This section has not been included because the problem tackled in it is very exciting. On the contrary, I feel tempted to remark that the problem is perhaps too trivial to act as a good testing ground for an orderly approach to the problem of program composition. This section has been included because it contains a true eye-witness account of what happened in the classroom. It should be interpreted as a partial answer to the question that is often posed to me, viz. to what extent I can teach programming style. (I never used the "Notes on Structured Programming"—mainly addressed to myself and perhaps to my colleagues—in teaching. The classroom experiment described in this section took place at the end of a course entitled "Introduction into the Art of Programming", for which separate lecture notes—with exercises and all that—were written. As at the moment of writing the students that followed this course have still to pass their examination, it is for me still an open question how successful I have been. They liked the course, I have heard that they described my programs as "logical poems", so I have the best of hopes.)

17. The Problem Of The Eight Queens

This last section is adapted from my lecture notes "Introduction into the Art of Programming". I owe the example—as many other good ones—to Niklaus Wirth. This last section is added for two reasons.

Firstly, it is a second effort to do more justice to the process of invention. (As a matter of fact I start where the student is not familiar with the concept of backtracking and aim at discovering it as I go along.)

Secondly, and that is more important, it deals with recursion as a programming technique. In preceding sections (particularly in "On a program model") I have reviewed the subroutine concept; there it emerged as an embodiment of what I have also called "operational abstraction". In the relation between main program and subroutine we can distinguish quite clearly two different semantic levels. On the level of the main program the subroutine represents a primitive action; on that level it is used on account of "what it does for us" and on that same level it is irrelevant "how it works". On the level of the subroutine body we are concerned with how it works but can—and should—abstract from how it is used. This clear separation of the two semantic levels "what it does" and "how it works" is denied to the designer of a recursive procedure. As a result of this circumstance the design of a recursive routine requires a different mental skill, justifying the inclusion of the current section in this manuscript. The recursive procedure has to be understood and conceived on a single semantic level: as such it is more like a sequencing device, comparable to the repetitive clauses.

It is requested to make a program generating all configurations of eight queens on a chessboard of 8*8 squares such that no queen can take any of the others. This means that in the configurations sought, no two queens may be on the same row, on the same column or on the same diagonal.

We don't have an operator generating all these configurations, this operator is precisely what we have to make. Now there is a very general way (cf. "On grouping and sequencing") of tackling such a problem, which is as follows.

Call the set of configurations to be generated: set A. Look for a set B of configurations with the following properties:

(1) set A is a subset of set B

(2) given an element of set B it is not too difficult to decide whether it belongs to set A as well

(3) we can make an operator generating all elements of set B.

With the aid of the generator (3) for the elements of set B, all elements of set B can then be generated in turn; they will be subjected to the decision criterion (2) which decides whether they have to be skipped or handed over, thus generating elements of set A. Thanks to (1) this algorithm will produce *all* elements of set A.

Three remarks are in order.

(1) If the whole approach is to make sense, set B is not identical to set A, and as it must contain set A as a (true) subset, it must be larger than set A. For reasons of efficiency, however, it is advisable to choose set B "as small as possible": the more elements it has, the more elements of it have to be skipped on account of the decision criterion (2).

(2) We should look for a decision criterion that is cheap to apply, at least the discovery that an element of B does *not* belong to A should (on the average) be cheap. Also this is dictated by efficiency considerations, as we may expect set B to be an order of magnitude larger than set A, i.e. the majority of the elements of B will have to be rejected.

(3) The assumption is that the generation of the elements of set B is easier than a direct generation of the elements of set A. If, nevertheless, the generation of the elements of set B still presents difficulties, we can repeat our pattern of thinking, re-apply the trick and look for a still larger set C of configurations that contains B as a subset etc. (And, as the careful reader will observe, we shall do so in the course of this example.)

Above, we have sketched a very general approach, applicable to many, very different problems. Faced with a particular problem, i.e. faced with a specific set A, the problem of course is what to select for our set B.

In a moment of optimism one could think that this is an easy matter, as we might consider the following technique. We list all the mutually independent conditions that our elements of set A must satisfy and omit one of them. Sometimes this works but as a general technique it is too naïve: its shortcomings become apparent when we apply it blindly to the problem of the eight queens. We can characterise our solutions by the two conditions

(1) there are 8 queens on the board

(2) no two of the queens can take each other.

Omitting either of them gives for set B the alternatives

B1: all configurations with N queens on the board such that no two queens can take eachother

B2: all configurations of 8 queens on the board.

But both sets are so ludicrously huge that they lead to utterly impractical algorithms. So we have to be smarter. The burning question is: "How?".

Well, at this stage of our considerations, being slightly at a loss, we are not so much concerned with the efficiency of our final program as with the efficiency of our own thought processes! So, if we decide to make a list of properties of solutions, in the hope of finding a useful clue, this is a rather undirected search and therefore we should not invest too much mental energy in such a search, that is: for a start we should restrict ourselves to their *obvious* properties.

(I gave the puzzle as a sobering exercise to one of the staff members of the Department of Mathematics at my University, because he expressed the opinion that programming was easy. He violated the above rule and, being, apart from a pure, perhaps also a poor mathematician, he started to look for interesting, non-obvious properties. He conjectured that if the chessboard were divided in four squares of 4*4 fields, each square should contain two queens, and then he started to prove this conjecture without having convinced himself that he could make good use of it. He still has not solved the problem and, as far as I know, has not yet discovered that his conjecture is false.)

Well, let us go ahead and let us list the obvious properties we can think of.

(a) No row may contain more than one queen, 8 queens are to be placed and the chessboard has exactly 8 rows. As a result we conclude that each row will contain precisely one queen.

(b) Similarly we conclude that each column will contain precisely on queen.

(c) There are 15 "upward" diagonals, each of them containing at most one queen, i.e. 8 upward diagonals contain precisely one queen and 7 upward diagonals are empty.

(d) Similarly we conclude that 8 downward diagonals contain precisely one queen and 7 are empty.

(e) Given any non-empty configuration of queens such that no two of them can take each other, removal of any one of these queens will result in a configuration sharing that property.

Now the last property is very important. (To be quite honest: here I feel unable to buffer the shock of invention!) In our earlier terminology it tells us something about any non-empty configuration from set $B1$. If we start with a solution (which is an 8-queen configuration from set $B1$) and take away one queen we get a 7-queen configuration from set $B1$; taking away a next queen will leave again a configuration from set $B1$ and we can repeat this process until the chessboard is empty. We could have taken a motion picture of this process: playing it back backwards it would show how, starting from an empty board, via configurations from set $B1$ that solution can be built up by adding one queen at a time. (Whether the trick of the motion picture played backwards is of any assistance for my readers is not for me to judge; I only mention it because I know that such devices help me.) When making the picture, any solution could be reduced to the empty board in many ways, in exactly the same number of ways—while playing it backwards—each solution can be built up. Can we exploit this freedom? We have rejected set $B1$ because it is too large, but maybe we can find a suitable subset of it, such that each non-empty configuration of the subset is a one-queen extension of only one other configuration of the subset. The "extension property" suggests that we are willing to consider configurations with less than 8 queens on the board and that we would like to form new configurations by adding a queen to an existing configuration—a relatively simple operation presumably. Well, this draws our attention immediately to the *generation* of the elements of the (still mysterious) set B. For instance, in what order? And this again raises a question to which, as yet, we have not paid the slightest attention: in what order are we to generate the solutions, i.e. the elements of set A? Can we make a reasonable suggestion in the hope of deriving a clue from it? (In my experience such a question about order is usually very illuminating. It is not only that we have to make a sequential program that by definition will generate the solutions in some order, so that the decision about the order will have to be taken at some stage of the game. The decision about the order usually provides the clue to the proof that the program will generate *all* solutions and each solution only *once*.)

Prior to that we should ask ourselves: how do we characterise solutions once we have them? To characterise a solution we must give the positions of 8 queens. The queens themselves are unordered, but the rows and the columns are not: we may assume them to be numbered from 0 through 7.

Thanks to property (a) which tells us that each row contains precisely one queen, we can order the queens according to the number of the row they occupy. Then each configuration of 8 queens can be given by the value of the **integer array** x [0:7], where

$x[i]$ = the number of the column occupied by the queen in the ith row.

Each solution is then a "8-digit word" ($x[0] \ldots x[7]$) and the only sensible order in which to generate these words that I can think of is the alphabetical order.

Note. As a consequence we open the way to algorithms in which rows and columns are treated differently, while the original problem was symmetrical in rows and columns! To consider asymmetric algorithms is precisely what the above considerations have taught us!

Returning to the alphabetical order: now we are approaching familiar ground. If the elements of set A are to be generated in alphabetical order and they have to be generated by selection from a larger set B, then the standard technique is to generate the elements of set B in alphabetical order as well and to produce the elements of the subset in the order in which they occur in set B.

First we have to generate all solutions with $x[0] = 0$ (if any), then those with $x[0] = 1$ (if any) etc.; of the solutions with $x[0]$ fixed, those with $x[1] = 0$ (if any) have to be generated first, followed by those with $x[1] = 1$ (if any) etc. In other words: the queen of row 0 is placed in column 0—say the square in the bottom left corner—and remains there until all elements of A (and B) with queen 0 in that position have been generated and only then is she moved one square to the right to the next column. For each position of queen 0, queen 1 will walk from left to right in row 1—skipping the squares that are covered by queen 0—for each combined position of the first two queens, queen 2 walks along row 2 from left to right, skipping all squares covered by the preceding queens, etc.

But now we have found set B! It is indeed a subset of $B1$, set B consists of all configurations with one queen in each of the first N rows, such that no two queens can take each other.

The criterion deciding whether an element of B belongs to A as well is that $N = 8$.

Having established our choice for set B, we find ourselves faced with the task of generating its elements in alphabetical order. We could try to do this via an operator "GENERATE NEXT ELEMENT OF B" with a program of the form

```
INITIALISE EMPTY BOARD;
repeat GENERATE NEXT ELEMENT OF B;
  if N = 8 then PRINT CONFIGURATION
until B EXHAUSTED
```

(Here we have used the fact that the empty board belongs to B, but not to A, and is not B's only element. We have made no assumptions about the existence of solutions.)

But for two reasons a program of the above structure is less attractive. Firstly, we don't have a ready-made criterion to recognise the last element of B when we meet it and in all probability we have to generalise the operator "GENERATE NEXT ELEMENT OF B" in such a way that it will produce the indication "B EXHAUSTED" when it is applied to the last "true" element of B. Secondly, it is not too obvious how to make the operator "GENERATE NEXT ELEMENT OF B": the number of queens on the board may remain constant, it may decrease and it may increase.

So that is not too attractive. What can we do about it? As long as we regard the sequence of configurations of set B as a single, monotonous sequence, not subdivided into a succession of subsequences, the corresponding program structure will be a single loop as in the program just sketched. If we are looking for an alternative program structure, we must *therefore* ask ourselves "How can we group the sequence of configurations from set B into a succession of subsequences?".

Realising that the sequence of configurations from set B have to be generated in alphabetical order and thinking about the main subdivision in a dictionary—viz. by first letter—the first grouping is obvious: by position of queen 0.

Generating all elements of set B—for the moment we forget about the printing of those configurations that belong to set A as well—then presents itself as

```
INITIALISE EMPTY BOARD;
h:= 0;
repeat SET QUEEN ON SQUARE[0,h];
       GENERATE ALL CONFIGURATIONS WITH QUEEN 0
         FIXED;
       REMOVE QUEEN FROM SQUARE[0,h];
       h:= h + 1
until h = 8 .
```

But now the question repeats itself: how do we group all configurations with queen 0 fixed? We have already given the answer: in order of increasing column number of queen 1, i.e.

```
h1 := 0;
repeat if SQUARE[1, h1] FREE do
  begin SET QUEEN ON SQUARE[1,h1];
        GENERATE ALL CONFIGURATIONS WITH FIRST
          2 QUEENS FIXED;
        REMOVE QUEEN FROM SQUARE[1,h1]
  end;
  h1 := h1 + 1
until h1 = 8 .
```

For "GENERATE ALL CONFIGURATIONS WITH FIRST 2 QUEENS FIXED" we could write a similar piece of program and so on; inserting them inside each other would result in a correct program with eight nested loops, but they would all be very, very similar. To do so has two disadvantages

(1) it takes a cumbersome amount of writing

(2) it gives a program solving the problem for a chessboard of 8*8 squares, but to solve the same puzzle for a board of, say, 10*10 squares would require a new, still longer program.

We are looking for a way in which all the loops can be executed under control of the same program text. Can we make the text of the loops identical? Can we exploit their identity?

Well, to start with, we observe that the outermost and the innermost loops are exceptional.

The outermost loop is exceptional in the sense that it does not test whether square[0,*h*] is free because we know it is free. But because we know it is free, there is no harm in inserting the conditional clause

```
if SQUARE[0,h] FREE do
```

and this gives the outermost loop the same pattern as the next six loops.

The innermost loop is exceptional in the sense that as soon as 8 queens have been placed on the board, there is no point in generating all configurations with those queens fixed, because we have a full board. Instead the configuration should be printed, because we have found an element of set *B* that is also an element of set *A*. We can map the innermost cycle and the embracing seven upon each other by replacing the line "GENERATE" by

if BOARD FULL **then** PRINT CONFIGURATION
else GENERATE ALL CONFIGURATIONS EXTENDING THE CURRENT ONE

For this purpose we introduce a global variable, "n" say, counting the number of queens currently on the board. The test "BOARD FULL" becomes "$n = 8$" and the operations on squares can then have "n" as first subscript.

By now the only difference between the eight cycles is that each has "its private h". By the time that we have reached this stage, we can give an affirmative answer to the question whether we can exploit the identity of the loops. The sequencing through the eight nested loops can be evoked with the aid of a recursive procedure, "generate" say, which describes the cycle once. Using it, the program itself collapses into

INITIALISE EMPTY BOARD; $n := 0$;
generate

while "generate" is recursively defined as follows:

```
procedure generate;
begin integer h;
  h:= 0;
  repeat if SQUARE[n,h] FREE do
    begin SET QUEEN ON SQUARE[n,h]; n:= n + 1;
      if n = 8 then PRINT CONFIGURATION
        else generate;
      n:= n - 1; REMOVE QUEEN FROM SQUARE[n,h]
    end;
    h:= h + 1
  until h = 8
end
```

Each activation of "generate" will introduce its private local variable h, thus catering for h, $h1$, ... , $h8$ that we would need when writing eight nested loops.

Our program—although correct to this level of detail—is not yet complete, i.e. it has not been refined up to the standard degree of detail that is required by our programming language. In our next refinement we should decide upon the conventions according to which we represent the configurations on the board. We have already decided more or less that we shall use the

integer array $x[0:7]$

giving in order the column numbers occupied by the queens, and also that

integer n

should be used to represent the number of queens on the board. More precisely

n = the number of queens on the board

$x[i]$ for $0 \leqslant i < n$ = the number of the column occupied by the queen in the ith row.

The array x and the scalar n are together sufficient to fix any configuration of the set B and those will be the only ones on the chessboard. As a result we have no *logical* need for more variables; yet we shall introduce a few more, because from a practical point of view we can make good use of them. The problem is that with only the above material the (frequent) analysis whether a given square in the next free row is uncovered is rather painful and time-consuming. It is here that we look for the standard technique as described in the section "On trading storage space for computation speed" (see page 42). The role of the stored argument is here played by the configuration of queens on the board, but this value does not change wildly—oh no, the only thing we do is to add or remove a queen. And we are looking for additional tables (whose contents are a function of the current configuration) such that they will assist us in deciding whether a square is free, and also such that they can be updated easily when a queen is added to or removed from a configuration.

How? Well, we might think of a boolean array of 8*8, indicating for each square whether it is free or not. If we do this for the full board, adding a queen might imply dealing with 28 squares. Removing a queen, however, is then a painful process, because it does not follow that all squares no longer covered by *her* are indeed free: they might be covered by one or more of the other queens that remain in the configuration. There is a remedy (again standard) for this, viz. associating with each square not a boolean variable, but an integer counter, counting the number of queens covering the square. Adding a queen then means increasing up to 28 counters by 1, removing a queen means decreasing them by 1 and a square is free when its associated counter equals zero. We could do it that way, but the question is whether this is not overdoing it: 28 adjustments is indeed quite a heavy overhead on setting or removing a queen.

Each square in the freedom of which we are interested covers a row (which is free by definition, so we need not bother about that), covers one of the 8 columns (which must still be empty), covers one of the 15 upward diagonals (which must still be empty) and one of the 15 downward diagonals (which must still be empty). This suggests that we should keep track of

(1) the columns that are free

(2) the upward diagonals that are free

(3) the downward diagonals that are free.

As each column or diagonal is covered only once we do not need a counter for each, a boolean variable is sufficient. The columns are readily identified by their column number and for the columns we introduce

boolean array col[0:7]

where "col[i]" means that the ith column is still free.

How do we identify the diagonals? Well, along an upward diagonal the difference between row number and column number is constant; along a downward diagonal their sum is constant. As a result, difference and sum respectively are the easiest index by which to distinguish the diagonals and we introduce therefore

boolean array up[-7:$+7$], down[0:14]

to keep track of which diagonals are free.

The question whether square[n,h] is free becomes

col[h] **and** up[$n-h$] **and** down[$n+h$] ,

setting and removing a queen both imply the adjustment of three booleans, one in each array.

In the final program the variable "k" is introduced for general counting purposes, statements and expressions are labeled (in capital letters). Note that we have merged two levels of description: what were statements and functions on the upper level, now appear as explanatory labels.

With the final program we come to the end of the last section. We have attempted to show the pattern of reasoning by which one could discover backtracking as a technique, and also the pattern of reasoning by which one could discover a recursive procedure describing it. The most important moral of this section is perhaps that all that analysis and synthesis could be carried out before we had decided how (and how redundantly) a configuration would be represented inside the machine. It is true that such considerations only bear fruit when eventually a convenient representation for configurations can be found. Yet the mental isolation of a level of abstraction in which we allow ourselves not to bother about it seems crucial.

Finally, I would like to thank the reader that has followed me up till here for his patience.

```
begin integer n, k; integer array x[0:7]; boolean array col[0:7], up[-7:+7], down[0:14];
    procedure generate;
    begin integer h;
        h:= 0;
        repeat if SQUARE[n,h] FREE: (col[h] and up[n-h] and down[n+h]) do
            begin SET QUEEN ON SQUARE[n,h]:
                x[n]:= h; col[h]:= false; up[n-h]:= false; down[n+h]:= false; n:= n + 1;
                if BOARD FULL: (n = 8) then
                begin PRINT CONFIGURATION:
                    k:= 0; repeat print(x[k]); k:= k + 1 until k = 8; newline
                end
                                    else generate;
                n:= n - 1; REMOVE QUEEN FROM SQUARE[n,h]:
                        down[n+h]:= true; up[n-h]:= true; col[h]:= true
            end;
            h:= h + 1
        until h = 8
    end;
    INITIALISE EMPTY BOARD:
      n:= 0;
      k:= 0; repeat col[k]:= true; k:= k + 1 until k = 8;
      k:= 0; repeat up[k-7]:= true; down[k]:= true; k:= k + 1 until k = 15;
    generate
end
```