

# COLDFUSION Developer's Journal

ColdFusionJournal.com

October 2000 Volume:2 Issue: 10

**XML DevCon FALL 2000** November 12-15, 2000  
**Announcing...** **Wireless DevCon** December 3-5, 2000

### Editorial

*The Wonderful World of Wireless*

Robert Diamond page 5

### Foundations

*Making Assertions*

Hal Helms page 14

### Product Reviews

*AuctionBuilder Pro! 1.0 from AbleCommerce*

Carey Lilly page 22

*CommonSpot from PaperThin*

Dave Horan page 56

### Guest Editorial

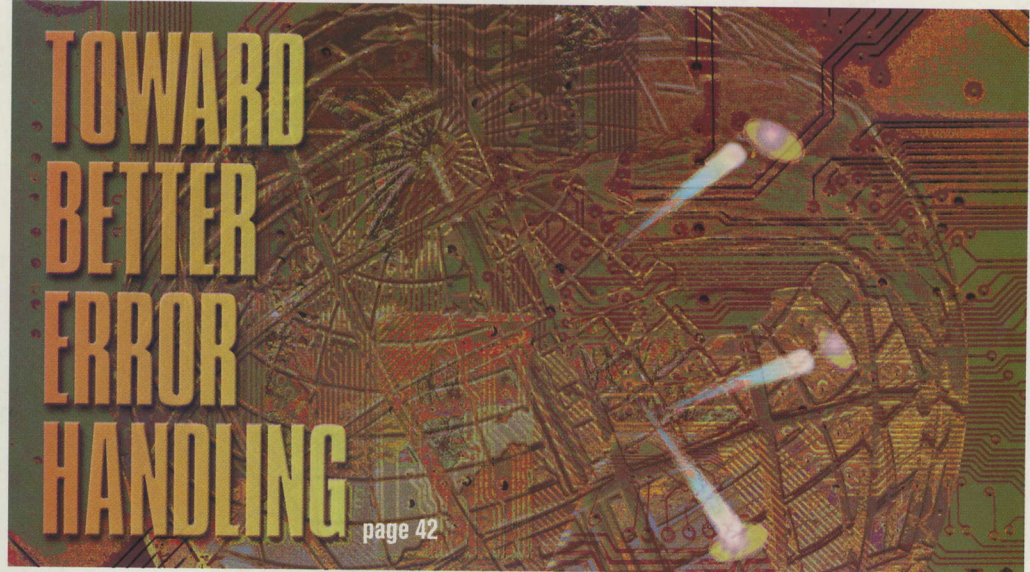
*Great Philosophers of Software Development*

Steven D. Drucker page 26

### CF Conference

*CFUN-2k = CF Party*

Charles Arehart page 36



**CFDJ Feature: Recursive Custom Tags** 6  
*A study of algorithms leads to superior application design* Mark Cyzyk

**<BF> on <CF>: 'The Ten Commandments' - Revisited** 18  
*Even the holiest rules need an update* Ben Forta

**CFDJ Feature: A ColdFusion Based Oracle Database Monitor** 28  
*Limitless possibilities with Web-enabled client/server applications* Kailasnath Awati & Mario Techera

**CF Tools: Use WDDX to Store Complex Variables for Clustered Web Servers** 36  
*Powerful new tool helps CF developers meet Web needs* Alan McCullough

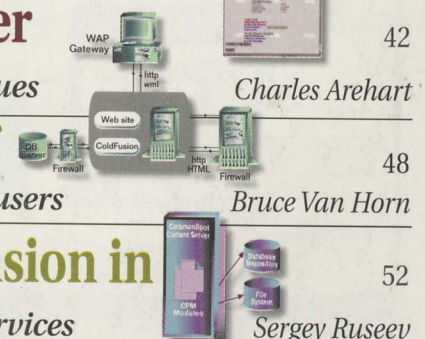
**CFDJ Feature: Toward Better Error Handling** 42  
*Tips and Techniques* Charles Arehart

**CFDJ Feature: ColdFusion Training Staff** 48  
*A new, interactive column for ColdFusion users* Bruce Van Horn

**WAP & ColdFusion: ColdFusion in WAP Architecture** 52  
*Create new services* Sergey Ruseev

MAIL ADDRESS: TOMSON 1609 COTTAGE LN MD 21286-8202  
 MARK CYZYK NINE WEB LLC  
 CFMJ 058528 E 99NOV 25  
 \*\*\*\*\*AULTX3-DIGIT 212  
 PRESENTED TO: SIMONAD US POSTAL FND 76  
 SEVEN FRANK

0 71486 02689 1





BY MARK CYZYK

I have a confession to make: I wasn't a computer science major in college – I was a philosophy major. While the two disciplines have much in common (conceptual acrobatics, a high degree of abstraction, logical and analytical rigor, obtuse and convoluted texts), I'm finding now that, as a Web applications developer, a study of the basic tenets of computer science can help me create more sophisticated applications.

Case in point: the study of data structures and algorithms.

I recently had a programming problem to solve, one that lent itself perfectly to a solution offered in any data structures textbook. My problem was this: I was working with librarians on our campus on a project, part of which entailed the creation of a subject terms list. Once compiled, this hierarchical list of subject terms would be used as a controlled vocabulary to index items in our database. Although the list of terms the librarians came up with consisted of two levels (a general term and a specific term), there was no guarantee that in the future there wouldn't be a need to further subdivide the second level into a third level, and that level into a fourth and so on. In short, the number of levels in the hierarchy could be variable. So how should the data for this project be stored, and how should I then programmatically extract it as needed?

After pondering several scenarios involving multiple tables and confusing looping mechanisms, I decided to consult a colleague for advice. He took one look at the situation and, having had solid training in computer science theory, planted two buzzwords in my head: *tree* and *recursion*. Armed with this information, I scoured the Web and the library in search of information about trees as data structures and how recursive functions can be used to navigate them.

### Storing a Tree Structure

Insofar as the list of subject terms is in a hierarchy, with a term serving as root and subterms serving as branches, it is properly described as a tree data structure. My solution with respect to how it should be stored in the back-end database is to store the entire tree in a single table consisting of three fields. The first field, "SUBJECTID", serves as the primary key, uniquely identifying each row. The second field, "SUBJECT", holds the text of the subject term itself, for example, "Italian Landscapes." The third field, "PARENTID", serves as a pointer to the next item up the hierarchy from the current term. Thus the parent of the term "Italian Landscapes" might be something like "Italian Art," and the parent of the term "Italian Art" might be something like "Art History" and so forth. In this manner, each term in the table is linked to its conceptual parent all the way to the top of the tree where there are base terms that themselves don't have parents. These terms represent the starting points in the tree structure and have, by default, PARENTIDs of 0. Figure 1 represents how this tree structure is stored in a table (the "lookup-Subjects" table) in the back-end database.

### Recursion

But now that the data is properly stored, the question remains: How do we programmatically access it? For instance, chances are at some point we'd want to print out the entire table in outline form, with those subject terms lacking parents at the base and other items down the branches printing in indented form. How to do this? There must be some sort of loop involved here, but what is the nature of this loop? How does it know, in a structure where the length of each branch is variable, when to stop? The conceptual key to doing this is known as *recursion*.

A recursive function is a function (or method, custom tag or other chunk of code) that calls itself. It is essentially a loop, executing over and over again until some sort of “base condition” is met. In fact, the tasks that many recursive functions are used to accomplish can instead be carried out using various FOR and WHILE loops, but using a recursive function is often thought to be a more elegant solution.

To print out the contents of the lookupSubject table in outline form, I created a custom tag called “simpleBuildTree”. This tag results in the output represented by Figure 2.

Listing 1 is the code for the cf\_simpleBuildTree custom tag. It is worth going over line by line.

```
cf_simpleBuildTree
```

First, this custom tag is called in the usual way:

```
<cf_simpleBuildTree>
```

Line 1 of the tag sets a default value of 0 for a variable, #theID#, which is used to retrieve records from the database later on. If, however, #theID# is being sent as an attribute to the tag, it is rescoped in lines 3–5 so that it can simply be referred to as #theID# instead of #attributes.theID#.

Lines 7–12 represent the “getCurrentItems” query. This query selects all records from the table whose PARENTIDs match the value of the current #PARENTID# variable. This value by default is set to 0, so the first time this query is run it will return all records appearing at the very top of the tree structure, that is, all records that are themselves not children of any other parent record. Referring back to Figure 1 for a minute, only two records fit this criterion – the record for “Arts and Humanities” and the record for “General and Reference.” Thus these two entries will serve as roots in the tree structure – and from these two roots all branches will grow.

Line 14 begins an unordered list, and line 16 begins a loop through the records retrieved by the getCurrentItems query (our two records).

Lines 18–20 output the current value of #getCurrentItems.subject# as a list item within the unordered list.

Line 22, however, is where the fun begins.

It’s possible that the current value of #getCurrentItems.subject# does not have any children; that is, it’s not the conceptual parent of any other subject term in the table. But it’s also possible that it has conceptual children, in which case we need to traverse down the tree, finding and outputting them all. The “checkForChild” query on lines 22–26 lets us know whether or not the current item has children attached to it. It does this by selecting records whose PARENTIDs match the current #SUBJECTID#.

On line 28 the recordcount for the checkForChild query is compared to 0. If it’s greater than 0 we know there must be children. If there are children, we need to loop back through the whole process, finding and outputting each one along the way. This is what the recursive call on lines 29–31 does: it calls the cf\_simpleBuildTree tag itself, this time passing a new value for #theID# that is equal to the current value of #getCurrentItems.subjectid#. At this point execution within the tag occurs within a completely new context – the context of the next item down the hierarchy.

Within this context line 14 begins a new unordered list nested within the unordered list a level up the hierarchy. Because HTML deals with nested lists properly, we’re presented with the correctly indented outline illustrated in Figure 2.

To back up for a minute to our first iteration through the getCurrentItems recordset, if the checkForChild query is run on the current value and comes up with nothing, nothing else

subjectID	subject	parentID
1	Arts and Humanities	0
2	General and Reference	0
3	Art History	1
4	Classics	1
5	English and American Literature	1
6	French	1
7	Film and Media Studies	1
8	Biography	2
9	Italian Art	3
10	Italian Portraiture	9
11	Italian Landscapes	9
12	Russian Art	3
13	Ancient Greek Literature	4
14	Ancient Roman Literature	4
*(AutoNumber)		0

FIGURE 1: The lookupSubjects table

```
Subject Terms - cf_simpleBuildTree
```

- Arts and Humanities
  - Art History
    - Italian Art
      - Italian Landscapes
      - Italian Portraiture
    - Russian Art
  - Classics
    - Ancient Greek Literature
    - Ancient Roman Literature
  - English and American Literature
  - Film and Media Studies
  - French
- General and Reference
  - Biography

FIGURE 2: Output of cf\_simpleBuildTree

```
Subject Terms - cf_reverseTree (SUBJECTID = 9)
```

Arts and Humanities - Art History - Italian Art

FIGURE 3: Calling cf\_reverseTree by itself

```
Subject Terms - cf_selectboxBuildTree - Selectbox Mode
```

New subject:

New subject is the child of:

- New subject is at top of tree (not a child of any parent subject)
- Arts and Humanities
- Arts and Humanities - Art History
- Arts and Humanities - Art History - Italian Art
- Arts and Humanities - Art History - Italian Art - Italian Landscapes

Submit

FIGURE 4: cf\_selectboxBuildTree on an HTML form

occurs; the code simply outputs the current value of #SUBJECT# as a root entry in the tree and moves on to the next record.

In this manner recursion can be used to process data contained in a tree structure. The important thing to remember here is that (1) a recursive call is being used to create a loop, and (2) there is a base condition that can be tested against to end the loop; in this case the base condition is reached when the checkForChild query returns a recordcount of 0, indicating that traversal down that particular branch has reached an end.

## cf\_buildTree and cf\_reverseTree

cf\_simpleBuildTree works great and outputs the data in a nice outline form. But suppose we wanted to additionally output each branch as one long string? Suppose we wanted output to look like:

```
Arts and Humanities
Arts and Humanities – Art History
Arts and Humanities – Art History – Italian Art
```

The first thing we'd need to do is put some switches in to let the program know whether we're in "outline" or "string" mode, then pass in an appropriate value indicating which mode we want. So if we just copy cf\_simpleBuildTree to a new file, say, cf\_buildTree, and make the edits, it can be called by:

```
<cf_buildTree
mode="outline"
>
```

The output in this mode is exactly the same as that in Figure 2.

Then it's a matter of editing it and putting in the aforementioned switches. Listing 2 contains the final code for cf\_buildTree, with any code referring to outline mode wrapped in the appropriate conditional code.

The major change required to output subject terms in string mode begins on line 36, within the checkForChild.recordcount conditional. Notice that, within string mode, this tag never actually outputs anything. Rather, beginning on line 37, it calls another custom tag, "cf\_reverseTree", passing it a value for the #theID# variable.

Here's what's going on: to output the subject terms in a string format such as "Arts and Humanities – Art History – Italian Art," the cf\_buildTree tag must somehow keep track of the entire list of these terms as it traverses the tree. But the recursive function it employs really doesn't do that; once it has traversed a branch to its end, it simply backs up one step and processes any other children left in the previous loop. If none are found, it keeps sequentially backing up the hierarchy, processing each branch, until it hits the root. Then it begins processing the next root term and all of its branches. The key thing to note here is that it's not backing all the way to the root each time – just one or two (or three or four...) levels until it finds a child in need of processing. Because this particular tree is an "unbalanced tree," that is, a tree whose respective branches may be of differing lengths, one cannot simply push and pop items onto a stack and then expect that stack to accurately represent a particular branch in string format.

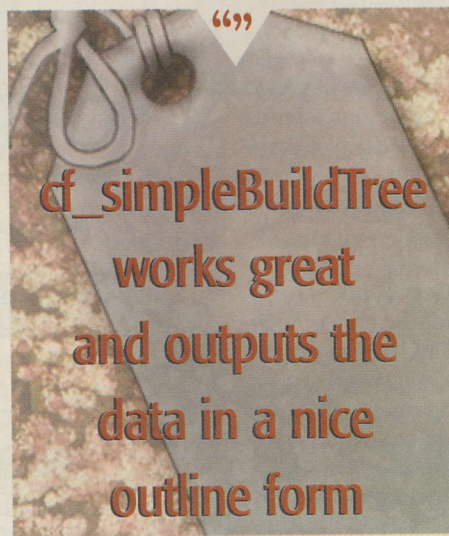
The recursion employed by this tag and the cf\_simpleBuildTree tag determines the end nodes or leaves of each tree – it's what's known as a "depth first" search. Once we know this, we must do a reverse lookup of the parents of the leaf all the way back to the root. This is what the cf\_reverseTree tag does on lines 37–39. Code for this tag is shown in Listing 3.

First, notice that this tag can be called directly with the following call:

```
<cf_reverseTree
theID = 9
>
```

This results in the output shown on Figure 3.

How does cf\_reverseTree work? Pretty much the same way that cf\_buildTree does, only in reverse order. It takes as an attribute a SUBJECTID and, based on that SUBJECTID, it traverses up the tree to the root. As it does this it prepends the value of the #SUBJECT# variable to a list that is then passed to any other recursive iteration of itself. When the top of the tree is reached, it outputs the contents of the list. This output represents the entire branch in string format, from root to leaf.



One last thing to note about this tag is that, on line 25, a <CFEXIT> tag is called. This is done so that, if there is a parent to the current child, the contents of #theList# are not output; it ensures that #theList# is output only once, when the top of the tree is reached.

Returning to cf\_buildTree, we see that, on lines 35–45, if checkForChild.recordcount is not 0, and if the mode of operation is "string" the cf\_reverseTree tag is called, which outputs the branch in string mode. However, if there are no children to the current node, cf\_reverseTree is likewise called. This occurs on lines 45–51.

## Further Modification: cf\_selectboxBuildTree and cf\_selectboxReverseTree

How might the string mode of this tag be used? One use for it would be to dynamically populate an HTML selectbox.

Listings 4 and 5 illustrate the edited cf\_buildTree and cf\_reverseTree (saved as cf\_selectboxBuildTree and cf\_selectboxReverseTree, respectively) needed to do this. Listing 6 illustrates how this new tag is called. Its output is represented in Figure 4.

cf\_selectboxBuildTree is roughly the same as cf\_buildTree; the main difference is the new attribute passed to the cf\_selectboxReverseTree tag on lines 40 and 52. This attribute, named "initialID," passes in the value of the current SUBJECTID and will be used later by the cf\_selectboxReverseTree tag to output an HTML <OPTION> tag with the proper SUBJECTID value.

Taking a look at the code for cf\_selectboxReverseTree (see Listing 5), we see that the value of #INITIALID# is passed to every recursive iteration of the tag, on line 34, without its value ever being changed. Hence the value of the original SUBJECTID is retained and finally output as the value of the HTML <OPTION> tag on line 43.

In this manner a tree structure can be processed and output in "string" mode, and such string output can be used to dynamically populate an HTML selectbox.

## A Note on Performance

It is important to note that recursive functions are extremely resource intensive. This isn't surprising; each time a recursive call is initiated, the tag itself must be reexecuted while simultaneously retaining the state of previous iterations. Further, in the examples offered in this article there is the additional overhead of having the data stored in a back-end database. Processing a medium-sized tree structure with a recursive custom tag could result in literally hundreds of hits on the database server, which could be very time-consuming. Every effort should be made to speed up or eliminate these transactions. A significant performance gain can be made, in the cases illustrated above, by simply caching the queries for a brief period – especially the queries in the reverse lookup tags – so they are not duplicated. Then, if a query was previously run from within a recursive iteration, it need not be run again; rather, its results will simply and quickly be read from cache.

Even so, careful attention must be taken when deciding to use a recursive call to process data – and the decision to use a recursive call should be relative to the amount of data and the resources you have available for processing.


## Conclusion

Robert Lafore, in his lucid and readable book *Data Structures and Algorithms in Java*, points out that it's not enough to learn the syntax of a programming language – one must also learn how best to use that language to manipulate data in an efficient manner. And the study of data structures and the algorithms appropriate for processing them is the first step in that direction, a path that results

ultimately in superior application design. For someone who wasn't a computer science major in college, that's solid advice.

## Acknowledgments

The author wishes to thank his colleagues for their special contributions to his understanding of data structure and algorithms: Ian Goh for pointing the way and Craig Turkington for his insightful

comments on this article in draft. Both were computer science majors. 

## About the Author

Mark Cyzyk is an Allaire certified ColdFusion developer and Web developer at Johns Hopkins University and Nine Web, LLC, where he uses ColdFusion to create dynamic database-driven applications for his clients.

MCZYK@JHU.EDU

### Listing 1: cf\_simpleBuildTree

```
1 <cfparam name="theID" default="0">
2
3 <cfif IsDefined("attributes.theID")>
4 <cfset theID = attributes.theID>
5 </cfif>
6
7 <cfquery datasource=#application.datasource#
  password=#application.password# name="getCurrentItems">
8 SELECT *
9 FROM LOOKUPSUBJECTS
10 WHERE PARENTID = #THEID#
11 ORDER BY SUBJECT
12 </cfquery>
13
14 <ul>
15
16 <cfloop query="GetCurrentItems">
17
18 <cfoutput>
19 <li>#subject#
20 </cfoutput>
21
22 <cfquery datasource=#application.datasource#
  password=#application.password# cachedwithin=#Create-
  TimeSpan(0,0,0,2)# name="checkForChild">
23 SELECT SUBJECTID
24 FROM LOOKUPSUBJECTS
25 WHERE PARENTID = #GETCURRENTITEMS.SUBJECTID#
26 </cfquery>
27
28 <cfif checkForChild.RecordCount gt 0>
29 <cf_simpleBuildTree
30 theID = "#GETCURRENTITEMS.SUBJECTID#"
31 >
32 </cfif>
33
34 </cfloop>
35
36 </ul>
```

### Listing 2: cf\_buildTree

```
1 <cfparam name="theID" default="0">
2 <cfparam name="mode" default="outline">
3
4 <cfif IsDefined("attributes.theID")>
5 <cfset theID = attributes.theID>
6 </cfif>
7
8 <cfif IsDefined("attributes.mode")>
9 <cfset mode = attributes.mode>
10 </cfif>
11
12 <cfquery datasource=#application.datasource#
  password=#application.password# name="getCurrentItems">
13 SELECT *
14 FROM LOOKUPSUBJECTS
15 WHERE PARENTID = #THEID#
16 ORDER BY SUBJECT
17 </cfquery>
18
19 <cfif mode IS "outline"><ul></cfif>
20
21 <cfloop query="getCurrentItems">
```

```
22
23 <cfif mode IS "outline">
24 <cfoutput>
25 <li>#subject#
26 </cfoutput>
27 </cfif>
28
29 <cfquery datasource=#application.datasource#
  password=#application.password# cachedwithin=#Create-
  TimeSpan(0,0,0,2)# name="checkForChild">
30 SELECT SUBJECTID
31 FROM LOOKUPSUBJECTS
32 WHERE PARENTID = #GETCURRENTITEMS.SUBJECTID#
33 </cfquery>
34
35 <cfif checkForChild.RecordCount gt 0>
36 <cfif mode IS "string">
37 <cf_reverseTree
38 theID = "#GETCURRENTITEMS.SUBJECTID#"
39 >
40 </cfif>
41 <cf_buildTree
42 theID = "#GETCURRENTITEMS.SUBJECTID#"
43 mode = "#MODE#"
44 >
45 <cfelse>
46 <cfif mode IS "string">
47 <cf_reverseTree
48 theID = "#GETCURRENTITEMS.SUBJECTID#"
49 >
50 </cfif>
51 </cfif>
52
53 </cfloop>
54
55 <cfif mode IS "outline"></ul></cfif>
```

### Listing 3: cf\_reverseTree

```
1 <cfparam name="theList" default="">
2
3 <cfif IsDefined("attributes.theID")>
4 <cfset theID = attributes.theID>
5 </cfif>
6
7 <cfif IsDefined("attributes.theList")>
8 <cfset theList = attributes.theList>
9 </cfif>
10
11 <cfquery datasource=#application.datasource#
  password=#application.password# cachedwithin=#Create-
  TimeSpan(0,0,0,2)# name="getCurrentItems">
12 SELECT *
13 FROM LOOKUPSUBJECTS
14 WHERE SUBJECTID = #THEID#
15 </cfquery>
16
17 <cfset theItem = #getcurrentitems.subject#>
18 <cfset theList = listPrepend(theList, "#theItem#")>
19
20 <cfif #getcurrentitems.parentID# IS NOT 0>
21 <cf_reverseTree
22 theID = "#GETCURRENTITEMS.PARENTID#"
23 theList = "#theList#"
24 >
25 <cfexit>
26 </cfif>
```

```

27
28 <cfset theList = listChangeDelims(theList, " - ")>
29
30 <cfoutput>
31 #theList#<br>
32 </cfoutput>

```

#### Listing 4: cf\_selectboxBuildTree

```

1 <cfparam name="theID" default="0">
2 <cfparam name="mode" default="outline">
3
4 <cfif IsDefined("attributes.theID")>
5 <cfset theID = attributes.theID>
6 </cfif>
7
8 <cfif IsDefined("attributes.mode")>
9 <cfset mode = attributes.mode>
10 </cfif>
11
12 <cfquery datasource=#application.datasource#
   password=#application.password# name="getCurrentItems">
13 SELECT *
14 FROM LOOKUPSUBJECTS
15 WHERE PARENTID = #THEID#
16 ORDER BY SUBJECT
17 </cfquery>
18
19 <cfif mode IS "outline"><ul></cfif>
20
21 <cfloop query="GetCurrentItems">
22
23 <cfif mode IS "outline">
24 <cfoutput>
25 <li>#subject#
26 </cfoutput>
27 </cfif>
28
29 <cfquery datasource=#application.datasource#
   password=#application.password# cachedwithin=#Create-
   TimeSpan(0,0,0,2)# name="checkForChild">
30 SELECT SUBJECTID
31 FROM LOOKUPSUBJECTS
32 WHERE PARENTID = #GETCURRENTITEMS.SUBJECTID#
33 </cfquery>
34
35 <cfif CheckForChild.RecordCount IS NOT 0>
36 <cfif (mode IS "string") OR (mode IS "selectbox")>
37 <cf_selectboxReverseTree
38   theID = "#GETCURRENTITEMS.SUBJECTID#"
39   mode = "#MODE#"
40   initialID = "#GETCURRENTITEMS.SUBJECTID#"
41 >
42 </cfif>
43 <cf_selectboxBuildTree
44   theID = "#GETCURRENTITEMS.SUBJECTID#"
45   mode = "#MODE#"
46 >
47 <cfelse>
48 <cfif (mode IS "string") OR (mode IS "selectbox")>
49 <cf_selectboxReverseTree
50   theID = "#GETCURRENTITEMS.SUBJECTID#"
51   mode = "#MODE#"
52   initialID = "#GETCURRENTITEMS.SUBJECTID#"
53 >
54 </cfif>
55 </cfif>
56
57 </cfloop>
58
59 <cfif mode IS "outline"></ul></cfif>

```

#### Listing 5: cf\_selectboxReverseTree

```

1 <cfparam name="theList" default="">
2
3 <cfif IsDefined("attributes.theID")>
4 <cfset theID = attributes.theID>
5 </cfif>
6
7 <cfif IsDefined("attributes.theList")>

```

```

8 <cfset theList = attributes.theList>
9 </cfif>
10
11 <cfif IsDefined("attributes.mode")>
12 <cfset mode = attributes.mode>
13 </cfif>
14
15 <cfif IsDefined("attributes.initialID")>
16 <cfset initialID = attributes.initialID>
17 </cfif>
18
19 <cfquery datasource=#application.datasource#
   password=#application.password# cachedwithin=#Create
   TimeSpan(0,0,0,2)# name="getCurrentItems">
20 SELECT *
21 FROM LOOKUPSUBJECTS
22 WHERE SUBJECTID = #THEID#
23 </cfquery>
24
25 <cfset theItem = #getcurrentitems.subject#>
26
27 <cfset theList = listPrepend(theList, "#theItem#")>
28
29 <cfif #getcurrentitems.parentID# IS NOT 0>
30 <cf_selectboxReverseTree
31   theID = "#GETCURRENTITEMS.PARENTID#"
32   theList = "#THELIST#"
33   mode = "#MODE#"
34   initialID = "#INITIALID#"
35 >
36 <cfexit>
37 </cfif>
38
39 <cfset theList = listChangeDelims(theList, " - ")>
40
41 <cfoutput>
42 <cfif mode IS "selectbox">
43 <option value=#INITIALID#>
44 </cfif>
45 #theList#
46 </cfoutput>
47 <cfif mode IS NOT "selectbox">
48 <br>
49 </cfif>

```

#### Listing 6: A form calling cf\_selectboxBuildTree

```

1 <fieldset>
2 <legend><b>Subject Terms - cf_selectboxBuildTree - Select
   box Mode</b></legend>
3 <p>
4 <cfform action="action.cfm">
5 <dl>
6 <dt>New subject:
7 <dd><cfinput name="subject" size=25 required message="You
   must include a subject!">
8 <dt>New subject is the child of:
9 <dd><select name="parentID" size=5">
10 <option value=0 selected>New subject is at top of tree
   (not a child of any parent subject)
11 <cf_selectboxBuildTree
12   mode="selectbox"
13 >
14 </select>
15 <p>
16 <input type="submit" value="Submit">
17 </cfform>
18 <p>
19 </fieldset>

```