Lecture 2: Procedures

Review

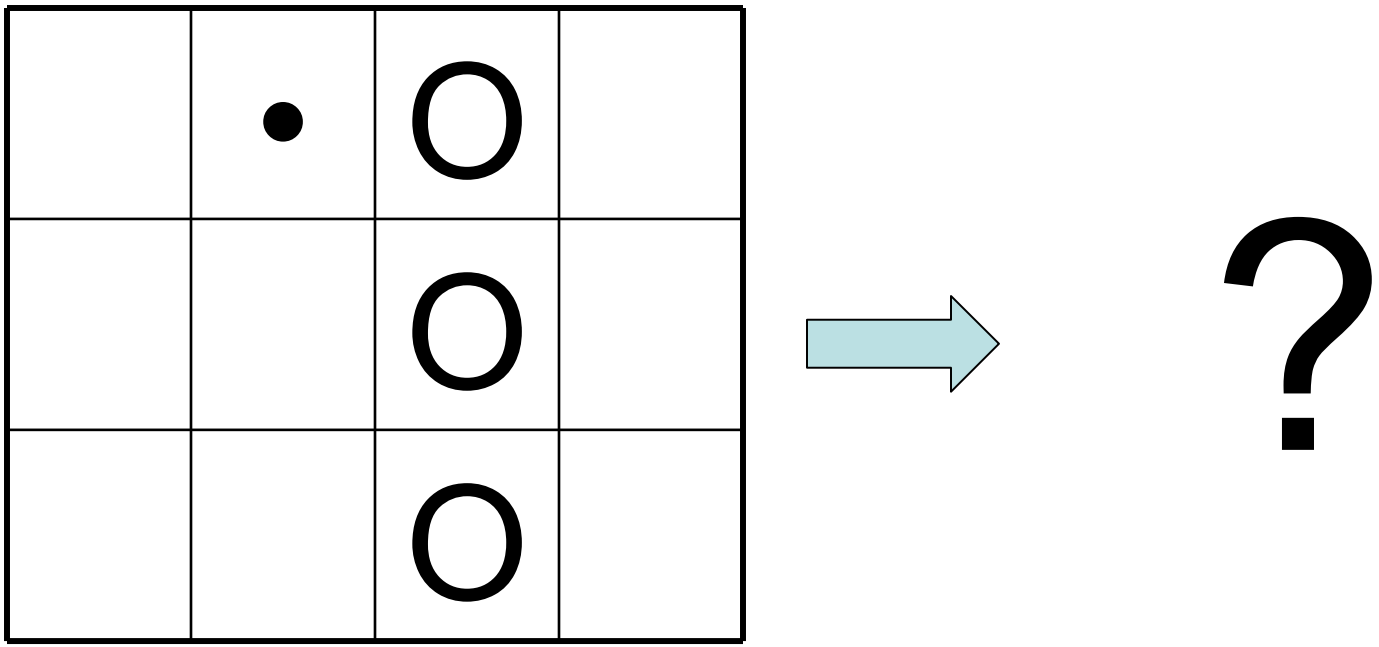
- Substitution Model
- Names
- lambda

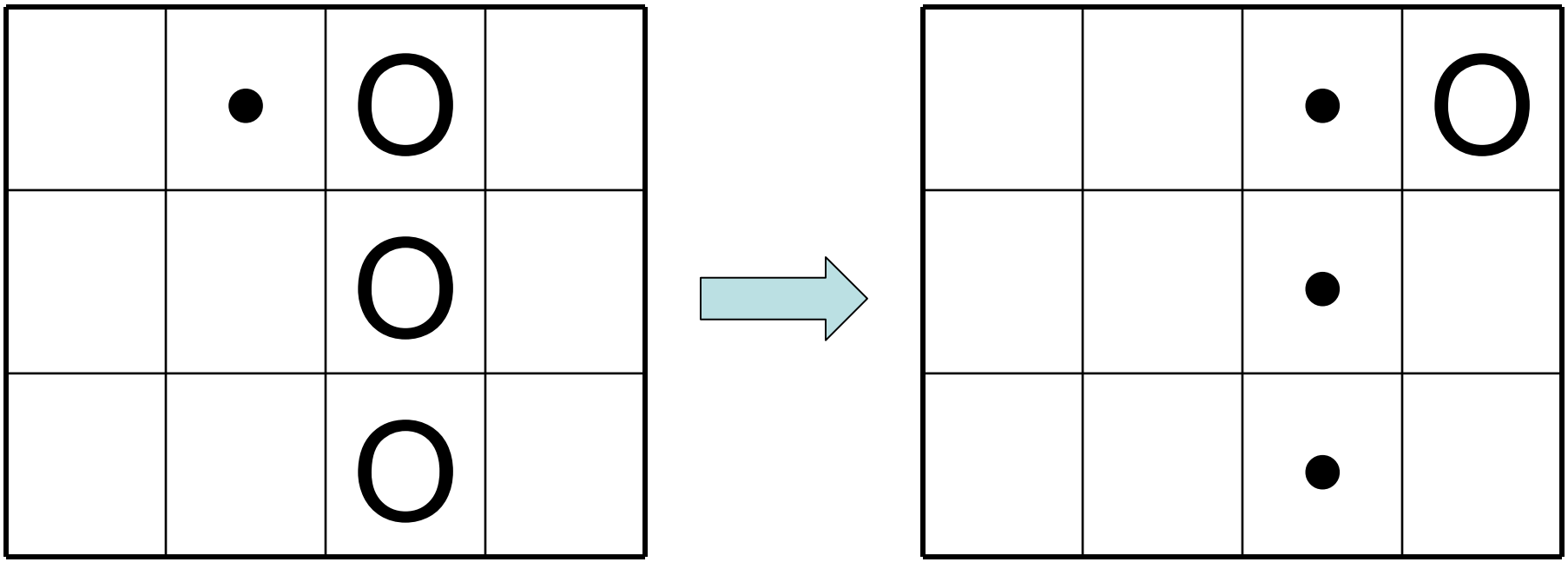
Abstracting computation

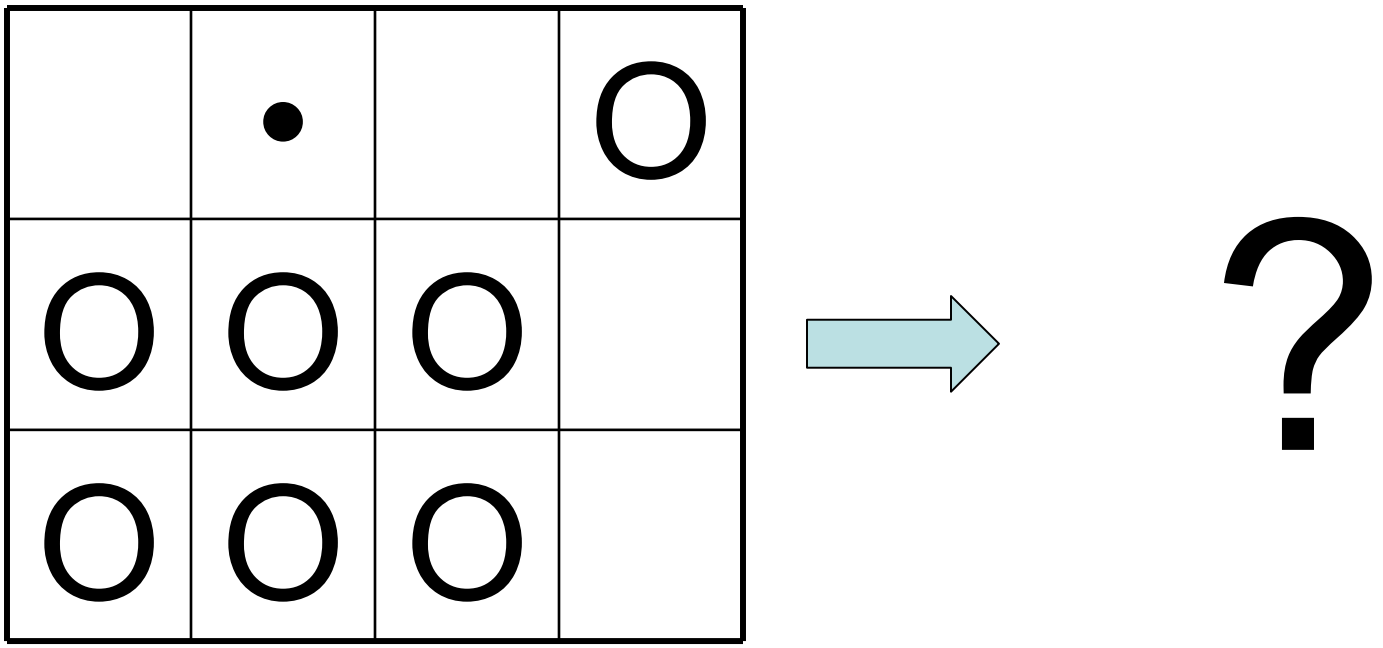
- We've already learned how to abstract data using names
- Now we want to abstract computation
- Why?

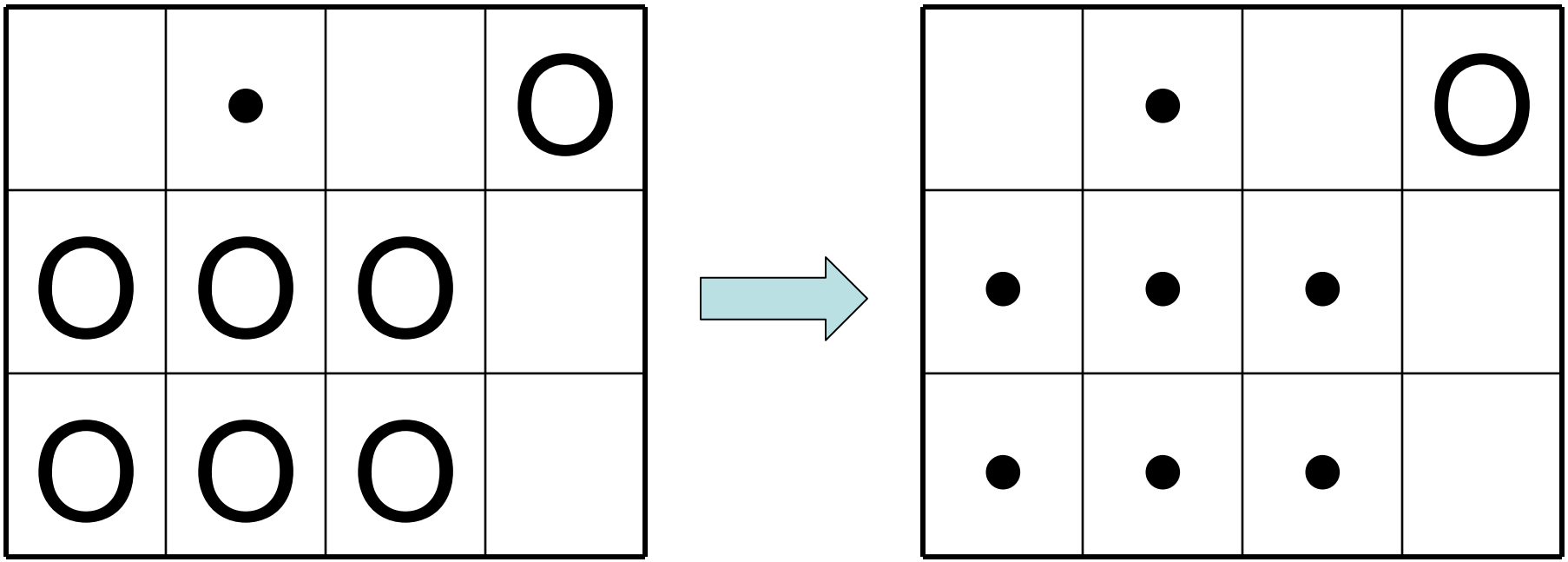
HiQ Game

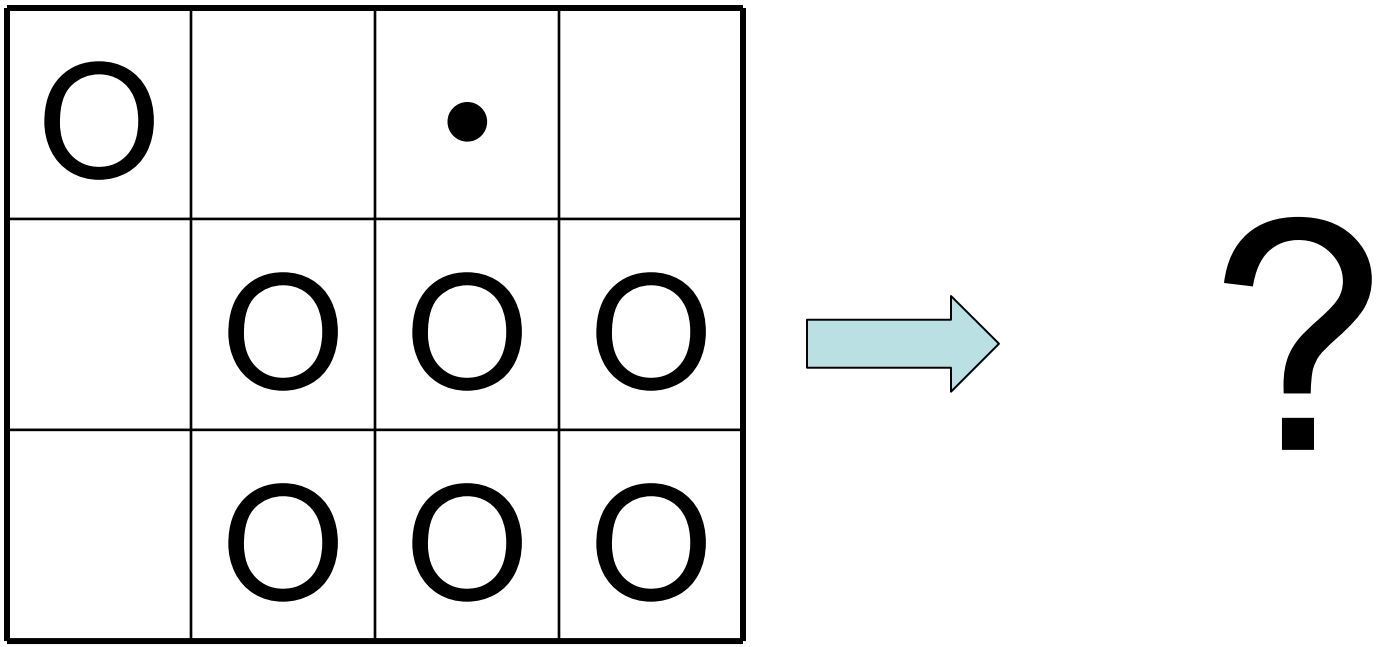
- Lets use the pegboard as a model of computation

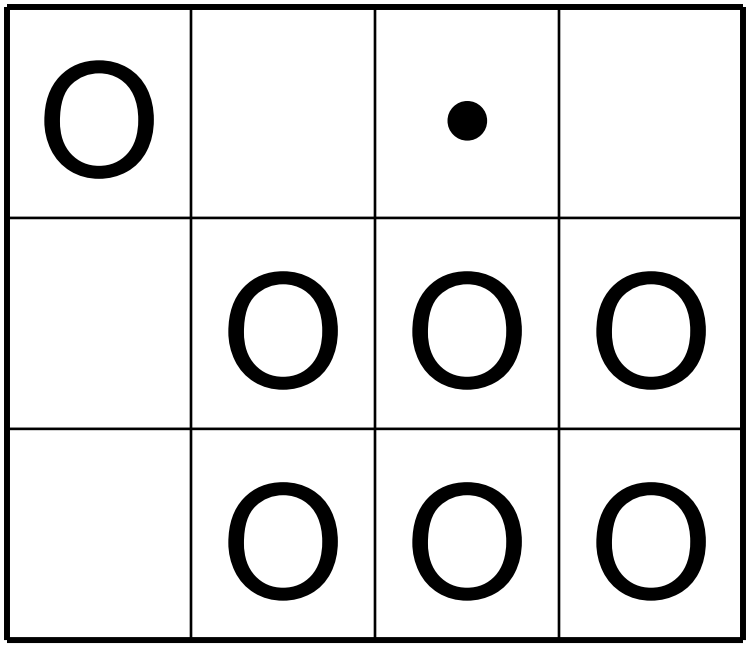


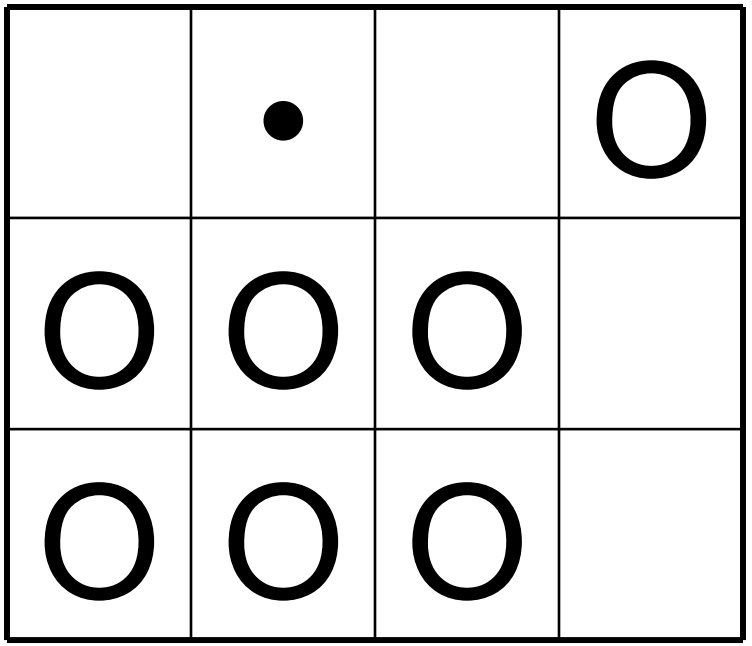







O
O O O
O O O
?


O
O O O
O O O


O
O O O
O O O

Abstracting computation

- We've already learned how to abstract data using names
- Now we want to abstract computation
- Use the **lambda** special form

lambda

- Special form that creates a procedure
- (**lambda** (*args) body*)
- Example:
  - (**lambda** (x) (* x x))
- The value returned by this special form is a **procedure**

lambda

- We can then apply this procedure using combinations
- ((lambda (x) (* x x)) 5)
- Three things happen when we evaluate this procedure
  1.
  2.
  3.

Lambda special form

- lambda syntax `(lambda (x y) (/ (+ x y) 2))`

- 1st operand position: the parameter list `(x y)`
  - a list of names (perhaps empty) `()`
  - determines the number of operands required

- 2nd operand position: the body `(/ (+ x y) 2)`
  - may be any expression
  - not evaluated when the lambda is evaluated

(from 6.001 slides)

lambda

- **Lambda** creates a *compound procedure* that is made up of the primitive procedures that are built into Scheme

Scheme Basics

- Rules for *evaluation*

1. If **self-evaluating,** return value.
2. If a **name,** return value associated with name in environment.
3. If a **special form,** do something special.
4. If a **combination,** then
   a. *Evaluate* all of the subexpressions of combination (in any order)
   b. *apply* the operator to the values of the operands (arguments) and return result

- Rules for *application*

1. If procedure is **primitive procedure,** just do it.
2. If procedure is a **compound procedure,** then:
   evaluate the body of the procedure with each formal parameter replaced by the corresponding actual argument value.

(from 6.001 Slides)

Abstracting Procedures

- In Scheme a procedure is a type like numbers, strings, etc.
- We can use **define** to assign a name to a procedure
- (define add1 (lambda (x) (+ x 1)))
- This expression does two things

Another example

- (define double (lambda (x) ….))))
- Doubles the argument x
- (define not (lambda (x)

(* 2 x)))

Lecture Problem

- Write a procedure sq that returns the cube of its input.
- `(define sq`

Lecture Problem

- Write a procedure cube that returns the cube of it's input.
- `(define cube`

Lecture Problem

- Write a procedure neg that takes a number and negates it
- `(define neg`

Lecture Problem

- Given a margin width m, which is both the top, bottom, left, and right margin of the page, write a procedure that computes the ”usable” (non margin) area of the 8.5in by 11in sheet of paper.

```
(usable-page 0)
;Value: 93.5
(usable-page 1)
;Value: 58.5
(define usable-page
```

- Now modify usable-page to take four arguments, top, bottom, left, right. Compute the page size using these separate margins

if

- Special form
- (**if** *test consequent alternative*)
- Type this into Dr. Scheme
- (if (> 6 5) 0 1)

Lecture Problem

- Write a procedure the-answer?, which returns true (#t) if the input is the number 42.

```
(define the-answer?
```

Recursion

- Sometimes, computation can’t be expressed as a fixed set of steps.
- Functions can call themselves
- This is called recursion

Example

- Sum the numbers from 1 to n
- (**define** sumton (lambda (n)

  .....
- Now what? How can we go about solving this problem?

Sum to n

- Are there any cases where we know the answer in a fixed set of computations?
- Write the expressions to compute sumton in these cases

- We'll call these the **base cases**

Sum to n

- Let's compute the sum from 1 to 5
- First, assume that **sumton** works for numbers less than 5.
- How can we combine it with some simple computation to get an answer?
- This is called the recursive case

Putting it all together

- We need three things to put it all together
  - 1. A way to solve the base case
  - 2. A way to combine an application of **sumton** with some simple operations
  - 3. A way to choose between the case to use

Remember **if**?

- (**if** *test consequent alternative*)

- Basic idea:
  - (define sumton
    (lambda (n)
    (**if** *base_case*
    base_expression
    (*op* (sumton (*op* n)))))))

My Answer

– (define sumton
    (lambda (n)
        (**if** *(*<= n 1)
            n
            (+ 1 (sumton (- n 1)))))))

What's going on from Scheme's perspective?

- We'll use the stepper

Writing factorial

- The same pattern applies to calculating factorials
- Let's figure out a plan:
  - What's the base case?
  - What's the recursive case?

Exponentiation

- Calculate x^n where n>=0 and n is an integer
- Base case
- Recursive case?

Remainder

Recursion is like proof by induction

- Prove: sum from 1 to n = n*(n+1)/2

- Base Cases:
  - 1 : 1
  - 2 : 3