Lists Template B/listt

Purpose

Code to support the list of... kind of value constructor.

---

§**1. Head.** As ever: if there is no heap, there are no lists (in this sense).

```
#IFDEF MEMORY_HEAP_SIZE; ! Will exist if any use is made of heap
```

§**2. KOV Support.** See the "BlockValues.i6t" segment for the specification of the following routines.

```
[ LIST_OF_TY_Support task arg1 arg2 arg3;
    switch(task) {
        CREATE_KOVS:     arg3 = LIST_OF_TY_Create(arg2);
                         if (arg1) LIST_OF_TY_CopyRawArray(arg3, arg1, 2, 0);
                         return arg3;
        CAST_KOVS:       rfalse;
        DESTROY_KOVS:    return LIST_OF_TY_Destroy(arg1);
        PRECOPY_KOVS:    return LIST_OF_TY_PreCopy(arg1, arg2);
        COPY_KOVS:       return LIST_OF_TY_Copy(arg1, arg2);
        COMPARE_KOVS:    return LIST_OF_TY_Compare(arg1, arg2);
        READ_FILE_KOVS:  rfalse;
        WRITE_FILE_KOVS: rfalse;
        HASH_KOVS:       return LIST_OF_TY_Hash(arg1);
    }
];
```

§**3. Creation.** A list is a multiple-block value with word-sized entries: the first few entries of the block are used for details about the list; the items in the the list then follow. Thus, to convert an item index to an array entry index, add `LIST_ITEM_BASE`.

Lists are by default created empty but in a block-value with enough capacity to hold 26 items, this being what's left in a 32-word block once all overheads are taken care of: 4 words are consumed by the header, then 2 more by the list metadata entries below.

```
Constant LIST_ITEM_KOV_F = 0; ! Entry 0: the kind of the list
Constant LIST_LENGTH_F = 1; ! Entry 1: length, i.e., number of items
Constant LIST_ITEM_BASE = 2; ! List items begin at this entry

[ LIST_OF_TY_Create skov list;
    skov = KindBaseTerm(skov, 0);
    list = BlkAllocate(28*WORDSIZE, LIST_OF_TY, BLK_FLAG_MULTIPLE + BLK_FLAG_WORD);
    BlkValueWrite(list, LIST_ITEM_KOV_F, skov);
    BlkValueWrite(list, LIST_LENGTH_F, 0);
    return list;
];
```

§**4. Setting Up.** NI needs to compile code which will create constant lists such as `{1, 4, 9}` at run-time: the following routine is convenient for that. A "raw array" in this routine's sense is an array `raw` such that `raw-->2` contains the number of items, `raw-->1` the kind of value, and `raw-->3` onwards are the items themselves.

```
[ LIST_OF_TY_CopyRawArray list arr rea cast len i ex bk v w;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    ex = BlkValueExtent(list);
    len = arr-->2;
    if ((len+LIST_ITEM_BASE > ex) &&
        (BlkValueSetExtent(list, len+LIST_ITEM_BASE) == false)) return 0;
    BlkValueWrite(list, LIST_LENGTH_F, len);
    if (rea == 2) bk = BlkValueRead(list, LIST_ITEM_KOV_F);
    else {
        bk = arr-->1;
        BlkValueWrite(list, LIST_ITEM_KOV_F, bk);
    }
    for (i=0:i<len:i++) {
        v = arr-->(i+3);
        if (KindAtomic(bk) == LIST_OF_TY) {
            w = LIST_OF_TY_Create(v-->1);
            LIST_OF_TY_CopyRawArray(w, v, 0, KindBaseTerm(bk, 0));
            BlkValueWrite(list, i+LIST_ITEM_BASE, w);
        } else {
            if ((cast) && (cast ~= bk)) {
                if (KOVIsBlockValue(cast)) v = BlkValueCreate(cast, v, bk);
            } else {
                if (KOVIsBlockValue(bk)) v = BlkValueCreate(bk, v);
            }
            BlkValueWrite(list, i+LIST_ITEM_BASE, v);
        }
    }
    if ((cast) && (cast ~= bk)) BlkValueWrite(list, LIST_ITEM_KOV_F, cast);
    #ifdef SHOW_ALLOCATIONS;
    print "Copied raw array to list: "; LIST_OF_TY_Say(list, 1); print "^";
    #endif;
    return list;
];
```

§**5. Destruction.** If the list items are themselves block-values, they must all be freed before the list itself can be freed.

```
[ LIST_OF_TY_Destroy list no_items i;
    if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F))) {
        no_items = BlkValueRead(list, LIST_LENGTH_F);
        for (i=0; i<no_items; i++) BlkFree(BlkValueRead(list, i+LIST_ITEM_BASE));
    }
    return list;
];
```

§**6. Copying.** Again, if the list contains block-values then they must be duplicated rather than bitwise copied as pointers.

Note that we use the pre-copy stage to remember the kind of value stored in the list. Type-checking ordinarily means that one list can only be copied into another if they have the same kind of contents: but there is one exception, which is when the list being copied is the empty list.

```
Global precopied_list_kov;

[ LIST_OF_TY_PreCopy lto lfrom list;
    precopied_list_kov = BlkValueRead(lto, LIST_ITEM_KOV_F);
];

[ LIST_OF_TY_Copy lto lfrom list no_items i nv bk val splk;
    no_items = BlkValueRead(lfrom, LIST_LENGTH_F);
    bk = BlkValueRead(lfrom, LIST_ITEM_KOV_F);
    if (precopied_list_kov ~= 0 or 1) BlkValueWrite(lto, LIST_ITEM_KOV_F, precopied_list_kov);
    else BlkValueWrite(lto, LIST_ITEM_KOV_F, bk);
    if ((precopied_list_kov == INDEXED_TEXT_TY) && (bk == TEXT_TY)) {
        for (i=0; i<no_items; i++) {
            nv = BlkValueCreate(INDEXED_TEXT_TY);
            INDEXED_TEXT_TY_Cast(BlkValueRead(lfrom, i+LIST_ITEM_BASE), TEXT_TY, nv);
            BlkValueWrite(lto, i+LIST_ITEM_BASE, nv);
        }
    } else {
        if (KOVIsBlockValue(bk)) {
            for (i=0; i<no_items; i++) {
                val = BlkValueRead(lfrom, i+LIST_ITEM_BASE);
                nv = BlkValueCreate(BlkType(val));
                BlkValueCopy(nv, val);
                BlkValueWrite(lto, i+LIST_ITEM_BASE, nv);
            }
        }
    }
    precopied_list_kov = 0;
];
```

§**7. Comparison.** Lists of a given kind of value are always grouped together, in this comparison: but the effect of that is unlikely to be noticed since NI's type-checker will probably prevent comparisons of lists of differing items in any case. The next criterion is length: a short list precedes a long one. Beyond that, we use the list's own preferred comparison function to judge the items in turn, stopping as soon as a pair of corresponding items differs: thus we sort lists of equal size in lexicographic order.

Since the comparison function depends only on the KOV, it may seem wasteful of a word of memory to store it in the list, given that we are already storing the KOV in any case. But we do this because comparisons have to be fast: we don't want to incur the overhead of translating KOV to comparison function.

```
[ LIST_OF_TY_Compare listleft listright delta no_items i cf;
    delta = BlkValueRead(listleft, LIST_LENGTH_F) - BlkValueRead(listright, LIST_LENGTH_F);
    if (delta) return delta;
    no_items = BlkValueRead(listleft, LIST_LENGTH_F);
    if (no_items == 0) return 0;
    delta = BlkValueRead(listleft, LIST_ITEM_KOV_F) - BlkValueRead(listright, LIST_ITEM_KOV_F);
    if (delta) return delta;
    cf = LIST_OF_TY_ComparisonFn(listleft);
    if (cf == 0 or UnsignedCompare) {
```

```
        for (i=0; i<no_items; i++) {
            delta = BlkValueRead(listleft, i+LIST_ITEM_BASE) -
                BlkValueRead(listright, i+LIST_ITEM_BASE);
            if (delta) return delta;
        }
    } else {
        for (i=0; i<no_items; i++) {
            delta = cf(BlkValueRead(listleft, i+LIST_ITEM_BASE),
                BlkValueRead(listright, i+LIST_ITEM_BASE));
            if (delta) return delta;
        }
    }
    return 0;
];
[ LIST_OF_TY_ComparisonFn list;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    return KOVComparisonFunction(BlkValueRead(list, LIST_ITEM_KOV_F));
];
[ LIST_OF_TY_Distinguish txb1 txb2;
    if (LIST_OF_TY_Compare(txb1, txb2) == 0) rfalse;
    rtrue;
];
```

§8. Hashing.

```
[ LIST_OF_TY_Hash list  len kov rv i;
    rv = 0;
    len = BlkValueRead(list, LIST_LENGTH_F);
    kov = BlkValueRead(list, LIST_ITEM_KOV_F);
    for (i=0: i<len: i++)
        rv = rv * 33 + KOVHashValue(kov, BlkValueRead(list, i+LIST_ITEM_BASE));
    return rv;
];
```

§**9. Printing.** Unusually, this function can print the value in one of several formats: 0 for a comma-separated list; 1 for a braced, set-notation list; 2 for a comma-separated list with definite articles, which only makes sense if the list contains objects; 3 for a comma-separated list with indefinite articles. Note that a list in this sense is *not* printed using the "ListWriter.i6t" code for elaborate lists of objects, and it doesn't use the "listing contents of..." activity in any circumstances.

```
[ LIST_OF_TY_Say list format no_items v i bk;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    bk = KindAtomic(BlkValueRead(list, LIST_ITEM_KOV_F));
    ! print "kov=", BlkValueRead(list, LIST_ITEM_KOV_F), ":";
    if (format == 1) print "{";
    for (i=0:i<no_items:i++) {
        v = BlkValueRead(list, i+LIST_ITEM_BASE);
        switch (format) {
            2: print (the) v;
            3: print (a) v;
            default:
```

```
                if (bk == LIST_OF_TY) LIST_OF_TY_Say(v, 1);
                else if ((bk == TEXT_TY or INDEXED_TEXT_TY) && (format == 1)) {
                    print "~"; PrintKindValuePair(bk, v); print "~";
                }
                else PrintKindValuePair(bk, v);
        }
        if (i<no_items-2) print ", ";
        if (i==no_items-2) {
            if (format == 1) print ", "; else {
                #ifdef SERIAL_COMMA; if (no_items ~= 2) print ","; #endif;
                print (string) LISTAND2__TX;
            }
        }
    }
    if (format == 1) print "}";
];
```

§**10. List From Description.** That completes the compulsory services required for this KOV to function: from here on, the remaining routines provide definitions of stored action-related phrases in the Standard Rules.

Given a description `D` which applies to some objects and not others – say, "lighted rooms adjacent to the Transport Hub" – we can cast this into a list of all objects satisfying `D` with the following routine. Slightly wastefully of time, we have to iterate through the objects twice in order first to work out the length of list we will need, and then to transcribe them.

```
[ LIST_OF_TY_Desc list desc kov obj no_items ex len i;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    ex = BlkValueExtent(list);
    len = desc(-3);
! for (len=0, obj=desc(-2, nothing, len): obj: len++, obj=desc(-2, obj, len)) ;
! len++;
    if (len+LIST_ITEM_BASE > ex) {
        if (BlkValueSetExtent(list, len+LIST_ITEM_BASE) == false)
            return 0;
    }
    if (kov) BlkValueWrite(list, LIST_ITEM_KOV_F, kov);
    else BlkValueWrite(list, LIST_ITEM_KOV_F, OBJECT_TY);
    BlkValueWrite(list, LIST_LENGTH_F, len);
    obj = 0;
    for (i=0: i<len: i++) {
        obj = desc(-2, obj, i);
        ! print "i = ", i, " and obj = ", obj, "^";
        BlkValueWrite(list, i+LIST_ITEM_BASE, obj);
    }
    return list;
];
```

§**11. Find Item.** We test whether a list `list` includes a value equal to `v` or not. Equality here is in the sense of the list's comparison function: thus for indexed texts or other lists, say, deep comparisons rather than simple pointer comparisons are performed. In other words, one copy of "Alert" is equal to another.

```
[ LIST_OF_TY_FindItem list v i no_items cf;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) rfalse;
    cf = LIST_OF_TY_ComparisonFn(list);
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if (cf == 0 or UnsignedCompare) {
        for (i=0; i<no_items; i++)
            if (v == BlkValueRead(list, i+LIST_ITEM_BASE)) rtrue;
    } else {
        for (i=0; i<no_items; i++)
            if (cf(v, BlkValueRead(list, i+LIST_ITEM_BASE)) == 0) rtrue;
    }
    rfalse;
];
```

§**12. Insert Item.** The following routine inserts an item into the list. If this would break the size of the current block-value, then we extend by at least enough room to hold at least another 16 entries.

In the call `LIST_OF_TY_InsertItem(list, v, posnflag, posn, nodups)`, only the first two arguments are compulsory.

(a) If `nodups` is set, and an item equal to `v` is already present in the list, we return and do nothing. (`nodups` means "no duplicates".)
(b) Otherwise, if `posnflag` is `false`, we append a new entry `v` at the back of the given `list`.
(c) Otherwise, when `posnflag` is `true`, `posn` indicates the insertion position, from 1 (before the current first item) to $N+1$ (after the last), where N is the number of items in the list at present.

```
[ LIST_OF_TY_InsertItem list v posnflag posn nodups i no_items ex nv;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    if (nodups && (LIST_OF_TY_FindItem(list, v))) return list;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if ((posnflag) && ((posn<1) || (posn > no_items+1))) {
        print "*** Couldn't add at entry ", posn, " in the list ";
        LIST_OF_TY_Say(list, true);
        print ", which has entries in the range 1 to ", no_items, " ***^";
        RunTimeProblem(RTP_LISTRANGEERROR);
        rfalse;
    }
    ex = BlkValueExtent(list);
    if (no_items+LIST_ITEM_BASE+1 > ex) {
        if (BlkValueSetExtent(list, ex+16) == false) return 0;
    }
    if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F))) {
        nv = BlkValueCreate(BlkValueRead(list, LIST_ITEM_KOV_F));
        BlkValueCopy(nv, v);
        v = nv;
    }
    if (posnflag) {
        posn--;
        for (i=no_items:i>posn:i--) {
            BlkValueWrite(list, i+LIST_ITEM_BASE,
```

```
                BlkValueRead(list, i-1+LIST_ITEM_BASE));
        }
        BlkValueWrite(list, posn+LIST_ITEM_BASE, v);
    } else {
        BlkValueWrite(list, no_items+LIST_ITEM_BASE, v);
    }
    BlkValueWrite(list, LIST_LENGTH_F, no_items+1);
    return list;
];
```

§**13. Append List.** Instead of adjoining a single value, we adjoin an entire second list, which must be of a compatible kind of value (something which NI's type-checking machinery polices for us). Except that we have a list `more` rather than a value `v` to insert, the specification is the same as for `LIST_OF_TY_InsertItem`.

```
[ LIST_OF_TY_AppendList list more posnflag posn nodups v i j no_items msize ex nv;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    if ((more==0) || (BlkType(more) ~= LIST_OF_TY)) return list;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if ((posnflag) && ((posn<1) || (posn > no_items+1))) {
        print "*** Couldn't add at entry ", posn, " in the list ";
        LIST_OF_TY_Say(list, true);
        print ", which has entries in the range 1 to ", no_items, " ***^";
        RunTimeProblem(RTP_LISTRANGEERROR);
        rfalse;
    }
    msize = BlkValueRead(more, LIST_LENGTH_F);
    ex = BlkValueExtent(list);
    if (no_items+msize+LIST_ITEM_BASE > ex) {
        if (BlkValueSetExtent(list, no_items+msize+LIST_ITEM_BASE+8) == false)
            return 0;
    }
    if (posnflag) {
        posn--;
        for (i=no_items+msize:i>=posn+msize:i--) {
            BlkValueWrite(list, i+LIST_ITEM_BASE,
                BlkValueRead(list, i-msize+LIST_ITEM_BASE));
        }
        ! BlkValueWrite(list, posn, v);
        for (j=0: j<msize: j++) {
            v = BlkValueRead(more, j+LIST_ITEM_BASE);
            if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F))) {
                nv = BlkValueCreate(BlkValueRead(list, LIST_ITEM_KOV_F));
                BlkValueCopy(nv, v);
                v = nv;
            }
            BlkValueWrite(list, posn+j+LIST_ITEM_BASE, v);
        }
    } else {
        for (i=0, j=0: i<msize: i++) {
            v = BlkValueRead(more, i+LIST_ITEM_BASE);
            if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F))) {
                nv = BlkValueCreate(BlkValueRead(list, LIST_ITEM_KOV_F));
                BlkValueCopy(nv, v);
```

```
                v = nv;
            }
            if ((nodups == 0) || (LIST_OF_TY_FindItem(list, v) == false)) {
                BlkValueWrite(list, no_items+j+LIST_ITEM_BASE, v);
                j++;
            }
        }
    }
    BlkValueWrite(list, LIST_LENGTH_F, no_items+j);
    return list;
];
```

§**14. Remove Value.** We remove every instance of the value `v` from the given `list`. If the optional flag `forgive` is set, then we make no complaint if no value of `v` was present in the first place: otherwise, we issue a run-time problem.

Note that if the list contains block-values then the value must be properly destroyed with `BlkFree` before being overwritten as the items shuffle down.

```
[ LIST_OF_TY_RemoveValue list v forgive i j no_items odsize f cf delendum;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) rfalse;
    cf = LIST_OF_TY_ComparisonFn(list);
    no_items = BlkValueRead(list, LIST_LENGTH_F); odsize = no_items;
    for (i=0; i<no_items; i++) {
        delendum = BlkValueRead(list, i+LIST_ITEM_BASE);
        if (cf == 0 or UnsignedCompare)
            f = (v == delendum);
        else
            f = (cf(v, delendum) == 0);
        if (f) {
            if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F)))
                BlkFree(delendum);
            for (j=i+1; j<no_items; j++)
                BlkValueWrite(list, j-1+LIST_ITEM_BASE,
                    BlkValueRead(list, j+LIST_ITEM_BASE));
            no_items--; i--;
            BlkValueWrite(list, LIST_LENGTH_F, no_items);
        }
    }
    if (odsize ~= no_items) rfalse;
    if (forgive) rfalse;
    print "*** Couldn't remove: the value ";
    PrintKindValuePair(BlkValueRead(list, LIST_ITEM_KOV_F), v);
    print " was not present in the list ";
    LIST_OF_TY_Say(list, true);
    print " ***^";
    RunTimeProblem(RTP_LISTRANGEERROR);
];
```

§**15. Remove Item Range.** We excise items `from` to `to` from the given `list`, which numbers its items upwards from 1. If the optional flag `forgive` is set, then we truncate a range overspilling the actual list, and we make no complaint if it turns out that there is then nothing to be done: otherwise, in either event, we issue a run-time problem.

Once again, we destroy any block-values whose pointers will be overwritten as the list shuffles down to fill the void.

```
[ LIST_OF_TY_RemoveItemRange list from to forgive i d no_items;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) rfalse;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if ((from > to) || (from <= 0) || (to > no_items)) {
        if (forgive) {
            if (from <= 0) from = 1;
            if (to >= no_items) to = no_items;
            if (from > to) return list;
        } else {
            print "*** Couldn't remove entries ", from, " to ", to, " from the list ";
            LIST_OF_TY_Say(list, true);
            print ", which has entries in the range 1 to ", no_items, " ***^";
            RunTimeProblem(RTP_LISTRANGEERROR);
            rfalse;
        }
    }
    to--; from--;
    d = to-from+1;
    if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F)))
        for (i=0; i<d; i++)
            BlkFree(BlkValueRead(list, from+i+LIST_ITEM_BASE));
    for (i=from: i<no_items-d: i++)
        BlkValueWrite(list, i+LIST_ITEM_BASE,
            BlkValueRead(list, i+d+LIST_ITEM_BASE));
    BlkValueWrite(list, LIST_LENGTH_F, no_items-d);
    return list;
];
```

§**16. Remove List.** We excise all values from the removal list `rlist`, wherever they occur in `list`. Inevitably, given that we haven't sorted these lists and can spare neither time nor storage to do so, this is an expensive process with a running time proportional to the product of the two list sizes: we accept that as an overhead because in practice the `rlist` is almost always small in real-world use.

If the initial lists were disjoint, so that no removals occur, we always forgive the user: the request was not necessarily a foolish one, it only happened in this situation to be unhelpful.

```
[ LIST_OF_TY_Remove_List list rlist i j k v w no_items odsize rsize cf f;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) rfalse;
    no_items = BlkValueRead(list, LIST_LENGTH_F); odsize = no_items;
    rsize = BlkValueRead(rlist, LIST_LENGTH_F);
    cf = LIST_OF_TY_ComparisonFn(list);
    for (i=0: i<no_items: i++) {
        v = BlkValueRead(list, i+LIST_ITEM_BASE);
        for (k=0: k<rsize: k++) {
            w = BlkValueRead(rlist, k+LIST_ITEM_BASE);
            if (cf == 0 or UnsignedCompare)
```

```
                f = (v == w);
            else
                f = (cf(v, w) == 0);
            if (f) {
                if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F)))
                    BlkFree(v);
                for (j=i+1: j<no_items: j++)
                    BlkValueWrite(list, j+LIST_ITEM_BASE-1,
                        BlkValueRead(list, j+LIST_ITEM_BASE));
                no_items--; i--;
                BlkValueWrite(list, LIST_LENGTH_F, no_items);
                break;
            }
        }
    }
    rfalse;
];
```

§17. Get Length.

```
[ LIST_OF_TY_GetLength list;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    return BlkValueRead(list, LIST_LENGTH_F);
];
[ LIST_OF_TY_Empty list;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) rfalse;
    if (BlkValueRead(list, LIST_LENGTH_F) == 0) rtrue;
    rfalse;
];
```

§18. Set Length. This is rather harder: it might lengthen the list, in which case we have to pad out with the default value for the kind of value stored – padding a list of numbers with 0s, a list of texts with copies of the empty text, and so on – creating such block-values as might be needed; or else it might shorten the list, in which case we must cut items, destroying them properly if they were block-values.

`LIST_OF_TY_SetLength(list, newsize, this_way_only, truncation_end)` alters the length of the given `list` to `newsize`. If `this_way_only` is 1, the list is only allowed to grow, and nothing happens if we have asked to shrink it; if it is -1, the list is only allowed to shrink; if it is 0, the list is allowed either to grow or shrink. In the event that the list does have to shrink, entries must be removed, and we remove from the end if `truncation_end` is 1, or from the start if it is -1.

```
[ LIST_OF_TY_SetLength list newsize this_way_only truncation_end no_items ex i dv;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    if (newsize < 0) "*** Cannot resize a list to negative length ***";
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if (no_items < newsize) {
        if (this_way_only == -1) return list;
        ex = BlkValueExtent(list);
        if (newsize+LIST_ITEM_BASE > ex) {
            if (BlkValueSetExtent(list, newsize+LIST_ITEM_BASE) == false)
                return 0;
        }
        dv = DefaultValueOfKOV(BlkValueRead(list, LIST_ITEM_KOV_F));
```

```
        for (i=no_items: i<newsize: i++)
            BlkValueWrite(list, LIST_ITEM_BASE+i, dv);
        BlkValueWrite(list, LIST_LENGTH_F, newsize);
    }
    if (no_items > newsize) {
        if (this_way_only == 1) return list;
        if (truncation_end == -1) {
            if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F)))
                for (i=0: i<no_items-newsize: i++)
                    BlkFree(BlkValueRead(list, LIST_ITEM_BASE+i));
            for (i=0: i<newsize: i++)
                BlkValueWrite(list, LIST_ITEM_BASE+i,
                    BlkValueRead(list, LIST_ITEM_BASE+no_items-newsize+i));
        } else {
            if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F)))
                for (i=newsize: i<no_items: i++)
                    BlkFree(BlkValueRead(list, LIST_ITEM_BASE+i));
        }
        BlkValueWrite(list, LIST_LENGTH_F, newsize);
    }
    return list;
];
```

§19. Get Item.

```
[ LIST_OF_TY_GetItem list i forgive no_items;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if ((i<=0) || (i>no_items)) {
        if (forgive) return false;
        print "*** Couldn't read from entry ", i, " of a list which";
        switch (no_items) {
            0: print " is empty ***^";
            1: print " has only one entry, numbered 1 ***^";
            default: print " has entries numbered from 1 to ", no_items, " ***^";
        }
        RunTimeProblem(RTP_LISTRANGEERROR);
        if (no_items >= 1) i = 1;
        else return false;
    }
    return BlkValueRead(list, LIST_ITEM_BASE+i-1);
];
```

§**20. Write Item.** The slightly odd name for this function comes about because our usual way to convert an rvalue such as `LIST_OF_TY_GetItem(L, 4)` is to prefix `Write`, so that it becomes `WriteLIST_OF_TY_GetItem(L, 4)`.

```
[ WriteLIST_OF_TY_GetItem list i val no_items;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if ((i<=0) || (i>no_items)) {
        print "*** Couldn't write to list entry ", i, " of a list which";
        switch (no_items) {
            0: print " is empty ***^";
            1: print " has only one entry, numbered 1 ***^";
            default: print " has entries numbered from 1 to ", no_items, " ***^";
        }
        return RunTimeProblem(RTP_LISTRANGEERROR);
    }
    BlkValueWrite(list, LIST_ITEM_BASE+i-1, val);
];
```

§**21. Put Item.** Higher-level code should not use `Write_LIST_OF_TY_GetItem`, because it does not properly keep track of block-value copying: the following should be used instead.

```
[ LIST_OF_TY_PutItem list i v  no_items nv;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F))) {
        nv = BlkValueCreate(BlkValueRead(list, LIST_ITEM_KOV_F));
        BlkValueCopy(nv, v);
        v = nv;
    }
    if ((i<=0) || (i>no_items)) return false;
    BlkValueWrite(list, LIST_ITEM_BASE+i-1, v);
];
```

§**22. Multiple Object List.** The parser uses one data structure which is really a list: but which can't be represented as such because the heap might not exist. This is the multiple object list, which is used to handle commands like TAKE ALL by firing off a sequence of actions with one of the objects taken from entries in turn of the list. The following converts it to a list structure.

```
[ LIST_OF_TY_Mol list len i;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    len = multiple_object-->0;
    LIST_OF_TY_SetLength(list, len);
    for (i=1: i<=len: i++)
        LIST_OF_TY_PutItem(list, i, multiple_object-->i);
    return list;
];
[ LIST_OF_TY_Set_Mol list len i;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    len = BlkValueRead(list, LIST_LENGTH_F);
    if (len > 63) len = 63;
    multiple_object-->0 = len;
```

```
    for (i=1: i<=len: i++)
        multiple_object-->i = BlkValueRead(list, LIST_ITEM_BASE+i-1);
];
```

§**23. Reversing.** Reversing a list is, happily, a very efficient operation when the list contains block-values: because the pointers are rearranged but none is duplicated or destroyed, we can for once ignore the fact that they are pointers to block-values and simply move them around like any other data.

```
[ LIST_OF_TY_Reverse list no_items i v;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if (no_items < 2) return list;
    for (i=0;i*2<no_items;i++) {
        v = BlkValueRead(list, LIST_ITEM_BASE+i);
        BlkValueWrite(list, LIST_ITEM_BASE+i,
            BlkValueRead(list, LIST_ITEM_BASE+no_items-1-i));
        BlkValueWrite(list, LIST_ITEM_BASE+no_items-1-i, v);
    }
    return list;
];
```

§**24. Rotation.** The same is true of rotation. Here, "forwards" rotation means towards the end of the list, "backwards" means towards the start.

```
[ LIST_OF_TY_Rotate list backwards  no_items i v;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if (no_items < 2) return list;
    if (backwards) {
        v = BlkValueRead(list, LIST_ITEM_BASE);
        for (i=0:i<no_items-1:i++)
            BlkValueWrite(list, LIST_ITEM_BASE+i,
                BlkValueRead(list, LIST_ITEM_BASE+i+1));
        BlkValueWrite(list, no_items-1+LIST_ITEM_BASE, v);
    } else {
        v = BlkValueRead(list, no_items-1+LIST_ITEM_BASE);
        for (i=no_items-1:i>0:i--)
            BlkValueWrite(list, LIST_ITEM_BASE+i,
                BlkValueRead(list, LIST_ITEM_BASE+i-1));
        BlkValueWrite(list, LIST_ITEM_BASE, v);
    }
    return list;
];
```

§**25. Sorting.** And the same, again, is true of sorting: but we do have to take note of block values when it comes to performing comparisons, because we can only compare items in the list by looking at their contents, not the pointers to their contents.

`LIST_OF_TY_Sort(list, dir, prop)` sorts the given `list` in ascending order if `dir` is 1, in descending order if `dir` is −1, or in random order if `dir` is 2. The comparison used is the one for the kind of value stored in the list, unless the optional argument `prop` is supplied, in which case we sort based not on the item values but on their values for the property `prop`. (This only makes sense if the list contains objects.)

```
Global LIST_OF_TY_Sort_cf;

[ LIST_OF_TY_Sort list dir prop cf  i j no_items v;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if (dir == 2) {
        if (no_items < 2) return;
        for (i=1:i<no_items:i++) {
            j = random(i+1) - 1;
            v = BlkValueRead(list, LIST_ITEM_BASE+i);
            BlkValueWrite(list, LIST_ITEM_BASE+i, BlkValueRead(list, LIST_ITEM_BASE+j));
            BlkValueWrite(list, LIST_ITEM_BASE+j, v);
        }
        return;
    }
    SetSortDomain(ListSwapEntries, ListCompareEntries);
    if (cf) { LIST_OF_TY_Sort_cf = BlkValueCompare; ! dir = -dir;
    }
    else LIST_OF_TY_Sort_cf = 0;
    SortArray(list, prop, dir, no_items, false, 0);
];

[ ListSwapEntries list i j v;
    if (i==j) return;
    v = BlkValueRead(list, LIST_ITEM_BASE+i-1);
    BlkValueWrite(list, LIST_ITEM_BASE+i-1, BlkValueRead(list, LIST_ITEM_BASE+j-1));
    BlkValueWrite(list, LIST_ITEM_BASE+j-1, v);
];

[ ListCompareEntries list col i j d cf;
    if (i==j) return 0;
    i = BlkValueRead(list, LIST_ITEM_BASE+i-1);
    j = BlkValueRead(list, LIST_ITEM_BASE+j-1);
    if (I7S_Col) {
        if (i provides I7S_Col) i=i.I7S_Col; else i=0;
        if (j provides I7S_Col) j=j.I7S_Col; else j=0;
        cf = LIST_OF_TY_Sort_cf;
    } else {
        cf = LIST_OF_TY_ComparisonFn(list);
    }
    if (cf == 0)
        return i - j;
    else
        return cf(i, j);
];

#IFNOT; ! IFDEF MEMORY_HEAP_SIZE

[ LIST_OF_TY_Support t a b c; rfalse; ];
[ LIST_OF_TY_Say list; ];
```

```
[ LIST_OF_TY_FindItem list v; rfalse; ];
[ LIST_OF_TY_Empty list; rfalse; ];
[ LIST_OF_TY_SetLength l n; rfalse; ];
[ LIST_OF_TY_InsertItem a b c d e; rfalse; ];
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```