The List Set Generator: A Construct for Evaluating Set Expressions

STUART C. SHAPIRO
University of Wisconsin, Madison, Wisconsin*

The list set generator is defined and algorithms for its use are given. The list set generator is a construct which may be added to a list processing system or any system that handles sets. It efficiently generates the set which results from any expression involving sets and set operators. The efficiency derives from evaluating the expression as a whole and in parallel, rather than evaluating subexpressions and then using those sets to arrive at the final result.

KEY WORDS AND PHRASES: set manipulating, list processing, set generation, sets, lists, file processing
CR CATEGORIES: 3.73, 3.74, 4.22

In designing and implementing an associational net structure to be used as the data structure for a natural language question answering system [4], it became obvious that the operations of intersecting and unioning arbitrary numbers of sets would be performed frequently. It was, therefore, necessary to discover a very efficient method for doing this. This paper describes a generalization of the method which was found, which generalization allows for the very efficient evaluation of set expressions of arbitrary length and complexity. The techniques described below would be useful in language systems that have a set data type and in systems for manipulating ordered files as well as in associative data systems. Various versions of these algorithms have been programmed in Extended Algol for Burroughs B5500 and in PL/I for IBM's System/360.

Unordered sets may be represented as lists which do not contain duplicate elements. The set operations will be performed more efficiently if the lists are ordered on some internal code (see, for example [2]). The set operations difference (relative complement), union and intersection, which could be performed only very inefficiently on unordered lists representing sets can be done efficiently on these ordered lists. For example to intersect two unordered lists takes an amount of time proportional to the product of their lengths while to intersect two ordered lists takes an amount of time proportional to the sum of lengths. When intersecting more than two lists, even more time could be saved by reading all the lists in parallel rather than intersecting them by pairs. If three lists were to be intersected of lengths m, n, and r and the first two had s elements in common, intersecting them two at a time as unordered lists would take an amount of time proportional to $mn + sr$; the time to intersect them as ordered lists two at a time would be proportional to $m + n + s + r$; but the time to intersect them by comparing all three at once would be proportional to $m + n + r$. The same results would hold for the other set operations.

In this paper, we define a generalization of the list reader (see Weizenbaum's reader [5] and Knowlton's "bug" [3]) which, as it is incremented, produces the new set determined by set operations on given sets. The algorithms for incrementing the generalized reader embody the efficient parallel methods for performing the set operations on ordered lists.

We first introduce some basic terminology.

D1. A *list set* is an ordered, finite list no two of whose elements are equal.

The ordering relation used in list sets is immaterial. In fact, different orderings may be used on different lists and any equivalence relation may be used for equating elements of different lists. The restriction is that if two elements are equivalent, then no element that appears after one of them on some list set shall be equivalent to any element that appears on any list set before the other. This restriction, of course, induces a common ordering relation on all elements of all lists in any operation, but this might not be one that is easily applied directly to some of the sets in question.

In any implementation of these algorithms, it would be possible to represent ordered sets by having the user provide a function which, given two elements, returns one of three codes depending on whether the first element is greater than, equivalent to, or less than the second and using this function whenever two elements are to be compared. It would also be possible to use these algorithms on ordered attribute-value lists (or any list where the ordering is on every nth element with the $(n + 1)$-th through the $(2n - 1)$-th elements always following the nth). For the purposes of this discussion, we will assume that all lists are ordered on an internal numeric code, smallest number first, and we will use identity as the equivalence relation.

Since, in the algorithms given below, a list is often searched for the smallest element equal to or greater than a given element, even more speed can be achieved if the lists are organized so that binary and/or bucket searches may be used. There would be no changes required in the

* Computer Sciences Department. This paper is a version of a section of the author's Ph.D. thesis [4]. The research reported herein was partially supported by a grant from the National Science Foundation (GP-7069) and partially by USAF Proj. RAND (project #1116). Use of the University Computing Center was made possible through support, in part, from the National Science Foundation and the Wisconsin Alumni Research Foundation (WARF) through the University of Wisconsin Research Committee.

algorithms given below since the only changes needed would be in the design of the reader and the routine to manipulate the reader.

A *reader*, as used in this discussion, may contain only a pointer to a list element or additional information as well. The essential requirements are that the reader be able to identify a unique element of some list (which we will refer to as the element currently pointed at by the reader) and that it be possible to retrieve the datum of that element, to *increment* the reader so that it points to the next element in the list, and to recognize when the element it is pointing at is the last in the list set.

We can consider a reader as a generator of the set represented by the list it reads. We will define three other *list set generators*. A difference list set generator is used to generate a set which is the difference between the sets generated by two list set generators. A union list set generator is used to generate a set which is the union of the sets generated by a number of list set generators. An intersect list set generator is used to generate a set which is the intersection of the sets generated by a number of list set generators. Figures 1–4 demonstrate the use of these generators. The algorithms used are given below.

D2. A *list set generator* (LSG) is defined recursively as:

1. (a) A *primitive LSG* (PLSG) is a reader.
   (b) A PLSG is an LSG.
2. (a) A *difference LSG* (DLSG) is an ordered pair of LSGs.
   (b) A DLSG is an LSG.
3. (a) A *union LSG* (ULSG) is an ordered, finite list of LSGs, no two of which have equal *data* (see below). The list is ordered so that if L_1 and L_2 are on the list and have data d_1 and d_2 respectively, then $d_1 < d_2$ if and only if L_1 is before L_2 in the list.
   (b) A ULSG is an LSG.
4. (a) An *intersect LSG* (ILSG) is an arbitrarily ordered,[1] finite list of LSGs.
   (b) An ILSG is an LSG.
5. The only LSGs are those defined by 1–4.

For various purposes, an LSG at any given time will be considered to be identifying a unique *datum*.

D3. The *datum of an LSG* is defined recursively as follows:

1. The datum of a PLSG is the datum of the list set element currently pointed at by the reader.
2. The datum of a DLSG, ULSG, or ILSG is the datum of the first LSG of which it is composed.

The datum of a DLSG or an ILSG may or may not be an element of the list set the LSG is generating. It will be, if the last operation performed on the LSG was *initializing*, *incrementing*, *incrementing to or past a datum*, or *incrementing past a datum* as these operations are described below. It may not be, if the last operation was *checking a datum against the LSG* or some operation not defined here. The datum of a PLSG or a ULSG will always be an element of the list set being generated.

[1] If the ILSG is ordered on the size of the sets to be generated by the component LSGs, smallest first, all operations on the ILSG will be significantly faster than otherwise.

$L_1 - L_2$

$L_1 = \{0, 1, 2, 5, 6, 8, 9\}$ $L_2 = \{0, 1, 2, 3, 7, 9\}$

Step	DLSG	Generated set
1	($_D$ 1:0*)	{ }
2	($_D$ 1:0, 2:0)	{ }
3	($_D$ 1:1, 2:0)	{ }
4	($_D$ 1:1, 2:1)	{ }
5	($_D$ 1:2, 2:1)	{ }
6	($_D$ 1:2, 2:2)	{ }
7	($_D$ 1:5, 2:2)	{ }
8	($_D$ 1:5, 2:7)	{5}
9	($_D$ 1:6, 2:7)	{5, 6}
10	($_D$ 1:8, 2:7)	{5, 6}
11	($_D$ 1:8, 2:9)	{5, 6, 8}
12	($_D$ 1:9, 2:9)	{5, 6, 8}
13	the *PLSG* for L_1 finishes	

* A *PLSG* will be represented as a list set identifier followed by ":" followed by the datum of the *PLSG*.

FIG. 1. Example of a *DLSG* being used to generate a set which is the difference between two sets

$L_1 \cup L_2 \cup L_3 \cup L_4$

$L_1 = \{0, 1, 2, 5, 6, 8, 9\}$ $L_3 = \{2, 3, 6, 8, 9\}$

$L_2 = \{0, 2, 3, 4, 5\}$ $L_4 = \{0, 1, 2, 3, 7, 9\}$

Step	ULSG	Generated set
1	($_U$ 1:0)	{ }
2	($_U$ 1:0, 2:2)	{ }
3	($_U$ 1:0, 2:2, 3:3)	{ }
4	($_U$ 1:0, 4:1, 2:2, 3:3)	{0}
5	($_U$ 4:1, 2:2, 3:3, 1:5)	{0, 1}
6	($_U$ 2:2, 3:3, 1:5, 4:7)	{0, 1, 2}
7	($_U$ 3:3, 2:4, 1:5, 4:7)	{0, 1, 2, 3}
8	($_U$ 2:4, 1:5, 3:6, 4:7)	{0, 1, 2, 3, 4}
9	($_U$ 1:5, 3:6, 4:7)	{0, 1, 2, 3, 4, 5}
10	($_U$ 3:6, 4:7, 1:8)	{0, 1, 2, 3, 4, 5, 6}
11	($_U$ 4:7, 1:8, 3:9)	{0, 1, 2, 3, 4, 5, 6, 7}
12	($_U$ 1:8, 3:9)	{0, 1, 2, 3, 4, 5, 6, 7, 8}
13	($_U$ 3:9)	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
14	finishes	

FIG. 2. Example of a *ULSG* being used to generate a set which is the union of four sets

The operations described below, except initialization, may be performed repeatedly on an LSG in order to consider successive elements in a list set. The elements will be generated in the order used for ordering the list sets, and once passed, an element will not again be the datum of the LSG. Thus, eventually, an LSG will have been moved past all the elements of the list set it generates. When this occurs, we say the LSG is *finished*. An LSG may finish during any of the operations described below, in which case the operation concludes, returning an appropriate flag. Instead of giving the finishing conditions in every algorithm below, we give them here once since they are the same for all.

D4. *Finishing conditions* are defined as follows:

1. A PLSG *finishes* when an attempt is made to increment it when it already points at the last element of its list set.

$L_1 \cap L_2 \cap L_3 \cap L_4$

$L_1 = \{0, 1, 2, 5, 6, 8, 9\}$ $L_3 = \{2, 3, 6, 8, 9\}$

$L_2 = \{0, 2, 3, 4, 5\}$ $L_4 = \{0, 1, 2, 3, 7, 9\}$

Step	*ILSG*	*Generated set*
1	($_I$ 1:0)	{ }
2	($_I$ 2:0, 1:0)	{ }
3	($_I$ 3:2, 2:0, 1:0)	{ }
4	($_I$ 4:2, 3:2, 2:0, 1:0)	{ }
5	($_I$ 4:2, 3:2, 2:2, 1:0)	{ }
6	($_I$ 4:2, 3:2, 2:2, 1:2)	{2}
7	($_I$ 4:3, 3:2, 2:2, 1:2)	{2}
8	($_I$ 4:3, 3:3, 2:2, 1:2)	{2}
9	($_I$ 4:3, 3:3, 2:3, 1:2)	{2}
10	($_I$ 4:3, 3:3, 2:3, 1:5)	{2}
11	($_I$ 4:7, 3:3, 2:3, 1:5)	{2}
12	($_I$ 4:7, 3:8, 2:3, 1:5)	{2}
13	($_I$ 4:9, 3:8, 2:3, 1:5)	{2}
14	($_I$ 4:9, 3:9, 2:3, 1:5)	{2}
15	the *PLSG* for L_2 finishes	

FIG. 3. Example of an *ILSG* being used to generate a set which is the intersection of four sets

$(L_1 \cup L_2) - (L_3 \cap L_4)$

$L_1 = \{2, 3, 6, 8, 9\}$ $L_3 = \{0, 1, 2, 5, 6, 8, 9\}$

$L_2 = \{0, 1, 2, 3, 7, 9\}$ $L_4 = \{0, 2, 3, 4, 5, 8\}$

Step	*LSG*	*Generated set*
1	($_D$($_U$ 1:2))	{ }
2	($_D$($_U$ 2:0, 1:2))	{ }
3	($_D$($_U$ 2:0, 1:2) ($_I$ 3:0))	{ }
4	($_D$($_U$ 2:0, 1:2) ($_I$ 4:0, 3:0))	{ }
5	($_D$($_U$ 2:1, 1:2) ($_I$ 4:0, 3:0))	{ }
6	($_D$($_U$ 2:1, 1:2) ($_I$ 4:2, 3:0))	{1}
7	($_D$($_U$ 1:2, 2:3) ($_I$ 4:2, 3:0))	{1}
8	($_D$($_U$ 1:2, 2:3) ($_I$ 4:2, 3:2))	{1}
9	($_D$($_U$ 2:3, 1:6) ($_I$ 4:2, 3:2))	{1}
10	($_D$($_U$ 2:3, 1:6) ($_I$ 4:3, 3:2))	{1}
11	($_D$($_U$ 2:3, 1:6) ($_I$ 4:3, 3:5))	{1, 3}
12	($_D$($_U$ 1:6, 2:7) ($_I$ 4:3, 3:5))	{1, 3}
13	($_D$($_U$ 1:6, 2:7) ($_I$ 4:8, 3:5))	{1, 3, 6}
14	($_D$($_U$ 2:7, 1:8) ($_I$ 4:8, 3:5))	{1, 3, 6, 7}
15	($_D$($_U$ 1:8, 2:9) ($_I$ 4:8, 3:5))	{1, 3, 6, 7}
16	($_D$($_U$ 1:8, 2:9) ($_I$ 4:8, 3:8))	{1, 3, 6, 7}
17	($_D$ 2:9 ($_I$ 4:8, 3:8))	{1, 3, 6, 7}
18	2:9	{1, 3, 6, 7, 9}
19	finishes	{1, 3, 6, 7, 9}

FIG. 4. An example using all three *LSG*s

2. A DLSG *finishes* when its first LSG finishes.
3. A ULSG *finishes* when it is composed of one LSG and that LSG finishes.
4. An ILSG *finishes* when any of its LSGs finishes.

In two cases an LSG may be discarded in favor of a component LSG: (1) if the second LSG of a DLSG finishes, the first LSG replaces the DLSG; (2) when a ULSG is composed of only one LSG, that LSG is used in place of the ULSG. These cases may arise during the algorithms described below, but we do not mention them again.

The first algorithm to be described is *initializing an LSG*. When an LSG is initialized, its datum will be the first element of the list set the LSG generates. If that list set is null, the LSG will finish during the initialization process.

A1. *Initializing an LSG*

1. PLSG: Initialize the reader so that it points at the first element of its list.
2. DLSG (see Figure 1 steps 1–8):
   (a) Initialize the first LSG.
   (b) Initialize the second LSG at or past the datum of the first (i.e. its datum will be equal to or larger than the datum of the first LSG).
   (c) If the data of the two LSGs are equal, increment the DLSG.
3. ULSG (see Figure 2 steps 1–4):
   Initialize each LSG in turn, placing them in the ULSG in the proper order (but not placing one that finishes). If an LSG is initialized with a datum equal to the datum of an LSG already in the ULSG, increment it until it has a datum not already represented. Then place it in the ULSG in the proper order.
4. ILSG (see Figure 3 steps 1–6):
   (a) Initialize one LSG and place it in the ILSG.
   (b) Initialize each successive LSG (in any order) at or past the datum of the previous LSG and place it as the first LSG of the ILSG.
   (c) When all LSGs have been initialized and inserted, if their data are not all equal, increment the ILSG to or past the datum of its first LSG.

Once an LSG is initialized, it can be repeatedly *incremented*, and after each step its datum will be the next greatest element of the list set it generates (see Figures 1–3). If some operation was performed on an LSG so that its datum is not an element of the set it generates, and the LSG is then incremented, its datum after being incremented will be the smallest element of the set it generates which is larger than the datum of the LSG before it was incremented.

A2. *Incrementing an LSG*

1. PLSG:
   The reader is incremented so that it points at the next element in its list.
2. DLSG:
   (a) Increment the first LSG.
   (b) Check the current datum of the first LSG against the second LSG. If the check fails, the increment is done. If the check succeeds, go back to step (a).
3. ULSG:
   (a) Remove the first LSG from the ULSG.
   (b) Increment the LSG removed in step (a). If it finishes, the increment is done.
   (c) If the datum of the LSG incremented in step (b) is equal to the datum of any other LSG in the ULSG, go to step (b).
   (d) Return the LSG to the ULSG in its proper order according to its current datum.
4. ILSG:
   (a) Increment the first LSG of the ILSG.
   (b) Let D be the datum of the first LSG and i be 1.
   (c) Let $i = i + 1$.
   (d) If there is no ith LSG the increment is done.
   (e) Increment the ith LSG to or past D.

(f) If the datum of the ith LSG equals D, go to step (c).
(g) Let D be the datum of the ith LSG and i be 0.
(h) Go to step (c).

There are times when we are not interested in the next element to be generated by an LSG but in the next element equal to or greater than a given one (e.g. A2.4(e)) or the next element strictly greater than a given one. Such an element could be found by repeatedly incrementing the LSG, but it would be more efficient to make full use of the information of what datum we wish to equal or exceed and increment the LSG to or past (or just past) the datum in one operation.

A3. *Incrementing an LSG (to or) past a datum, D*

1. PLSG:
   Increment the reader (zero or more times) until it points to an element whose datum is (equal to or) greater than D.
2. DLSG:
   (a) Increment the first LSG (to or) past D.
   (b) Check the datum of the first LSG against the second LSG. If the check is successful increment the DLSG. If the check is not successful, the increment is done.
3. ULSG:
   (a) Remove the first LSG from the list.
   (b) Increment the LSG removed in step (a) (to or) past D. If it is finished, go to step (e).
   (c) If the datum of the LSG incremented in step (b) is equal to the datum of any other LSG in the ULSG, increment it. If this finishes the LSG, go to (e).
   (d) Return the LSG to its proper place in the ULSG according to its current datum.
   (e) If the datum of the LSG which is now first in the ULSG is not (equal to or) larger than D, go to step (a); otherwise the increment is done.
4. ILSG:
   This is exactly the same as incrementing an ILSG (A2.4), except that in step (a), the first LSG is incremented (to or) past D.

If it is desired to determine if a given element is a member of the set generated by an LSG, this can, in most cases, be done more quickly than by incrementing the LSG to or past the element and then looking at the datum of the LSG, if it is acceptable that, when the check is finished, the datum of the LSG might not be a member of the set it generates. To subsequently produce an unknown member of the set generated by the LSG, it would be necessary to perform one of the incrementing operations on it.

A4. *Checking a datum, D, against an LSG*

1. PLSG:
   Increment the reader to or past D. If the reader finishes or it ends up with a datum which is larger than D, the check is unsuccessful. If the PLSG ends up with a datum equal to D, the check is successful.
2. DLSG:
   (a) Check D against the first LSG. If this check is unsuccessful, the check against the DLSG is unsuccessful.
   (b) If the check against the first LSG was successful, check D against the second LSG. If this check is successful, the check against the DLSG is unsuccessful and vice versa.
3. ULSG:
   (a) If the datum of any LSG in the ULSG equals D, the check is successful.
   (b) Remove the first LSG from the ULSG.
   (c) Increment the LSG removed in step (b) to or past D. If it finishes, go to step (f).
   (d) If the datum of the LSG incremented in step (c) is equal to the datum of any other LSG in the ULSG, increment it. If this finishes the LSG, go to step (f).
   (e) Return the LSG to its proper place in the ULSG according to its current datum.
   (f) If the datum of the LSG incremented in step (c) was equal to D after it was incremented, the check is successful. Otherwise, if the datum of the LSG, which is now first in the ULSG, is larger than D, the check is unsuccessful. If neither of the above two cases holds, go to step (b).
4. ILSG:
   Check D against each LSG that makes up the ILSG in turn. As soon as one is found for which the check is unsuccessful, the check against the ILSG is unsuccessful. If all checks are successful, the check against the ILSG is successful.

It should be remembered that the list sets are ordered and the LSGs generate them in order. The only way to generate all the members of a set is by successive incrementing with no other operations interposed. An LSG cannot be "backed up" to an element it has already passed. If several elements are to be checked against an LSG, this must be done in the proper order to avoid the necessity of initializing several LSGs.

The example in Figure 4 shows how LSGs are used to evaluate a set expression. Because of their generality, LSGs would be extremely useful as part of a language system allowing sets as a data type. Moreover, since any ordered, sequential file fits the definition of list set given above, LSGs may be used for traditional file handling and will lead to great efficiency when arbitrary numbers of files are to be handled simultaneously. For these purposes, the DLSG is used for purging records from a file, the ULSG is used for merging files[2] and all the LSGs may be used for information retrieval.

Acknowledgment. The author expresses his thanks to Professor Larry E. Travis of the Computer Sciences Department, University of Wisconsin for his continuing help and guidance.

RECEIVED AUGUST, 1970

REFERENCES

1. BROOKS, F. P. JR., AND IVERSON, K. E. *Automatic Data Processing: System/360 Edition*. Wiley, New York, 1969.
2. FELDMAN, J. A., AND ROVNER, P. D. An ALGOL-based associative language. *Comm. ACM 12*, 8 (Aug. 1969), 439–449.
3. KNOWLTON, K. C. A programmer's description of L^6 *Comm. ACM 9*, 8 (Aug. 1966), 616–625.
4. SHAPIRO, S. C. A data structure for semantic information processing. Ph.D. Th., Comput. Sci. Dep., U. Wisconsin, Madison, Wis., 1970 (in preparation).
5. WEIZENBAUM, J. Symmetric list processor. *Comm. ACM 6*, 9 (Sept. 1963), 524–544.

[2] The ULSG merges in the manner described as "m-way merge with ranking sort" [1].