Debugging Programs in Emacs and Allegro CL

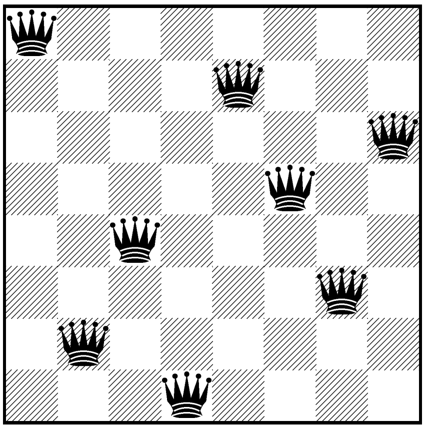
António Menezes Leitão

14 de Abril de 2000

1 The Problem

Let's consider the problem of placing 8 queens on a chess board in such a way that none is under attack.

Here is one example of a possible board configuration:



2 Abstract Data Types

To represent this problem, we will create a few abstract data types. We will need to represent positions on the chess board, sequences of positions to represent a solution and a chess board to register the positions occupied by the placed queens.[1]

2.1 Position

A position is a pair made of a line and a column:

```
(defun make-position (line column)
  (list line colunm))

(defun position-line (position)
  (caar position))

(defun position-column (position)
  (caddr position))
```

[1] The representations we will use are just for presentation purposes and they aren't the most appropriate for a more efficient implementation.

2.2 Positions

Positions are sequences of elements, each being a position:

```
(defun make-positions ()
  (list))

(defun join-position (position positions)
  (cons position positions))
```

2.3 Board

The chess board will contain the occupied positions and will allow us to check whether some position is occupied.

To this end, we will explore a mathematical property about chess boards that says that all positions in a diagonal that goes up-left have the same coordinate difference and all positions in a diagonal that goes up-right have the same coordinate sum.

	0	**1**	**2**	**3**
0	0	-1	-2	-3
1	1	0	-1	-2
2	2	1	0	-1
3	3	2	1	0

(a) Coordinate Difference

	0	**1**	**2**	**3**
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

(b) Coordinate Sum

The relevant operations for this type are the following:

```
(defun make-board ()
  (list (list) (list) (list)))

(defun join-queen (position board)
  (let ((l (position-line position))
        (c (position-column position)))
    (list (cons c (car board))
          (cons (+ l c) (cadr board))
          (cons (- l c) (caddr board)))))

(defun attacked-queen-p (position board)
  (let ((l (position-line position))
        (c (position-column position)))
    (or (member c (car board))
        (member (+ l c) (cadr board))
        (member (- l c) (caddr board)))))
```

2.4 Program

The program implements an backtracking algorithm that attempts to place a queen in a given line, checking column by column until it finds the non-attacked position or until it runs out of columns. In this last case, it returns false (i.e., `nil`), otherwise, it places the queen in the chess board and it tries to solve the problem for the remaining queens (on the next lines). If a solution if found for this subproblem (i.e., the recursive call didn't return false), it joins the found position to the solution, otherwise it tries another column.

```
(defun queens (n)
  (place-queens n n n (make-board)))

(defun place-queens (n i j board)
  (cond ((= i 0)
         (make-positions))
        ((= j 0)
         nil)
        ((attacked-queen (make-position i j) board)
         (place-queens n i (1- j) board))
        (t
         (let ((result
                (place-queens n
                              (1- i)
                              n
                              (join-queen (make-position i j)
                                          board))))
           (if result
             (join-position (make-position i j) result)
             (place-queens n i (1- j) board))))))
```

3 Tests

After we compile the above definitions we are in position to test the code. To this end, we move to the *Lisp Listener* and we write:

```
USER(1): (queens 4)
Error: Attempt to take the value of the unbound variable ‘COLUNM’.
  [condition type: UNBOUND-VARIABLE]

Restart actions (select using :continue):
 0: Try evaluating COLUNM again.
 1: Set the symbol-value of COLUNM and use its value.
 2: Use a value without setting COLUNM.
[1] USER(2):
```

Unfortunately, we have an error in our program. As we don't know where is it, we will inspect the *stack* with the command `CTRL-c s`. Here is the result:

```
   (ERROR #<UNBOUND-VARIABLE @ #x204d4f72>)
 ->(SYS::..CONTEXT-SAVING-RUNTIME-OPERATION)
   (MAKE-POSITION 4 4)
   (PLACE-QUEENS 4 4 ...)
   (QUEENS 4)
   [... EXCL::%EVAL ]
   (EVAL (QUEENS 4))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
```

The arrow shows the place where our debugging commands will have effect. This place is designated the current frame. As the error seems to be in the `make-position` function, we change the cursor to the line bellow, we select that frame with the command `.` and we edit the source code with the command `e`. The debugging environment shows us the function:

```
(defun make-position (line column)
  (list line colunm))
```

The error is obviously a typo (`colunm` should have been `column`). We correct:

```
(defun make-position (line column)
  (list line column))
```

and we recompile the function with `META-CTRL-x` and we return to the *debugger* where we will try again to execute the function that caused the error by using the command `R` (that means `restart`) in the frame `-> (MAKE-POSITION 4 4)`. We are then taken again to the *listener* where we see:

```
[1] USER(2):
Error: attempt to call `ATTACKED-QUEEN' which is an undefined function.
  [condition type: UNDEFINED-FUNCTION]

Restart actions (select using :continue):
 0: Try calling ATTACKED-QUEEN again.
 1: Return a value instead of calling ATTACKED-QUEEN.
 2: Try calling a function other than ATTACKED-QUEEN.
 3: Setf the symbol-function of ATTACKED-QUEEN and call it again.
[1] USER(3):
```

Again, we move to the *debugger* (`CTRL-c s`):

```
   (ERROR #<UNDEFINED-FUNCTION @ #x205069b2>)
 ->(ATTACKED-QUEEN (4 4) NIL)
   (PLACE-QUEENS 4 4 ...)
   (QUEENS 4)
   [... EXCL::%EVAL ]
   (EVAL (QUEENS 4))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
```

The error must be in the caller function so we move the current frame to the line bellow and we edit the code (as before, `.` to select the frame and `e` to edit the code).

```
(defun place-queens (n i j board)
  (cond ((= i 0)
         (make-positions))
        ((= j 0)
         nil)
        ((attacked-queen (make-position i j) board)
         (place-queens n i (1- j) board))
        (t
         ...)))
```

The error is that the correct function to call is called `attacked-queen-p` and not `attacked-queen`. We correct:

```
(defun place-queens (n i j board)
  (cond ((= i 0)
         (make-positions))
        ((= j 0)
         nil)
        ((attacked-queen-p (make-position i j) board)
         (place-queens n i (1- j) board))
        (t
         ...)))
```

Now, `META-CTRL-x` to compile, `CTRL-x o` to move to the *debugger* window and we re-execute the line `-> (PLACE-QUEENS 4 4 ...)` using the command `R` (`restart`). We then return to the *listener*, where another error pops up:

```
[1] USER(3):
Error: Attempt to take the car of 4 which is not listp.
  [condition type: SIMPLE-ERROR]
[1] USER(4):
```

Moving to the *debugger*, we see:

```
   (ERROR SIMPLE-ERROR :FORMAT-CONTROL ...)
 ->(SYS::..CONTEXT-SAVING-RUNTIME-OPERATION)
   (POSITION-LINE (4 4))
   (ATTACKED-QUEEN-P (4 4) NIL)
   (PLACE-QUEENS 4 4 ...)
   (QUEENS 4)
   [... EXCL::%EVAL ]
   (EVAL (QUEENS 4))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
```

Editing the function `position-line` we see that the error is an extra `car`:

```
(defun position-line (position)
  (caar position))
```

The line is the first element of the position list so we should obtain it just with a `car`. The correction is trivial:

```
(defun position-line (position)
  (car position))
```

After we compile the function (with `META-CTRL-x`), we return to the *debugger* and, since we know what the function should return in this case, we use the `r` command (that means `return`) with the current frame on line `-> (POSITION-LINE (4 4))` to return the correct value and continue. The environment asks about the value to return and we write the value `4`:

```
Form (evaluated in the Lisp environment): 4
```

and we obtain ... another error:

```
[1] USER(4):
Error: 'NIL' is not of the expected type 'NUMBER'
  [condition type: TYPE-ERROR]
[1] USER(5):
```

Again, let's move to the *debugger*:

```
   (ERROR TYPE-ERROR :DATUM ...)
 ->(ATTACKED-QUEEN-P (4 4) NIL)
   (PLACE-QUEENS 4 4 ...)
   (QUEENS 4)
   [... EXCL::%EVAL ]
   (EVAL (QUEENS 4))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
```

This time, the message is not sufficiently clear and it is convenient to get some more information. One simple way to achieve that is to call the function in its *interpreted* form. To this end, we edit the function (with the command `e`):

```
(defun attacked-queen-p (position board)
  (let ((l (position-line position))
        (c (position-column position)))
    (or (member c (car board))
        (member (+ l c) (cadr board))
        (member (- l c) (caddr board)))))
```

and we execute `CTRL-u META-CTRL-x`. This command changes the function so that, when invoked, it executes under the interpreter.

We now execute a `restart` and the error arises again (as expected, obviously), but now the *debugger* shows some extra information:

```
   (ERROR TYPE-ERROR :DATUM ...)
 ->(+ 4 NIL)
   (OR (MEMBER C #) (MEMBER # #) ...)
   [... EXCL::EVAL-AS-PROGN ]
   (LET (# #) (OR # # ...))
   (ATTACKED-QUEEN-P (4 4) NIL)
   (PLACE-QUEENS 4 4 ...)
   (QUEENS 4)
   [... EXCL::%EVAL ]
   (EVAL (QUEENS 4))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
```

We immediately see a problem on the second argument to the sum. We will inspect the lexical environment (with the command `l` that means `lexicals`), and the environments shows us:

```
Interpreted lexical environment:
L: 4
C: NIL
BOARD: NIL
POSITION: (4 4)
Compiled lexical environment:
0(REST): EXCL::ARGS: (4 NIL)
1(LOCAL): L: 4
2(LOCAL): C: NIL
3(LOCAL): BOARD: NIL
4(LOCAL): POSITION: (4 4)
5(LOCAL): :UNKNOWN: (4 NIL)
6(LOCAL): :UNKNOWN: 4
7(LOCAL): :UNKNOWN: (NIL)
8(LOCAL): :UNKNOWN: NIL
9(UNKNOWN): :UNKNOWN: NIL
```

We see that the variable that refers to the column of the position has the value `nil`, meaning that there is a problem in the corresponding selector. We will check its behavior using the `trace` operation. To this end, we move temporarily to the *listener* (that still shows that we are under a debugging session):

```
[1] USER(5): (trace position-column)
(POSITION-COLUMN)
[1] USER(6):
```

We return to the *debugger* and, to be sure that the selector will be called again, we execute a `restart` some frames bellow, on the line that contains `-> (ATTACKED-QUEEN-P (4 4) NIL)` and we get the error again (as expected), but before that the function `position-column` shows its invocation:

```
[1] USER(6):
 0: (POSITION-COLUMN (4 4))
 0: returned NIL
Error: ‘NIL’ is not of the expected type ‘NUMBER’
  [condition type: TYPE-ERROR]
[1] USER(7):
```

It is clear that the error is in the `position-column` selector. We edit the this function (with `CTRL-c .` over the name of the function):

```
(defun position-column (position)
  (caddr position))
```

To obtain the second element of the list that represents a position the correct operation to use is `cadr` and not `caddr`:

```
(defun position-column (position)
  (cadr position))
```

We compile this function (and, by the way, the `attacked-queen-p` to make it run compiled and not interpreted) and we return to the *stack* a few frames below and we execute another `restart` (with the command `R`) on the line containing `-> (ATTACKED-QUEEN-P (4 4) NIL)`. We then return to the *listener* where we get the answer:

```
[1] USER(7):
NIL
USER(8):
```

Unfortunately, the answer is wrong. In fact, there are several solutions to the 4 by 4 board. To understand the problem, let’s visualize the execution of the `place-queens` function. To this end, we should ask for an interpreted execution (issuing the `CTRL-u META-CTRL-x` command in the function definition) because in a compiled function the recursive calls might not show up in the trace. Then, we do `META-T` on the function name to `trace` it. The re-evaluation of the original expression now produces:

```
USER(8): (queens 4)
 0: (PLACE-QUEENS 4 4 4 NIL)
   1: (PLACE-QUEENS 4 3 4 ((4) (8) (0)))
     2: (PLACE-QUEENS 4 3 3 ((4) (8) (0)))
       3: (PLACE-QUEENS 4 3 2 ((4) (8) (0)))
         4: (PLACE-QUEENS 4 2 4 ((2 4) (5 8) (1 0)))
           5: (PLACE-QUEENS 4 2 3 ((2 4) (5 8) (1 0)))
             6: (PLACE-QUEENS 4 2 2 ((2 4) (5 8) (1 0)))
               7: (PLACE-QUEENS 4 2 1 ((2 4) (5 8) (1 0)))
                 8: (PLACE-QUEENS 4 2 0 ((2 4) (5 8) (1 0)))
                 8: returned NIL
               7: returned NIL
             6: returned NIL
           5: returned NIL
         4: returned NIL
         4: (PLACE-QUEENS 4 3 1 ((4) (8) (0)))
           5: (PLACE-QUEENS 4 2 4 ((1 4) (4 8) (2 0)))
             6: (PLACE-QUEENS 4 2 3 ((1 4) (4 8) (2 0)))
               7: (PLACE-QUEENS 4 1 4 ((3 1 4) (5 4 8) (-1 2 0)))
...
USER(9):
```

That's too much information. Let's use a smaller example:

```
USER(9): (queens 1)
 0: (PLACE-QUEENS 1 1 1 NIL)
   1: (PLACE-QUEENS 1 0 1 ((1) (2) (0)))
   1: returned NIL
   1: (PLACE-QUEENS 1 1 0 NIL)
   1: returned NIL
 0: returned NIL
NIL
USER(10):
```

In fact, there's something wrong. The board 1 × 1 has an obvious solution that it is not found. To analyze the problem, we will execute the function step-by-step. First, we remove the trace (again, using `META-T` on the function name) because the step-by-step shows more or less the same information. Then, we ask the environment to interpret also the functions that call `place-queens`, otherwise we will not see the first call. In our case, we just have to do a `CTRL-u META-CTRL-x` on function `queens`. Finally, we write on the *listener*:

```
USER(10): :step place-queens
USER(11): (queens 1)
 1: (PLACE-QUEENS N N N (MAKE-BOARD))
[STEP] USER(11):
  2: N => 1
  2: N => 1
  2: N => 1
  2: (MAKE-BOARD)
[STEP] USER(11):
```

Each step shows what was evaluated and what will be evaluated. The simplest command is simply to the key `RET` that asks for another step:

```
[STEP] USER(11):
  result 2: NIL
  2: (BLOCK PLACE-QUEENS
       (COND ((= I 0) (MAKE-POSITIONS))
             ((= J 0) NIL)
             ((ATTACKED-QUEEN-P # BOARD)
              (PLACE-QUEENS N I # BOARD))
             (T (LET # #))))
[STEP] USER(11):
   3: (COND ((= I 0) (MAKE-POSITIONS))
            ((= J 0) NIL)
            ((ATTACKED-QUEEN-P (MAKE-POSITION I J) BOARD)
             (PLACE-QUEENS N I (1- J) BOARD))
            (T (LET (#) (IF RESULT # #))))
[STEP] USER(11):
    4: (= I 0)
[STEP] USER(11):
     5: I => 1
     5: 0 => 0
    result 4: NIL
    4: (= J 0)
[STEP] USER(11):
     5: J => 1
     5: 0 => 0
    result 4: NIL
    4: (ATTACKED-QUEEN-P (MAKE-POSITION I J) BOARD)
[STEP] USER(11):
```

As we know that the function `attacked-queen-p` should be correct, we can evaluate its call in just one step, by issuing the command `:sover` that abbreviates to `:so`. The follow up produces:

```
[STEP] USER(11): :so
    result 4: NIL
    4: T => T
    4: (LET ((RESULT (PLACE-QUEENS N # N #)))
         (IF RESULT
           (CONS (MAKE-POSITION I J) RESULT)
           (PLACE-QUEENS N I (1- J) BOARD)))
[STEP] USER(12):
     5: (PLACE-QUEENS N (1- I) N
                          (JOIN-QUEEN (MAKE-POSITION I J) BOARD))
[STEP] USER(12):
      6: N => 1
      6: (1- I)
[STEP] USER(12): :so
      result 6: 0
      6: N => 1
      6: (JOIN-QUEEN (MAKE-POSITION I J) BOARD)
[STEP] USER(13): :so
      result 6: ((1) (2) (0))
      6: (BLOCK PLACE-QUEENS
           (COND ((= I 0) (MAKE-POSITIONS))
                 ((= J 0) NIL)
                 ((ATTACKED-QUEEN-P # BOARD)
                  (PLACE-QUEENS N I # BOARD))
                 (T (LET # #))))
```

```
[STEP] USER(14):
       7: (COND ((= I 0) (MAKE-POSITIONS))
                ((= J 0) NIL)
                ((ATTACKED-QUEEN-P (MAKE-POSITION I J) BOARD)
                 (PLACE-QUEENS N I (1- J) BOARD))
                (T (LET (#) (IF RESULT # #))))
[STEP] USER(14):
        8: (= I 0)
[STEP] USER(14):
         9: I => 0
         9: 0 => 0
        result 8: T
        8: (MAKE-POSITIONS)
```

This is the point where the function returns the solution for the trivial case. The next step is:

```
[STEP] USER(14):
        result 8: NIL
       result 7: NIL
      result 6: NIL
     result 5: NIL
     5: (IF RESULT
          (CONS (MAKE-POSITION I J) RESULT)
          (PLACE-QUEENS N I (1- J) BOARD))
[STEP] USER(14):
```

And here is the problem: when the function finds the trivial case, the returned solution by `make-positions` is an empty list (`()`) that, in Common Lisp, is the same as the false value (`nil`).

Let's edit the function `make-positions` (with `CTRL-c .`) and let's return something different from `nil`:

```
(defun make-positions ()
  (list 'end))
```

We can now recompile the function and try again:

```
USER(15): (queens 1)
((1 1) END)
```

It looks better now. Let's get to our business:

```
USER(16): (queens 8)
((8 8) (7 4) (6 1) (5 3) (4 6) (3 2) (2 7) (1 5) END)
```