NESTED ITERATORS AND RECURSIVE BACKTRACKING

by

Raphael Finkel
Marvin Solomon

Computer Science Technical Report #388

June 1980

Nested Iterators and Recursive Backtracking

Raphael Finkel
Marvin Solomon

University of Wisconsin--Madison
Technical Report 388

Abstract

This paper introduces a new programming language construct called the _nested iterator_, which is useful in coding iterative solutions to backtracking problems that are usually attacked by more complex recursive methods. We show how to generate code from nested iterators and present several programs that employ nested iterators.

TABLE OF CONTENTS

Nested Iterators and Recursive Backtracking

1. Introduction

Consider the following elementary programming task:

> Problem S: A is an n by n array of real numbers. Compute the sum of its elements.

A reasonable program to accomplish this task is shown below: (The syntactic details of the examples in this paper are not meant to represent any particular existing programming language.)

```
Program S1
sum := Ø;
for i from 1 to n do
    for j from 1 to n do
        sum := sum + A[i,j]
    od
od
```

An extremely peculiar program employs recursion on the number of dimensions:

```
Program S2
var i : array[1..2] of integer;
procedure advance(k : integer):
    if k > 2 then
        sum := sum + A[i[1],i[2]]
    else
        for i[k] from 1 to n do advance(k+1) od
    fi
sum := Ø;
advance(1)
```

Although Program S2 seems strange, it might be used if the number of dimensions of the array were quite large or not known at compile time. Algorithms that search large spaces by backtracking often have exactly those properties: The number of dimensions of the search space is high, or the depth of the search is not known at compile time. For this reason, backtracking is almost always implemented by recursive procedures.

We will show how backtracking search and enumeration algorithms usually implemented with complex recursive procedures in the style of Program S2 can be recast in the iterative style of Program S1 with the aid of a new control structure. The translation often yields a more perspicuous program. Moreover, since recursive procedure calls are replaced by simple control flow, the program is likely to be more efficient.

2. Nested Iterators

Let us begin with a standard example of backtracking. The "eight queens" problem [1] was known to Gauss:

> Problem Q: Find all placements of eight queens on a chessboard so that no two share a row, column, or diagonal.

A first brute-force approach is to attempt all $\binom{64}{8} = 4{,}426{,}165{,}368$ placements of eight queens on 64 squares. An enormous reduction in effort is derived from the fact that there must be exactly one queen in each column; it suffices to consider the $8^8 = 16{,}777{,}216$ choices of a row for each queen. Usually, once this reduction has been made, a program is written that uses recursion to achieve the backtrack. However, we may restrict ourselves to iteration:

```
Program Q1
for q[1] from 1 to 8 do
    for q[2] from 1 to 8 do
        .
         .
          .
            for q[8] from 1 to 8 do
                if ok(q[1..8]) then
                    yield q[1..8] fi
            od
          .
         .
        .
    od
od
```

Ok is a Boolean procedure that verifies that its argument represents a non-attacking set of queens. We are using _yield_ in the CLU [2] sense; the program might print out the result at this point or otherwise make it available to the caller.

If the first few queens have a conflict, no placement of the remaining queens can lead to a solution. This observation leads to a second dramatic reduction in the number of cases that must be considered:

```
Program Q2
for q[1] from 1 to 8 do
    if ok(q[1..1]) then
        for q[2] from 1 to 8 do
            if ok(q[1..2]) then
            .
             .
              .
                if ok(q[1..8]) then
                    yield q[1..8]
                fi
              .
             .
            .
            fi
        od
    fi
od
```

Program Q2 suffers from two defects: It is rather unwieldy, and it does not readily extend to

> Problem Q': Solve Problem Q with n queens on an n by n chessboard, for n = 3, 4, 5, ..., 16.

The first defect may be remedied by a macro preprocessor (analogous to our use of ellipsis above, and suggested in [3], section 4.1.4), but the second defect remains: If the number of levels of nesting is not known at compile time, the macro approach is inadequate.

We introduce the nest programming language construct to abbreviate Program Q2 and simultaneously generalize to Problem Q':

```
Program Q3
for n from 3 to 16 do
    nest i from 1 to n <<
        for q[i] from 1 to n do
            if ok(q[1..i]) then
                inner
            fi
        od
    >> do yield q[1..n] od
od
```

This program yields all solutions to n-queens problems for n in [3..16].

The general form of _nest_ is:

```
nest <control variable> from <lower limit> to <upper limit>
    << <statement list> >>
    do <statement list> od
```

The reserved word _inner_ must appear as a statement exactly once within the first statement list. It marks where that same statement list should be inserted as the next level. If all levels have been finished, then the second statement list (enclosed by _do_ and _od_) is inserted instead.

The eight queens problem is only one example of a large class of algorithms that operate by extending a partial solution in all possible ways until a solution is obtained. If the number of steps in a direct path to a solution is bounded by the value max (in the eight queens problem, max is 8), a general program might look like this:

```
Program R1
initialize;
nest i from 1 to max <<
    for state[i] in extensions to state[1..i-1] do
        if state[1..i] is feasible then
            inner
        fi
    od >>
do yield state[1..max] od
```

In some cases, primarily tree searches, the depth of search has no _a priori_ bound. Such cases might use an unbounded _nest_:

```
Program R2
initialize;
nest i from 1 <<
    for state[i] in extensions to state[1..i-1] do
        if state[1..i] is feasible then
            if state[1..i] is a solution then
                yield state[1..i]
            else inner
            fi
        fi
    od >>
```

This program consists of an infinite nest of loop-test pairs. In any given execution, control should never penetrate more than a finite number of levels before the inner if succeeds, the outer if fails, or the for loop has an empty set of possible extensions. Therefore, the unbounded nest has no do part.

3. Implementation

The implementation of nest is related to the implementation of for, which we will consider first. The loop

```
for i from 1 to n do S od
```

may be considered an abbreviation for the sequence of statements

$$S_1;\ S_2;\ \ldots;\ S_n,$$

where S_k denotes a version of S in which the variable i (presumably occurring in S) has the value k. Figure 1 shows a flow chart for one standard interpretation of the for loop. (We do not mean to imply that Figure 1 is the best way to implement the for loop. This topic has been discussed elsewhere [4]. Our discussion is equally applicable to any of the proposed variants.)

We have grouped the actions dealing with loop control in the dashed box. The two arcs entering the box are labelled i and n, for initial and next, and the two exits are labelled s and f, for successor and final. The box delimits what is sometimes called a generator or iterator [2,5]. We will call any such construct with two inputs and two outputs an iterator. When iterators are nested, the flow graph can be grouped as shown in Figure 2. Figure 3 shows a nest of k iterators schematically. The successor exit from box j is connected to the initial entrance to box j+1 if j<k and to S otherwise. Similarly, the final exit of box j goes to the next entrance of box j-1 if j>1 and to the final exit of the nest otherwise. The initial entrance of the entire nest is connected to the initial entrance of box 1; the next entrance (the arc from S) is connected to the next entrance of box k.

We can simulate Figure 3 with the flow graph of Figure 4. The structure of Figure 4 has three parts: The center is the body of the nested iterator, on the left is an iterator that counts i upward from 1 to k, and on the right is an iterator that counts i downward from k to 1. (We demand that the statements within the nest body not modify its control variable i, just as we demand that statements within a for loop not modify its control variable.)

One might conjecture that the flow graph of Figure 4 could be expressed (perhaps more clearly) by the conventional control constructs if, while, and exit. However, when Figure 1 is used to expand the for loop, this flow graph has a subgraph equivalent to one that has been shown impossible to represent using only if, loop, and multi-level exit without changing its execution sequence or increasing its length [6]. This fact suggests that nest should be included as a basic control structure in algorithmic languages, since it expresses a flow of control that could not otherwise be expressed without resort to goto statements. The nest structure may be translated into lower-level tests and conditional branches; we discuss this translation later.

Nesting can easily be extended beyond for statements to any iterator. For example, in Program Q3, the iterator includes an if statement. It is, nonetheless, a two-input/two-output fragment. Figure 5 shows the flow chart of the nest in Program Q3.

A bounded nested iterator, not including its do part, is itself an iterator. An unbounded nest, on the other hand, is a compound statement rather than an iterator; its flow graph has a single entrance and exit, as shown in Figure 6.

4. Examples

We now present some examples of generation and search techniques that are easy to express using nested iterators.

4.1 Permutations and Combinations

Here is a program that generates, in lexicographic order, all $\binom{n}{k}$ combinations of k integers chosen from {1, ..., n}:

```
program Choose
var Choices : array[0..k] of 1..n;
Choices[0] := 0;  { dummy for initialization }
nest i from 1 to k
<<
    for Choices[i] := Choices[i-1] + 1 to n+k-i
        do inner od
>> do yield Choices[1..k] od
```

This algorithm is identical to the one given in [3] in section 5.2, but our program is easier to read.

A similar method may be used to generate all permutations in lexicographic order on a set of numbers. The overall structure of the program is as follows:

```
program Permutations(S : sorted set of integer);
    nest i from 1 to size(S)
    <<
        forall P[i] in S - {P[1], ..., P[i-1]}
        do inner od
    >>
        do yield P[1..size(S)] od
```

A doubly-linked list may be used to implement the sorted set, and P[i] may be deleted and then restored around the inner statement. A more efficient program [7] can be derived from some facts about the sequence of permutations produced. If $S = \{a_1, \ldots, a_n\}$, where $a_1 < a_2 < \ldots < a_n$, then the sequence of permutations of n in lexicographic order is a_1s_{11}, a_1s_{12}, ..., a_1s_{1m}, a_2s_{21}, ..., a_2s_{2m}, ..., a_ns_{n1}, where $m = (n-1)!$ and $s_{i1}, \ldots, s_{im}$ is the sequence of permutations of $S - \{a_i\}$ in lexicographic order. Now $a_is_{im} = a_ia_na_{n-1} \ldots a_{i+1}a_{i-1} \ldots a_1$, and $a_{i+1}s_{i+1,1} = a_{i+1}a_1 \ldots$

$a_{i-1}a_i a_{i+2} \ldots a_n$. Therefore, if the current permutation is kept in an array P, the translation from $a_i s_{im}$ to $a_{i+1} s_{i+1,1}$ can be achieved by swapping the first element of P with the element in position n-i+1 to obtain $a_{i+1}a_n \ldots a_{i+2}a_i \ldots a_1$ and then reversing position 2 through n. These observations allow us to replace the forall loop by a loop on i = rank(j), the rank of P[j] among P[j..n]:

```
program Permutations
for j from 1 to n do P[j] := j od;
nest j from 1 to n
<<
    for rank[j] from 1 to n-j+1
    do
        inner;
        if rank[j] < n-j+1 then
            swap(j,n-rank[j]+1);
            reverse(j+1,n)
        fi
    od
>> do yield P[1..n] od
```

"Swap" interchanges two elements of P, and "reverse" reverses all elements of P between the indicated positions.

4.2 Trees

The set of all directed d-ary trees on n nodes can be generated in many ways. One, given by [8] for d=2, may be generalized in the following way: The set of d-ary trees is in 1-1 correspondence with the set of sequences of n red markers and (d-1)n black markers having the property that the ratio between the numbers of black and red markers in any initial segment is at most d-1. (Therefore, the first position is never black and the last position is never red.) Generation of legal marker sequences is similar to the solution of the 8-queens problem:

```
program Marker Positions
    Count[red] := Ø;
    Count[black] := Ø;
    nest i from 1 to d*n
    <<
        for Marker[i] := black to red
        do
            Count[Marker[i]] := Count[Marker[i]] + 1;
            if (Count[black] <= Count[red]*(d-1)) and
                (Count[red] <= n)
            then inner fi;
            Count[Marker[i]] := Count[Marker[i]] - 1;
        od
    >> do yield Marker[1..d*n] od
```

A tree can be generated from a legal marker sequence by calling MakeTree(1):

```
program MakeTree(var Pos : integer) : tree;
{ Generate a tree from a subsequence of Marker, starting in
position Pos.  Leave Pos set to the first unused position in
Marker. }
var Answer : tree;
begin
    assert Marker[Pos] = red;
    Pos := Pos + 1;
    new(Answer);
    for c := 1 to d
    do
        if Marker[Pos] = black then
            Answer^.child[c] := nil;
            Pos := Pos + 1
        else Answer^.child[c] := MakeTree(Pos)
        fi
    od;
    MakeTree := Answer;
end; { MakeTree }
```

4.3 Partitions and Compositions

Partitions of an integer n are multisets of positive integers that sum to n [3]. We will represent each partition by a sequence sorted in non-decreasing order. An unbounded nest can be used to generate all partitions of n in lexicographic order:

```
program Partitions
Remainder := n;
P[Ø] := 1;  { dummy to start off }
nest i from 1
<<
    for P[i] := P[i-1] to Remainder
    do
        Remainder := Remainder - P[i];
        if Remainder = Ø then
            yield P[1..i]
        else inner fi;
        Remainder := Remainder + P[i];
    od
>>
```

Compositions of n are sequences of positive integers that sum to n. They differ from partitions in that the order of the integers is significant. Changing the lower limit of the *for* loop above to 1 yields the compositions of n instead of the partitions. The k-part compositions are sequences of k non-negative integers that sum to n. They are generated by a similar program:

```
program Compositions
Remainder := n;
nest i from 1 to k-1
<<
    for C[i] := Ø to Remainder
    do
        Remainder := Remainder - C[i];
        inner;
        Remainder := Remainder + C[i]
    od
>>
    do
        C[k] := Remainder;
        yield C[1..k]
    od
```

A more compact representation of a partition lists each distinct integer together with its multiplicity. The following program represents a partition by a strictly decreasing sequence P of integers together with a corresponding sequence M of their multiplicities. It also uses an array Remainder with the property that Remainder[i] = n - sum{P[j]*M[j] | j < i}. The algorithm is similar to one presented in [3].

```
program Compressed Partitions
Remainder[1] := n;  { full value }
P[Ø] := n + 1; { dummy for consistency }
nest i from 1
<<
    P[i] := 1;
    M[i] := Remainder[i];
    yield P[1..i], M[1..i];
    for P[i] := 2 to min(Remainder[i],P[i-1]-1)
    do
        Remainder[i+1] := Remainder[i];
        for M[i] := 1 to Remainder[i] div P[i]
        do
            Remainder[i+1] :=
                Remainder[i+1] - P[i];
            if Remainder[i+1] = Ø
            then yield Permut[1..i],M[1..i]
            else inner
            fi
        od
    od
>>
```

4.4 Alpha-beta search

Alpha-beta search is a technique for evaluating positions in two-player games [9]. The alpha-beta search algorithm can be described easily with an unbounded nest:

```
program Alpha Beta Search
var path : sequence of node;
path[Ø].alpha := -INFINITY; path[Ø].beta := INFINITY;
path[Ø].pos := InitialPosition;
```

```
nest depth from Ø
<<
    if path[depth].pos is a terminal position then
        path[depth].value :=
            StaticEvaluation(path[depth].pos)
    else
        for NextPos[depth] in
            SuccessorPositions(path[depth].pos)
        do
            path[depth+1].alpha := -path[depth].beta;
            path[depth+1].beta := -path[depth].alpha;
            path[depth+1].pos := NextPos[depth];
            inner;
            path[depth].alpha :=
                max(-path[depth+1].value,path[depth].alpha);
            if path[depth].alpha >= beta
                then exitloop fi
        od
    fi
>>
```

5. Generating Code

This section presents a method for generating efficient code for nested iterators. Code is generated for each statement in two segments. For iterators, the first code segment starts with the initial entry and ends with the successor exit; the second starts with the next entry and ends with the final exit. We allow jumps between the segments. The code for other statement types is divided somewhat arbitrarily into two segments for consistency in the translation algorithm.

We present the generated code by means of an S-attributed translation [1Ø]. Each non-terminal symbol representing a statement has three synthesized attributes: T1 and T2 are the two segments of the generated code and I is a Boolean attribute that is true only for iterators. The translation of the corresponding code fragment is the concatenation of T1 with T2. Each other non-terminal symbol (for example, <var>) has one attribute representing its translation. In this case, we omit the grammar and attribute rules and use the symbol itself to represent its translation. The source language is the *ad hoc* language used in all the examples in this paper. The only control structures in the target language are conditional and unconditional *goto* statements.

For clarity, we will display the target language in a high-level syntax, although a practical translator would produce some form of intermediate code or machine language. We also assume that Genlabel is a procedure that assigns a new label to its argument.

```
<statement> ::= inner

    <statement>.I = true
    <statement>.T1 =   {empty translation}
    <statement>.T2 =   {empty translation}


<statement> ::= nest <var> from <expr1> to <expr2>
    '<<' <statement list1> '>>' do <statement list2> od

    if not <statement list1>.I then error
    Genlabel(L1); Genlabel(L2); Genlabel(L3); Genlabel(L4);
```

```
    <statement>.I = <statement list2>.I
    <statement>.T1 =
            <var> := <expr1>
        L1 :
            IF <var> > <expr2> THEN GOTO L2
            <statement list1>.T1
            <var> := <var> + 1
            GOTO L1
        L2 :
            <statement list2>.T1
    <statement>.T2 =
            <statement list2>.T2
            <var> := <expr2>
        L3 :
            IF <var> < <expr1> THEN GOTO L4
            <statement list1>.T2
            <var> := <var> - 1
            GOTO L3
        L4 :
```

<statement> ::= <u>nest</u> <var> <u>from</u> <expr> '<<' <statement list> '>>'

```
    Genlabel(L1); Genlabel(L2); Genlabel(L3);

    <statement>.I = false
    <statement>.T1 =
            <var> := <expr>
        L1 :
            <statement list>.T1
            <var> := <var> + 1
            GOTO L1
    <statement>.T2 =
        L2 :
            IF <var> < <expr> THEN GOTO L3
            <statement list>.T2
            <var> := <var> - 1
            GOTO L2
        L3 :
```

```
<statement list1> ::= <statement> ; <statement list2>

    if <statement>.I then
        if <statement list2>.I then error
        else
            <statement list1>.I = true
            <statement list1>.T1 =
                <statement>.T1
            <statement list1>.T2 =
                <statement>.T2
                <statement list2>.T1
                <statement list2>.T2
        fi
    elif <statement list2>.I then
        <statement list1>.I = true;
        <statement list1>.T1 =
            <statement>.T1
            <statement>.T2
            <statement list2>.T1
        <statement list1>.T2 =
            <statement list2>.T2
    else
        <statement list1>.I = false;
        <statement list1>.T1 =
            <statement>.T1
            <statement>.T2
        <statement list1>.T2 =
            <statement list2>.T1
            <statement list2>.T2
    fi


<statement> ::= while <bool> do <statement list> od

    Genlabel(L1); Genlabel(L2);
    <statement>.I = <statement list>.I
    <statement>.T1 =
    L1 :
        IF NOT ( <bool> ) THEN GOTO L2
        <statement list>.T1
    <statement>.T2 =
        <statement list>.T2
        GOTO L1
    L2 :


<statement> ::= if <bool> then <statement list1>
    else <statement list2> fi
```

```
    if <statement list1>.I then
        if <statement list2>.I then error
        else
            Genlabel(L1) Genlabel(L2);
            <statement>.I = true
            <statement>.T1 =
                IF <bool> THEN GOTO L1
                <statement list2>.T1
                <statement list2>.T2
                GOTO L2
            L1 :
                <statement list1>.T1
            <statement>.T2 =
                <statement list1>.T2
            L2 :
        fi
    else
        Genlabel(L1) Genlabel(L2);
        <statement>.I = <statement list2>.I
        <statement>.T1 =
            IF NOT ( <bool> ) THEN GOTO L1
            <statement list1>.T1
            <statement list1>.T2
            GOTO L2
        L1 :
            <statement list2>.T1
        <statement>.T2 =
            <statement list2>.T2
        L2 :
    fi


<statement> ::= for <var> from <expr1> to <expr2>
    do <statement list> od

    Genlabel(L1); Genlabel(L2);

    <statement>.I = <statement list>.I
    <statement>.T1 =
        <var> := <expr1>
    L1 :
        IF <var> > <expr2> THEN GOTO L2
        <statement list>.T1
    <statement>.T2 =
        <statement list>.T2
        <var> := <var> + 1
        GOTO L1
    L2 :
```

For example, consider the following code:

```
nest i from 1 to n
<<
    if q[i]
    then s1
    else
        s2;
        inner;
        s3
    fi
>> do s4
```

The translation is:

```
    i := 1;
L11 :
    IF i > n THEN GOTO L12;
    IF NOT ( q[i] ) THEN GOTO L21;
    s1;
    GOTO L22;
L21 :
    s2;
    i := i + 1;
    GOTO L11;
L12 :
    s4;
    i := n;
L13 :
    IF i < 1 THEN GOTO L14;
    s3;
L22 :
    i := i - 1;
    GOTO L13;
L14 :
```

6. Conclusions

Iterative solutions to problems are often preferable to recursive solutions for several reasons:

1) They correspond directly to a natural description of the problem. An iterative solution is usually shorter and easier to understand than a recursive solution.

2) They do not hide information in a recursion stack. For example, a successful search is represented by a path to the goal. In a recursive implementation, the path is on the recursion stack; it must be duplicated in a separate array to make it accessible to a printing routine. Iterative solutions have no recursion stack and therefore do not duplicate this information.

3) The code generated by a compiler for a nested iterator solution is likely to be substantially faster than the code for the best recursive solution. Moreover, more compiler optimizations are possible without inter-procedural data flow analysis.

An obstacle to using iterative techniques is the fact that the iterative program corresponding to a backtracking algorithm cannot be easily expressed in traditional high-level control constructs. The nesting constructs introduced in this paper remove that obstacle.

7. Bibliography

[1] N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall (1976).

[2] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in CLU," Communications of the ACM 20, 8, pp. 564-576 (August 1977).

[3] E. M. Reingold, J. Nievergelt, and N. Deo, Combinatorial Algorithms: Theory and Practice, Prentice-Hall (1977).

[4] F. Baskett, "The best simple code generation technique for WHILE, FOR, and DO loops," Sigplan Notices 13, 4, pp. 31-32 (April 1978).

[5] M. Shaw, W. Wulf, and R. London, "Abstraction and Verification in Alphard: Iteration and Generators," Technical Report, Carnegie-Mellon University Department of Computer Science (August 1976).

[6] W. W. Peterson, T. Kasami, and N. Tokura, "On the capabilities of while, repeat, and exit statements," Communications of the ACM 16, 8, pp. 503-512 (August 1973).

[7] R. J. Ord-Smith, "Algorithm 323: Generation of Permutations in Lexicographic Order," Communications of the ACM 11, p. 117 (1968).

[8] A. Proskurowski, "On the generation of binary trees," Journal of the ACM 27, 1, pp. 1-2 (January 1980).

[9] D. E. Knuth and R. W. Moore, "An Analysis of Alpha-Beta Pruning," Artificial Intelligence 6, 4, pp. 293-326 (Winter 1975).

[10] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns, "Attributed translations," Journal of Computer and System Sciences 9, pp. 279-307 (1974).

8. Figures

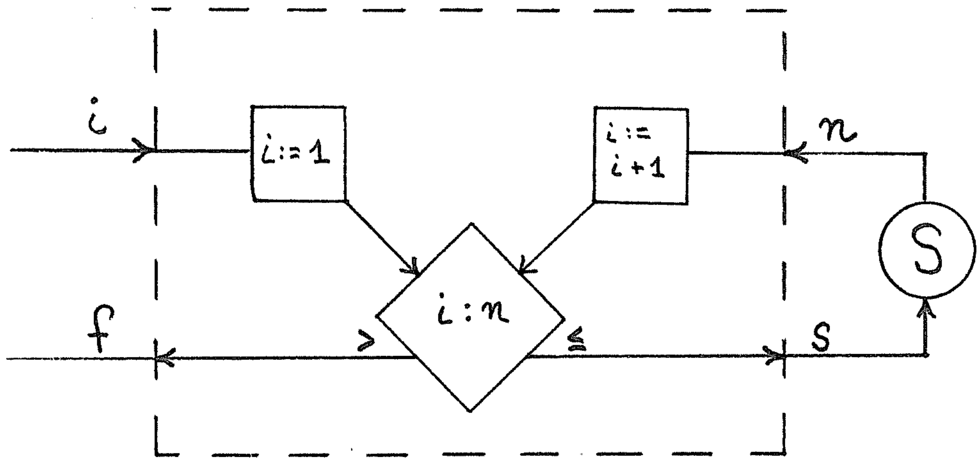

Figure 1: <u>for</u> i <u>from</u> 1 <u>to</u> n <u>do</u> S <u>od</u>

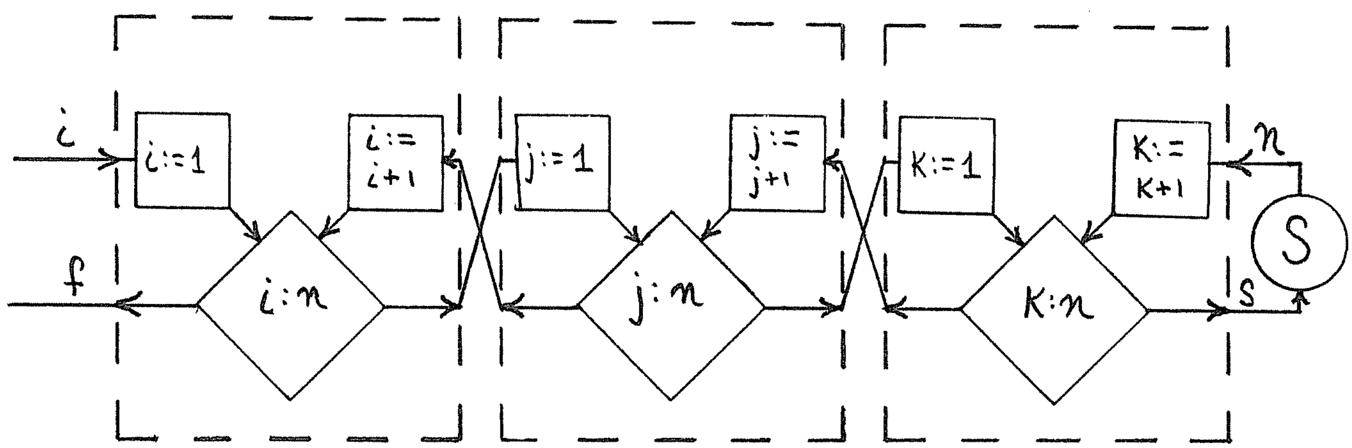

Figure 2:
<u>for</u> i <u>from</u> 1 <u>to</u> n <u>do</u>
  <u>for</u> j <u>from</u> 1 <u>to</u> n <u>do</u>
    <u>for</u> k <u>from</u> 1 <u>to</u> n <u>do</u> S <u>od</u>
  <u>od</u>
<u>od</u>

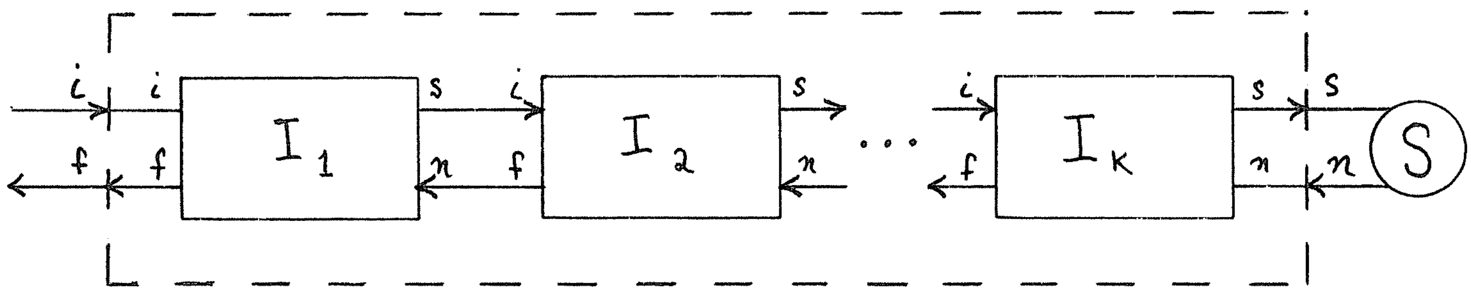

Figure 3: k nested iterators

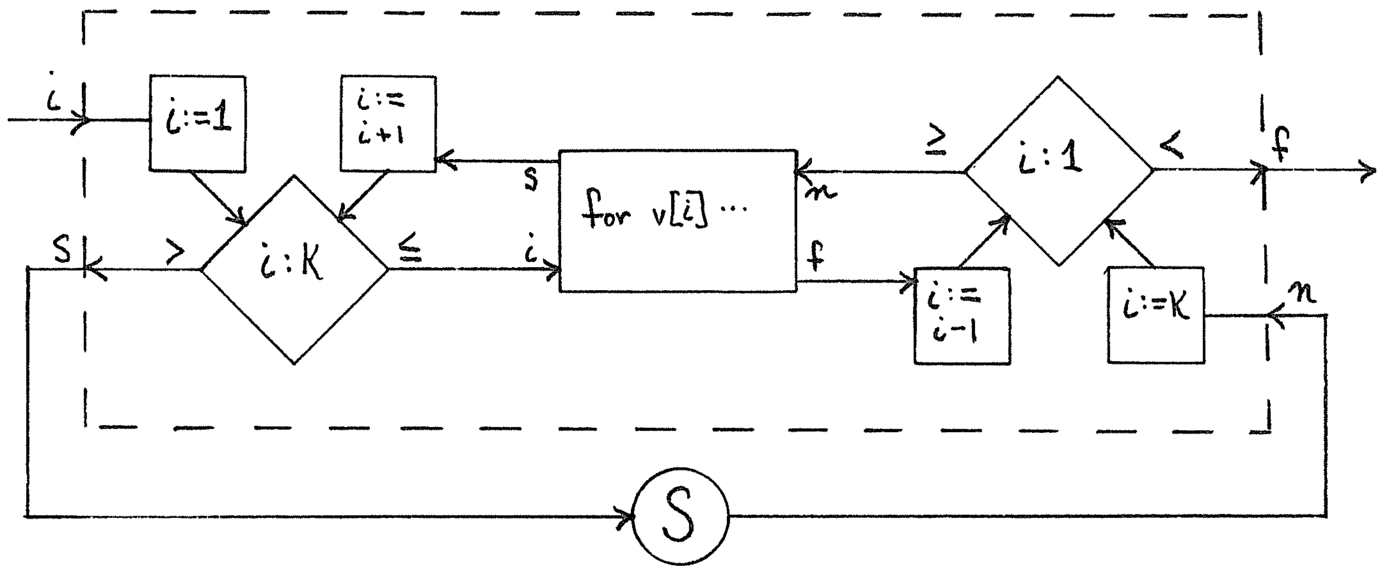

Figure 4

nest i from 1 to k
<<for v[i] ... >> do S od

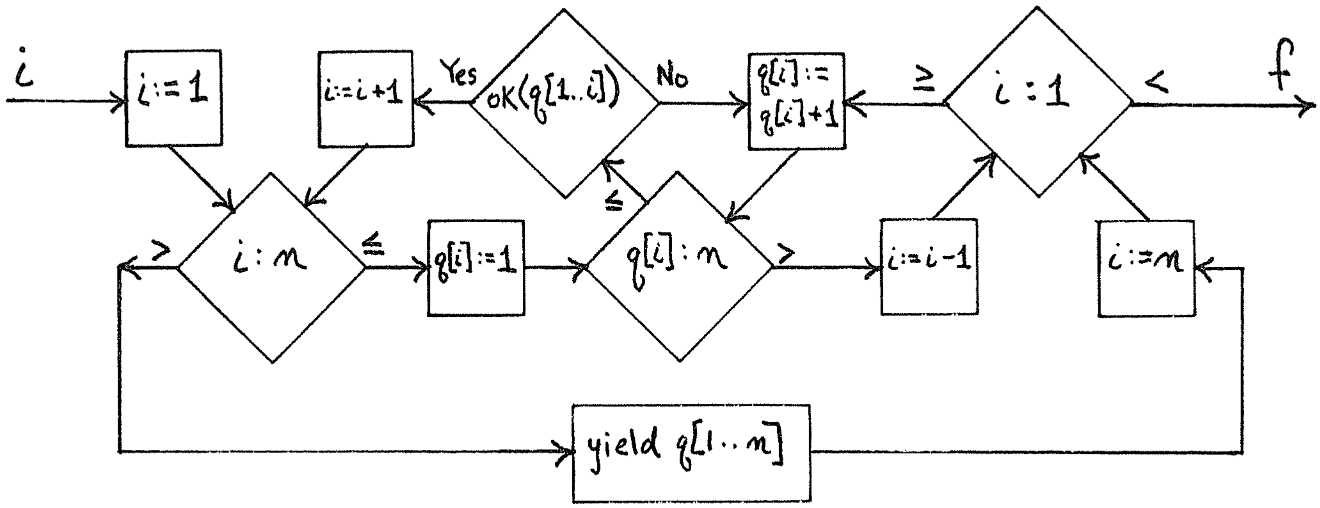

Figure 5

```
nest i from 1 to n
<<
    for q[i] from 1 to n do
        if ok(q[1..i]) then inner fi
    od
>>
do yield q[1..n] od
```

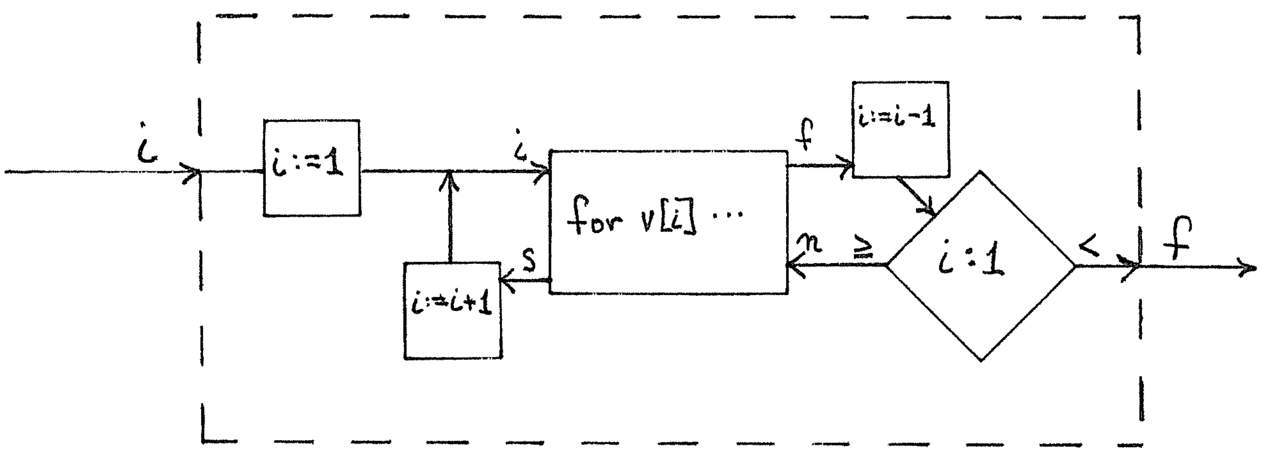

Figure 6: nest i from 1 << for v[i] ... >>