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.

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.

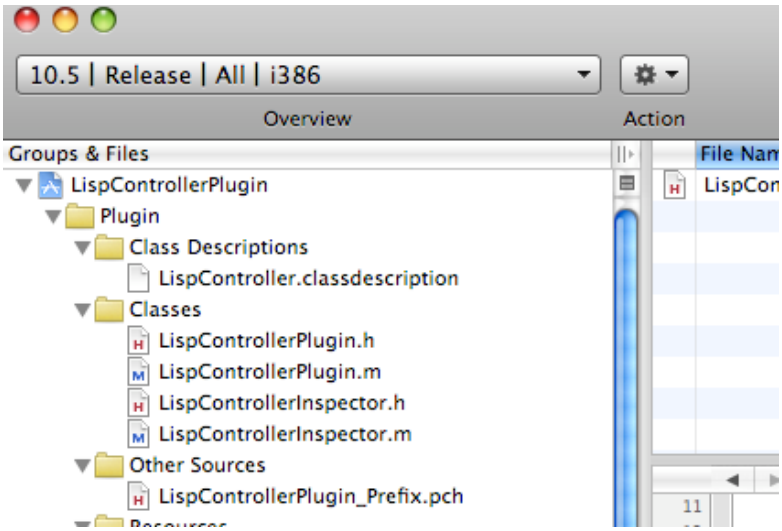

Figure 1

In your version of the pull down menu labeled "10.5 | Release | All | 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.

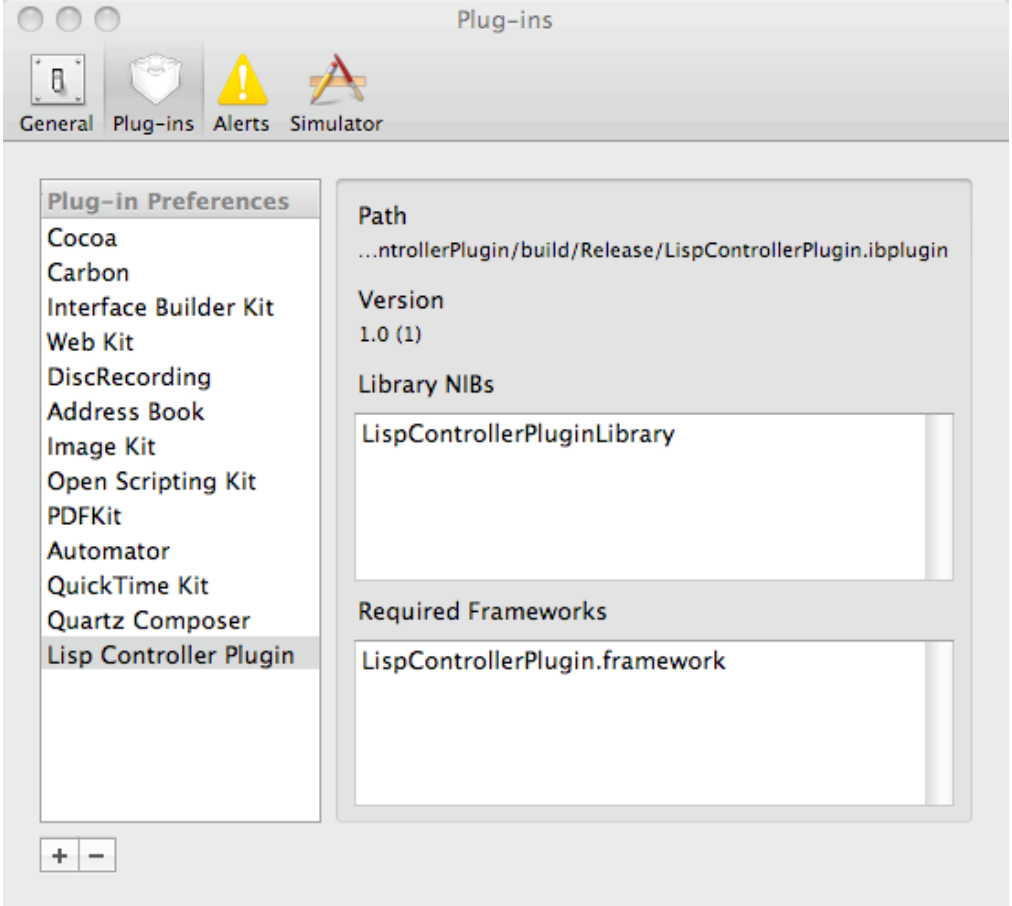

Figure 2

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.

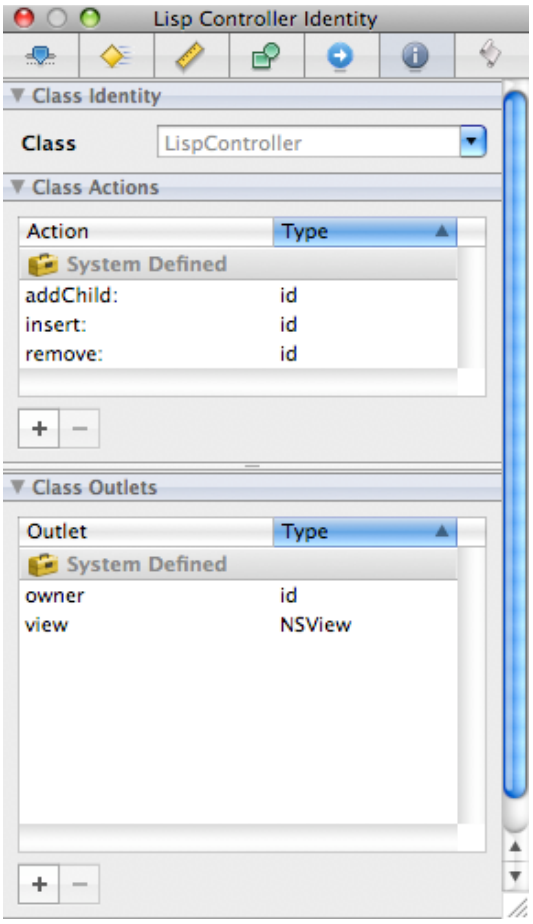

Figure 3: Lisp Controller Identity Inspector

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:

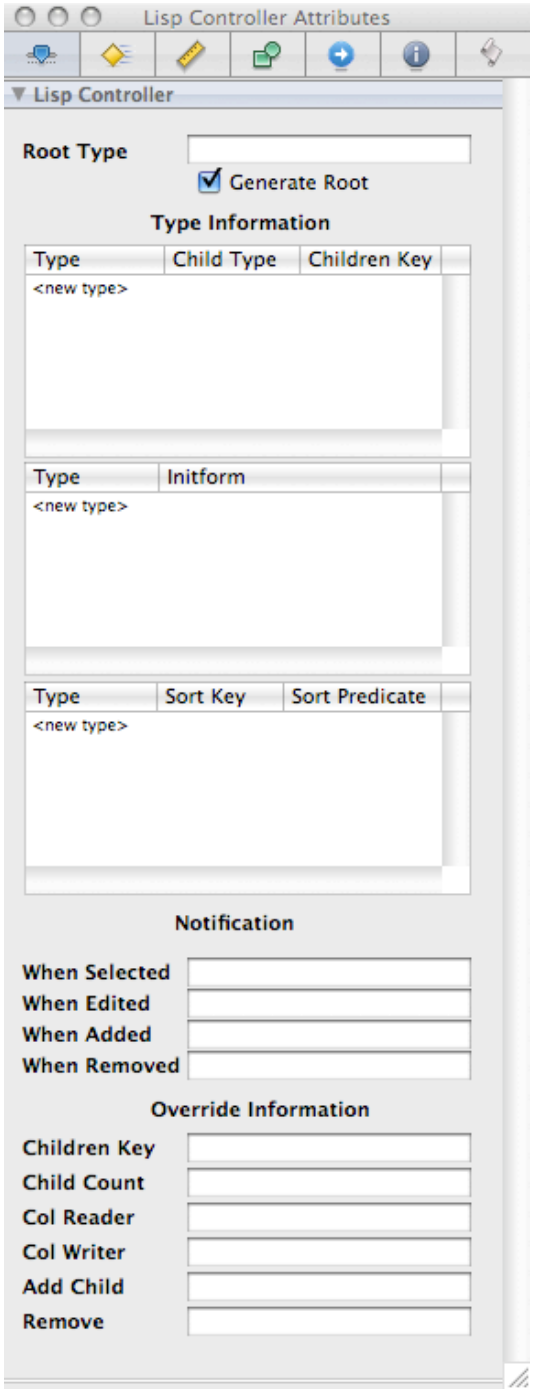

Figure 4: Lisp Controller Attributes Inspector

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.

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.

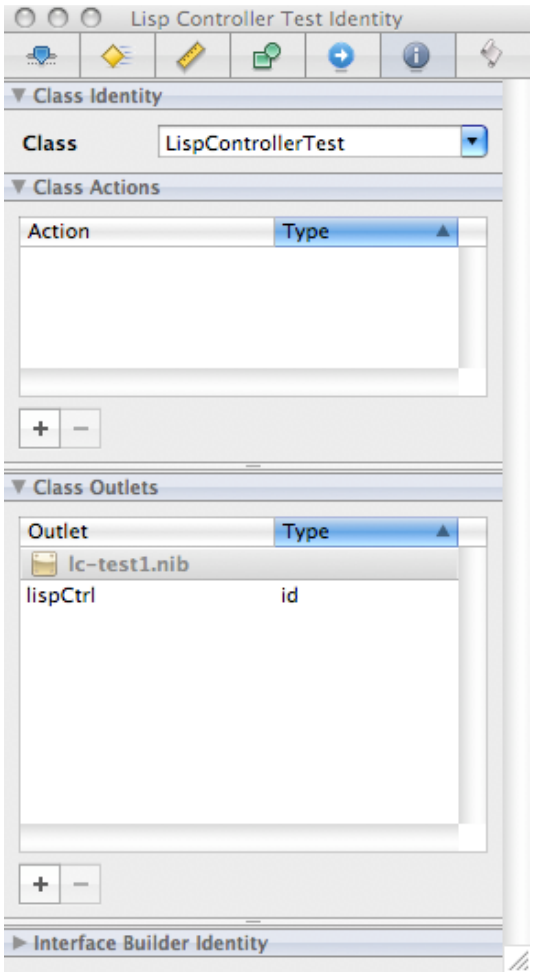

Figure 5: LispControllerTest Identity

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.

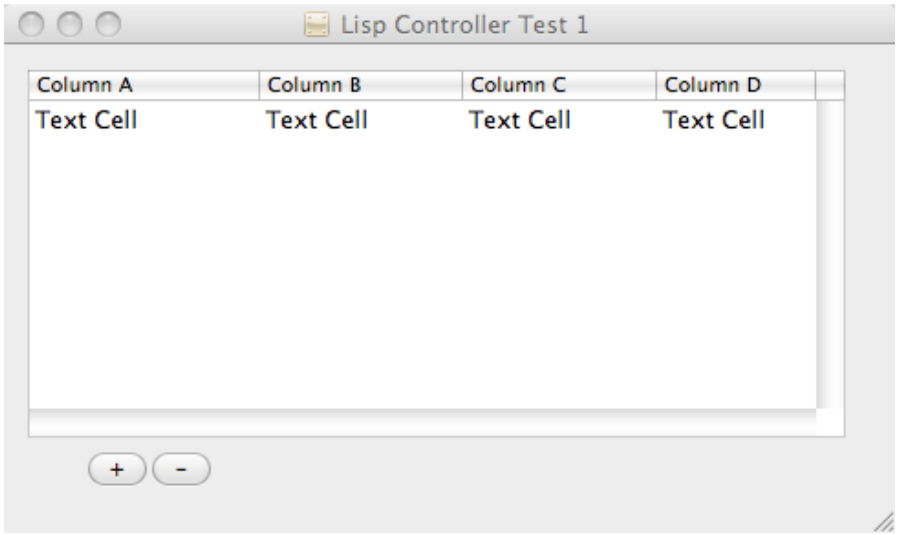

Figure 6: Lisp Controller Test 1 window

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.

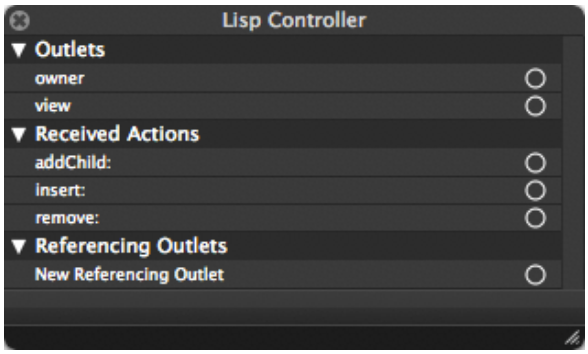

Figure 7: Lisp Controller bindings

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.

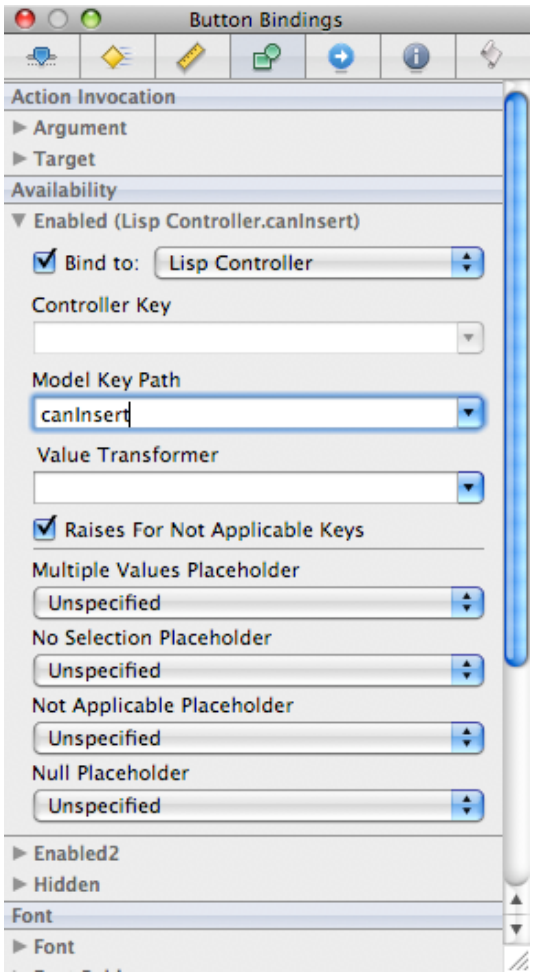

Figure 8: "+" button bindings

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.

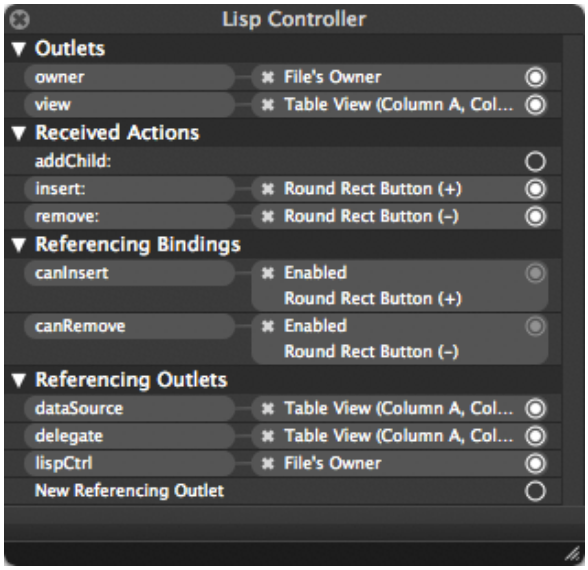

Figure 9: Lisp Controller after linking

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.

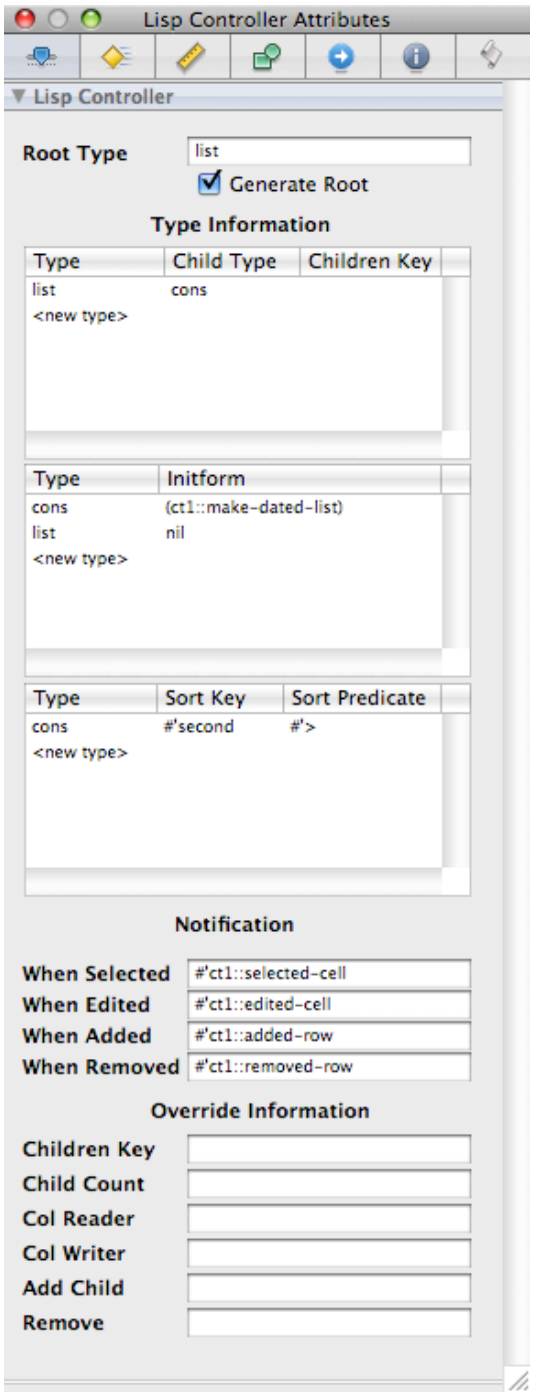

Figure 10: Completed Lisp Controller attributes

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.

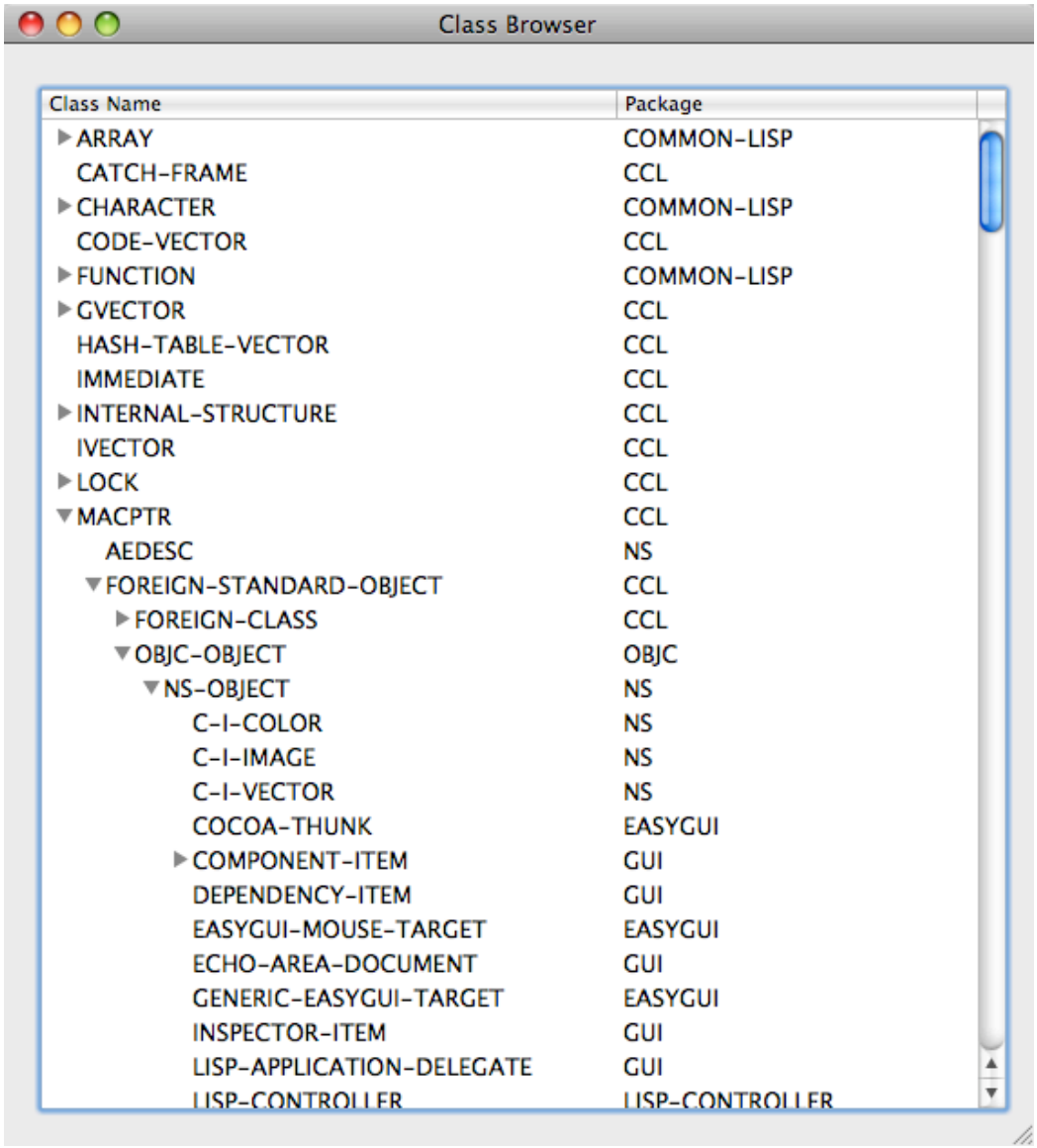

Figure 11: The Class Browser Window

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.

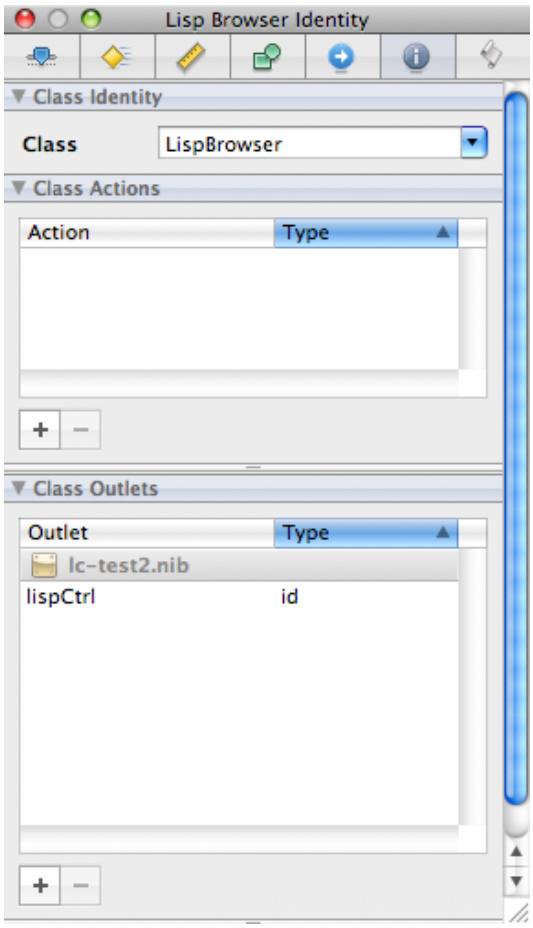

Figure 12: LispBrowser Identity Inspector

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.

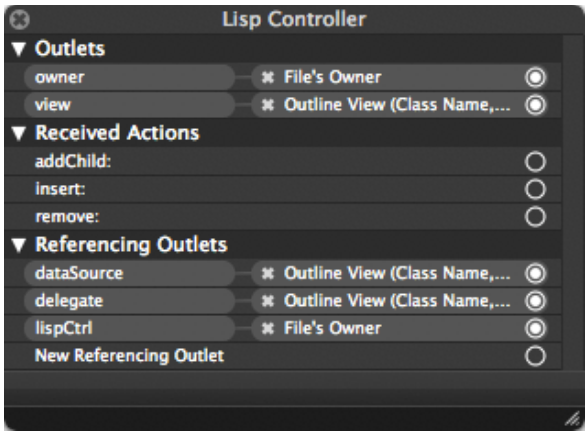

Figure 14: Lisp Controller links

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.

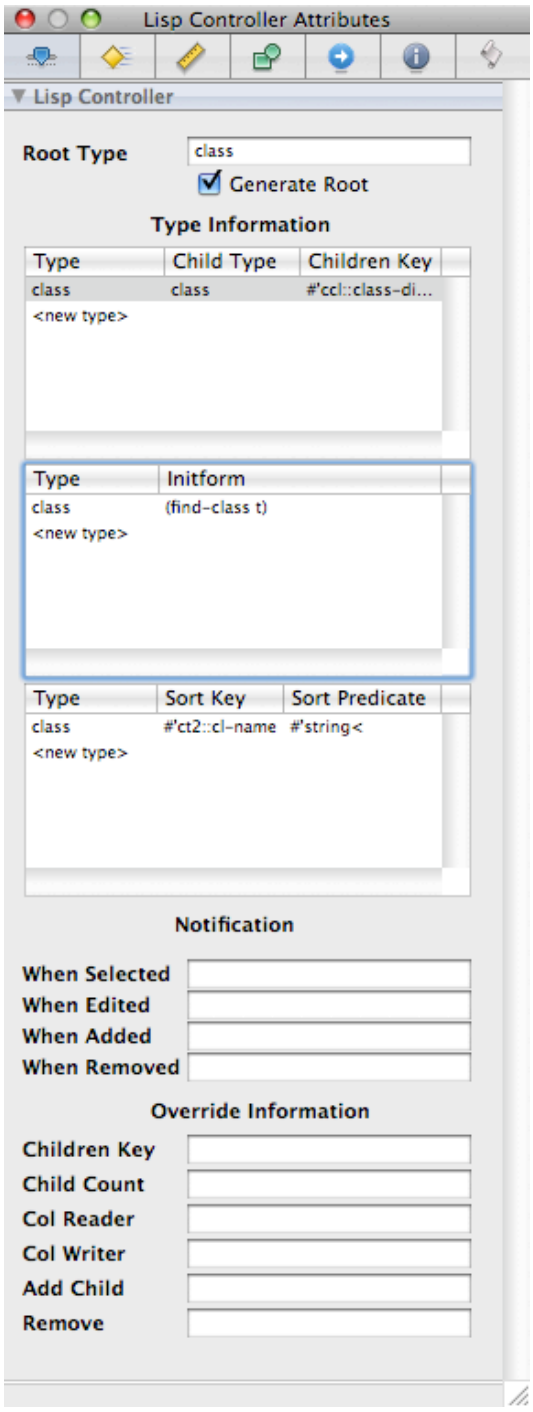

Figure 13: Lisp Controller Attribute Inspector

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 15 below:

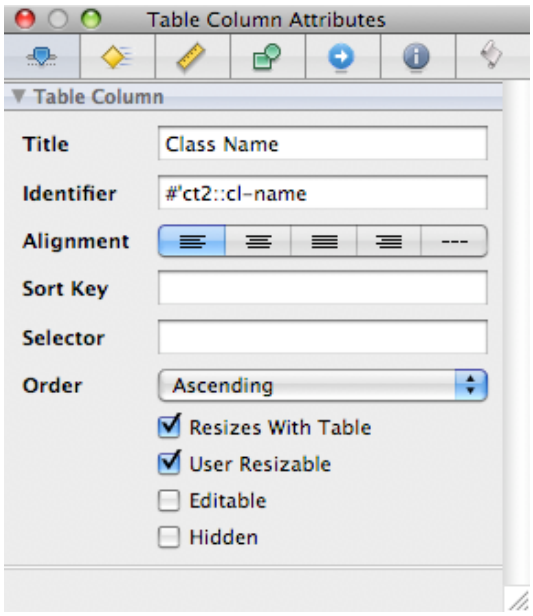

Figure 15: Attribute Inspector for the first column

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 16 below.

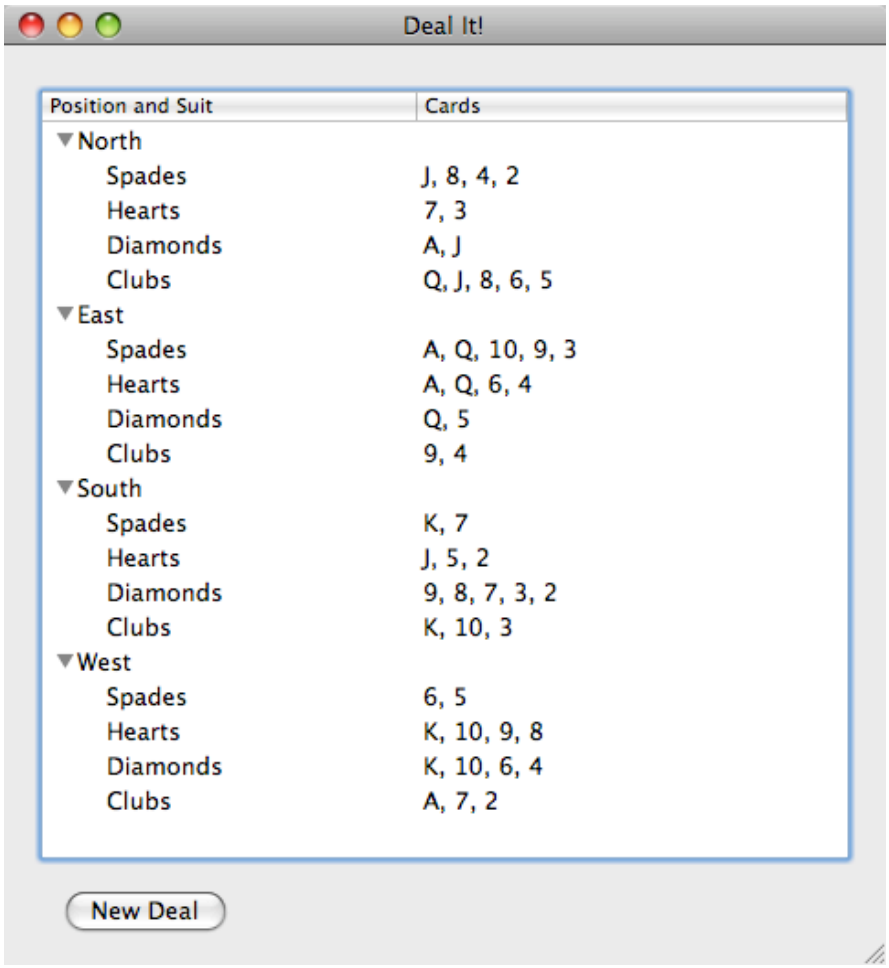

Figure 16: The Card Dealer window

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 didd, 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 17 below.

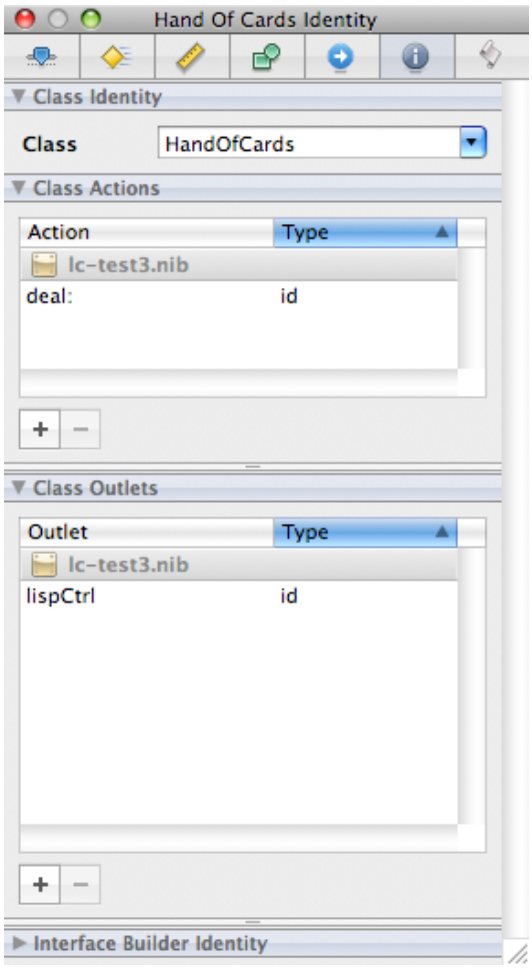

Figure 17: Identity of File's Owner

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 13 below.

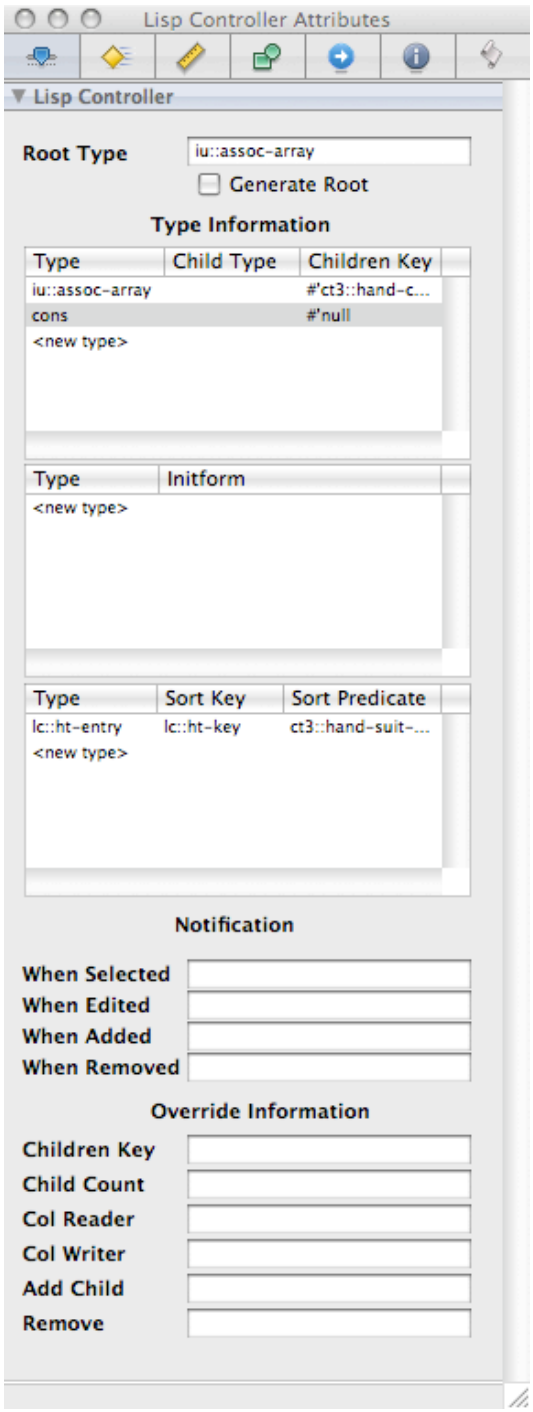

Figure 18: Lisp Controller Attribute Inspector

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 19 below:

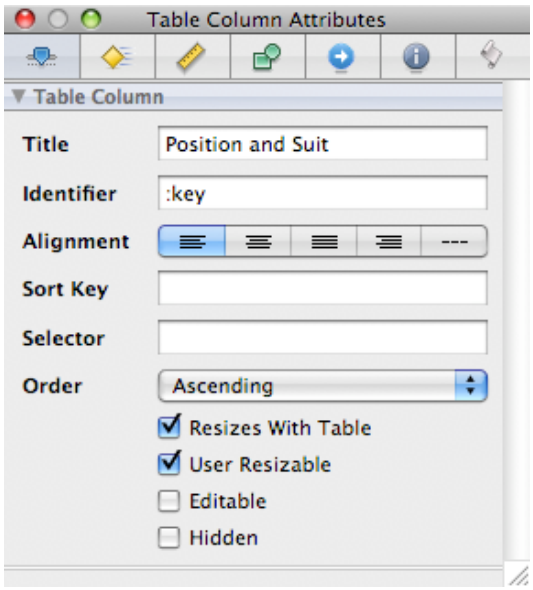

Figure 19: Attribute Inspector for the first column

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.

6.0 Final Notes

Although this is the end of this tutorial 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.