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

Version 2.0 April 2013

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 latter name when discussing Lisp interactions.

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

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. Overview of lisp-controller functionality
2. Configuring the lisp-controller
    2.1 Access Functions and make-instance Keywords
    2.2 Initialization
    2.3 Actions
    2.4 Enabling buttons using lisp-controller bindings
    2.5 Key-Value Coding (KVC) through the LispController
3. Data Conversion
4. Configuring Column Accessors
4. Example Code
    5.1 Controller Test 1: Auto generated list displayed in an NSTableView
    5.2 Controller Test 2: Class Browser
    5.3 Controller Test 3: Card Dealer
6.0 Implementation
7.0 Final Notes

1. Overview of lisp-controller functionality

The lisp-controller class is primarily intended to be a data source for ns:ns-table-view and ns:ns-outline-view objects and to act as a delegate for those classes as well. It is generally unnecessary for the developer to convert Lisp data into an Objective-C form for display. The lisp-controller instance does automatic data translation between Lisp and Objective-C data objects. It configures appropriate *undo* actions when configured to do so. It will add or remove table rows in response to messages from interface controls (e.g. buttons) that have been configured to send the appropriate message. It provides slots to which such controls can bind so that they are enabled and disabled appropriately.

2. Configuring the lisp-controller

This section will provide a description of each field that can be configured when calling make-instance 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 Access Functions and make-instance Keywords

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 interface-packages.lisp as follows:

```
((defpackage :lisp-controller
  (:nicknames :lc)
  (:use :ccl :common-lisp :iu)
  (:import-from :gui
                cgfloat)
  (:export

   ;; lisp-controller.lisp
   added-func
   add-child-func
   can-add-child
   can-insert
   can-remove
   children-func
   content-class
   count-func
   current-selection
   delete-func
   edited-func
   func-owner
   gen-root
   ht-key
   ht-value
   lisp-controller
   lisp-controller-changed
   modified-bound-value
   objects
   reader-func
   removed-func
   root
   root-type
   select-func
   undo-doc
   undo-name
   view
   writer-func))
```

Most of these accessors should not be needed. Values will generally be provided using keyword arguments provided to the make-instance call for the lisp-controller. There are a few exceptions. One is the function "lisp-controller-changed" which is used to inform 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.

The keyword arguments that are acceptable to make-instance include the following:

:add-child-func
  Developer-supplied Lisp override function called to add a child to some table entry.

:added-func
  Lisp function called when a table entry has been added by the user.
  See the *notification functions* section below for more information.

:child-key
  Lisp function that returns the children of both the root object and row objects at level 0.

:child-key-<n>
  Lisp function that returns the children of row objects at level <n> of an outline view.

:col-ids
  List of table column identifiers (0, 1, 2 ... used by default).

:col-keys
  List of lisp functions that return values for their respective column.

:col-keys-<n>
  List of lisp functions that return values for their respective column for rows at level <n> of an outline view.

:count-func
  Lisp function that provides a row count if root object has no obvious length.

:edited-func
  Lisp function called when table entry has been edited by the user.
  See the *notification functions* section below for more information.

:func-owner
  Lisp object on which select-func, edited-func, added-func, and removed-func functions are called.

:initform
  Lisp function called to create a new row entry.

:initform-<n>
  Lisp function called to create a new row entry for rows at level <n> of an outline view.

:removed-func
  Lisp function called when a table entry has been removed by the user.
  See the *notification functions* section below for more information.

:root
  The primary data content accessed through a lisp-controller.

:root-child-key
  Lisp function that when applied to the controller's root returns a list of child objects.

:row-height
  Default height of all rows in the table.

:row-height-<n>
  Height of rows at level <n>.

:search-key
  Function to apply to row objects before applying search-test function.

:search-test
  Function of two arguments. First is result of applying :search-key function to a row object. Second is a search string.

:select-func
  Lisp function called when table selection changes.
  See the *notification functions* section below for more information.

:sort-key
  Lisp function.

:sort-key-<n>
  Lisp function for rows at level <n> of an outline view.

:sort-pred
  Lisp predicate function.

:sort-pred-<n>
  Lisp predicate function for rows at level <n> of an outline view.

:undo-doc
  Document that owns the undo-manager that will be used for table undo actions

:undo-name
  Name of the table that will be used in undo menu choices

2.2 Initialization

Lisp-controller objects are primarily configures with the keyword arguments to make-instance shown above. This section will provide additional information about these. Several of the arguments can be made level-specific using a <keyword>-<n> format. The lisp-controller remembers the highest level specified in any such keyword. Let's call that the *max-level*. In some cases you may want it to the case that a specified keyword is acceptable at level N and all higher levels. To do that, you must make sure that the option is specified at the max-level. If some specified argument should be acceptable up to level N, but no higher level, then the max-level must be greater than N. That is, you must have some other level-specific argument for a level greater than N. Level 0 always refers to the least deeply embedded level displayed in a table. Those would be the children of the root object.

Root and Row Object Initialization

:root
The *root* slot of a lisp-controller must contain an object from which all other objects to be displayed can be derived. The root object is not itself explicitly displayed. When the lisp-controller is used as a data source for an ns:ns-table-view the lisp-controller must provide data for each row/column cell in the table. It does this by maintaining a list of row objects to which column-specific accessor functions are applied to get the data value for a cell in a specified column for a specified row. The row objects are called the *children* of the root object.

The root object can be specified when the lisp-controller is initialized by using the :root keyword argument. If that is not possible, it can be set at runtime using a form such as:

```
      (setf (root <lisp-controller>) <root-object>)
```

This will cause the lisp-controller to re-display the table using the new root object.

A root object 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. 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.

:initform

The :initform keyword argument can be either a function of 0 arguments or a form acceptable to eval. This is called whenever a new child is added at any level in the data hierarchy except the root (which is specified using the :root argument). Note that works whenever the parent object is of a type that is understood by the lisp-controller (i.e. a sequence or hash-table). For other types of root or row-objects to which you want to permit the addition of a new child, you must provide an :add-child-func value that will be called to add the new child.

:initform-<n>

You may want to permit the addition of new children at multiple different levels of the table with the initial new child being different at each level. You can specify a unique initform for each level using these keyword arguments.

Child Information

Each *child* of the root object is treated as a row that is to be displayed within an ns:ns-table-view or or a top-level row to be displayed within an ns:ns-outline-view. There are a few default ways of locating the children of a root object if the root is one of the recognized types. If the root object is a sequence, 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. Each ht-entry object encapsulates a single key/value pair. The ht-key and ht-value functions can be used to retrieve the key and value respectively from an ht-entry object. There are corresponding setf methods for these that result in making changes to the original hash-table. This makes it possible to permit users to edit a displayed hash table and have edit changes reflected in the hash-table that is the root object.

:child-key
:child-key-<n>
The :child-key argument is used to find the children of objects that a lisp-controller cannot handle by its default methods. The function provided with :child-key is used to access children at all levels of the table unless a :child-key-<n> argument is provided for some levels.

If the parent object is a hash-table, then a list of ht-entry objects is created which represents the set of children. When applied to a root object, those children represent the rows of an ns:ns-table-view or the top-level rows of an ns:ns-outline-view. When applied to some other more deeply nested object they represent the children of that more deeply indented object.

If you wish to permit users to add new children and/or remove existing children, then there must be setf function for the function specified with the :child-key argument.

:root-child-key

If you require separate functions to retrieve the children for the root and for higher levels, then the :root-child-key provides a way to specify a level-specific child key that applies to the root object.

:count-func

This keyword is applicable when the lisp-controller is managing an ns:ns-table-view (not an ns:ns-outline-view). It takes three arguments: the object specified with the :func-owner argument, the lisp-controller-object, and the root object for which the count is needed. It should return an integer value. This is only required when the root is not a sequence or hash-table.

:add-child-func
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 notification function specified using the :added-func keyword, 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.

Table Information

These arguments provide additional about the table being controlled that is needed by the lisp-controller

:col-ids

This argument provides a list of strings to be used as column identifiers. By default those identifiers will be the string representation of integers starting with 0 (i.e. "0", "1", etc.). These identifiers must match the identifiers specified when the ns-table-column objects are created for the table being managed so that when the table requests data for a particular column, the lisp-controller will know what data to use.

:col-keys
:col-keys-<n>

These arguments provide a list of accessor functions that take a row-object argument and return a column-specific value to be displayed. The order of the functions in the list should match the order of the columns specified in the :col-ids argument. This need not be the same order as the physical order in which columns are displayed in the table.

:row-height
:row-height-<n>

This value is used by the lisp-controller in its capacity as a *delegate* for a table. It is returned when requested. The value is used to specify the height of rows (either all rows or at some level)

:undo-doc

When a table is used to display some of the data associated with a user document, you may choose to provide undo and redo functionality. This argument should provide an ns-document object that has an undo manager associated with it. If the window can display different documents at different times, then you will have to explicitly set this value for the lisp-controller when the document being displayed in the window is changed.

:undo-name

This argument should provide a string that is used to make undo menu choices comprehensible. It should identify the table that is being managed by this lisp-controller. Additional information will be added to the undo string by the lisp-controller to indicate the specific action that can be undone or redone. If this is not specified, the generic string "table" will be used.

Sort and Search Information

:sort-key
:sort-key-<n>

The :sort-key arguments allow you to specify how the rows are to be sorted (either top-level rows or more deeply nested rows of children within ns:ns-outline-views). The value of this argument should be a function. 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 (value of the :sort-pred argument) to the results.

This is pretty much the same as an ordinary lisp sort, however there are some subtle differences. 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 :child-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-pred
:sort-pred-<n>

The sort predicate specified should be 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.

:search-key

The :search-key argument provides a way to search through all the row-objects displayed in a table (at any level). The argument value must be a function that is applicable to any row at any level. That value will be compared to a string provided by the user in some search text field. You should bind the value of an appropriate text field to the the lisp-controller's search-string slot. When that value is changed, the lisp-controller will use the :search-key function to extract a value from each row.

Note that the search-key need not return anything that is displayed in the table view. It can be some attribute of a row that is not otherwise displayed.

:search-test

This should supply a predicate that takes two arguments. The first argument provided will be the result of applying the :search-key to a row and the second will be the string that is the current value of the lisp-controller's search-string slot. By default the search predicate will be #'string-equal. The lisp-controller will find all matches and make it possible to see next and previous matches as described later.

<u>Notification Functions</u>

If desired, there are four types of notifications that the lisp-controller can provide to the user.

:func-owner

All notification functions are called using the value of this initialization keyword as the first argument.

:select-func

The ns-table-view and ns-outline-view 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 was provided by the :func-owner keyword argument
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 title: the column title if a row was selected or "" 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.

:edited-func

If you have permitted editing of a column, then after cell editing is complete (whether or not a change was actually made) the function specified by this argument will be called with eight arguments. The first six are identical to those described above, with the exception that both the row number is guaranteed to be a non-negative number and the column title is guaranteed to be a valid column title. 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

:added-func
New objects can be added as children to the root object. For ns-table-views that means adding a new row to the table. For ns-outline-views 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 ns-outline-views. After that has been done, the :added-func notification function is called with five arguments:
1) owner object: This is whatever was provided by the :func-owner keyword argument
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 ns-table-view, 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.

:removed-func

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-func except that the last argument represents the child that was removed rather than the child that was added.

2.4 lisp-controller actions

Lisp-controllers have five actions that can be triggered by Objective-C messages sent by window controls (typically buttons). Those messages include:

insert:

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

addChild:

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

remove:

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

searchNext:

When received, this action results in scrolling the table so that the next result found to match the search-key is visible and selecting that row.

searchPrev:

When received, this action results in scrolling the table so that the previous result found to match the search-key is visible and selecting that row.

2.4 Enabling buttons using LispController bindings

There are five 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
canSearchNext
    indicates it is appropriate to request the next search result
canSearchPrev

indicates it is appropriate to request the previous search result

These are used by binding a button's enabled field to the LispController's canRemove, canInsert, canAddChild, canSearchNext, or canSearchPrev path respectively.

2.5 Binding lisp-controller Fields

KVC is a mechanism that Cocoa provides which allows object values to be *bound* together. When the value of one of the bound objects is modified, then the other is changed to reflect that new value. This ability was used in several of the projects in the tutorial. The only requirement for binding between Objective-C and Lisp slots is that the latter must be KVC and KVO compliant. This can be done for Lisp slots by using the :kvo slot option as described in the "Lisp-KVO Reference and Tutorial.rtf" document. This section discusses different ways in which binding lisp-controller fields is useful.

One of the most useful ways to use binding with lisp-controllers is to specify a binding between some value of an interface element and some data object by routing through the "root" of the lisp-controller. If the root is changed over the life of the table, this can provide a way to bind interface object values to slots in an object that is determined dynamically at runtime and may change. When the root object changes, so does the value of the interface object.

A second useful lisp-controller field is its selection. You may want some secondary field or table to automatically reflect information that is relevant to whatever is currently selected in a table. By specifying a binding that goes through the currently selected object you can do just that.

It is also useful to bind the root value of a lisp-controller to some slot value that changes at runtime. A couple of the examples provided in the *UserInterfaceTutorial* document demonstrate this technique for changing what a table displays in response to what a user selects in another table. It could equally well be bound to the value of some Lisp slot that contains something new as a consequence of a computation that is triggered by some other user action.

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 this section 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 "ns-object-utils.lisp".

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

When converting Lisp objects to appropriate ns-objects 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, 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 ns-object 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 ns-object, 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 ns-decimal objects to pass data back and forth and the lisp object is an integer, then the lisp-controller assumes that a format defined in "decimal.lisp" is being used. This format is more thoroughly described for Loan Calc example in the 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 ns-objects to Lisp objects: ns-to-lisp-object

Conversion from ns-objects 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 ns-object, then the value is left unconverted and passed through as the resulting Lisp object.

If the object is an ns-decimal, 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 ns-decimal 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 Objective-C classes are converted appropriately.

An ns-date object os converted to a corresponding Lisp date.

If the previous object was a string, then the ns-object 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.