to-child-seq parent children new-child)
                       parent))
        (when (subtypep (controller-type-of self parent) 'hash-table)
          (setf (ht new-child) parent)
          (setf (gethash (ht-key new-child) parent) (ht-value new-child))))
      ;; else see if there is a user-specified function to add a child
      (when (add-child-func self)
        (setf new-children (funcall (add-child-func self) parent))
        (when new-children
          (setf new-child (elt new-children (1- (length new-children)))))))
    (when (added-func self)
      ;; notify by calling the function specified in IB
      (let ((last-child (if (typep new-child 'ht-entry)
                            (list (ht-key new-child) (ht-value new-child))
                            new-child)))
        (when last-child
          (funcall (added-func self)
                   (owner self)
                   self
                   (root self)
                   parent last-child))))
    (sort-sequence self new-children)))
```

First off, all the information needed to create a new child is gathered together to assure that we have everything needed. That includes:
1) The type of the parent object because that determines what type of child it will have
2) The type of the child, given the type of the parent
3) The key that is used to retrieve the children from the parent
4) The initform for the type of the child
5) The function used to set the children of the parent

If we have all of those pieces, then we go ahead and create a new child and add it to the list of children for the parent. If the new child is an ht-entry we set the parent hash-table for it and make sure that the parent hash-table is updated.

If we don't have all those pieces, but do have an override function that can be called to add a new child, then we call it.

Next, if the developer identified a notification function that was to be called after a new child was added, then we go ahead and call it.

Last, but not least, we sort the sequence of children so that the new child will be displayed in the correct location relative to other children and return that list from the function.

The #/addChild: method is similar to the #/insert: method, but will only add a child to the currently selected object. So it is never called to add a child to the root. Therefore we can ignore the special case considerations that were needed in the #/insert: method to handle root peculiarities. That makes this method much simpler and should be self-explanatory.

```
(objc:defmethod (#/addChild: :void)
                ((self lisp-controller) (button :id))
  (declare (ignore button))
  ;; add a new child to the currently selected item
  (let* ((row-num (#/selectedRow (view self)))
         (parent (object-at-row self row-num)))
    (add-child-to self parent))
  (#/reloadData (view self)))
```

Finally we will consider the method called to delete selected objects:

```
(objc:defmethod (#/remove: :void)
                ((self lisp-controller) (button :id))
  (declare (ignore button))
  (let ((row-num (#/selectedRow (view self))))
    (multiple-value-bind (child parent) (object-at-row self row-num)
      (when parent
        (remove-child-from self parent child)
        (#/reloadData (view self))))))
```

The bulk of the work done here is by the call to remove-child-from, so let's have a look at that function;

```
(defmethod remove-child-from ((self lisp-controller) parent child)
  (let* ((parent-type (controller-type-of self parent))
         (child-key (assoc-aref (type-info self) parent-type :child-key))
         (set-child-func (assoc-aref (type-info self) parent-type :child-setf-key))
         (parent-is-root (eq parent (root self)))
         (new-children nil))
```

```
    (if (delete-func self)
      (setf new-children (funcall (delete-func self) (owner self) self parent child))
      (when (and child-key set-child-func)
        (let ((children (funcall child-key parent)))
          (setf new-children
                (funcall set-child-func
                         (delete-from-child-seq children child)
                         parent)))))
    (when (and parent-is-root (null new-children))
      ;; The only time this actually does something is when the
      ;; objects were a list and we just deleted the last child.
      (if (listp parent) (setf (root self) nil))
      (setf (objects self) nil))
    (when (removed-func self)
      (funcall (removed-func self) (owner self) self (root self) parent child))
    (sort-sequence self new-children)))
```

If the developer specified an override function to delete objects, then we just call that function and let it return a list of new children that we'll use. Otherwise, assuming that we can gather up all the necessary functions we will call the delete-from-child-seq function to remove the child object. If the parent of the selected object is the root object, then it is possible that we are removing the very last object from the root. In that case only we may need to set the root object if it was a list and to set the objects slot to nil as well.

Finally we will call a delete notification function if one was specified and then re-sort the new list of children and return it.

The delete-from-child-seq function is implemented somewhat differently for different types of sequences. We already looked at the version used when the children are a list of ht-entry objects. For vectors, the object is removed and the remaining elements of the vector are moved down. For lists, the object is removed in a way that leaves all references to the list valid (as long as the last item in the list isn't removed). The reader is invited to find these functions and examine them to see exactly how they are implemented.

7.0 Final Notes

Although this is the end of the lisp-controller reference as of April 2010, it isn't yet really complete. Several additional Tests are needed to completely validate all of the functionality. Under normal circumstances I would probably have waited until I had a few more of these before checking in this code. I rushed it a bit to hopefully get it into the 1.5 CCL release.

Many things have yet to be thoroughly tested. In particular, the use of override functions has not yet been tested to any extent, so I would not be surprised to find some type of bug there. Also, code to add new children of developer-specified type with developer-specified initforms has only been lightly tested. I will add more examples as quickly as possible.