UMass Boston Computer Science

CS450 High Level Languages

Backtracking Design Recipe

Thursday, May 8, 2025

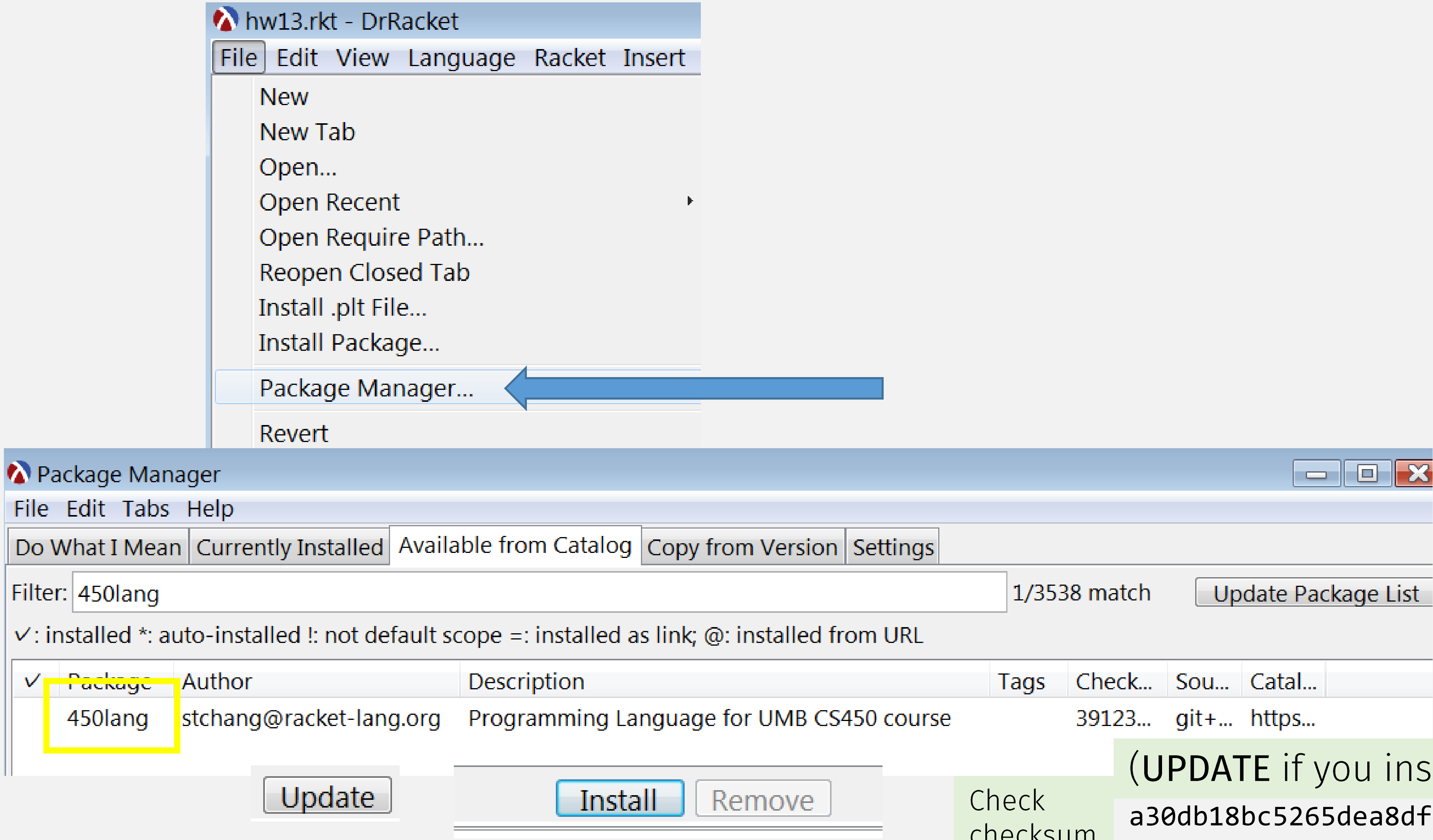
5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Logistics

- HW 13 out
  - Due: **Tues 5/13 11am EST**
  - Last hw!
  - Must use #lang 450lang

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Installing “450 Lang”



Filter: 450lang   1/3538 match

✓: installed *: auto-installed !: not default scope =: installed as link; @: installed from URL

✓	Package	Author	Description	Tags	Check...	Sou...	Catal...
	450lang	stchang@racket-lang.org	Programming Language for UMB CS450 course		39123...	git+...	https...

(**UPDATE** if you installed last week)

Check checksum

`a30db18bc5265dea8df619959c67869d4b333f`

`https://pkgs.racket-lang.org/package/450lang`

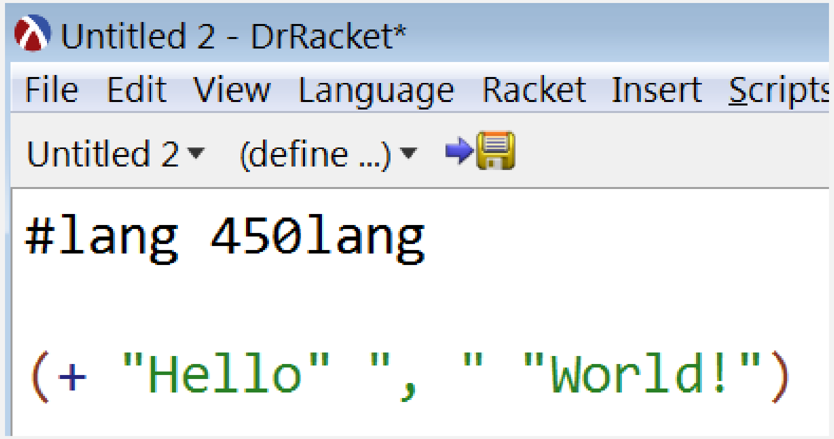
Using “450 Lang”

Read the **Programming Language Specification** linked from HW description!

<u>Added features:</u>

- Lists
- More arith fns: `-`, `abs`
- Logical operations: ¬, ∧, ∨
- “top-level” `bind/rec` Like `define`
- `rackunit` equivalents

Not as “good” as Racket



```
#lang 450lang

(+ "Hello" ", " "World!")
```

“quotes” are implicitly inserted by the language

Taking requests …

Ask for additional primitives in `INIT-ENV`

Design Recipe <u>even more important</u> now

DO NOT “save” writing tests until the end!!

(you’ve been warned)

Generative (non-structural) Recursion Design Recipe

1. Name, Signature
2. Description
   - Must include **Termination Argument**
3. Examples
   - Even more important now!
4. Code (No structural template, but can use a “general” template)

5. Tests

Previously

Generative (non-structural) Recursion Design Recipe

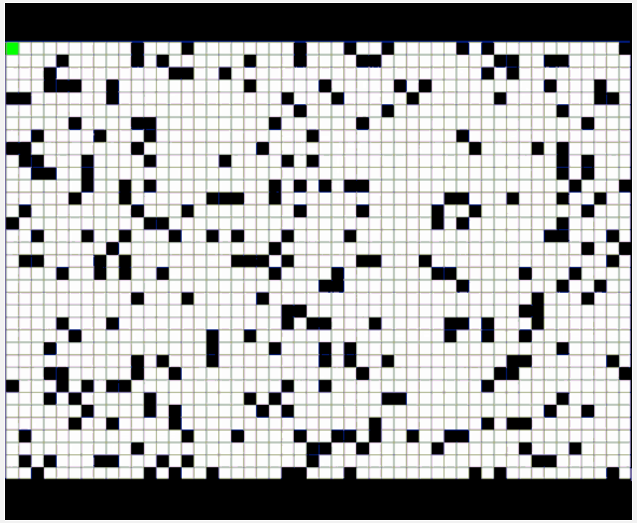
1. Name, Signature
2. Description
   - Must include **Termination Argument**
3. Examples
   - Even more important now!
4. Code (No structural template, but can use a "general" template)
   a) **Break problems into <u>smaller</u> problems to** (recursively) **solve**
   b) **Determine how to combine smaller solutions**
   c) **"trivially solvable" problem is base case!**
5. Tests

Previously

Previously

Backtracking

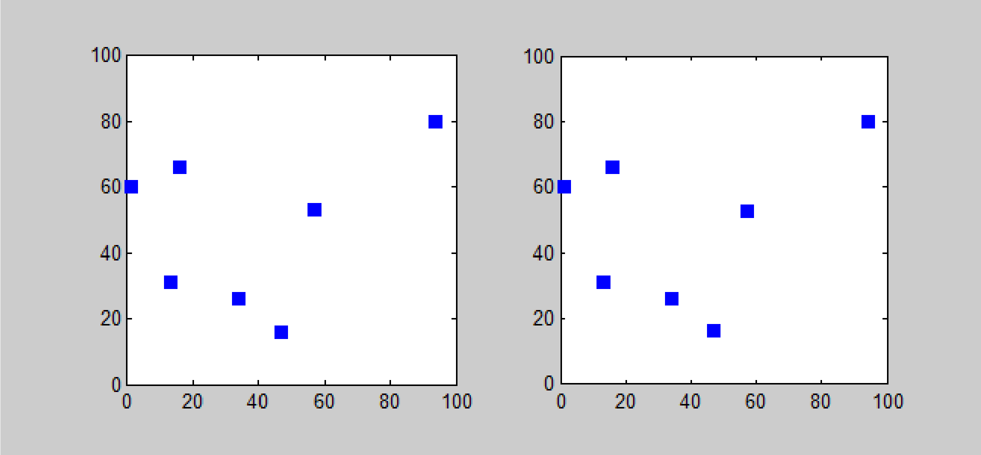
- A **recursive algorithm** for **finding solutions to many computational problems that …**
  - **… tries potential solutions optimistically …** but **“backtracks” when stuck**
  - **Graph algorithms,** e.g., **Path finding**



Previously

Backtracking

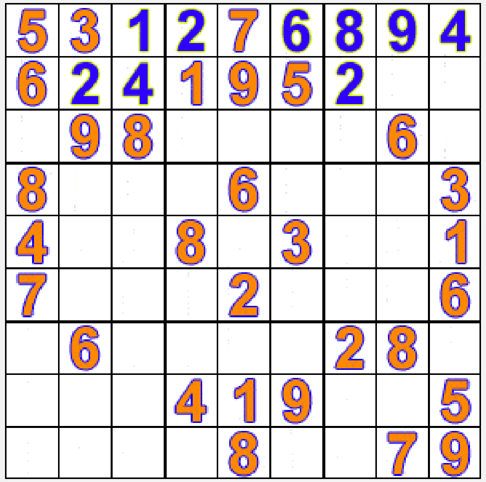
- A **recursive algorithm** for **finding solutions to many computational problems that …**
  - **… tries potential solutions optimistically …** but **“backtracks” when stuck**
  - **Graph algorithms,** e.g., **Path finding**
  - **Optimization,** e.g., **knapsack, “traveling salesman”**



Previously

Backtracking

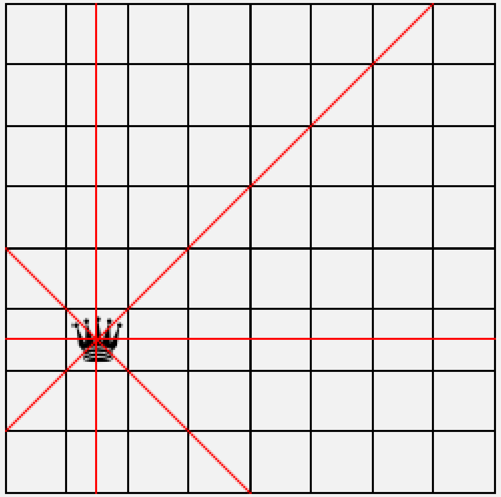
- A **recursive algorithm** for **finding solutions to many computational problems that …**
  - **… tries potential solutions optimistically …** but **“backtracks” when stuck**
  - **Graph algorithms,** e.g., **Path finding**
  - **Optimization,** e.g., **knapsack, “traveling salesman”**
  - **Solving puzzles,** e.g., **Sudoku, n-queens**



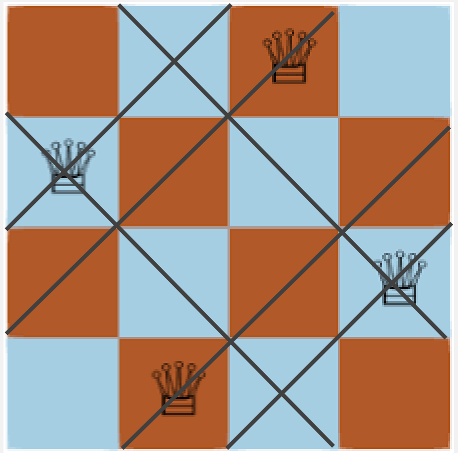
5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

N-Queens problem

- **Place** n **queens** on an n **x** n **chess board** so that **no queen “threatens” another …**



All the positions “threatened” by a queen



All queens safe

N-Queens problem – solving …

- **Place n queens** on an **n x n chess board** so that no queen “threatens” another …

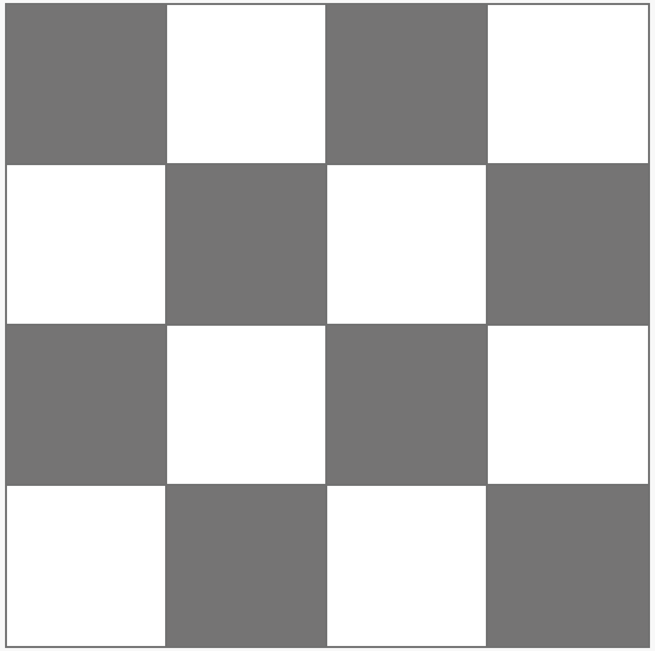
- To find a solution …

- … optimistically “place” each queen in non-threatening position on board …

- … and hope it works out ???

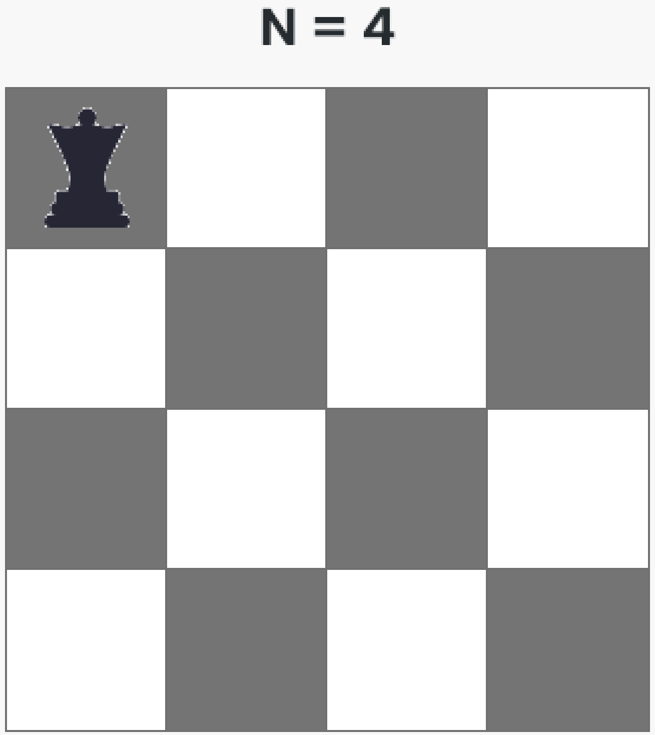
Example: 4-queens

N = 4



4 x 4 Chess Board

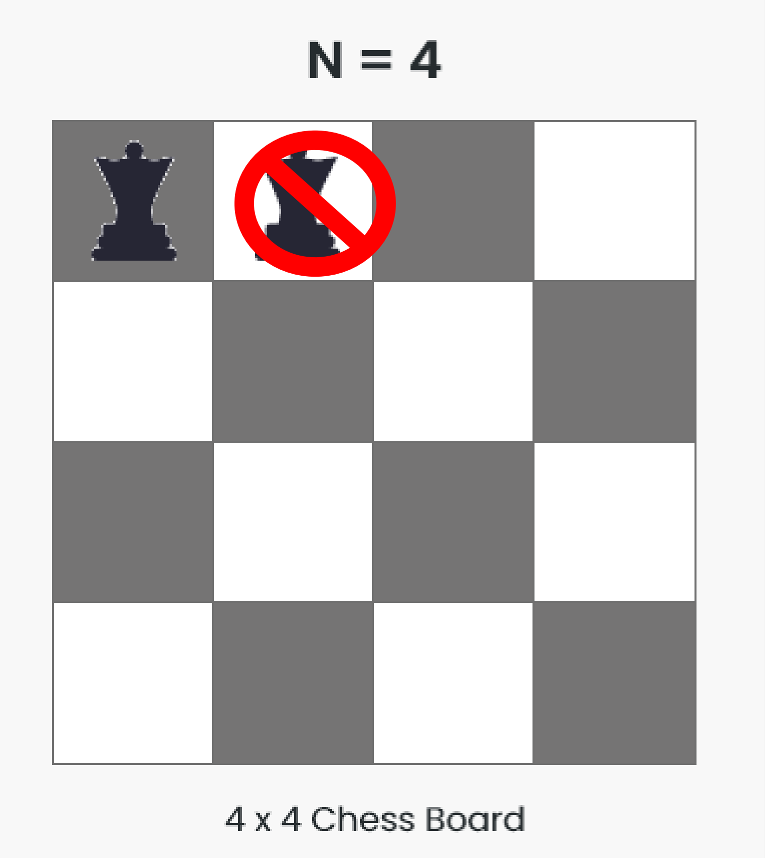
Example: 4-queens



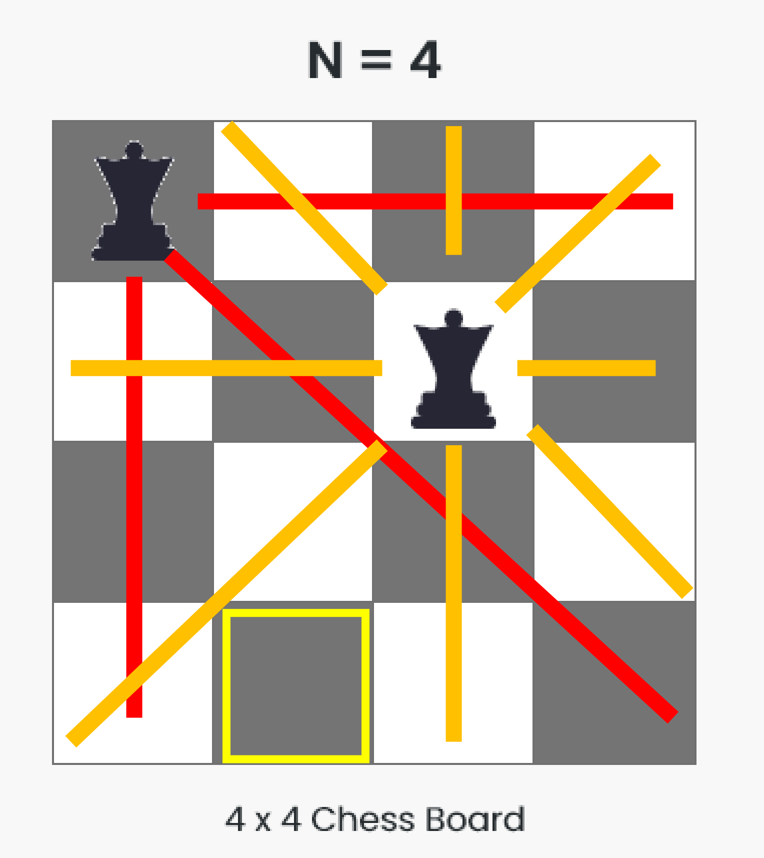
N = 4

4 x 4 Chess Board

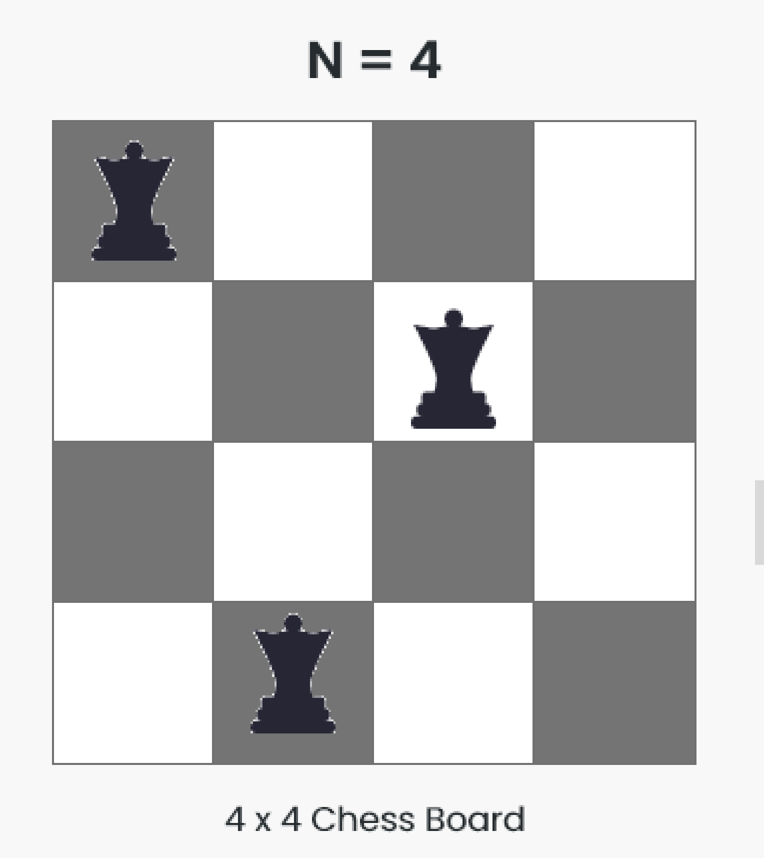
Example: 4-queens



Example: 4-queens



Example: 4-queens



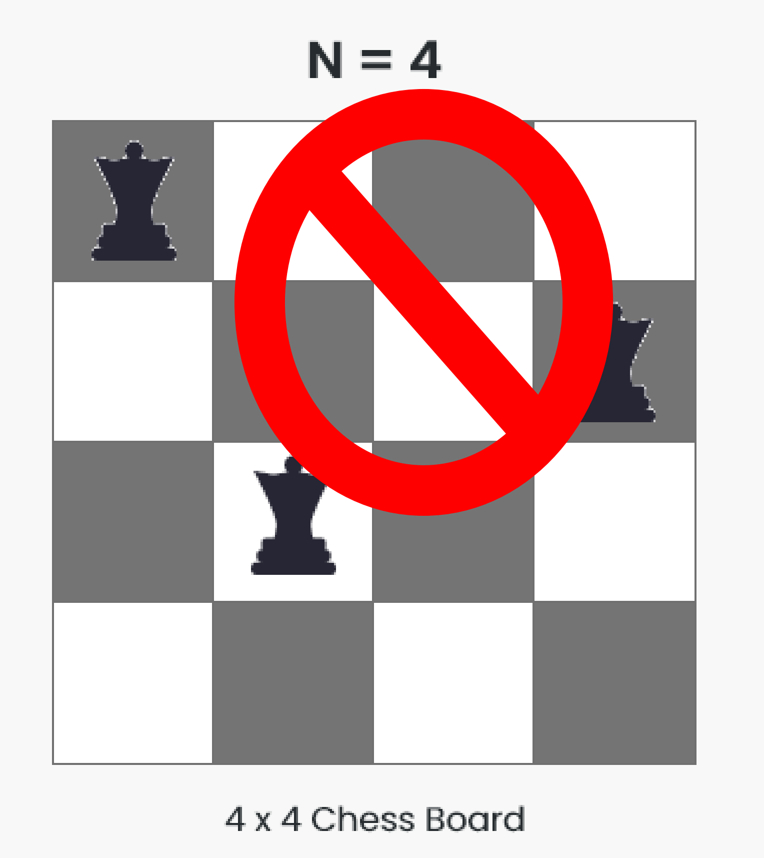
But ... need to place 4 queens!

FAIL???

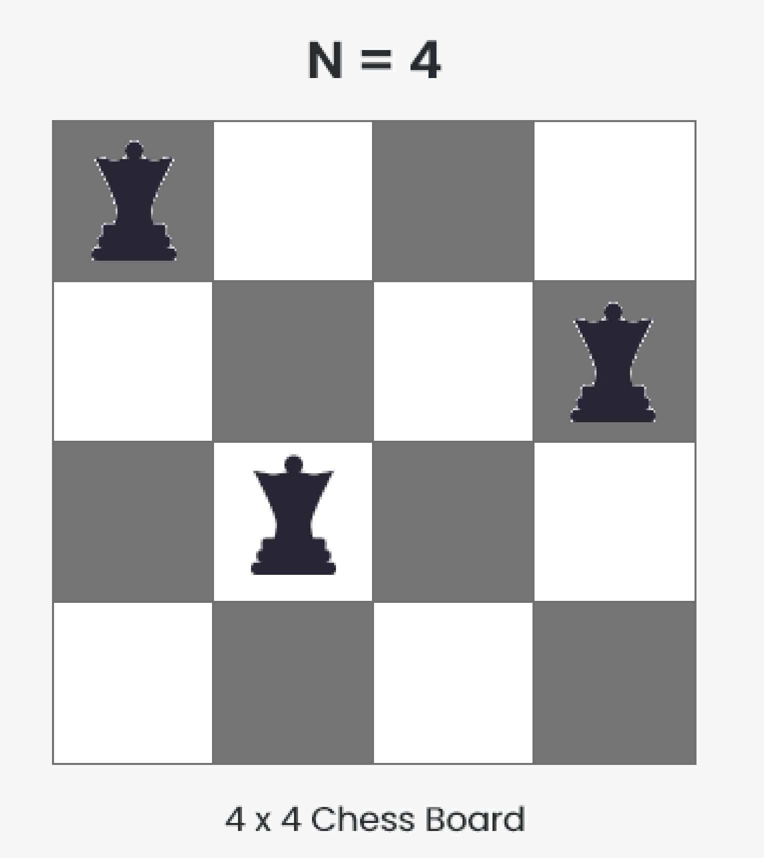
No, we havent tried all solutions ...

... need to go backwards

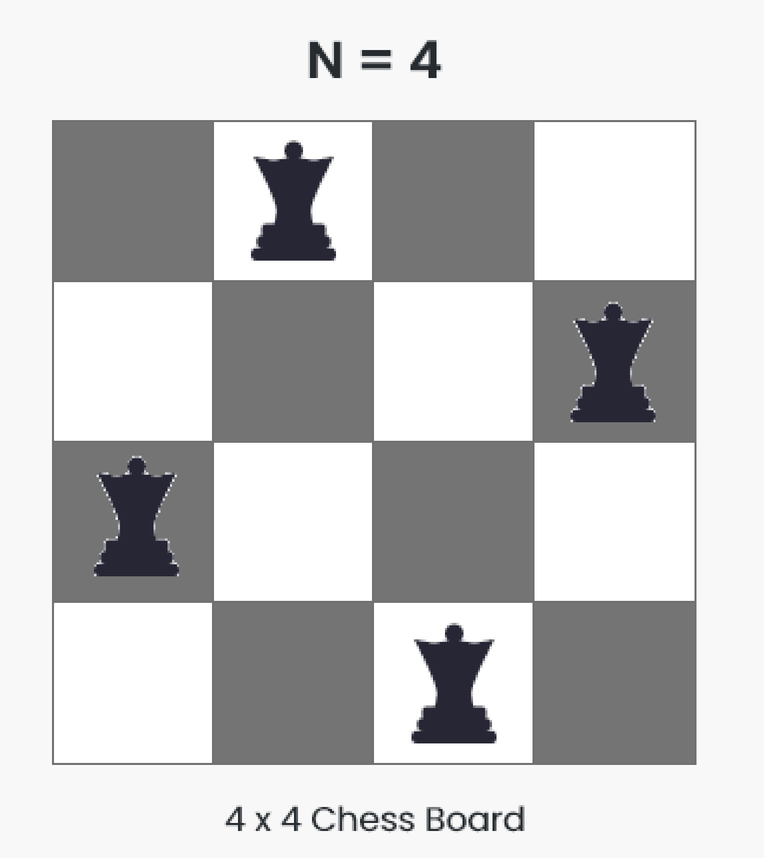
Example: 4-queens - Backtracking



Example: 4-queens - Backtracking



Example: 4-queens - Backtracking



N = 4

4 x 4 Chess Board

☑

Backtracking Design Recipe

- Combination of other “recipes”
  - **Accumulator** – for “current solution”
  - **Generative Recursion**
    - Description must include **Termination Argument**
- Code “Template”
  - 2 base cases
    - Success
    - Fail
  - Recursive call …
    - Should optimistically move forward towards potential solution by placing a queen …
    - … but **result must be checked!** And **backtrack if fail …**

Example: 4-queens – as code

```
;; termination argument:
;; recursive calls “smaller” bc ...
(define (find-sol x y …)
  (cond
   [(done? …) … DONE …]
   [(at-last-col? … x …) (find-sol FIRST-X (next y) …)]
   [(no-solution? … ) … FAIL-RESULT … ]
```

Example: 4-queens – as code

```
;; termination argument:
;; recursive calls “smaller” bc ...
(define (find-sol x y curr-solution)
  (cond
   [(done? curr-solution …) … DONE …] ;; base case - success
   [(at-last-col? … x …) (find-sol FIRST-X (next y) …)] ; try next
   [(no-solution? … ) … FAIL-RESULT … ] ;; base case - fail
```

Accumulator!

Example: 4-queens – as code

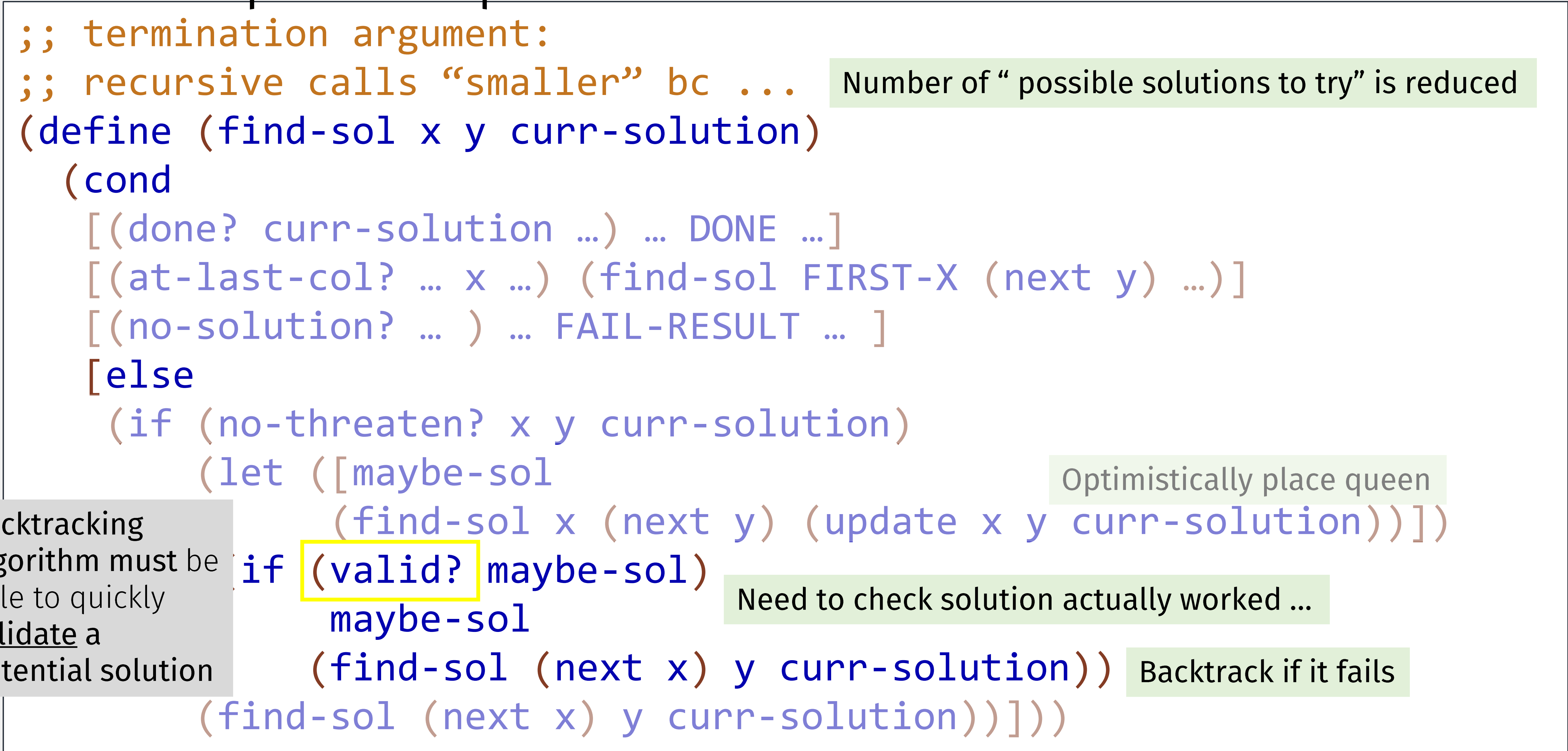
```
;; termination argument:
;; recursive calls “smaller” bc ...
(define (find-sol x y curr-solution)
  (cond
   [(done? curr-solution …) … DONE …]
   [(at-last-col? … x …) (find-sol FIRST-X (next y) …)]
   [(no-solution? … ) … FAIL-RESULT … ]
   [else
    (if (no-threaten? x y curr-solution)
        (let ([maybe-sol
               (find-sol x (next y) (update x y curr-solution))])
         (if (valid? maybe-sol)
              maybe-sol
             (find-sol (next x) y curr-solution))
        (find-sol (next x) y curr-solution))]))
```

???

Number of “ possible solutions to try” is reduced

Optimistically place queen

Example: 4-queens – as code



```
;; termination argument:
;; recursive calls “smaller” bc ...
(define (find-sol x y curr-solution)
  (cond
   [(done? curr-solution …) … DONE …]
   [(at-last-col? … x …) (find-sol FIRST-X (next y) …)]
   [(no-solution? … ) … FAIL-RESULT … ]
   [else
    (if (no-threaten? x y curr-solution)
        (let ([maybe-sol
               (find-sol x (next y) (update x y curr-solution))])
         (if (valid? maybe-sol)
              maybe-sol
             (find-sol (next x) y curr-solution))
        (find-sol (next x) y curr-solution))]))
```

Number of “ possible solutions to try” is reduced

Optimistically place queen

Need to check solution actually worked …

Backtrack if it fails

Backtracking algorithm must be able to quickly validate a potential solution

Example: 4-queens – as code

```
;; termination argument:
;; recursive calls “smaller” bc ... less possible solutions to try
(define (find-sol x y curr-solution)
  (cond
   [(done? curr-solution …) … DONE …]
   [(at-last-col? … x …) (find-sol FIRST-X (next y) …)]
   [(no-solution? … ) … FAIL-RESULT … ]
   [else
    (if (no-threaten? x y curr-solution)
        (let ([maybe-sol
              (find-sol x (next y) (update x y curr-solution))])
          (if ( false? maybe-sol)
              maybe-sol
             (find-sol (next x) y curr-solution))
        (find-sol (next x) y curr-solution))]))
```

Produce “false” value to indicate no solution

Backtracking algorithm must be able to quickly <u>validate</u> a potential solution

Example: 4-queens – as code

```
;; nqueens : Nat -> List<Queen>
;;    …     …
(define (find-sol x y cu
  (cond
   [(done? curr-solution …) … DONE …]
   [(at-last-col? … x …) (find-sol FIRST-X (next y) …)]
   [(no-solution? … ) … FAIL-RESULT … ]
   [else
    (if (no-threaten? x y curr-solution)
        (let ([maybe-sol
              (find-sol x (next y) (update x y curr-solution))])
         (if (false? maybe-sol)
             maybe-sol
            (find-sol (next x) y curr-solution))
        (find-sol (next x) y curr-solution))]))
```

```
;; A Queen is a
;; … row and column …
```

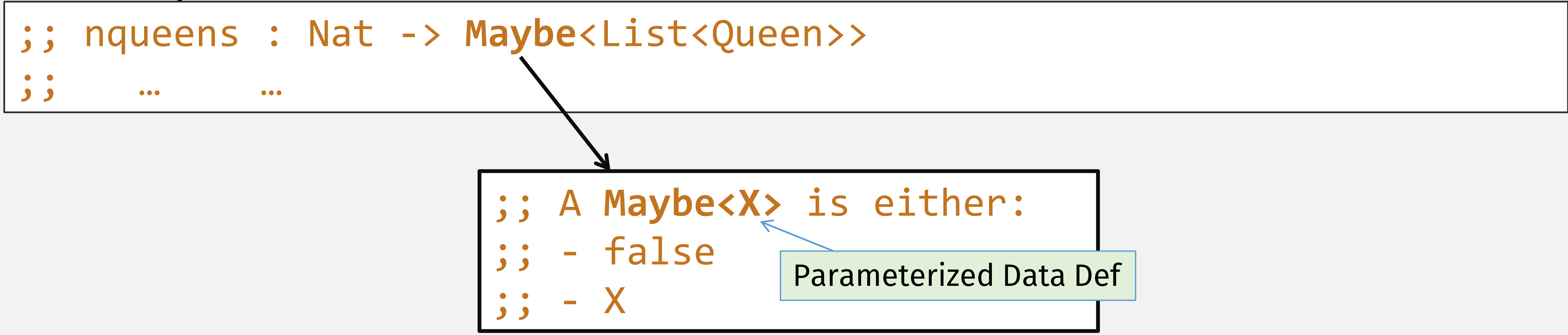
Produce “false” value to indicate no solution

Example: 4-queens – as code

```
;; nqueens : Nat -> Maybe<List<Queen>>
;;   …    …
(define (find-sol x y curr-solution)
  (cond
   [(done? curr-solution …) … DONE …]
   [(at-last-col? … x …) (find-sol FIRST-X (next y) …)]
   [(no-solution? … ) … FAIL-RESULT … ]
   [else
    (if (no-threaten? x y curr-solution)
        (let ([maybe-sol
              (find-sol x (next y) (update x y curr-solution))])
         (if ( false?  maybe-sol)
              maybe-sol
             (find-sol (next x) y curr-solution))
        (find-sol (next x) y curr-solution))]))
```

Produce “false” value to indicate no solution

Maybe Data Definitions



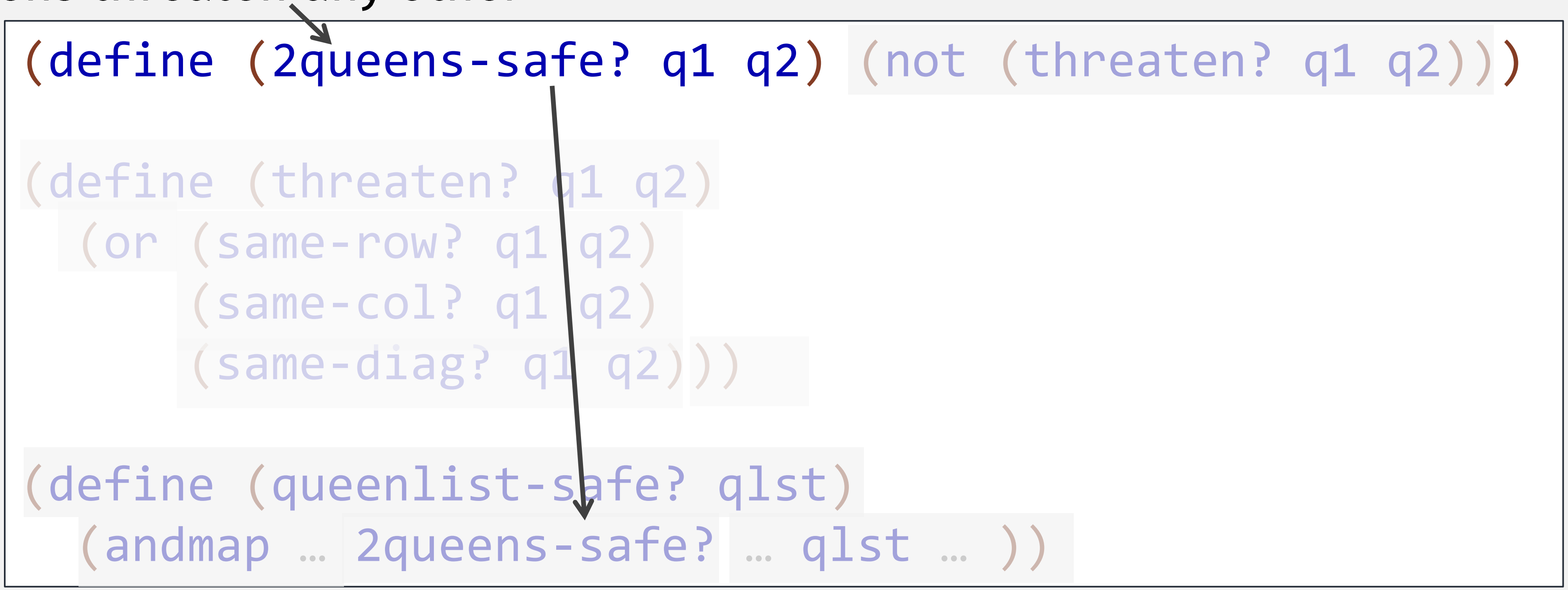
```
;; nqueens : Nat -> Maybe<List<Queen>>
;;     …      …
```

```
;; A Maybe<X> is either:
;; - false
;; - X
```

Parameterized Data Def

N-queens Solution Validation

- Still useful to write a **`valid?`** predicate, i.e., for testing
- A “valid” n-queens solution has
  - n (unique) queens
  - No queens threaten any other



```
(define (2queens-safe? q1 q2) (not (threaten? q1 q2)))

(define (threaten? q1 q2)
  (or (same-row? q1 q2)
      (same-col? q1 q2)
      (same-diag? q1 q2)))

(define (queenlist-safe? qlst)
  (andmap … 2queens-safe? … qlst … ))
```