The 8-queens problem

CS 5010 Program Design Paradigms
"Bootcamp"

Lesson 8.7

Introduction

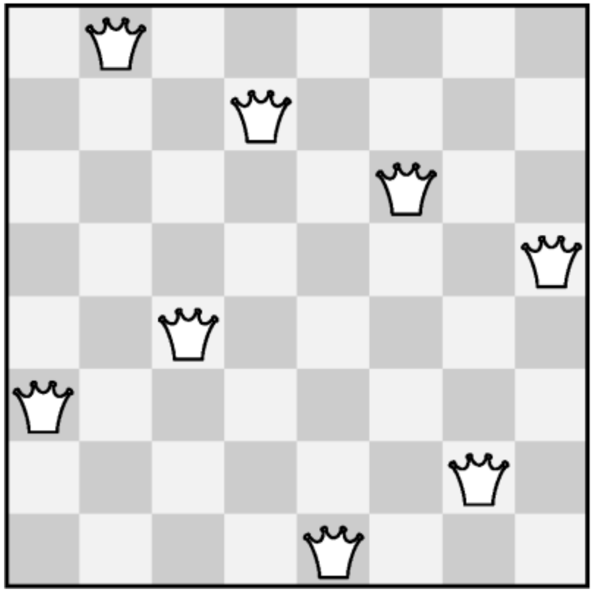
- In this lesson, a classic example of general recursion: the eight queens problem.
- Along the way we'll learn something more about *layered design*.

Layered Design

- In layered design, we write a data design and a set of procedures for each data type.
- We try to manipulate the values of the type only through the procedures.
- We already did this once– we hooked things up so that our graph programs (**reachables** and **path?**) didn't care how the graphs were represented, so long as we had a **successor** function that gave right answers.
- In general, we start with the lowest-level pieces and work our way up.

The problem for this lesson: 8-queens

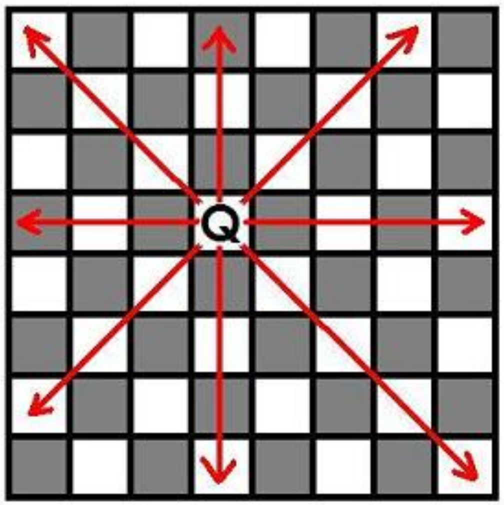
- Find a placement of 8 queens on a chessboard so that no queen can capture another queen.
- Here's one solution:



source

What can a queen capture?

- A queen can move any number of spaces horizontally, vertically, or diagonally



© 2009 Bigriddles

What can a queen capture?

- If the queen is at row r and column c, then it can attack any square (r', c') such that
- r' = r (horizontal movement)
- c' = c (vertical movement)
- r'+c' = r+c (northwest-southeast movement)
- r'-c' = r-c (northeast-southwest movement)

Of course, we'll generalize to boards of other sizes

- and our data representation should be independent of board size.
- If we need information about the board size, we'll put that in an invariant.

Data Design for Queen

```
;; Queens:
(define-struct queen (row col))
;; A Queen is a (make-queen PosInt PosInt)

;; Queen Queen -> Boolean
;; STRATEGY: Use template for Queen on q1 and q2
(define (threatens? q1 q2)
  (or
    (= (queen-row q1) (queen-row q2))
    (= (queen-col q1) (queen-col q2))
    (=
      (+ (queen-row q1) (queen-col q1))
      (+ (queen-row q2) (queen-col q2)))
    (=
      (- (queen-row q1) (queen-col q1))
      (- (queen-row q2) (queen-col q2)))))

;; Queen ListOfQueen -> Boolean
;; STRATEGY: Use HOF ormap on other-queens
(define (threatens-any? this-queen other-queens)
  (ormap
    (lambda (other-queen) (threatens? this-queen other-queen))
    other-queens))
```

Data Design

- Define a legal configuration to be a set of queens on squares that can't attack each other.
- Since no two queens can occupy the same row, we'll only represent legal configurations of the form

  {(1,c1), ..., (k, c_k)}

  for some k.
- We'll represent them as a list in reverse order:

  ((k c_k) (k-1, c_k-1) ... (1, c1))

Operations on configurations

```
;; : -> LegalConfig
(define empty-config empty)

;; legal-to-add-queen? : PosInt LegalConfig -> Bool
;; GIVEN: a column col and a legal configuration
;;   ((k, c_k), (k-1, c_k-1), ... (1, c1))
;; RETURNS: true iff adding a queen at row k+1 and column col
;; would result in a legal configuration.
;; STRATEGY: Cases on whether the configuration is empty.
(define (legal-to-add-queen? col config)
  (or
    (empty? config) ;; first queen is always legal
    (local
      ((define next-row (+ 1 (length config)))
       (define new-queen (make-queen next-row col)))
      (not (threatens-any? new-queen config)))))
```

None of the old queens threaten each other, so we only need to check whether the new queen threatens any of the old queens.

Operations on Configurations (2)

```
;; place-queen : PosInt LegalConfig -> LegalConfig
;; GIVEN: a column col
;;        and a legal config of some length k
;; WHERE: a new queen at (k+1, col)  wouldn’t threaten
;; any of the existing queens.
;; RETURNS: the given configuration with a new queen
;; added at (k+1,col)
;; STRATEGY: Cases on whether config is empty
(define (place-queen col config)
  (if (empty? config)
      (list (make-queen 1 1))
      (local
        ((define next-row (+ 1 (length config)))
         (define new-queen (make-queen next-row col)))
        (cons new-queen config))))
```

It turns out to be useful to separate out legal-to-add-queen? as a separate function.

Operations on configurations (3)

```
;; Config PosInt -> Boolean
;; RETURNS: Is the configuration complete for a board of
;;  size n?
;; STRATEGY: combine simpler functions

(define (config-complete? config size)
  (= size (length config)))
```

The General Problem

```
;; complete-configuration :
;;    LegalConfig PosInt-> MaybeLegalConfig
;; GIVEN: a legal configuration and the size of the board
;; RETURNS: an extension of the given configuration to the given
;; size, if there is one, otherwise false.
;; STRATEGY: Recur on  each legal placement of next queen.
;; DETAILS: Given ((k, c_k), (k-1, c_k-1), ... (1, c1)), we
;;  generate all the configurations
;;  ((k+1, c_k+1), (k, c_k), (k-1, c_k-1), ... (1, c1))
;;  and recur on each of them until we find one that works.
;; HALTING MEASURE: (- size (length config))
```

In other words, ...

Algorithm

- If **config** is already complete, it is its own completion: the problem is trivial.
- Otherwise, look at each of the successors of **c** in turn, and choose the first completion.

Top Level

```
;; Nat -> MaybeLegalConfig
;; STRATEGY: Call a more general function
(define (nqueens n)
  (complete-configuration empty-config n))
```

Function Definition

```
;; HALTING MEASURE: (- size (length config))
(define (complete-configuration config size)
  (cond
    [(= (length config) size) config]
    [else
      (first-success
        (lambda (next-config)
          (complete-configuration next-config size))
        (legal-successors config size))]))
```

legal-successors

```
;; LegalConfig Nat -> ListOfLegalConfig
;; GIVEN a legal configuration
;;   ((k, c_k), (k-1, c_k-1), ... (1, c1))
;; RETURNS: the list of all legal configurations
;;   ((k+1, col), (k, c_k), (k-1, c_k-1), ... (1, c1))
;; for col in [1,size]
;; STRATEGY: Use HOF filter on [1,n] to find all places on
;;  which it is legal to place next queen.  Use map on the
;;  result to construct each such configuration.

(define (legal-successors config size)
  (map
    (lambda (col) (place-queen col config))
    (filter
      (lambda (col) (legal-to-add-queen? col config))
      (integers-from 1 ncols))))
```

Help Functions

```
;; integers-from : Integer Integer -> ListOfInteger
;; GIVEN: n, m
;; RETURNS: the list of integers in [n,m]
;; STRATEGY: recur on n+1;  halt when n > m.
;; HALTING MEASURE: max(0,m-n).

(define (integers-from n m)
  (cond
    [(> n m) empty]
    [else (cons n (integers-from (+ n 1) m))]))

;; (X -> MaybeY) ListOfX -> MaybeY
;; first elt of lst s.t. (f elt) is not false; else false
;; STRATEGY: Use template for ListOfX on lst

(define (first-success f lst)
  (cond
    [(empty? lst) false]
    [else
     (local ((define y (f (first lst))))
       (if (not (false? y))
           y
           (first-success f (rest lst))))]))
```

first-success is like **ormap**, but in ISL **ormap** requires **f** to be **(X -> Bool)**, not **(X -> MaybeY)**. In full Racket, we could just use **ormap**.

Output

```
> (nqueens 1)
(list (make-queen 1 1))
> (nqueens 2)
#false
> (nqueens 3)
#false
> (nqueens 4)
#false
> (nqueens 5)
(list
 (make-queen 5 4)
 (make-queen 4 2)
 (make-queen 3 5)
 (make-queen 2 3)
 (make-queen 1 1))
> (nqueens 6)
#false
> (nqueens 7)
(list
 (make-queen 7 6)
 (make-queen 6 4)
 (make-queen 5 2)
 (make-queen 4 7)
 (make-queen 3 5)
 (make-queen 2 3)
 (make-queen 1 1))
>
```

```
> (nqueens 8)
(list
 (make-queen 8 4)
 (make-queen 7 2)
 (make-queen 6 7)
 (make-queen 5 3)
 (make-queen 4 6)
 (make-queen 3 8)
 (make-queen 2 5)
 (make-queen 1 1))
> (nqueens 9)
(list
 (make-queen 9 5)
 (make-queen 8 7)
 (make-queen 7 9)
 (make-queen 6 4)
 (make-queen 5 2)
 (make-queen 4 8)
 (make-queen 3 6)
 (make-queen 2 3)
 (make-queen 1 1))
```

```
> (nqueens 10)
(list
 (make-queen 10 7)
 (make-queen 9 4)
 (make-queen 8 2)
 (make-queen 7 9)
 (make-queen 6 5)
 (make-queen 5 10)
 (make-queen 4 8)
 (make-queen 3 6)
 (make-queen 2 3)
 (make-queen 1 1))
> (nqueens 11)
(list
 (make-queen 11 10)
 (make-queen 10 8)
 (make-queen 9 6)
 (make-queen 8 4)
 (make-queen 7 2)
 (make-queen 6 11)
 (make-queen 5 9)
 (make-queen 4 7)
 (make-queen 3 5)
 (make-queen 2 3)
 (make-queen 1 1))
```

```
> (nqueens 12)
(list
 (make-queen 12 4)
 (make-queen 11 9)
 (make-queen 10 7)
 (make-queen 9 2)
 (make-queen 8 11)
 (make-queen 7 6)
 (make-queen 6 12)
 (make-queen 5 10)
 (make-queen 4 8)
 (make-queen 3 5)
 (make-queen 2 3)
 (make-queen 1 1))
```

You should check by hand to see that there are no solutions for n = 2,3,4, and 6.

Layered Design

These were the only operations used by the configuration functions

- We designed our system in 3 layers:
  1. Queens. The operations were **make-queen**, **queen-row**, and **threatens?**
  2. Configurations. The operations were **empty-config**, **config-complete?**, **legal-to-add-queen?**, and **place-queen.**
  3. Search. This was the main function **complete-configuration** and its helper **legal-successors**.

These were the only operations on configurations used by layer 3.

Information-Hiding

- At each level, we could have referred to the implementation details of the lower layers, but we didn't need to.
- We only needed to refer to the procedures that manipulated the values in the lower layers.
- So when we code the higher layers, we don't need to worry about the details of the lower layers.

Information-Hiding (2)

- We could have written 3 files: queens.rkt, configs.rkt, and search.rkt, with each file **provide**-ing just those few procedures.
- In larger systems this is a must. It is the major topic of Managing System Design (aka Bootcamp 2)

Information-Hiding (3)

- These procedures form an *interface* to the values in question.
- If you continue along this line of analysis, you will be led to objects and classes (next week's topic!).

Information-Hiding (4)

- You use information-hiding every day.
- Example: do you know how Racket *really* represents numbers? Do you care? Ans: No, so long as the arithmetic functions give the right answer.
- Similarly for file system, etc: so long as **fopen**, **fclose**, etc. do the right thing, you don't care how files are actually implemented.

Except for performance, of course.

Summary

- In this lesson, we wrote a solution to the n-queens problem.
  - we used generative recursion
  - with a list of subproblems.
- We constructed our solution in layers
  - At each layer, we got to forget about the details of the layers below
  - This enables us to control complexity: to solve our problem while juggling less stuff in our brains.

Next Steps

- Study the file 08-9-queens.rkt in the Examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 8.5
- Go on to the next lesson