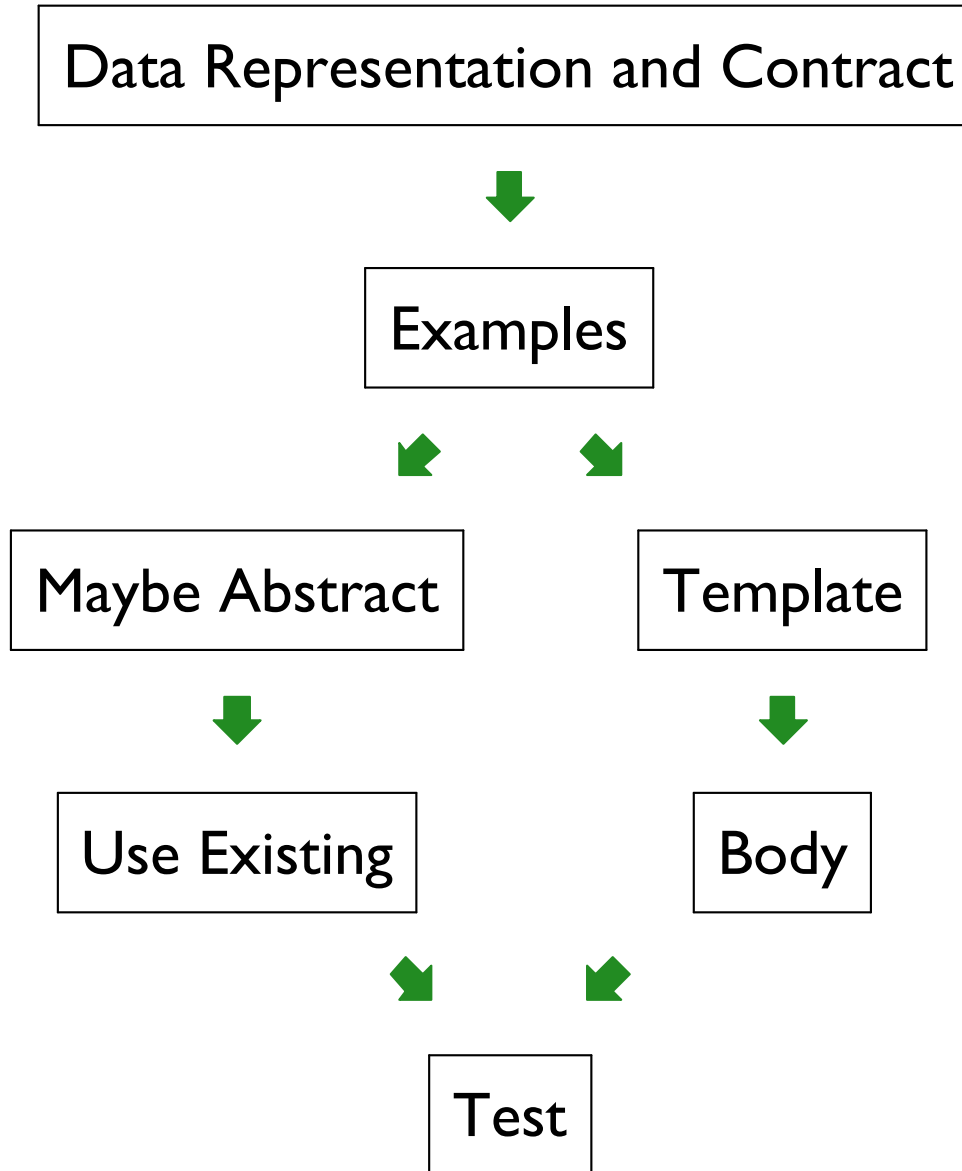


How to Design A Program (So Far)



Challenge Problem

Implement the function **odd-items** which takes a list-of-X and produces a list-of-X containing every other item in the given list (including the first item)

Data Representation and Contract

Already done for us:

```
; odd-items : list-of-X -> list-of-X
```

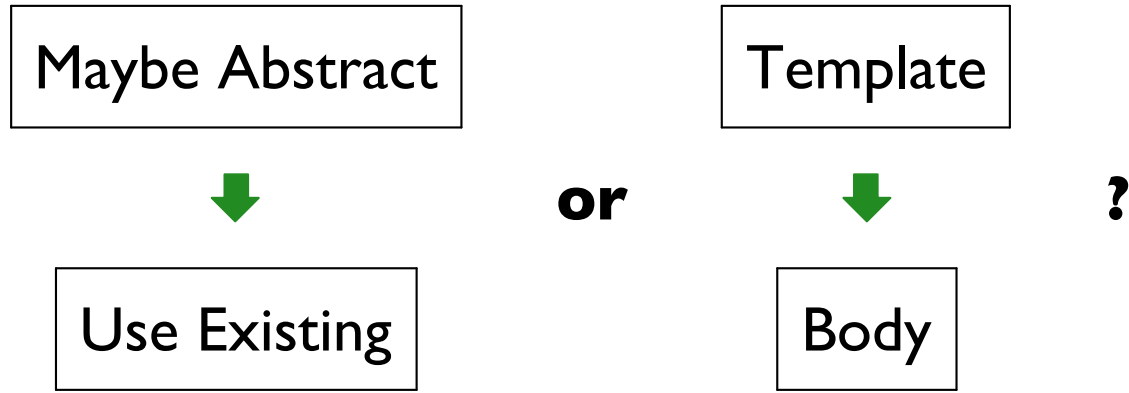
Examples

```
(check-expect (odd-items empty) empty)
```

```
(check-expect (odd-items '(1 2 3 4 5))  
              '(1 3 5))
```

```
(check-expect (odd-items '(apple banana cherry))  
              '(apple cherry))
```

```
(check-expect (odd-items (list true false))  
              (list true))
```



We know that **foldr** captures the template for **list-of-X**, so choose the left branch — and abstraction is done already!

Maybe Abstract



Use Existing

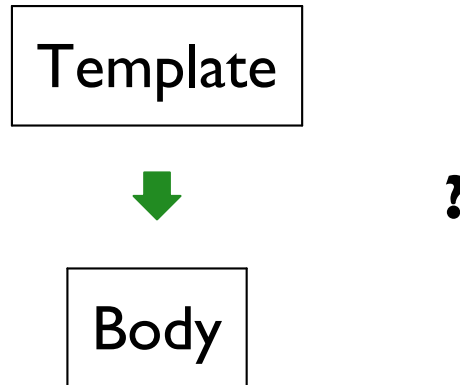
```
(define (odd-items l)
  (foldr (lambda (item odd-rest)
          ...)
        empty l))
```

Problem: the odd items of the rest of the list are useless for the odd items of the whole list

```
(check-expect (odd-items '(1 2 3 4)) '(1 3))
```

but

```
(check-expect (odd-items '(2 3 4)) '(2 4))
```



```
(define (odd-items l)
  (cond
    [(empty? l) empty]
    [(cons? l)
     ... (first l)
     ... (odd-items (rest l)) ...]))
```

Same problem — it's not just a reuse problem...

Structural Recursion

- For recursively defined data, our recipe so far always produces **structurally recursive** programs
- In a sense, it always works:

```
(define (odd-items l)
  (first
   (foldr (lambda (item odds+evens)
            (list (cons item
                    (second odds+evens))
                  (first odds+evens)))
          (list empty empty) l)))
```

But making structural recursion work sometimes requires more creativity than solving the problem a different way

Generative Recursion

Structural recursion is a powerful tool, but we need more tools

Our new tool is ***generative recursion***:

```
(define (func v)
  (cond
    [(trivially-solvable? v) ...]
    [else ...
     (func generated-v1)
      .
      .
     (func generated-vn)
     ...]))
```

Structural recursion is a special case of generative recursion that is especially common

Back to Odd Items

When the list given to `odd-items` has less than two items, the problem is trivial to solve:

```
(define (odd-items l)
  (cond
    [(or (empty? l)
         (empty? (rest l)))
     l]
    [else ...]))
```

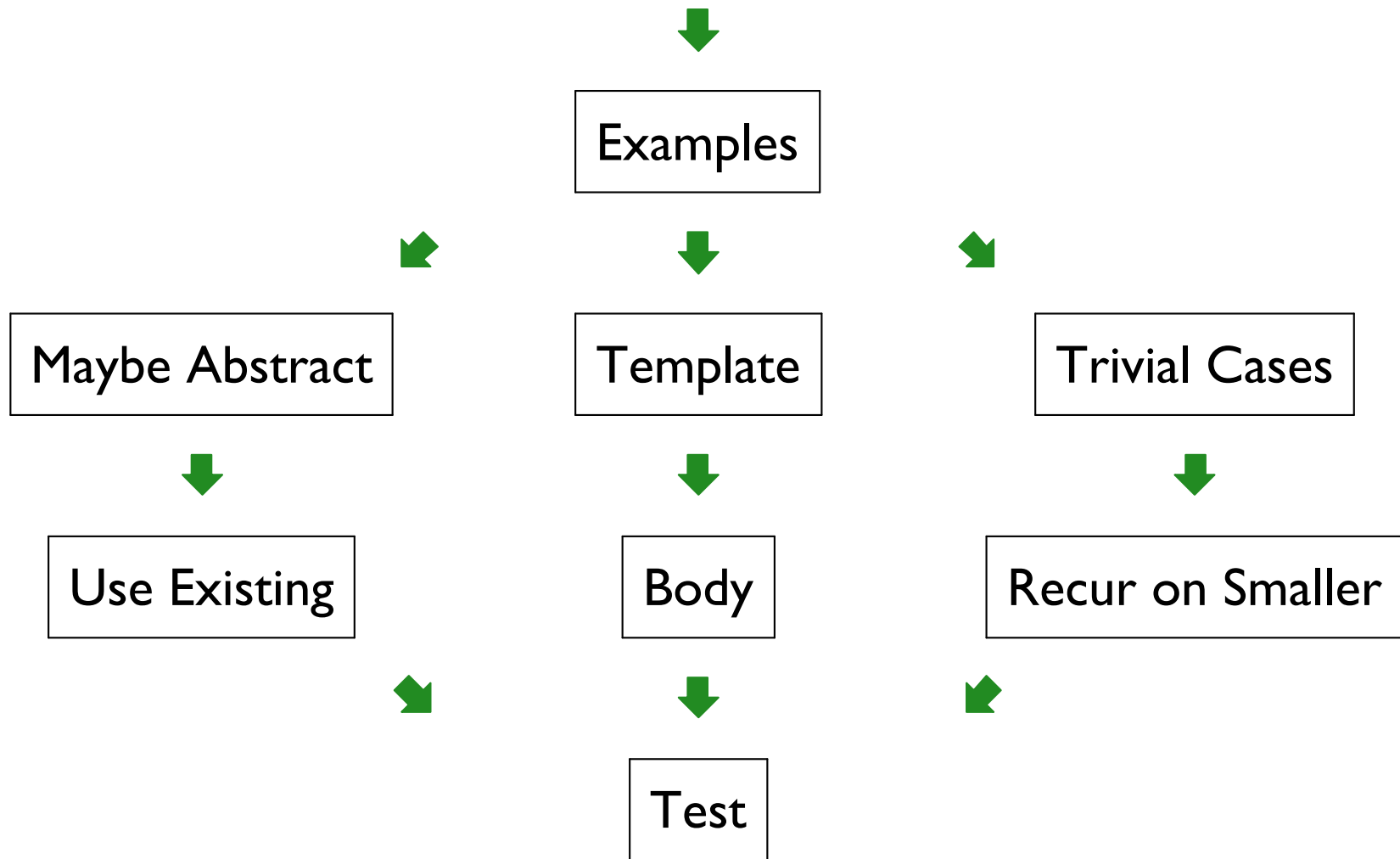
Back to Odd Items

Otherwise, it's helpful to have the *rest* of the *rest*:

```
(define (odd-items l)
  (cond
    [(or (empty? l)
         (empty? (rest l)))
     l]
    [else (cons
            (first l)
            (odd-items (rest (rest l))))]))
```

How to Design A Program

Data Representation and Contract



Guessing a Number

```
; make-secret-checker : num -> (num -> sym)
(define (make-secret-checker n)
  (local [(define secret (random n))]
    (lambda (m)
      (cond
        [(= m secret) 'perfect]
        [< m secret) 'too-small]
        [> m secret) 'too-large]))))
```

Implement the function **discover-number** which takes a number *n* and a function produced by **(make-secret-checker n)**, and returns the secret number in the function

Data Representation and Contract

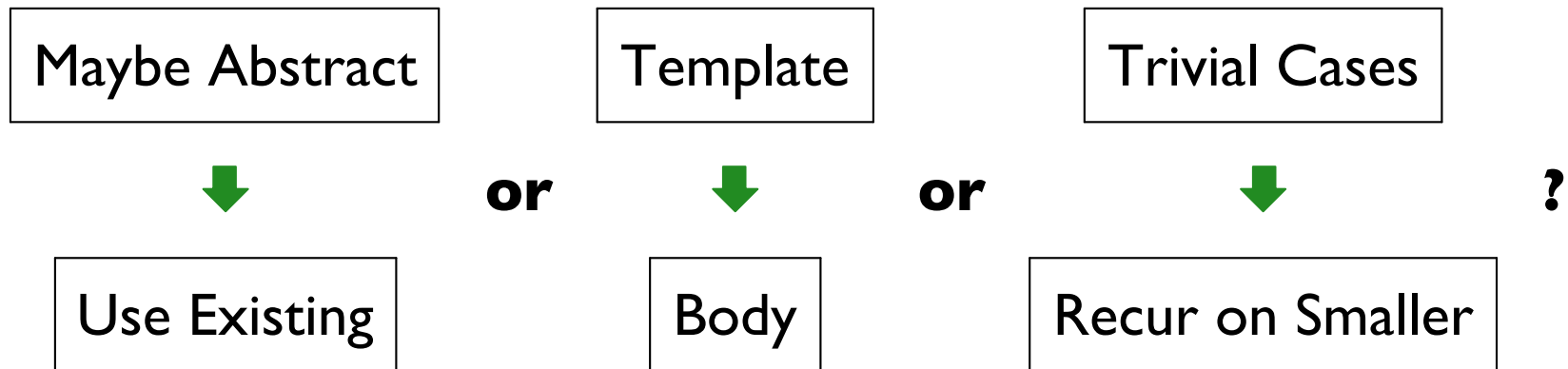
Apparently done already:

```
; discover-number : num (num -> sym) -> num
```

Examples

```
(check-expect  
  (discover-number 1 (make-secret-checker 1))  
  0)
```

```
(check-expect  
  (discover-number 3 (make-secret-checker 3))  
  ... 0 or 1 or 2 ...)
```



- Abstract/reuse: nothing obvious
- Template: nothing for **num**

... but is it really **nat**?

Yes, starting from 1

Template



Body

```
; discover-number : nat (nat -> sym) -> nat
; n is <= number given to checker,
; checker's secret is < n
(define (discover-number n checker)
  (cond
    [(= n 1) ...]
    [else
     ...
     (discover-number (sub1 n) checker)
     ...])))
```

Template



Body

```
; discover-number : nat (nat -> sym) -> nat
; n is <= number given to checker,
; checker's secret is < n
(define (discover-number n checker)
  (cond
    [(= n 1) 0]
    [else
     ...
     (discover-number (sub1 n) checker)
     ...]))
```

Template



Body

```
; discover-number : nat (nat -> sym) -> nat
; n is <= number given to checker,
; checker's secret is < n
(define (discover-number n checker)
  (cond
    [(= n 1) 0]
    [else
     (cond
       [(symbol=? (checker (sub1 n)) 'perfect) (sub1 n)]
       [else
        (discover-number (sub1 n) checker)]))]))
```

Template



Body

```
; discover-number : nat (nat -> sym) -> nat
; n is <= number given to checker,
; checker's secret is < n
(define (discover-number n checker)
  (cond
    [(= n 1) 0]
    [else
     (cond
       [(symbol=? (checker (sub1 n)) 'perfect) (sub1 n)]
       [else
        (discover-number (sub1 n) checker)]))]))
```

This works, but is there a better way?

Guessing a Number

If you know a number is between 0 and 9:



and you only get 'perfect' or 'imperfect' answers to guesses, there's no better way to find the number

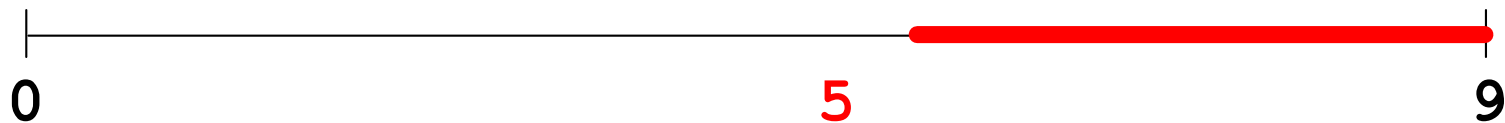


Guessing a Number

If you know a number is between 0 and 9:



and you only get 'perfect' or 'imperfect' answers to guesses, there's no better way to find the number



'perfect'

Guessing a Number

If you know a number is between 0 and 9:



but you get 'perfect, 'too-small, or 'too-large answers, it's better to guess in the middle



Guessing a Number

If you know a number is between 0 and 9:



but you get 'perfect, 'too-small, or
'too-large answers, it's better to guess in the
middle



'perfect

Trivial Cases



Recur on Smaller



'perfect'

- Trivially solvable if mid-point is 'perfect'
- Otherwise, mid-point results cuts the range in half — try again

Guessing A Number with Generative Recursion

```
(define (discover-number n checker)
  (discover-in-range 0 (sub1 n) checker))

; discover-in-range : nat nat (nat -> bool) -> num
; Finds the number between lo and hi (inclusive)
(define (discover-in-range lo hi checker)
  (cond
    [trivial? ...]
    [else
     ... (discover-in-range ...)
     ...])))
```

Guessing A Number with Generative Recursion

```
(define (discover-number n checker)
  (discover-in-range 0 (sub1 n) checker))

; discover-in-range : nat nat (nat -> bool) -> num
; Finds the number between lo and hi (inclusive)
(define (discover-in-range lo hi checker)
  (local [(define mid (quotient (+ lo hi) 2))])
    (cond
      [trivial? ...]
      [else
       ... (discover-in-range ...)
       ...])))
```

Guessing A Number with Generative Recursion

```
(define (discover-number n checker)
  (discover-in-range 0 (sub1 n) checker))

; discover-in-range : nat nat (nat -> bool) -> num
; Finds the number between lo and hi (inclusive)
(define (discover-in-range lo hi checker)
  (local [(define mid (quotient (+ lo hi) 2))])
    (cond
      [(symbol=? (checker mid) 'prefect) mid]
      [else
       ... (discover-in-range ...)
       ...])))
```

Guessing A Number with Generative Recursion

```
(define (discover-number n checker)
  (discover-in-range 0 (sub1 n) checker))

; discover-in-range : nat nat (nat -> bool) -> num
; Finds the number between lo and hi (inclusive)
(define (discover-in-range lo hi checker)
  (local [(define mid (quotient (+ lo hi) 2))])
    (cond
      [(symbol=? (checker mid) 'prefect) mid]
      [else
       ... (discover-in-range lo mid checker)
       ... (discover-in-range mid hi checker) ...])))
```

Guessing A Number with Generative Recursion

```
(define (discover-number n checker)
  (discover-in-range 0 (sub1 n) checker))

; discover-in-range : nat nat (nat -> bool) -> num
; Finds the number between lo and hi (inclusive)
(define (discover-in-range lo hi checker)
  (local [(define mid (quotient (+ lo hi) 2))])
    (cond
      [(symbol=? (checker mid) 'perfect) mid]
      [else
       (cond
         [(symbol=? (checker mid) 'too-large)
          (discover-in-range lo mid checker)]
         [else
          (discover-in-range mid hi checker)]]]))))
```

Running the Guesser

```
(discover-number 10 check-7)
```

→

```
(discover-in-range 0 9 check-7)
```

```
using (define (discover-number n checker)
        (discover-in-range 0 (sub1 n) checker))
```

Running the Guesser

```
(discover-in-range 0 9 check-7)
```

→

```
(cond
  [(symbol=? (check-7 4) 'perfect) 4]
  [else
   (cond
    [(symbol=? (check-7 4) 'too-large)
     (discover-in-range 0 4 check-7)]
    [else
     (discover-in-range 4 9 check-7) ])]])
```

```
using (define (discover-in-range lo hi checker)
       (local [(define mid (quotient (+ lo hi) 2))]
         (cond
          [(symbol=? (checker mid) 'perfect) mid]
          [else
           (cond
            [(symbol=? (checker mid) 'too-large)
             (discover-in-range lo mid checker)]
            [else
             (discover-in-range mid hi checker) ])])))
```


Running the Guesser

```
(cond
  [(symbol=? (check-7 4) 'perfect) 4]
  [else
   (cond
     [(symbol=? (check-7 4) 'too-large)
      (discover-in-range 0 4 check-7)]
     [else
      (discover-in-range 4 9 check-7)]))])
```

→

```
(cond
  [(symbol=? (check-7 4) 'too-large)
   (discover-in-range 0 4 check-7)]
  [else
   (discover-in-range 4 9 check-7)])
```

Running the Guesser

```
(cond
  [(symbol=? (check-7 4) 'too-large)
   (discover-in-range 0 4 check-7)]
 [else
  (discover-in-range 4 9 check-7)])
```

→

```
(discover-in-range 4 9 check-7)
```

Running the Guesser

```
(discover-in-range 4 9 check-7)
```

→

```
(cond  
  [(symbol=? (check-7 6) 'perfect) 6]  
  [else  
    (cond  
      [(symbol=? (check-7 6) 'too-large)  
       (discover-in-range 4 6 check-7)]  
      [else  
       (discover-in-range 6 9 check-7) ])]])
```

Running the Guesser

```
(cond
  [(symbol=? (check-7 6) 'perfect) 6]
  [else
   (cond
    [(symbol=? (check-7 6) 'too-large)
     (discover-in-range 4 6 check-7)]
    [else
     (discover-in-range 6 9 check-7) ])]])
```

→

```
(discover-in-range 6 9 check-7)
```

Running the Guesser

```
(discover-in-range 6 9 check-7)
```

→

```
(cond
  [(symbol=? (check-7 7) 'perfect) 7]
  [else
   (cond
    [(symbol=? (check-7 7) 'too-large)
     (discover-in-range 6 7 check-7)]
    [else
     (discover-in-range 7 9 check-7) ])]])
```

Running the Guesser

```
(cond
  [(symbol=? (check-7 7) 'perfect) 7]
  [else
   (cond
     [(symbol=? (check-7 7) 'too-large)
      (discover-in-range 6 7 check-7)]
     [else
      (discover-in-range 7 9 check-7) ])]])
```

→

7

Running the Guesser Again

```
(discover-number 3 check-2)
```

→

```
(discover-in-range 0 2 check-2)
```

Running the Guesser Again

```
(discover-in-range 0 2 check-2)
```

→

```
(cond
  [(symbol=? (check-2 1) 'perfect) 1]
  [else
   (cond
    [(symbol=? (check-2 1) 'too-large)
     (discover-in-range 0 1 check-2)]
    [else
     (discover-in-range 1 2 check-2)])])])
```


Running the Guesser Again

```
(cond
  [(symbol=? (check-2 1) 'perfect) 1]
  [else
   (cond
    [(symbol=? (check-2 1) 'too-large)
     (discover-in-range 0 1 check-2)]
    [else
     (discover-in-range 1 2 check-2)]))])
```

→

```
(discover-in-range 1 2 check-2)
```

Running the Guesser Again

```
(discover-in-range 1 2 check-2)
```

→

```
(cond
  [(symbol=? (check-2 1) 'perfect) 1]
  [else
   (cond
    [(symbol=? (check-2 1) 'too-small)
     (discover-in-range 1 2 check-7)]
    [else
     (discover-in-range 1 2 check-2) ])]])
```

Running the Guesser Again

```
(cond
  [(symbol=? (check-2 1) 'perfect) 1]
  [else
   (cond
    [(symbol=? (check-2 1) 'too-small)
     (discover-in-range 1 2 check-7)]
    [else
     (discover-in-range 1 2 check-2) ])]])
```

→

```
(discover-in-range 1 2 check-2)
```

Running the Guesser Again

```
(discover-in-range 1 2 check-2)
```

→

```
(discover-in-range 1 2 check-2)
```

Running the Guesser Again

```
(discover-in-range 1 2 check-2)
```

→

```
(discover-in-range 1 2 check-2)
```

Infinite loop!

Generative Recursion and Termination

- With structural recursion, a program always ***terminates***
 - Every value is finite
- With generative recursion, termination becomes more tricky
 - You have to argue that the problem size definitely gets smaller for every recursive call

Guessing a Number, Corrected

```
(define (discover-in-range lo hi checker)
  (local [(define mid (quotient (+ lo hi) 2))]
    (cond
      [(symbol=? (checker mid) 'prefect) mid]
      [else
       (cond
         [(symbol=? (checker mid) 'too-large)
          (discover-in-range lo (sub1 mid))]
         [else
          (discover-in-range (add1 mid) hi)]))]))))
```

Algorithms

Our **discover-in-range** function is an example of a general **algorithm** called **binary search**

Many algorithms are less obvious than binary search

Mostly you'll use general algorithms, not invent them

- Algorithm textbooks are like “recipe” books
- Few people design new general algorithms

Generative recursion is far more common than general algorithms, and it's often merely structural recursion