HIGH LEVEL ITERATORS AND A METHOD FOR AUTOMATICALLY DESIGNING DATA STRUCTURE REPRESENTATION

by

Jay Earley

Memorandum No. ERL-M425

February 1974

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ABSTRACT

We discuss the thesis that one good way of achieving non-procedural or problem-oriented languages is by constructing higher and higher level procedural languages, and along with them, more sophisticated optimizers. We present a set of operations called iterators embedded in a programming language VERS2 which represent a higher level of description than currently exists. These are operations which, if written out, would normally involve an iteration over a group of objects. We then introduce a method (called iterator inversion) for automatically designing data structure representations for the data structures which are iterated over. This method has the effect of converting the part of a program to which it is applied from a batch to an incremental algorithm.

CR Categories: 4.12, 4.22

Key Words and Phrases: high level languages, iterators, high level operations, data structure representation, automatic programming, optimization

This research was supported under NSF Grant GJ-34342X.

HIGH LEVEL ITERATORS AND A METHOD FOR AUTOMATICALLY DESIGNING DATA STRUCTURE REPRESENTATION

Jay Earley
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720

Introduction

Automatic programming is a field of endeavor in which we strive to automate the work that is normally done by programmers, with the ultimate goal of making computers usable by people who are not necessarily trained in programming [1]. We feel that one important approach in this area is the automation of programming by developing high level languages which allow a solution to a problem to be stated more easily, and optimizers (we mean this very broadly) for these languages which will make their programs as efficient as ones written (with more difficulty) in a lower level language. In fact, we feel that there is a continuum of levels of description between the statement of a problem and the statement of an efficient solution to the problem, and we are working upwards along this continuum. In this paper we present some evidence that when an algorithmic description of a problem solution is expressed at a high enough level, it has most of the desirable properties of a problem statement. We also present evidence that sophisticated optimization methods are possible which translate these high level programs into efficient lower level ones.

This paper represents the next step in our on-going effort to design programming languages in which the semantics of algorithms may be described in as convenient a way as possible - ignoring, if desired, issues of efficiency. It is the sequel to a previous paper [2] and builds directly on the concepts presented in that paper. In the next section, we summarize the relevant points of the previous paper, but we recommend that it be read for a thorough understanding of the issues involved.

In this paper we confine ourselves to those high level operations which we call iterators and to a particular optimization technique which applies to them. In another paper [5] we cover the full range of high level operations which we consider important.

We would like to emphasize that there are two reasons for introducing high level operations into a programming language. The obvious one is that they permit programs to be smaller and more understandable, and they often allow a programmer (or a system which constructs programs) to think in terms of a single conceptual unit where before he would have had to think in terms of a complicated program with loops and tests. For instance, the iterator

$$\exists X \in S \mid P(X)$$

may be used in place of a loop with an initializing statement, a test, and a branch out of the loop. A second reason for using high level operations is that they make the program much more available for optimization. Thus in the example above, an optimizer knowing the form of P, might choose to represent S in such a way as to make the iteration possible without generating all of S. This would be next-to-impossible for an optimizer to discover if it had to work directly with the loop written out.

Some existing languages have made efforts in the direction of using high level operations which suppress sequencing. These are SETL [9], APL [10], and the artificial intelligence languages [11].

The operations described in this paper are being designed as part of an experimental language VERS2 which is being implemented as an extension of EL1 [4] and our examples will use the EL1 syntax. For those readers not familiar with it, the unusual features of the EL1 syntax are summarized in Appendix A.

Relational Level Data Structures

In our previous paper [2], we worked with four basic kinds of data structures in permitting a high level relational appraoch to data structure descriptions:

1) Tuples have a fixed number of named components, which are usually heterogeneous. <7,SPADE> might represent a playing card, and TUP(DENOM:INT, SUIT:SUIT) might represent the tuple type for cards.

2) Sequences are ordered collections of homogeneous values. ["T", "R", "E", "E"] might represent an English word, and SEQ(CHAR) might represent the type.

3) Sets are unordered collections of homogeneous values with no repeats. {1,2,3,4} is a set of four integers, and SET(INT) is the type.

4) Relations are sets of tuples defining mappings between values. They are written using the syntax given for sets and tuples, and relation types look like tuple types except that they use REL.

In addition to these, there are two more data structures in VERS2 which are relevant to this paper.

Functions are essentially relations which define a functional mapping which is always total. A function has any number of domains and one range, and there is always a range value defined for every member of the cartesian product of the domains. As an example, we would represent an $M \times N$ real array as a function as follows:

```
FUNC({1...M}, {1...N}, REAL←0)   .
```

The construct "{A...B}" represents the set of integers between A and B. The "←0" indicates that any component which has not yet been stored into is initially 0. We can store into a component as follows:

```
A(3,4) ← 7.6
```

or access a value as

```
A(2,6)   .
```

We can also (as with relations) access a row

```
A(1,*)
```

or a column

```
A(*,3)
```

producing in each case a function with one less domain. The important difference between relations and functions is that functions are total. This means that they have a fixed set of tuples in which only the range value may be modified by assignment. In relations, the set of tuples varies, and modifications are made by adding or deleting tuples rather than by assignment. In [2] we represented a two-dimensional array by a relation, but a function does a better job.

Multi-Sets (bags) are sets in which repeating elements are permitted. They are useful as arguments to n-ary addition or multiplication as in QA4 (see [3]). They may be obtained as a trivial modification of sets. All the same primitives are used with multi-sets as with sets; they just have a slightly different meaning. The distinction between the two can be made in the type declaration.

Iterators

Typical iterative operations are the quantifiers

$\exists X \in S | P(X)$ there exists X in S such that P(X) is true,

$\forall X \in S: P(X)$ for all X in S, P(X) is true

which yields boolean values for S a sequence or set. The existential quantifier also has the side effect of storing the existing value into the control variable X. An example program using these two is the following routine CAN\NULL which works on a BNF grammar:

```
DECL G:REL(DEF:NT, RTSIDE:SEQ(NT ∪ TERM))   .
```

Here G is a relation representing the grammar such that G(N), where N is a non-terminal, produces the set of all alternatives A of N. Alternatives are sequences of non-terminals or terminals. The routine

CAN\NULL decides whether a particular non-terminal N can generate the null string:

```
CAN\NULL ← EXPR(N:NT ∪ TERM; BOOL)
BEGIN
N ε TERM ⇒ FALSE;
∃A ε G(N)| ∀S ε A: CAN\NULL(S) ? TRUE
END
```

Here we are implicitly using another high level feature which checks to make sure that no recursive call on a routine is made with the same parameters as an existing uncompleted call. This error (as well as others) is caught by the "?" operator which specifies that if there is an error in its left hand argument that its right hand argument should be the value of the expression.

Another iterative operation is the set former:

{X ε S| P(X)} the set of all X in S such that P(X) is true .

We could use this with our grammar to produce the set of all null productions in G:

{P ε G| # P.RTSIDE = 0} .

"#S" yields the size of S, where S is a set or sequence. There is also a sequence former

[X ε S| P(X)] .

Before we introduce any more iterators, we need to distinguish between iterators and iterative operations: In the code

$$\exists X \in S \mid P(X)$$

"$X \in S \mid P(X)$" is an iterator, and "$\exists$" is an iterative operation. Iterators yield streams of values, and iterative operations do something with these values to yield a result. So far we have introduced four iterative operations: 1) existential quantifier, 2) universal quantifier, 3) set former, 4) sequence former. We have also used two forms of iterators:

a) "$X \in S$", b) "iterator | boolexp" .

The FOR iterative operation may be used to perform any action repeatedly:

FOR iterator DO exp

for example:

FOR $X \in S \mid P(X)$ DO PRINT(X) .

There is also a step iterator of the form

$A \leftarrow i, n(A), f$

which initializes A to i, and then repeatedly reassigns n(A) to it until it reaches the value f. The third argument may be left out, and it will terminate when it reaches NIL. Using this, we can do an Algol FOR statement:

FOR I ← 1, I+1, 10 DO ——

or we can search through a linked list L with a NEXT field

$$\text{FOR } P \leftarrow L,\ P.\text{NEXT DO} \text{ —— } .$$

There is a converge iterator

$$A \Leftarrow i,\ n(A),\ e$$

which is similar to the step iterator except that it terminates when the successive values generated converge. The third argument is a small epsilon to be used for real number convergence. Thus we can compute the square root of N by successive approximations as follows:

$$S \Leftarrow IG,\ (N/S+S)/2,\ .0001 \quad .$$

Here IG is the initial guess and S represents the successive approximation to the square root of N. For values other than reals, epsilon is omitted and the iterator terminates when two successive values are equal.

The "#" iterative operation computes the number of values generated by its iterator. Thus we could compute the number of alternatives of a particular non-terminal N:

$$\# \ P \in G \mid P.\text{DEF} = N \quad .$$

Another iterator allows us to generate one collection of values based on another iteration:

$$\text{exp FOR iterator} \quad .$$

The exp presumably is written in terms of the control variable of the iterator and yields one exp value for each value generated by the iterator. For example, we could have written out G(N) on our grammar as follows:

$$\{P.RTSIDE\ FOR\ P \in G |\ P.DEF = N\} \quad .$$

The iterative operation for <u>collection</u> is of the form

op / iterator

where op is an operator or routine which takes two arguments of some type and returns a result of the same type. It applies the operator successively to each value generated by the iterator and the previous result. Thus

$$+ / X \in S |\ P(X)$$

adds together all values in S for which P is true. To illustrate this on our grammar, consider the routine H which computes the set of all terminals which can be the first symbol of a string derived from N.

```
H ← EXPR(N:NT ∪ TERM; SET(TERM))
BEGIN
N ε TERM ⇒ {N};
∪ / H(FIRST(A)) ? {} FOR A ε G(N)
END
```

Notice that this only works if the grammar cannot contain null productions. Before we can modify this program to work in the presence of null productions we must introduce some new iterators.

The forms

iterator WHILE boolexp

iterator UNTIL boolexp

generate values as long as or until a condition is satisfied. We also need ways of combining iterators. The sequential combination:

iterator ; iterator

generates one set of values and then the second. Thus if S1 = [1,2,3] and S2 = [4,5,6], the expression [X ε S1; X ε S2] yields [1,2,3,4,5,6]. The nested combination

iterator * iterator

generates all of the second collection of values for each value from the first. So we get all combinations. Thus the expression [<A,B> FOR A ε S1 * B ε S2] yields [<1,4>, <1,5>, <1,6>, <2,4>, <2,5>, <2,6>, <3,4>, <3,5>, <3,6>]. Finally the parallel combination

iterator ∥ iterator

generates the first one of each, then the second, and so on. Thus the expression [<A,B> FOR A ε S1 ∥ B ε S2] yields [<1,4>, <2,5>, <3,6>]. Notice that the parallel iterator doesn't make sense over sets because they are unordered.

We can now write our expanded version of H using "UNTIL" and the nested combination and the CAN\NULL routine written previously.

```
H ← EXPR(N:NT ∪ TERM ; SET(TERM))
BEGIN
N ε TERM ⇒ {N} ;
 ∪ / H(S) ? {} FOR A ε G(N) * S ε A UNTIL NOT CAN\NULL(S)
END
```

We needed to use "UNTIL NOT" instead of "WHILE" because UNTIL generates the value which makes its boolean change value but WHILE does not.

Continuing, we have three similar iterative operations

```
THE   iterator
FIRST iterator
LAST  iterator   .
```

"THE" requires that there be exactly one value generated and returns it. The others return the first or last value generated, only. We can illustrate "THE" by writing out the relational access primitive on a grammar which maps in the opposite direction from what we have done so far. "G(*,A)", where A is an alternative, returns the non-terminal which is on the left side of A. It will return a single value and not a set if we have declared that G is functional from its second domain onto its first (which of course it is). We could write this out as

```
THE P.DEF FOR P ε G | P.RTSIDE=A
```

Now let's introduce a new problem domain for our examples. Suppose we are working with playing cards.

```
CARD :: TUP(SUIT:SUIT, DENOM: {1...13})
SUIT :: {SPADE, DIAM, HEART, CLUB}
HAND :: SET(CARD)
```

We have an iterator which sorts the values generated according to some relation on those values. In the simplest case, SORT X ε S

sorts the integers in S in ascending order. If S contains other values such as cards, then one may sort these based on some attribute of the card, i.e.

SORT X ε S BASED\ON X.DENOM

In addition, one can sort with respect to a relation other than "<" by specifying it explicitly, and a sort may also be based on primary <u>and</u> secondary keys. We will not bother with examples of these. The SORT iterator can be used with the sequence former to produce an actual sorted data structure, or with FIRST and LAST. Thus

FIRST SORT X ε S

produces the smallest value in S. In practice we will abbreviate "FIRST SORT" by "MIN" and "LAST SORT" by "MAX". This is illustrated by the following expression which produces the longest SUIT in a HAND H.

MAX S ε SUIT BASED\ON # C ε H | C.SUIT=S

The form "A1, A2 ε S" generates all pairs A1 and A2 from S. It is simply an abbreviation for "A1 ε S * A2 ε S". We also have a delete iterative operation

DEL iterator

which deletes all values generated, from the set or sequence they are generated from. Examples of these are in the next section. In addition we may write

```
DEL which iterator
```

where "which" is THE, FIRST, or LAST, with the obvious meaning. There is also a replace iterative operation.

```
REPL iterator WITH exp
REPL which iterator WITH exp
```

which replaces each value generated with the corresponding value of the exp (which presumably is written in terms of the control variable). And finally, the iterator "$X \subseteq S$" generates all subsets (subsequences) of S. This and the replace operation are used primarily in conjunction with pattern matching. In [5] the pattern matching facilities are developed and more complete examples of the use of iterators are given. The iterators are summarized in FIG. 1.

Iterator Inversion

Now let's consider the problem of taking a program which is written in terms of relational level data structures and iterators and making it efficient. This is a very difficult task. Clearly if we follow the usual procedure in programming languages and represent all instances of a particular kind of data structure (such as a set) in the same way, we will be far from the kind of efficiency that is possible. This is because we are dealing with structures which are fairly far removed from the machine. So we have set up VERS2 so that the user (and later the system) may choose his own data structure representations from a library (or even program his own if he chooses). We call this the implementation facility, and it is explained in more detail in [2].

Iterators

$A \in S$

$A \subseteq S$

$A \leftarrow i, n(A), F$

$A \Leftarrow i, n(A), e$

A, B

A = pattern

pattern

| in place of A above

iterator | boolexp

iterator WHILE boolexp

iterator UNTIL boolexp

exp FOR iterator

SORT iterator BASED\ON exp

iterator ; iterator

iterator * iterator

iterator ‖ iterator

Iterative Operations

∃ iterator

∀ iterator : boolexp

{iterator}

[iterator]

op/iterator

iterator

which iterator

DEL which iterator

REPL which iterator WITH exp

FOR iterator DO exp

(which is Λ|FIRST|LAST|THE)

Figure 1

Even this is not enough, however. We have a similar problem with iterators. If all iterators are implemented in the same way we will still be far away from the kind of efficiency possible. We would often like to be able to pick a representation for the set or sequence which is being iterated over which would cut down on the number of values being examined. This is more difficult than choosing a representation for a structure which is not involved in an iterator for the following reason: If the structure is not involved in an iterator we need to consider only the primitive operations which are applied to the structure, the frequency of executing each primitive, and the size of the structure. If an iterator is involved, we must also consider the kind of iterator, the form of the boolean expression (if there is one), and the iterative operation. With such a variety of different cases we can no longer just choose a representation from a library, instead we must design the representation to fit the iterator(s) involved.

Here we present a method for doing this called iterator inversion. This method can be applied to almost any iterator, but it will not always improve the efficiency of the program. This depends on the form of the iterator and the relative frequency of execution of the iterator vs. modification of the structures involved. Iterator inversion is only one such method, and a complete system should have other possibilities at its disposal, but we have found iterator inversion to be widely useful, and in some sense a "basic" method from which others might be derived. Using this method, we represent the set or sequence in such a way that whenever the iterator is to be executed, we can generate only those values which are needed by the iterative operation on that round.

For instance, if we have a set former

$$\{X \in S | \ P(X)\} \quad ,$$

we maintain, along with S, the set of values in S for which P is true, so that we just need to copy this set to execute the iterator. We check each time an element Y is added to (or deleted from) S to see if P(Y) is true. If it is, we add Y to (or delete Y from) the auxiliary set. In addition, we must update the auxiliary set whenever P changes, since P might be expressed in terms of data structures which can be modified. This becomes even more complicated if the expression P contains a free variable A; then we will maintain along with S, a data structure function T which maps values from the domain of A onto subsets of S for which P is true for that value of A. We call T the <u>inversion function</u>.

Let's consider a specific example:

$$\{X \in S | \ F(X) = A\}^{\dagger} \quad .$$

Here we will store, along with S, an inversion function T which maps values of A into subsets of S. In more detail, if ADOM is the domain of A, and SDOM the domain of S, then

DECL T:FUNC(ADOM, SET(SDOM) $\leftarrow$ {}) .

[†] Here F is a data structure function. In the remainder of the paper we will use this syntax (F(X)) to stand for either function access or tuple selection (normally written X.F). This is because they have identical semantics as far as iterator inversion is concerned and it allows one transformation to apply to both constructs.

This particular predicate can be modified by storing into A or storing in the F field of a member of S. We don't have to worry about modifications to A since our mapping is defined in terms of A, but we do have to worry about changes to F fields. So each time there is an assignment to the F field of a tuple in S, we fix up T to reflect the change. The following are the changes required to invert the above iterator.

```
A           original code                  after inversion

            {X ε S| F(X) = A}      →       T(A)

            ADD Y TO S             →       ADD Y TO S;
                                           ADD Y TO T(F(Y))

            DEL Y FROM S           →       DEL Y FROM S;
                                           DEL Y FROM T(F(Y))

            F(Y) ← Z               →       Y ε S → BEGIN
                                                   DEL Y FROM T(F(Y));
                                                   ADD Y TO T(Z)
                                                   END;
                                           F(Y) ← Z
```

Notice that this is similar to file inversion in data bases; this is where the name iterator inversion comes from.

Now let's consider an example program dealing with playing cards. Using the previous definitions of CARD, SUIT, and HAND, we might have a program containing the code

```
ADD C TO H
```

and

```
FOR C ε H| C.SUIT=S DO
```

where C is a CARD, H is a HAND, and S is a SUIT. To invert this iterator, we would define an inversion function for each hand

```
INV\HAND :: FUNC(SUIT, SET(CARD) ← {})
DECL IH:INV\HAND
```

then the above code would become

```
ADD C TO H; ADD C TO IH(C)
```

and

```
FOR C ε IH(C) DO   .
```

In this example, the changes produced by iterator inversion may seem slight, but in an algorithm with a number of iterators, it may radically change the processing involved. Here we present a full example using a problem taken from Knuth [7] -- the "topological sort" algorithm, which we also used as an example in [2]. The problem is to perform a topological sort of a partially ordered set of objects. The input is a sequence of pairs of objects. Each pair <B,C> defines an ordering relationship between the objects in the pair, so that B is less than C. The goal is to output the objects in such an order that an object appears before everything that it is less than in the ordering. See Knuth for a more detailed explanation. The algorithm picks an object A such that there are no pairs in which it is greater than another object. A is then one of the "least" objects we have. We then output A, delete all pairs containing A, and repeat the process until all the objects are exhausted. If we reach a condition in which there is no least object, but the objects are not exhausted, it means that the input pairs do not form a partial ordering, so we give an error message.

In the high level program we have just two sets: S is the set of all objects, and SP is the set of all pairs.

```
WHILE $\exists$A $\varepsilon$ S| $\nexists$<-,A> $\varepsilon$ SP DO
     BEGIN
     OUTPUT A;
     DEL A FROM S;
     DEL <A,-> $\varepsilon$ SP;
     S = {} $\Rightarrow$ RETURN
     END;
ERROR
```

In this program, <-,A> is a tuple pattern (see [5]) and the code

$\exists$<-,A> ε SP

is equivalent to

$$\exists P \in SP \mid P.SECOND = A$$

where SECOND is the second field of the pairs. There are three iterators in this program, and we will invert each of them, but first we must extend iterator inversion to include other iterative operations. With quantifiers and "#", we only need to store the number of elements in each subset in the inversion function rather than the subset itself. The transformations for this inversion are as follows:

```
B    original code                       after inversion

     #X ε S| F(X) = A          →       K(A)

     ADD Y TO S                →       Y ∉ S → K(F(Y)) ← K(F(Y))+1;
                                       ADD Y TO S

     DEL Y FROM S              →       Y ε S → K(F(Y)) ← K(F(Y))-1;
                                       DEL Y FROM S

     F(Y) ← Z                  →       Y ε S → BEGIN
                                               K(F(Y)) ← K(F(Y))-1;
                                               K(Z) ← K(Z)+1
                                               END;
                                       F(Y) ← Z
```

We also need an inversion to handle the case where the boolean expression is X.F = C, where C is constant:

```
C    original code                       after inversion

     {X ε S| F(X) = C}         →       T

     ADD Y TO S                →       ADD Y TO S;
                                       F(Y) = C → ADD Y TO T

     DEL Y FROM S              →       DEL Y FROM S;
                                       F(Y) = C → DEL Y FROM T
     original code                        after inversion

     F(Y) ← Z              →      Y ε S → BEGIN
                                          F(Y) = C → DEL·Y FROM T;
                                          Z = C → ADD Y TO T
                                          END;
                                  F(Y) ← Z
```

Now we can invert the three iterators in the topological sort program. Inverting the ∄ iterator using inversion B, we introduce a function COUNT which maps each object b onto an integer which is the number of pairs <a,b>. We then replace "∄<–,A> ε SP" with "COUNT(A) = 0". Now the line of code reads

```
WHILE ∃A ε S| COUNT(A) = 0 DO
```

This fits the pattern for inversion C since we must keep a subset rather than a number because we are using the side effect of storing the object found into X. This inversion introduces a set ZRCOUNT, which is the set of all objects for which the COUNT is 0. The line becomes

```
WHILE ∃A ε ZRCOUNT DO    .
```

We can invert the DEL iterator, using a variant of inversion A. This introduces a function SUCC (successors) which maps an object b onto the set of all pairs <b,a>. The line then becomes

```
FOR P ε SUCC(A) DO DEL P FROM SP    .
```

Notice that we don't actually want SUCC to map onto the pairs themselves, but rather onto the addresses within the representation of SP where the pairs reside, so that they can be deleted easily. The detailed effects of iterator inversion on topological sort are in Appendix B.

It is interesting to look at this in terms of the levels of data structure description introduced in [2]. Knuth contains a machine level description of the algorithm; [2] contains relational level and access path descriptions written before the development of iterator inversion. Iterator inversion exactly accomplishes the task of reducing the relational level description to the access path description, thereby presumably duplicating part of the original effort of designing Knuth's version of the algorithm.

Up till now, in all of our examples we have introduced an inversion function in addition to the ordinary representation of the set involved. This is not always necessary. In some cases the inversion function duplicates the original set S closely enough that we can use it in place of any other representation for S. This will be possible if the inversion maps onto disjoint subsets of S which cover all of its values. Then if we implement all the normal primitives on S so that they work on the representation used by the inversion function, the inversion function becomes the entire representation of S.

So far we have illustrated iterator inversion with programs which contained a number of iterators, but also contained a significant amount of other code. The most startling consequences of iterator inversion are apparent when it is applied to a program which consists entirely of iterators. That is, a program which is applicative, containing only iterators and associated expressions -- no explicit flow of control or assignment statements. Examples are the CAN\NULL and H routines presented earlier. If you take such a program and invert all the iterators in it (proceeding from the inside, out), it has the effect of converting the original batch algorithm into an incremental one. That is, if the original routine (say the CAN\NULL routine) works on some data structure which can change over time (in this case, the BNF grammar), and computes a result (a boolean) based on a parameter (a non-terminal), then the inverted algorithm will maintain a new data structure from which this result can be accessed directly (a function from non-terminals to booleans) and will update this structure each time the original structure (the grammar) is modified.

This enables a programmer to write all such routines in the simpler batch form and then have the decision made later about whether it should actually be called each time or handled incrementally. This is, of course, predicated on the assumptions that the routines can be written in applicative form and that we can invert most iterators. The latter issue will be dealt with in the next section; here we would like to state that some significant complex algorithms can be written in applicative form using iterators. We have coded a simple LR(1) parse constructor [8] in VERS2 as a completely applicative program. It has the form described earlier of a routine which works on a data structure which can change over time (again a grammar) and computes a result (parsing tables). Here there is no parameter besides the grammar itself, so the inverted data structure is just the parsing tables (plus auxilliary structures needed to maintain them incrementally). We have actually applied iterator inversion to this entire program, thereby producing an incremental parse constructor automatically!

Extending Iterator Inversion

Thus far we have shown how to invert only a very limited variety of iterators: those of the form "iterator|boolexp" for two specific forms of boolean expressions and for two cases of iterative operations (those needing the subset and those needing only a count). In this section we describe how it can be extended to cover almost all boolean expressions, iterators, and iterative operations. The exact details of this are very complicated and would not be of interest to most of our readers, so we will present here only main concepts and indicate which forms can be inverted.

First we extend the method to cover more boolean expressions. As an example, consider the iterator

$$\{X \varepsilon S | F(G(X)) = A\}$$

where A is a variable. If we add or delete an element in S, this is handled similarly to the way it is handled in transformation A, using the same inversion function T, but using F(G(Y)) in place of F(Y). Storing a value into G is also handled similarly. The difficult modification is "F(Y) ← Z." Here we must determine all the possible X values affected by this change and the corresponding A values for them, so that we can update T correctly. If we imagine that function G has an inverse G^{-1}, then the X values are $G^{-1}(Y)$ and the A value is F(Y), and we update by executing

```
DEL $G^{-1}$(Y) FROM T(F(Y))
ADD $G^{-1}$(Y) TO T(Z)
```

If G is one-to-one, there will be just one X value, and otherwise, there will be a set of them, and T will have to be updated for each member of the set.

Notice that we can construct explicitly the relevant restriction of G^{-1}; it is the inversion of the following iterator

$$\{X \varepsilon S | G(X) = Y\} \quad .$$

Furthermore, we can generalize this to other boolean expressions as follows. Consider

$$\{X \varepsilon S | F(E) = A\}$$

where E is any expression involving X and no free variables. Then in order to handle the modification

$$F(Y) \leftarrow Z$$

we invert the iterator

$$\{X \varepsilon S \mid E = Y\}$$

and use the resulting function, similarly to the way we used G^{-1}.

This illustrates an important point: Whenever a modification is to be processed, the crucial problem is to get from the values given in the modification to the values for X (the control variable) and each of the free variables in the boolean expression which are affected by the modification. This is because we will usually have set up the inversion function T so that it maps from these free variable values onto sets of X values. Thus if the iterator contained two free variables, A and B, the inversion function would be "$T(A,B)$".

This is not always the case, however. Consider the iterator

$$\{X \varepsilon S \mid F(X) = G(A)\} \quad .$$

Here we don't want to map from A, but rather from $G(A)$, because this makes the inversion easier to do. That is, in updating "$F(Y) \leftarrow Z$", the mapping to be updated is simply $T(F(Y))$. If we were mapping from A, it would be $T(G^{-1}(F(Y)))$ and this might even involve iterating over a set of values. Notice that we could map from an expression containing more than one free variable, such as $G(A,B)$, but not from one containing X as well, such as $G(A,X)$.

We might also decide not to map from a free variable at all if it is rarely stored into. Instead we could update the inversion function each time the variable is stored into. That is, in the previous example, we would store a subset of all X such that $F(X) = G(A)$ for the current value of A, and update it whenever A is stored into. For each free variable in the boolean expression we have three choices: 1) Map from it, 2) Map from an expression containing it, 3) Update the inversion function whenever it is stored into. The choice of which to do, and in fact, the choice of whether to invert at all, is dictated by a variety of factors which we discuss in the next section.

Using generalizations of the methods described so far, we can handle boolean expressions of the form $E_1 = E_2$, where the E's contain only functional accesses. Now we extend it to include other relational operators. If the iterator is $\{X \varepsilon S|\ F(X) < A\}$, we use the same inversion function as we did for "=", but instead of using T(A) in place of the iterator, we use

$$\cup/T(I) \text{ FOR } I \varepsilon \text{ ADOM}|\ I < A$$

where ADOM is the domain of A. The same thing is used for $\leq, >, \geq$. The operator "≠" is a little confusing; we could also just use the "=" inversion function and then take the union of all subsets except T(A). However, this doesn't buy us much unless the domain of A is very small and the subsets large. It may frequently be the case that we will not want to invert "≠" expressions.

Now consider an iterator involving arithmetic expressions:

$$\{X \varepsilon S|\ F(X)+G(X,B) = A\}$$

where A and B are free. It is easy to handle modifications to S or G, but stores into F require a second look. If we are handling "F(Y) ← Z", all B values are affected by the modification, and so the only way we can update the inversion function is to consider all possible values that B can take on. For each of these, we can calculate an A value and thereby update the function. So inversion is possible, but unless the domain of B is relatively small, it may not be desirable.

Finally, we extend our method to boolean operators:

$$\{X \in S \mid F(X) = A \text{ OR } G(X) = B\} \quad .$$

Here we create two different inversion functions, one for each disjunct, and take their union to get the result of the iteration. In the case of conjunction:

$$\{X \in S \mid F(X) = A \text{ AND } G(X) = B\}$$

we map from both A and B in the inversion function, T(A,B) and modifications are fairly straightforward, we just must take both conjuncts into consideration.

Now we complete our description of the handling of boolean expressions and define those which can be inverted.

First we require that the boolean expressions contain no computed routine calls, so that we may examine the body of any routine called from the expression in doing the inversion. Second, we require that the expression and any routines called from it are applicative -- that is, they contain only applications of routines (primitive or programmer defined) to arguments -- no explicit flow of control or assignment statements. Furthermore we require that any iterators in the expression be invertible. We then invert all these iterators, obtaining in their place either a constant or a function application, depending on the form of the iterator. Most of the higher level data structure operations (such as ε, $\cup$, $\subseteq$, relational access, etc.) can be expanded as applicative expression involving iterators. For example, "$A \subseteq B$" can be expanded as "$\forall X \varepsilon A: X \varepsilon B$". Then these iterators can be inverted. This leaves us with an expression consisting of arithmetic operators (+, -, *, \\), relational operators (=, >, <), boolean operators (AND, OR, NOT), functional access, and conditional expressions. We move all conditional expressions toward the outside by performing the transformation

$$F([\![A \Rightarrow B;\ C]\!]) \qquad \rightarrow \qquad [\![A \Rightarrow F(B);\ F(C)]\!]$$

then move these out of the relational operator (such as "=") by doing

$$[\![A \Rightarrow B;\ C]\!] = D \qquad \rightarrow \qquad [\![A \Rightarrow B = D;\ \ C = D]\!] \quad .$$

Now all expressions involved are boolean, so we can perform a final transformation.

$$[\![A \Rightarrow B;\ C]\!] \qquad \rightarrow \qquad \text{A AND B OR NOT A AND C}$$

We then convert the boolean expression to disjunctive normal form and move the NOT's in with the relational operators. This leaves us with a boolean expression of the following general form:

$$\begin{aligned}
\text{BE} &\sim \text{D OR} \cdots \text{OR D} \\
\text{D} &\sim \text{C AND} \cdots \text{AND C} \\
\text{C} &\sim \text{E ROP E} \\
\text{ROP} &\sim = | \neq | < | \leq | > | \geq
\end{aligned}$$

where E is an expression consisting solely of arithmetic operations and functional accesses.

The resulting expression consists of constructs for which we have already introduced inversion methods, so generalizing on these in fairly straightforward ways, we can invert the transformed iterator.

Now let's consider the case where the set being iterated over is not always the same:

$$\{X \in E \mid F(X) = A\}$$

Here E is an expression which yields a set as its value. We handle this case by mapping from both A and E, T(A,E). We now must update the inversion function whenever a set in the domain cf E is modified. In addition, when we are handling "F(Y) ← Z", we must find the identities of the sets which are affected by this. This is handled by the VERS2 primitive SUPER: SUPER(Y,D) yields the set or sets from domain D which contain Y. So the mapping to be updated is T(F(Y), SUPER(Y,EDOM)). Notice that SUPER(Y,D) is in fact the iterator

$$\{S \in E \mid Y \in S\} \quad ,$$

so it can be inverted to make the above code efficient.

In a number of the previous examples, we have used the domain of a variable or expression in inverting an iterator. For this to work well, the programmer must have declared the relevant variables in such a way that their types are known at compile time. This will usually be the case because the VERS2 language encourages programmers to type variables accurately, and other optimizations also depend on typing. Equally important, however, is that the type of an object accurately reflect its usage, so that in most cases the domain of the inversion function contains only the relevant objects. This is accomplished in VERS2 by allowing sets as domains, e.g.

```
DECL AGE: {1...12}
```

and by having "tagged types", so that an object is tagged with an indication of its intended usgae. For example, by writing

```
HAND :: SET(CARD)
DECL H: HAND
```

we know that the domain of H includes only HAND's and not any other sets of cards that may exist.

We will now briefly mention how other iterators are handled. Consider

$$\{F(X) \text{ FOR } X \in S \mid G(X) = A\} \quad .$$

We set up the inversion function as before but now it maps onto sets of F(X) values, rather than subsets of S. We must handle the additional case where F is stored into, but this is similar to previous examples.

Now consider

$$\{F(X,Y) \text{ FOR } X \in S \mid F(X) = A * Y \in T(X)\} \quad .$$

Here we map from A and T(X) onto sets of F(X,Y) values, but now modifications to F or S must find those T(X) sets which are affected, and modifications to T must find those A values affected. This is done using ideas similar to those already presented, so we will not go into more detail. The sequential iterator combination (";") is handled similarly. The step and converge iterators cannot be inverted because they do not iterate over a data structure at all. The "$\subseteq$" iterator is rarely used except in pattern matching, so we are not considering it.

This leaves the WHILE, UNTIL, SORT, and "$\|$" iterators. We are considering these together because they are the ones which involve ordering on the data structures involved. Thus far we have assumed that we were working exclusively with sets. In fact, in the previous examples, sequences could also be used without making a significant difference because the iterators did not make use of the ordering that a sequence has. These four iterators do, and therefore the inversion function must take this into account. In handling the WHILE and UNTIL iterators, the inversion function must map onto a pair consisting of 1) the subsequence which is the result, and 2) an integer which is the index of the sequence element at which the value of the boolean expression changes. Then whenever the original sequence is modified, we compare the index at which the modification takes place with the stored index to determine how to update the inversion function. The parallel iterator combination is treated almost the same as the other combinations except that the inversion function maps onto sequences, and a correspondence must be maintained between the elements of the original sequences and the elements of the inverted sequence so that modifications can be passed along efficiently. The SORT iterator is handled by mapping onto sorted versions of the original data structure. Then new items are simply inserted in their proper place, and any modification to the BASED\ON expression will cause a reordering to be done.

Now we consider the iterative operations. These mainly influence the type of value which the inversion function maps onto. The set former, the FOR statement, the THE operation, and existential quantifiers which use the value assigned to the control variable, all require the inversion function to map onto sets. The sequence former and the FIRST and LAST operations require it to map onto sequences. "∃" (when control variable value is not used), "∀", and "#" need only a count. With the universal quantifier it is most efficient to keep a count of those set elements which do <u>not</u> satisfy the boolean expression. Then it can easily be tested against 0. The DEL and REPL operations require a set to be stored, but not a set of the values involved, rather a set of pointers to the locations of the values in the representation of the set or sequence from which the values are to be deleted. The collection iterative operation ("/") can be handled nicely if the operator involved has an inverse. Thus if we are inverting

$$+/X \in S \mid P(X)$$

we map onto the actual sums. We need the inverse of + so that we can subtract the appropriate value from this sum whenever an element is deleted from S. Luckily the two commonly used operators are "+" and "$\cup$". Set union does not have an inverse, but with multi-sets, difference is the inverse of union. So in the case of "$\cup$" we store multi-sets in the inversion function and then use them like sets when they are accessed.

Efficiency Factors

In applying iterator inversion to a particular iterator, we must decide the best domains to map the inversion function from. And we must also decide if even the best mapping is good enough to be worthwhile -- that is, we must decide whether to invert the iterator or not. These decisions are influenced by many factors. The speed issues are the frequency of execution of the iterator vs. the frequency of execution of modifications to the iterator and the amount of time required to update the inversion function for each modification. Notice that in some cases (though not often), updating the inversion function requires iterating over a set of values, so it is quite possible that inversion would slow down a program, not speed it up. We are assuming that the frequency information we need is either supplied by the programmer directly or computed by the system by running sample data sets supplied by the programmer. So from this information we decide on the most efficient set of free variables or expressions to map from, and then we compare this with the non-inverted implementation to determine if it is faster and by how much.

Frequently the inverted iterator will be faster, but then we must subtract the appropriate value from this sum whenever an element is deleted from S. Luckily the two commonly used operators are "+" and "$\cup$". Set union does not have an inverse, but with multi-sets, difference is the inverse of union. So in the case of "$\cup$" we store multi-sets in the inversion function and then use them like sets when they are accessed.

Efficiency Factors

In applying iterator inversion to a particular iterator, we must decide the best domains to map the inversion function from. And we must also decide if even the best mapping is good enough to be worthwhile -- that is, we must decide whether to invert the iterator or not. These decisions are influenced by many factors. The speed issues are the frequency of execution of the iterator vs. the frequency of execution of modifications to the iterator and the amount of time required to update the inversion function for each modification. Notice that in some cases (though not often), updating the inversion function requires iterating over a set of values, so it is quite possible that inversion would slow down a program, not speed it up. We are assuming that the frequency information we need is either supplied by the programmer directly or computed by the system by running sample data sets supplied by the programmer. So from this information we decide on the most efficient set of free variables or expressions to map from, and then we compare this with the non-inverted implementation to determine if it is faster and by how much.

Frequently the inverted iterator will be faster, but then we must examine the space issues to see what price we are paying for the speed. The main issue here is the amount of duplication introduced. First we must determine whether the inversion function must exist along with the original set or whether it can replace it. This is only possible if the inversion function maps onto subsets, if they are disjoint, and if the crucial primitives on the original structure can be done efficiently on the inversion function.

There is also a duplication problem if there is more than one iterator on the same data structure -- this requires a separate inversion function for each one. Finally, the subsets being mapped onto may not be disjoint. The examples we have shown so far have each contained one (or no) free variables, and in each case, by mapping from that variable we have obtained a function which maps onto disjoint subsets of the original set. This is not always the case. Sometimes the subsets will overlap, causing values to be duplicated within the inversion function. The amount of duplication will depend on the size of the original set, the size of the subsets mapped onto, and the number of subsets. This information again can be gotten from the user or by gathering statistics on sample data sets.

Now if (as will frequently be the case) we are gaining some speed at an increase in space, we will have to rely on the user's desires as far as the relative importance of time and space for his program.

It is important to mention that the expansions produced by substituting the updating code in line will contain a lot of redundancy. This can be seen clearly in the example in the Appendix. In order to really obtain the efficiency that we expect from applying iterator inversion to parts of a program, we will need a fairly sophisticated optimizer to remove all the redundant code.

Conclusions

We have not yet implemented iterator inversion as part of the VERS2 system and gotten experience with it, so much remains to be learned about its practical use. In addition, we feel that, as powerful as it is, it is only the first of a number of methods of designing data structure representation from iterators which might be developed. Some of these may even use iterator inversion as a starting point.

As an example, consider the following problem: Given a word G from a dictionary, return the set of all words which contain at least one letter from G. Assuming that a word is a sequence of letters and a dictionary is a set of words, this can be written

$$\{W \in \mathrm{DICT} \mid \exists L \in W \mid L \in G\} \quad .$$

If we just inverted this directly, we would get a function which mapped each word in the dictionary onto the set of all words to be returned. For normal sizes of words and dictionaries, this would take an astronomical amount of space, so we would like to find another method. A reasonable one would be to construct a function mapping each letter onto the set of all words containing that letter. If this were WU, then the solution would be

$$\cup/\mathrm{WU}(L)\ \mathrm{FOR}\ L \in G \quad .$$

Now this might have been derived from the original code as follows:

We could transform the first line into the equivalent expression

$$\cup/\{W \varepsilon \text{ DICT} \mid L \varepsilon W\} \text{ FOR } L \varepsilon G$$

then if we invert just the inner iterator, we get WU as the inversion function and it yields the solution above. We haven't gone any further with this, but it seems like a promising direction.

When we write a program, such as the simple LR(1) parse constructor, in completely applicative form using iterators, we are doing something similar to Dijkstra's notion of structured programming [6], except carrying the notion further. Dijkstra urges that, in the interest of clarity and an enhanced ability to prove assertions about programs, we eliminate GOTO's and substitute higher level control structures for them. We are also eliminating assignment statements and the whole notion of explicit flow of control and substituting iterators for them. Just as in pure Lisp, there is no specified order of evaluation in an applicative program. The difference between applicative VERS2 and pure Lisp is that we have introduced enough high level control operations (iterators) that it is now possible to express complex algorithms in a natural way in a purely applicative language. One might call this "highly structured programming". This should make it much more feasible to automatically optimize and prove things about these programs. Iterator inversion is the first evidence of this, but we expect more to follow. We hope that this paper will encourage others (especially those interested in proving assertions about programs) to investigate these possibilities.

References

[1] Balzer, R. "Automatic programming," USC Information Sciences Institute, 1972.

[2] Earley, J. "Relational level data structures in programming languages," Computer Science, University of California, Berkeley, 1973.

[3] Rulifson, J.F., Derksen, J. and Waldinger, R.J. "QA4: A procedural calculus for intuitive reasoning," Stanford Research Institute, 1972.

[4] Wegbreit, B. "The treatment of data types in EL1," Harvard University, 1971.

[5] Earley, J. "High level operations in automatic programming," Computer Science, University of California, Berkeley, 1973.

[6] Dijkstra, E.W. "Notes on Structured Programming," in Structured Programming, Academic Press, 1972.

[7] Knuth, D.E. The Art of Computer Programming, Vol. I, Addison-Wesley, 1968, p. 258.

[8] DeRemer, F.L. "Simple LR(k) Grammars," Comm. ACM, July 1971.

[9] Schwartz, J.T. "On Programming, Installment 1: Generalities," Computer Science Department, New York University, 1973.

[10] Iverson, K.E. A Programming Language, Wiley, 1962.

[11] Bobrow, D.G. and Raphael, B. "New Programming Languages for AI Research," 3rd International Joint Conference on Artificial Intelligence, 1973.

Acknowledgements

I have benefitted greatly from discussions with B. Wegbreit, L. Barnes, S. Chernicoff, and J. Barth while formulating these ideas.

Appendix A: EL1 Syntax

The following are the unusual EL1 features:

```
R ← EXPR(A:A1,B:B1;C) BEGIN···END
```

declares R to be a routine with formal parameters A and B of types A1 and B1 respectively returning a result of type C.

```
DECL X:T
```

declares X to be a variable taking on values of type T.

```
A :: T
```

declares A to be a special name for type T.

```
A.B
```

selects the B field of tuple A.

```
A\B
```

is just an identifier. "\" is just used as a letter.

```
bool → exp
```

stands for "IF bool THEN exp".

```
bool ⇒ exp
```

means that if the bool is true return the value of the exp as the result of the enclosing block.

```
⟦···⟧
```

is a BEGIN···END block.

Appendix B: Iterator Inversion Applied to Topological Sort

The new code in each figure is underlined.

Figure 1

Original

```
DECL S:SET(T)
PAIR :: TUP(FIRST:T,SECOND:T)
DECL SP:SET(PAIR)
FOR P ε INPUT() DO
  BEGIN
  ADD FIRST(P) TO S
  ADD SECOND(P) TO S
  ADD P TO SP
  END
WHILE ∃A ε S| ∄<-,A> ε SP DO
  BEGIN
  OUTPUT A
  DEL <A,-> ε SP
  DEL A FROM S
  #S = 0 → RETURN()
  END
ERROR
```

Figure 2

Initial Expansions

```
DECL S:SET(T)
PAIR :: TUP(FIRST:T,SECOND:T)
DECL SP:SET(PAIR)
FOR P $\varepsilon$ INPUT() DO
  BEGIN
  ADD FIRST(P) TO S
  ADD SECOND(P) TO S
  ADD P TO SP
  END
WHILE $\exists A \; \varepsilon \; S \mid \nexists \underline{P \; \varepsilon \; SP \mid SECOND(P) = A}$ DO
  BEGIN
  OUTPUT A
  $\underline{\text{FOR } Q \; \varepsilon \; SP \mid FIRST(Q) = A \text{ DO DEL } Q \text{ FROM } SP}$
  DEL A FROM S
  $\#S = 0 \rightarrow$ RETURN()
  END
ERROR
```

Figure 3

$Q \in SP \mid FIRST(Q) = A$ becomes $Q \in SUCC(A)$

```
DECL S:SET(T)
PAIR ::= TUP(FIRST:T,SECOND:T)
DECL SP:SET(PAIR)
DECL SUCC:FUNC(T,SET(PAIR) ← {})
FOR P ε INPUT() DO
  BEGIN
  ADD FIRST(P) TO S
  ADD SECOND(P) TO S
  ADD P TO SP
  ADD P TO SUCC(B)
  END
WHILE ∃A ε S| ∄P ε SP| SECOND(P) = A DO
  BEGIN
  OUTPUT A
  FOR Q ε SUCC(A) DO
    BEGIN
    DEL Q FROM SP
    DEL Q FROM SUCC(FIRST(Q))
    END
  DEL A FROM S
  #S = 0 → RETURN()
  END
ERROR
```

Figure 4

$\nexists P \in SP \mid \text{SECOND}(P) = A$ becomes $\text{COUNT}(A) = 0$

```
DECL S:SET(T)
PAIR :: TUP(FIRST:T,SECOND:T)
DECL SP:SET(PAIR)
DECL SUCC:FUNC(T,SET(PAIR) ← {})
DECL COUNT:FUNC(T,INT ← 0)
FOR P ε INPUT() DO
  BEGIN
  ADD FIRST(P) TO S
  ADD SECOND(P) TO S
  P ∉ SP →
    BEGIN
    COUNT(SECOND(P)) ← COUNT(SECOND(P))+1
    ADD P TO SP
    END
  ADD P TO SUCC(B)
  END
WHILE ∃A ε S| COUNT(A) = 0 DO
  BEGIN
  OUTPUT A
  FOR Q ε SUCC(A) DO
    BEGIN
    Q ε SP →
      BEGIN
      COUNT(SECOND(Q)) ← COUNT(SECOND(Q))-1
      DEL Q FROM SP
      END
    DEL Q FROM SUCC(FIRST(Q))
    END
  DEL A FROM S
  #S = 0 → RETURN()
  END
ERROR
```

Figure 5

$A \in S \mid COUNT(A) = 0$ becomes $A \in ZRCOUNT$

```
DECL S:SET(T)
PAIR :: TUP(FIRST:T,SECOND:T)
DECL SP:SET(PAIR)
DECL SUCC:FUNC(T,SET(PAIR) ← {})
DECL COUNT:FUNC(T,INT ← 0)
DECL ZRCOUNT:SET(T)
FOR P ε INPUT() DO
  BEGIN
  ADD FIRST(P) TO S
  COUNT(FIRST(P)) = 0 → ADD FIRST(P) TO ZRCOUNT
  ADD SECOND(P) TO S
  COUNT(SECOND(P)) = 0 → ADD SECOND(P) TO ZRCOUNT
  P ∉ SP →
    BEGIN
    SECOND(P) ε S →
      BEGIN
      COUNT(SECOND(P)) = 0 → DEL SECOND(P) FROM ZRCOUNT
      COUNT(SECOND(P))+1 = 0 → ADD SECOND(P) TO ZRCOUNT
      END
    COUNT(SECOND(P)) ← COUNT(SECOND(P))+1
    ADD P TO SP
    END
  ADD P TO SUCC(B)
  END
WHILE ∃A ε ZRCOUNT DO
  BEGIN
  OUTPUT A
  FOR Q ε SUCC(A) DO
    BEGIN
    Q ε SP →
      BEGIN
```

```
        SECOND(Q) ε S →
          BEGIN
          COUNT(SECOND(Q)) = 0 → DEL SECOND(Q) FROM ZRCOUNT
          COUNT(SECOND(Q))-1 = 0 → ADD SECOND(Q) TO ZRCOUNT
          END
        COUNT(SECOND(Q)) ← COUNT(SECOND(Q))-1
        DEL Q FROM SP
        END
      DEL Q FROM SUCC(FIRST(Q))
      END
    DEL A FROM S
    COUNT(A) = 0 → DEL A FROM ZRCOUNT
    #S = 0 → RETURN()
    END
  ERROR
```

Figure 6

Redundancies Cleaned Up

```
DECL S:SET(T)
PAIR :: TUP(FIRST:T,SECOND:T)
DECL SUCC:FUNC(T,SET(PAIR) ← {})
DECL COUNT:FUNC(T,INT ← 0)
DECL ZRCOUNT:SET(T)
FOR P ε INPUT() DO
  BEGIN
  ADD FIRST(P) TO S
  COUNT(FIRST(P)) = 0 → ADD FIRST(P) TO ZRCOUNT
  ADD SECOND(P) TO S
  COUNT(SECOND(P)) ← COUNT(SECOND(P))+1
  ADD P TO SUCC(FIRST(P))
  END
WHILE  A ε ZRCOUNT DO
  BEGIN
  OUTPUT A
  FOR Q ε SUCC(A) DO
    BEGIN
    COUNT(SECOND(Q)) = 1 → ADD SECOND(Q) TO ZRCOUNT
    COUNT(SECOND(Q)) ← COUNT(SECOND(Q))-1
    DEL Q FROM SUCC(A)
    END
  DEL A FROM S
  DEL A FROM ZRCOUNT
  #S= 0 → RETURN()
  END
ERROR
```