Notes on Implementing List Ops in the Interpreter
26 January 2009

I believe the best way to represent lists in the interpreter is as C Lists of Values. By C List, I mean the list structure defined in the revised list.h and list.c.

Using Lists of Values will allow all lists to be handled uniformly, as far as list operations go. It may be useful to define a ValueList specialization of List, with ListElemData typedef'd as Value, and appropriate specializations of the List functions.

A list Value struct has the following data fields:

```
LoR = RVAL or LVAl, depending on context
tag = ListTag
type = pointer to full list TypeStruct
size = TypeSize(List*), i.e., always the size of just a pointer
val.ListVal = pointer to C List structure
```

This is a Lisp-style (Java-style) implementation of lists, as pointers to dynamically-allocated data. So, in any interpreter memory pool, a list value is allocated with only a pointer's worth of storage. This means there are no array-like list values, with inline storage and offsets for the elements.

Based on this list representation, here are some notes about implementing list operations:

Operator	**Description**	**Implementation Notes**
[e_1,...,e_n]	construction (elementwise)	Start by building a new List Value struct. Then build the list value with NewList, followed by a loop of PutList(interExpr(e_i)) for each element expression. Return the new Value.
[e_1 .. e_n]	construction (inclusive range)	Like elementwise construction, but for the range e_1-e_n. The type checker ensures that the type of e_1 and e_n are integer.
L[n]	selection (nth, from 1)	Use GetNthList to fetch the Value at the nth position in the list, and return that Value.
L[m..n]	selection (mth - nth)	Like range construction, but using a loop with GetNthList to get the elements at list positions m through n. Return the new list.

+	concatenation	Use the C function ListConcat. That function takes two lists, but FMSL list contat (with '+') is overloaded to allow three cases: (1) both args are lists (the way ListConcat is implemented) (2) first arg is an element, second arg is list (3) vice versa of case 2 For cases 2 and 3, the type checker ensures that the element arg is the correct type, i.e., it's of the base type of the list. In the interp, you'll need to wrap the element arg into a list, by creating a new List Value. Then you can call the ListConcat function. As explained in the documentation of list.h, ListConcat is the non-mutating version of list construction. It's OK to use the mutating PutList in the initial list construction, but the non-mutating ListConcat needs to be used for the FMSL list concat, in order for it to have non-mutating semantics.
-	deletion	Use the C functions InListWithFunction, SubList, ConcatList. We can't use the DelListNth function, because FMSL list deletion is non-mutating. For example, if l is an FMSL list of integer = `[1,2,3,4]`, then the expression `l - 3` returns `[1,2,4]`. Like list concat, it's a non-mutating function, meaning the value of l is not affected by the delete. The args for FMSL deletion are not overloaded like list concat. The first arg to delete must be a list and the second arg must be an element of the list's basetype. The type checker ensures this. There aren't three overloads like there are for list concat. So the implementation of the delete op starts by searching the list using InListWithFunction. If that returns a non-zero value, then a non-mutating version of DelListNth is performed. The code looks like this, for the list value l and delete position n: ConcatLists(SubList(l, 1, n-1), SubList(l, n+1, ListLen(l))); DelNthList is not itself called, but replaced with this chunk of code. It's the standard delete idiom for a functional language. Instead of mutating l by removing the nth element, a new list is created by concatenating the first n-1 elements of l, with the n+1st through the last element of l. I noted above that the implementation of list delete should use InListWithFunction, not just InList. The discussion immediately below abut the FMSL in operator clarifies this. I.e., it explains the function to use with InListWithFunction.

`in`	membership	Use the list function InListWithFunction. The deal is that all equality checks in FMSL must be deep. In Lisp terms, equality in FMSL is implemented as `equal`, not `eq` (if you happen to recall that distinction). As explained in the list.h documentation, there are two c functions that implement membership -- InList and InListWithFunction. When you used InListWithFunction, you supply a function that performs the deep equality test for two list elements. In the case of a Value struct as the list element type, the equals function uses the Valtag to determine how to compare the values. For the atomic types, it can use ==. I.e., the IntVal, RealVal, BoolVal can be compared with ==. String compare is used for StringVals. The equals function for lists works recursively, with == for atomic list elements and recursive descent into sublists. The testing example in int-list-test has the basic idea, in the example EqualsFunc.
#	length	Use the ListLen function.