ListWriter Template B/lwt

Purpose

A flexible object-lister taking care of plurals, inventory information, various formats and so on.

§**1. Specification.** The list-writer is called by one of the following function calls:

(1) `WriteListOfMarkedObjects(style)`, where the set of objects listed is understood to be exactly those with the `workflag2` attribute set, and
   (a) the `style` is a sum of `*_BIT` constants as defined in "Definitions.i6t".

(2) `WriteListFrom(obj, style, depth, noactivity, iterator)`, where only the first two parameters are compulsory:
   (a) the set of objects listed begins with `obj`;
   (b) the `style` is a sum of `*_BIT` constants as defined in "Definitions.i6t";
   (c) the `depth` is the recursion depth within the list – ordinarily 0, but by supplying a higher value, we can simulate a sublist of another list;
   (d) `noactivity` is a flag which forces the list-writer to ignore the "listing the contents of" activity (in cases where it would otherwise consult this): by default this is `false`;
   (e) `iterator` is an iterator function which provides the objects in sequence.

`WriteListOfMarkedObjects` is simply a front-end which supplies suitable parameters for `WriteListFrom`.

The iterator function is by default `ObjectTreeIterator`. This defines the sequence of objects as being the children of the parent of `obj`, in object tree sequence (that is: `child(parent(obj))` is first). Moreover, when using `ObjectTreeIterator`, the "listing the contents of" activity is carried out, unless `noactivity` is set.

We also provide the iterator function `MarkedListIterator`, which defines the sequence of objects as being the list in the word array `MarkedObjectArray` with length `MarkedObjectLength`. Here the "listing the contents of" activity is never used, since the objects are not necessarily the contents of any single thing. This of course is the iterator used by `WriteListOfMarkedObjects(style)`: it works by filling this array with all the objects having `workflag2` set.

The full specification for iterator functions is given below.

The list-writer automatically groups adjacent and indistinguishable terms in the sequence into plurals, and carries out the "printing the plural name" activity to handle these. Doing this alone would result in text such as "You can see a cake, three coins, an onion, and two coins here", where the five coins are mentioned in two clauses because they happen not to be adjacent in the list. The list-writer therefore carries out the activity "grouping together" to see if the user would like to tie certain objects together into a single entry: what this does is to set suitable `list_together` properties for the objects. NI has already given plural objects a similar `list_together` property. The net effect is that any entries in the list with a non-zero value of `list_together` must be adjacent to each other.

We could achieve that by sorting the list in order of `list_together` value, but that would result in drastic movements, whereas we want to upset the original ordering as little as possible. So instead we use a process called *coalescing* the list. This is such that for every value $L \neq 0$ of `list_together`, every entry with that value is moved back in the list to follow the first entry with that value. Thus if the original order is $x_1, x_2, ..., x_N$ then x_j still precedes x_k in coalesced order unless there exists $i < j < k$ such that $L(i) = L(k) \neq 0$ and $L(j) \neq L(i)$. This is as stable as it can be while achieving the "interval" property that non-zero L values occur in contiguous runs.

We therefore obtain text such as "You can see a cake, five coins, the tiles W, X, Y and Z from a Scrabble set, and an onion here." where the coins and the Scrabble tiles have been coalesced together in the list.

It's important to note that the default `ObjectTreeIterator` has the side-effect of actually reordering the object tree: it rearranges the children being listed so that they appear in the tree in coalesced order. The `MarkedListIterator` used by `WriteListOfMarkedObjects` has the same side-effect if the marked objects all happen to share a common parent. It might seem odd for a list-writer to have side effects at all, but the idea is that occasional coalescing improves the quality of play in many small ways – for instance, the sequence of matches to TAKE ALL is tidier – and that coalescing has a small speed cost, so we want to do it as little as possible. (The latter was more of a consideration for I6: interpreters are faster nowadays.)

This specification is somewhat stronger than that of the I6 library's traditional list-writer, because
(i) it supports arbitrary lists of objects, not just children of specific parents, while still allowing coalesced and grouped lists,
(ii) it can be used recursively in all cases,
(iii) it uses its own memory, rather than borrowing memory from the parser, so that it can safely be used while the parser is working, and
(iv) it manages this memory more flexibly and without silently failing by buffer overruns on unexpectedly large lists.

The I7 version of `WriteListFrom`, when using `ObjectTreeIterator`, differs from the I6 version in that the object `o` is required to be the `child` of its `parent`, that is, to be the eldest child. (So in effect it's a function of parents, not children, but we retain the form for familiarity's sake.) In practice this was invariably the way `WriteListFrom` was used even in I6 days.

Finally, the `ISARE_BIT` is differently interpreted in I7: instead of printing something like " are ducks and drakes", as it would have done in I6, the initial space is now suppressed and we instead print "are ducks and drakes".

§**2. Memory.** The list-writer needs to dynamically allocate temporary array space of a known size, in such a way that the array is effectively on the local stack frame: if only either the Z-machine or Glulx supported a stack in memory, this would be no problem, but they do not and we must therefore use the following.

The size of the stack is such that it can support a list which includes every object and recurses in turn to most other objects: in practice, this never happens. It would be nice to allocate more just in case, but the Z-machine is desperately short of array memory.

```
Constant REQUISITION_STACK_SIZE = 3*{-value:Data::Instances::count(K_object)};
Array requisition_stack --> REQUISITION_STACK_SIZE;
Global requisition_stack_pointer = 0;

[ RequisitionStack len top addr;
    top = requisition_stack_pointer + len;
    if (top > REQUISITION_STACK_SIZE) return false;
    addr = requisition_stack + requisition_stack_pointer*WORDSIZE;
    ! print "Allocating ", addr, " at pointer ", requisition_stack_pointer, "^";
    requisition_stack_pointer = top;
    return addr;
];

[ FreeStack addr;
    if (addr == 0) return;
    requisition_stack_pointer = (addr - requisition_stack)/WORDSIZE;
];
```

§**3. WriteListOfMarkedObjects.** This routine will use the `MarkedListIterator`. That means it has to create an array containing the object numbers of every object with the `workflag2` attribute set, placing the address of this array in `MarkedObjectArray` and the length in `MarkedObjectLength`. Note that we preserve any existing marked list on the stack (using the assembly-language instructions `@push` and `@pull`) for the duration of our use.

While the final order of this list will depend on what it looks like after coalescing, the initial order is also important. If all of the objects have a common parent in the object tree, then we coalesce those objects and place the list in object tree order. But if not, we place the list in object number order (which is essentially the order of traversal of the initial state of the object tree: thus objects in Room A will all appear before objects in Room B if A was created before B in the source text).

```
Global MarkedObjectArray = 0;
Global MarkedObjectLength = 0;

[ WriteListOfMarkedObjects style
    obj common_parent first mixed_parentage length;

    objectloop (obj ofclass Object && obj has workflag2) {
        length++;
        if (first == nothing) { first = obj; common_parent = parent(obj); }
        else { if (parent(obj) ~= common_parent) mixed_parentage = true; }
    }
    if (mixed_parentage) common_parent = nothing;

    if (length == 0) {
        if (style & ISARE_BIT ~= 0) print (string) IS3__TX, " ", (string) NOTHING__TX;
        else if (style & CFIRSTART_BIT ~= 0) print (string) NOTHING2__TX;
        else print (string) NOTHING__TX;
    } else {
        @push MarkedObjectArray; @push MarkedObjectLength;
        MarkedObjectArray = RequisitionStack(length);
        MarkedObjectLength = length;
        if (MarkedObjectArray == 0) return RunTimeProblem(RTP_LISTWRITERMEMORY);

        if (common_parent) {
            ObjectTreeCoalesce(child(common_parent));
            length = 0;
            objectloop (obj in common_parent) ! object tree order
                if (obj has workflag2) MarkedObjectArray-->length++ = obj;
        } else {
            length = 0;
            objectloop (obj ofclass Object) ! object number order
                if (obj has workflag2) MarkedObjectArray-->length++ = obj;
        }

        WriteListFrom(first, style, 0, false, MarkedListIterator);

        FreeStack(MarkedObjectArray);
        @pull MarkedObjectLength; @pull MarkedObjectArray;
    }
    return;
];
```

§4. About Iterator Functions. Suppose `Iter` is an iterator function and that we have a "raw list" $x_1, x_2, ..., x_n$ of objects. Of these, the iterator function will choose a sublist of "qualifying" objects. It is called with arguments

```
    Iter(obj, depth, L, function)
```

where the `obj` is required to be x_j for some j and the function is one of the `*_ITF` constants defined below:

(a) On `START_ITF`, we return x_1, or `nothing` if the list is empty.

(b) On `SEEK_ITF`, we return the smallest $k \geq j$ such that x_k qualifies, or `nothing` if none of $x_j, x_{j+1}, ..., x_n$ qualify.

(c) On `ADVANCE_ITF`, we return the smallest $k > j$ such that x_k qualifies, or `nothing` if none of $x_{j+1}, x_{j+2} ..., x_n$ qualify.

(d) On `COALESCE_ITF`, we coalesce the entire list (not merely the qualifying entries) and return the new x_1, or `nothing` if the list is empty.

Thus, given any x_i, we can produce the sublist of qualifying entries by performing `START_ITF` on x_i, then `SEEK_ITF`, to produce q_1; then `ADVANCE_ITF` to produce q_2, and so on until `nothing` is returned, when there are no more qualifying entries. `SEEK_ITF` and `ADVANCE_ITF` always return qualifying objects, or `nothing`; but `START_ITF` and `COALESCE_ITF` may return a non-qualifying object, since there's no reason why x_1 should qualify in any ordering.

The iterator can make its own choice of which entries qualify, and of what the raw list is; `depth` is supplied to help in that decision. But if `L` is non-zero then the iterator is required to disqualify any entry whose `list_together` value is not `L`.

```
Constant SEEK_ITF = 0;
Constant ADVANCE_ITF = 1;
Constant COALESCE_ITF = 2;
Constant START_ITF = 3;

! Constant DBLW; ! Uncomment this to provide debugging information at run-time
```

§5. Marked List Iterator. Here the raw list is provided by the `MarkedObjectArray`, which is convenient for coalescing, but not so helpful for translating the `obj` parameter into the i such that it is x_i. We simply search from the beginning to do this, which combined with other code using the iterator makes for some unnecessary $O(n^2)$ calculations. But it saves memory and nuisance, and the iterator is used only with printing to the screen, which never needs to be very rapid anyway (because the player can only read very slowly).

```
[ MarkedListIterator obj depth required_lt function i;
    if (obj == nothing) return nothing;
    switch(function) {
        START_ITF: return MarkedObjectArray-->0;
        COALESCE_ITF: return MarkedListCoalesce();
        SEEK_ITF, ADVANCE_ITF:
            for (i=0: i<MarkedObjectLength: i++)
                if (MarkedObjectArray-->i == obj) {
                    if (function == ADVANCE_ITF) i++;
                    for (:i<MarkedObjectLength: i++) {
                        obj = MarkedObjectArray-->i;
                        if ((required_lt) && (obj.list_together ~= required_lt)) continue;
                        if ((c_style & WORKFLAG_BIT) && (depth==0) && (obj hasnt workflag))
                            continue;
                        if ((c_style & CONCEAL_BIT) &&
                            ((obj has concealed) || (obj has scenery))) continue;
                        return obj;
                    }
                    return nothing;
                }
    }
    return nothing;
];
```

§6. Coalesce Marked List. The return value is the new first entry in the raw list.

```
[ MarkedListCoalesce o i lt l swap m;
    for (i=0: i<MarkedObjectLength: i++) {
        lt = (MarkedObjectArray-->i).list_together;
        if (lt ~= 0) {
            ! Find first object in list after contiguous run with this list_together value:
            for (i++: (i<MarkedObjectLength)&&((MarkedObjectArray-->i).list_together==lt): i++) ;
            ! If the contiguous run extends to end of list, the list is now perfect:
            if (i == MarkedObjectLength) return MarkedObjectArray-->0;
            ! And otherwise we look to see if any future entries belong in the earlier run:
            for (l=i+1: l<MarkedObjectLength: l++)
                if ((MarkedObjectArray-->l).list_together == lt) {
                    ! Yes, they do: so we perform a rotation to insert it before element i:
                    swap = MarkedObjectArray-->l;
                    for (m=l: m>i: m--) MarkedObjectArray-->m = MarkedObjectArray-->(m-1);
                    MarkedObjectArray-->i = swap;
                    ! And now the run is longer:
                    i++;
                    if (i == MarkedObjectLength) return MarkedObjectArray-->0;
                }
            i--;
        }
    }
    return MarkedObjectArray-->0;
];
```

§7. Object Tree Iterator. Here the raw list is the list of all children of a given parent: since the argument `obj` is required to be a member of the raw list, we can use `parent(obj)` to find it. Now seeking and advancing are fast, but coalescing is slower.

```
Global list_filter_routine;

[ ObjectTreeIterator obj depth required_lt function;
    if ((obj == nothing) || (parent(obj) == nothing)) return nothing;
    if (function == START_ITF) return child(parent(obj));
    if (function == COALESCE_ITF) return ObjectTreeCoalesce(obj);
    if (function == ADVANCE_ITF) obj = sibling(obj);
    for (:: obj = sibling(obj)) {
        if (obj == nothing) return nothing;
        if ((required_lt) && (obj.list_together ~= required_lt)) continue;
        if ((c_style & WORKFLAG_BIT) && (depth==0) && (obj hasnt workflag)) continue;
        if (obj hasnt list_filter_permits) continue;
        if ((c_style & CONCEAL_BIT) &&
            ((obj has concealed) || (obj has scenery))) continue;
        return obj;
    }
];
```

§**8. Coalesce Object Tree.** Again, the return value is the new first entry in the raw list.

```
[ ObjectTreeCoalesce obj memb lt later;
    #Ifdef DBLW; print "^^Sorting out: "; DiagnoseSortList(obj); #Endif;
    .StartAgain;
    for (memb=obj: memb~=nothing: memb=sibling(memb)) {
        lt = memb.list_together;
        if (lt ~= 0) {
            ! Find first object in list after contiguous run with this list_together value:
            for (memb=sibling(memb): (memb) && (memb.list_together == lt): memb = sibling(memb)) ;
            ! If the contiguous run extends to end of list, the list is now perfect:
            if (memb == 0) return obj;
            ! And otherwise we look to see if any future entries belong in the earlier run:
            for (later=sibling(memb): later: later=sibling(later))
                if (later.list_together == lt) {
                    ! Yes, they do: so we perform a regrouping of the list and start again:
                    obj = GroupChildren(parent(obj), list_together, lt);
                    #Ifdef DBLW; print "^^Sorted to: "; DiagnoseSortList(obj); #Endif;
                    jump StartAgain;
                }
        }
    }
    return obj;
];
#Ifdef DBLW;
[ DiagnoseSortList obj memb;
    for (memb=child(obj): memb~=nothing: memb=sibling(memb)) print memb, " --> "; new_line;
];
#Endif;
```

§**9. WriteListFrom.** And here we go at last. Or at any rate we initialise the quartet of global variables detailing the current list-writing process, and begin.

```
[ WriteListFrom first style depth noactivity iter a ol;
    @push c_iterator; @push c_style; @push c_depth; @push c_margin;
    if (iter) c_iterator = iter; else c_iterator = ObjectTreeIterator;
    c_style = style; c_depth = depth;
    c_margin = 0; if (style & EXTRAINDENT_BIT) c_margin = 1;
    objectloop (a ofclass Object) {
        give a list_filter_permits;
        if ((list_filter_routine) && (list_filter_routine(a) == false))
            give a ~list_filter_permits;
    }
    first = c_iterator(first, depth, 0, START_ITF);
    if (first == nothing) {
        print (string) NOTHING__TX;
        if (style & NEWLINE_BIT ~= 0) new_line;
    } else {
        if ((noactivity) || (iter)) {
            WriteListR(first, c_depth, true);
            say__p = 1;
        } else {
            objectloop (ol provides list_together) ol.list_together = 0;
            CarryOutActivity(LISTING_CONTENTS_ACT, parent(first));
        }
    }
    @pull c_margin; @pull c_depth; @pull c_style; @pull c_iterator;
];
```

§**10. Standard Contents Listing Rule.** The default for the listing contents activity is to call this rule in its "for" stage: note that this suppresses the use of the activity, to avoid an infinite regress. The activity is used only for the default `ObjectTreeIterator`, so there is no need to specify which is used.

```
[ STANDARD_CONTENTS_LISTING_R;
    WriteListFrom(child(parameter_object), c_style, c_depth, true);
];
```

§**11. Partitioning.** Given qualifying objects $x_1, ..., x_j$, we partition them into classes of the equivalence relation $x_i \sim x_j$ if and only

(i) they both have a `plural` property (not necessarily the same), and
(ii) neither will cause the list-maker to recurse downwards to show the objects inside or on top of them, and
(iii) if they cause the list-maker to add information about whether they are worn, lighted or open, then it will add the same information about both, and
(iv) they are considered identical by the parser, in that they respond to the same syntaxes of name: so that it is impossible to find a TAKE X command such that X matches one and not the other.

The equivalence classes are numbered consecutively upwards from 1 to n, in order of first appearance in the list. For each object x_i, `partition_classes->(i-1)` is the number of its equivalence class. For each equivalence class number c, `partition_class_sizes->c` is the number of objects in this class.

```
#Ifdef DBLW;
Global DBLW_no_classes; Global DBLW_no_objs;
[ DebugPartition partition_class_sizes partition_classes first depth i k o;
    print "[Length of list is ", DBLW_no_objs, " with ", k, " plural.]^";
    print "[Partitioned into ", DBLW_no_classes, " equivalence classes.]^";
    for (i=1: i<=DBLW_no_classes : i++) {
        print "Class ", i, " has size ", partition_class_sizes->i, "^";
    }
    for (k=0, o=first: k<DBLW_no_objs : k++, o = c_iterator(o, depth, lt_value, ADVANCE_ITF)) {
        print "Entry ", k, " has class ", partition_classes->k,
            " represented by ", o, " with L=", o.list_together, "^";
    }
];
#Endif;
```

§**12. Partition List.** The following creates the `partition_classes` and `partition_class_sizes` accordingly. We return n, the number of classes.

```
[ PartitionList first no_objs depth partition_classes partition_class_sizes
    i k l n m;
    for (i=0: i<no_objs: i++) partition_classes->i = 0;
    n = 1;
    for (i=first, k=0: k<no_objs: i=c_iterator(i, depth, lt_value, ADVANCE_ITF), k++)
        if (partition_classes->k == 0) {
            partition_classes->k = n; partition_class_sizes->n = 1;
            for (l=c_iterator(i, depth, lt_value, ADVANCE_ITF), m=k+1:
                (l~=0) && (m<no_objs):
                l=c_iterator(l, depth, lt_value, ADVANCE_ITF), m++) {
                if ((partition_classes->m == 0) && (ListEqual(i, l))) {
                    if (partition_class_sizes->n < 255) (partition_class_sizes->n)++;
                    partition_classes->m = n;
                }
            }
            if (n < 255) n++;
        }
    n--;
    #Ifdef DBLW;
    DBLW_no_classes = n; DBLW_no_objs = no_objs;
    DebugPartition(partition_class_sizes, partition_classes, first, depth);
    #Endif;
    return n;
];
```

§**13. Equivalence Relation.** The above algorithm will fail unless `ListEqual` is indeed reflexive, symmetric and transitive, which ultimately depends on the care with which `Identical` is implemented, which in turn hangs on the `parse_noun` properties compiled by NI. But this seems to be sound.

```
[ ListEqual o1 o2;
    if ((o1.plural == 0) || (o2.plural == 0)) rfalse;
    if (child(o1) ~= 0 && WillRecurs(o1) ~= 0) rfalse;
    if (child(o2) ~= 0 && WillRecurs(o2) ~= 0) rfalse;
    if (c_style & (FULLINV_BIT + PARTINV_BIT) ~= 0) {
        if ((o1 hasnt worn && o2 has worn) || (o2 hasnt worn && o1 has worn)) rfalse;
        if ((o1 hasnt light && o2 has light) || (o2 hasnt light && o1 has light)) rfalse;
        if (o1 has container) {
            if (o2 hasnt container) rfalse;
            if ((o1 has open && o2 hasnt open) || (o2 has open && o1 hasnt open))
                rfalse;
        }
        else if (o2 has container)
            rfalse;
    }
    return Identical(o1, o2);
];

[ WillRecurs o;
    if (c_style & ALWAYS_BIT ~= 0) rtrue;
    if (c_style & RECURSE_BIT == 0) rfalse;
    if ((o has supporter) || ((o has container) && (o has open or transparent))) rtrue;
    rfalse;
];
```

§**14. Grouping.** A "group" is a maximally-sized run of one or more adjacent partition classes in the list whose first members have a common value of `list_together` which is a routine or string, and which is not equal to `lt_value`, the current grouping value. (As we see below, it's by setting `lt_value` that we restrict attention to a particular group: if we reacted to that as a `list_together` value here, then we would simply go around in circles, and never be able to see the individual objects in the group.)

For instance, suppose we have objects with `list_together` values as follows, where R_1 and R_2 are addresses of routines:

`coin` (R_1), `coin` (R_1), `box` (R_2), `statuette` (0), `coin` (R_1), `box` (R_2)

Then the partition is 1, 1, 2, 3, 1, 2, so that the final output will be something like "three coins, two boxes and a statuette" – with three grouped terms. Here each partition class is the only member in its group, so the number of groups is the same as the number of classes. (The above list is not coalesced, so the R_1 values, for instance, are not contiguous: we need to work in general with non-coalesced lists because not every iterator function will be able to coalesce fully.)

But if we have something like this:

`coin` (R_1), `Q` (R_2), `W` (R_2), `coin` (R_1), `statuette` (0), `E` (R_2), `R` (R_2)

then the partition is 1, 2, 3, 1, 4, 5, 6 and we have six classes in all. But classes 2 and 3 are grouped together, as are classes 5 and 6, so we end up with a list having just four groups: "two coins, the letters Q and W from a Scrabble set, a statuette and the letters E and R from a Scrabble set". (Again, this list has not been fully coalesced: if it had been, it would be reordered `coin`, `coin`, `Q`, `W`, `E`, `R`, `statuette`, with partition 1, 1, 2, 3, 4, 5, 6, and three groups: "two coins, the letters Q, W, E and R from a Scrabble set and a statuette".)

The reason we do not group together classes with a common non-zero `list_together` which is *not* a routine or string is that low values of `list_together` are used in coalescing the list into a pleasing order (say, moving all of the key-like items together) but not in grouping: see `list_together` in the *Inform Designer's Manual*, 4th edition.

We calculate the number of groups by starting with the number of classes and then subtracting one each time two adjacent classes share `list_together` in the above sense.

```
[ NumberOfGroupsInList o no_classes depth partition_classes partition_class_sizes
    no_groups cl memb k current_lt lt;
    no_groups = no_classes;
    for (cl=1, memb=o, k=0: cl<=no_classes: cl++) {
        ! Advance to first member of class number cl
        while (partition_classes->k ~= cl) {
            k++; memb = c_iterator(memb, depth, lt_value, ADVANCE_ITF);
        }
        if (memb) { ! In case of accidents, but should always happen
            lt = memb.list_together;
            if ((lt ~= lt_value) && (lt ofclass Routine or String) && (lt == current_lt))
                no_groups--;
            current_lt = lt;
        }
    }
    #Ifdef DBLW; print "[There are ", no_groups, " groups.]^"; #Endif;
    return no_groups;
];
```

§15. Write List Recursively. The big one: `WriteListR` is the heart of the list-writer.

```
[ WriteListR o depth from_start
    partition_classes partition_class_sizes
    cl memb index k2 l m no_classes q groups_to_do current_lt;
    if (o == nothing) return; ! An empty list: no output
    if (from_start) {
        o = c_iterator(o, depth, 0, COALESCE_ITF); ! Coalesce list and choose new start
    }
    o = c_iterator(o, depth, 0, SEEK_ITF); ! Find first entry in list from o
    if (o == nothing) return;
    ! Count index = length of list
    for (memb=o, index=0: memb: memb=c_iterator(memb, depth, lt_value, ADVANCE_ITF)) index++;
    if (c_style & ISARE_BIT ~= 0) {
        if (index == 1 && o hasnt pluralname) print (string) IS3__TX;
        else                                  print (string) ARE3__TX;
        if (c_style & NEWLINE_BIT ~= 0)       print ":^";
        else                                  print (char) ' ';
        c_style = c_style - ISARE_BIT;
    }
    partition_classes = RequisitionStack(index/WORDSIZE + 2);
    partition_class_sizes = RequisitionStack(index/WORDSIZE + 2);
    if ((partition_classes == 0) || (partition_class_sizes == 0))
        return RunTimeProblem(RTP_LISTWRITERMEMORY);
    no_classes =
        PartitionList(o, index, depth, partition_classes, partition_class_sizes);
    groups_to_do =
        NumberOfGroupsInList(o, no_classes, depth, partition_classes, partition_class_sizes);
    for (cl=1, memb=o, index=0, current_lt=0: groups_to_do>0: cl++) {
        ! Set memb to first object of partition class cl
        while (partition_classes->index ~= cl) {
            index++; memb=c_iterator(memb, depth, lt_value, ADVANCE_ITF);
            if (memb==0) { print "*** Error in list-writer ***^"; break; }
        }
        #Ifdef DBLW;
        ! DebugPartition(partition_class_sizes, partition_classes, o, depth);
        print "^[Class ", cl, " of ", no_classes, ": first object ", memb,
            " (", memb.list_together, "); groups_to_do ", groups_to_do, ",
            current_lt=", current_lt, " listing_size=", listing_size,
            " lt_value=", lt_value, " memb.list_together=", memb.list_together, "]^";
        #Endif;
        if ((memb.list_together == lt_value) ||
            (~~(memb.list_together ofclass Routine or String))) current_lt = 0;
        else {
            if (memb.list_together == current_lt) continue;
            ! Otherwise this class begins a new group
            @push listing_size;
            q = memb; listing_size = 1; l = index; m = cl;
            while (m < no_classes && q.list_together == memb.list_together) {
                m++;
                while (partition_classes->l ~= m) {
```

```
                l++; q = c_iterator(q, depth, lt_value, ADVANCE_ITF);
            }
            if (q.list_together == memb.list_together) listing_size++;
        }
        if (listing_size > 1) {
            ! The new group contains more than one partition class
            WriteMultiClassGroup(cl, memb, depth, partition_class_sizes);
            current_lt = memb.list_together;
            jump GroupComplete;
        }
        current_lt = 0;
        @pull listing_size;
    }
    WriteSingleClassGroup(cl, memb, depth, partition_class_sizes->cl);
    .GroupComplete;
    groups_to_do--;
    if (c_style & ENGLISH_BIT ~= 0) {
        if (groups_to_do == 1) {
            if (cl <= 1) print (string) LISTAND2__TX;
            else print (string) LISTAND__TX;
        }
        if (groups_to_do > 1) print (string) COMMA__TX;
    }
}
FreeStack(partition_class_sizes);
FreeStack(partition_classes);
]; ! end of WriteListR
```

§16. Write Multiple Class Group. The text of a single group which contains more than one partition class. We carry out the “grouping together” activity, so that the user can add text fore and aft – this is how groups of objects such as “X, Y and Z” can be fluffed up to “the letters X, Y and Z from a Scrabble set” – but the default is simple: we print the grouped items by calling `WriteListR` once again, but this time starting from X, and where `lt_value` is set to the common `list_together` value of X, Y and Z. (That restricts the list to just the objects in this group.) Because `lt_value` is set to this value, the grouping code won't then group X, Y and Z again, and they will instead be individual classes in the new list – so each will end up being sent, in turn, to `WriteSingleClassGroup` below, and *that* is where they are printed.

```
[ WriteMultiClassGroup cl memb depth partition_class_sizes q k2 l;
    ! Save the style, because the activity below is allowed to change it
    q = c_style;
    if (c_style & INDENT_BIT ~= 0) PrintSpaces(2*(depth+c_margin));
    BeginActivity(GROUPING_TOGETHER_ACT, memb);
    if (ForActivity(GROUPING_TOGETHER_ACT, memb)) {
        c_style = c_style &~ NEWLINE_BIT;
    } else {
        if (memb.list_together ofclass String) {
            ! Set k2 to the number of objects covered by the group
            k2 = 0;
            for (l=0 : l<listing_size : l++) k2 = k2 + partition_class_sizes->(l+cl);
            EnglishNumber(k2); print " ";
            print (string) memb.list_together;
            if (c_style & ENGLISH_BIT ~= 0) print " (";
            if (c_style & INDENT_BIT ~= 0)  print ":^";
        } else {
            inventory_stage = 1;
            parser_one = memb; parser_two = depth + c_margin;
            if (RunRoutines(memb, list_together) == 1) jump Omit__Sublist2;
        }
        c_margin++;
        @push lt_value; @push listing_together; @push listing_size;
        lt_value = memb.list_together; listing_together = memb;
        #Ifdef DBLW; print "^^DOWN lt_value = ", lt_value, " listing_together = ", memb, "^^";
        @push DBLW_no_classes; @push DBLW_no_objs; #Endif;
        WriteListR(memb, depth, false);
        #Ifdef DBLW; print "^^UP^^"; @pull DBLW_no_objs; @pull DBLW_no_classes; #Endif;
        @pull listing_size; @pull listing_together; @pull lt_value;
        c_margin--;
        if (memb.list_together ofclass String) {
            if (q & ENGLISH_BIT ~= 0) print ")";
        } else {
            inventory_stage = 2;
            parser_one = memb; parser_two = depth+c_margin;
            RunRoutines(memb, list_together);
        }
        .Omit__Sublist2;
    }
    EndActivity(GROUPING_TOGETHER_ACT, memb);
    ! If the NEWLINE_BIT has been forced by the activity, act now
    ! before it vanishes...
    if (q & NEWLINE_BIT ~= 0 && c_style & NEWLINE_BIT == 0) new_line;
    ! ...when the original style is restored again:
    c_style = q;
];
```

§17. Write Single Class Group. The text of a single group which contains exactly one partition class. Because of the way the multiple-class case recurses, every class ends up in this routine sooner or later; this is the place where the actual name of an object is printed, at long last.

```
[ WriteSingleClassGroup cl memb depth size q;
    q = c_style;
    if (c_style & INDENT_BIT) PrintSpaces(2*(depth+c_margin));
    if (size == 1) {
        if (c_style & NOARTICLE_BIT ~= 0) print (name) memb;
        else {
            if (c_style & DEFART_BIT) {
                if ((cl == 1) && (c_style & CFIRSTART_BIT)) print (The) memb;
                else print (the) memb;
            } else {
                if ((cl == 1) && (c_style & CFIRSTART_BIT)) print (CIndefArt) memb;
                else print (a) memb;
            }
        }
    } else {
        if (c_style & DEFART_BIT) {
            if ((cl == 1) && (c_style & CFIRSTART_BIT)) PrefaceByArticle(memb, 0, size);
            else PrefaceByArticle(memb, 1, size);
        }
        @push listing_size; listing_size = size;
        CarryOutActivity(PRINTING_A_NUMBER_OF_ACT, memb);
        @pull listing_size;
    }
    if ((size > 1) && (memb hasnt pluralname)) {
        give memb pluralname;
        WriteAfterEntry(memb, depth);
        give memb ~pluralname;
    } else WriteAfterEntry(memb, depth);
    c_style = q;
];
```

§18. Write After Entry. Each entry can be followed by supplementary, usually parenthetical, information: exactly what, depends on the style. The extreme case is when the style, and the object, call for recursion to list the object-tree contents: this is achieved by calling `WriteListR`, using the `ObjectTreeIterator` (whatever the iterator used at the top level) and increasing the depth by 1.

```
[ WriteAfterEntry o depth
    p recurse_flag parenth_flag eldest_child child_count combo;
    inventory_stage = 2;
    if (c_style & PARTINV_BIT) {
        BeginActivity(PRINTING_ROOM_DESC_DETAILS_ACT);
        if (ForActivity(PRINTING_ROOM_DESC_DETAILS_ACT) == false) {
        combo = 0;
        if (o has light && location hasnt light) combo=combo+1;
        if (o has container && o hasnt open)     combo=combo+2;
        if ((o has container && (o has open || o has transparent))
            && (child(o)==0))                    combo=combo+4;
        if (combo) L__M(##ListMiscellany, combo, o);
        }
        EndActivity(PRINTING_ROOM_DESC_DETAILS_ACT);
    }   ! end of PARTINV_BIT processing
    if (c_style & FULLINV_BIT) {
        if (o has light && o has worn) { L__M(##ListMiscellany, 8, o);  parenth_flag = true; }
        else {
            if (o has light)           { L__M(##ListMiscellany, 9, o);  parenth_flag = true; }
            if (o has worn)            { L__M(##ListMiscellany, 10, o); parenth_flag = true; }
        }
        if (o has container)
            if (o has openable) {
                if (parenth_flag) {
                    #Ifdef SERIAL_COMMA; print ","; #Endif;
                    print (string) AND__TX;
                } else           L__M(##ListMiscellany, 11, o);
                if (o has open)
                    if (child(o)) L__M(##ListMiscellany, 12, o);
                    else          L__M(##ListMiscellany, 13, o);
                else
                    if (o has lockable && o has locked) L__M(##ListMiscellany, 15, o);
                    else                                L__M(##ListMiscellany, 14, o);
                parenth_flag = true;
            }
            else
                if (child(o)==0 && o has transparent)
                    if (parenth_flag) L__M(##ListMiscellany, 16, o);
                    else              L__M(##ListMiscellany, 17, o);
        if (parenth_flag) print ")";
    }   ! end of FULLINV_BIT processing
    child_count = 0;
    eldest_child = nothing;
    objectloop (p in o)
        if ((c_style & CONCEAL_BIT == 0) || (p hasnt concealed && p hasnt scenery))
            if (p has list_filter_permits) {
                child_count++;
                if (eldest_child == nothing) eldest_child = p;
            }
    if (child_count && (c_style & ALWAYS_BIT)) {
        if (c_style & ENGLISH_BIT) L__M(##ListMiscellany, 18, o);
        recurse_flag = true;
    }
    if (child_count && (c_style & RECURSE_BIT)) {
        if (o has supporter) {
            if (c_style & ENGLISH_BIT) {
                if (c_style & TERSE_BIT) L__M(##ListMiscellany, 19, o);
                else                     L__M(##ListMiscellany, 20, o);
                if (o has animate)       print (string) WHOM__TX;
                else                     print (string) WHICH__TX;
            }
            recurse_flag = true;
        }
        if (o has container && (o has open || o has transparent)) {
            if (c_style & ENGLISH_BIT) {
                if (c_style & TERSE_BIT) L__M(##ListMiscellany, 21, o);
                else                     L__M(##ListMiscellany, 22, o);
                if (o has animate)       print (string) WHOM__TX;
                else                     print (string) WHICH__TX;
            }
            recurse_flag = true;
        }
    }
    if (recurse_flag && (c_style & ENGLISH_BIT))
        if (child_count > 1 || eldest_child has pluralname) print (string) ARE2__TX;
        else                                                print (string) IS2__TX;
    if (c_style & NEWLINE_BIT) new_line;
    if (recurse_flag) {
        o = child(o);
        @push lt_value; @push listing_together; @push listing_size;
        @push c_iterator;
        c_iterator = ObjectTreeIterator;
```

```
        lt_value = 0;   listing_together = 0;   listing_size = 0;
        WriteListR(o, depth+1, true);
        @pull c_iterator;
        @pull listing_size; @pull listing_together; @pull lt_value;
        if (c_style & TERSE_BIT) print ")";
    }
];
```