

CSCI 334:  
Principles of Programming Languages

Lecture 11: Higher Order Functions

Instructor: Dan Barowy  
**Williams**

Topics

Higher Order Functions

Computability, part 1

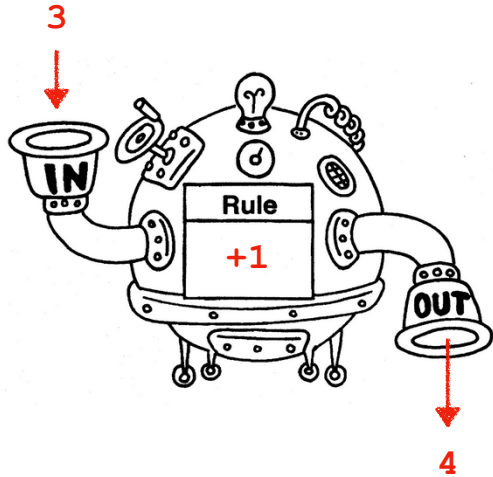
Your to-dos

1. Reading response, **due Wednesday 3/9.**
2. Lab 5, **due Sunday 3/13** (partner lab)  
(last one before midterm!)

Three amazing concepts from LISP

- First-class functions
- Higher-order functions
  - map
  - fold
- Garbage collection

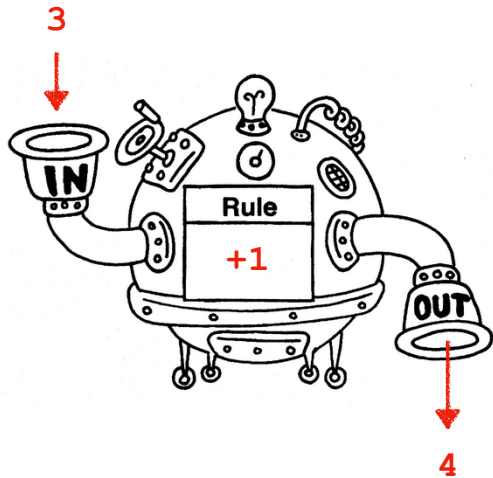
a function



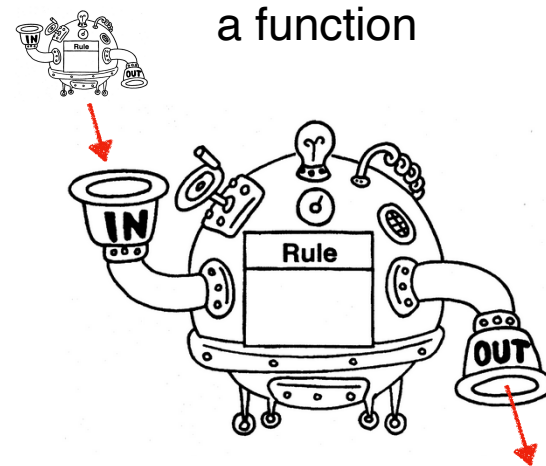
“first class” function

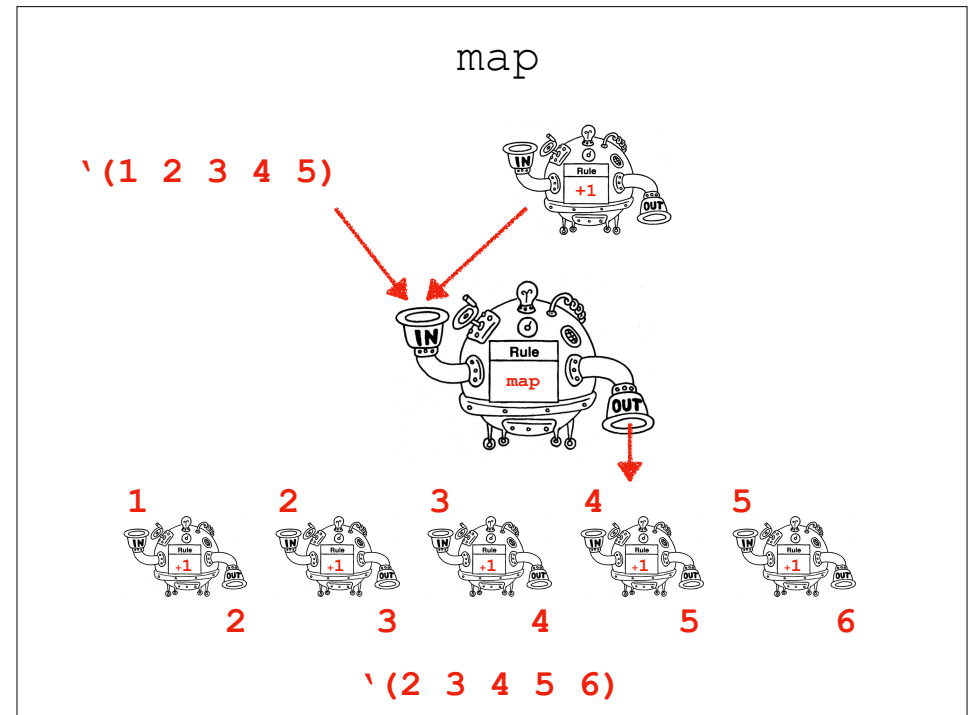
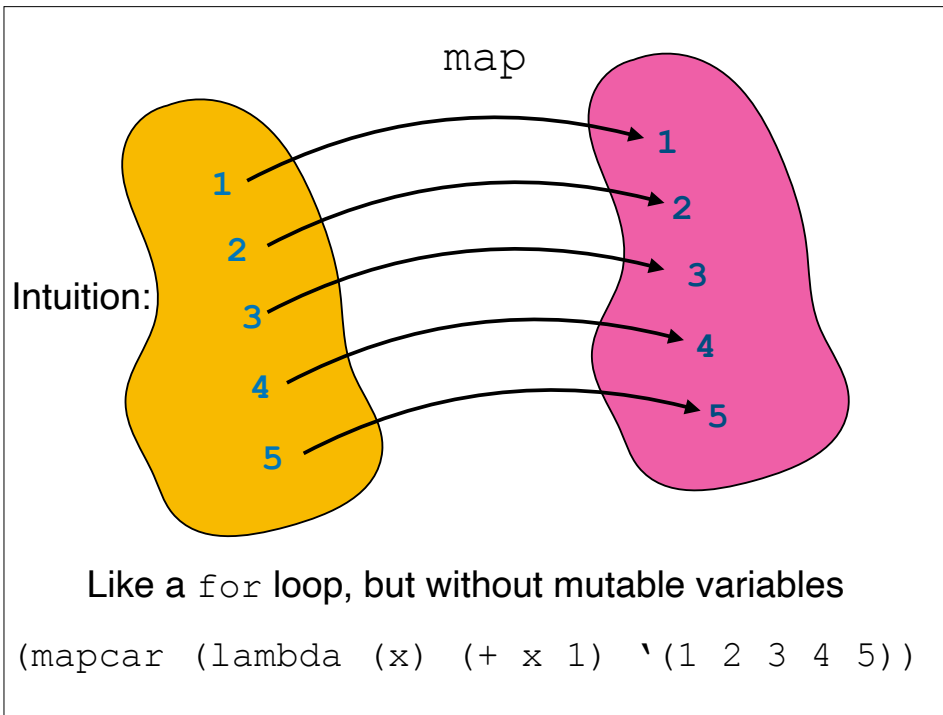
Function definitions are **values** in a functional programming language

a function



a function





## Activity

Write a function (using `mapcar`) that replaces the number 3 in a list with the number 6

```
(mapcar #'my-replace '(1 2 3 4 5 6))
      '(1 2 6 4 5 6)
```

## Activity

Write a function (using `mapcar`) that replaces the number 3 in a list with the number 6

```
(defun my-replace (x)
  (cond
    ((equal x 3) 6)
    (t x)
  )
)

(mapcar #'my-replace '(1 2 3 4 5 6))
      '(1 2 6 4 5 6)
```

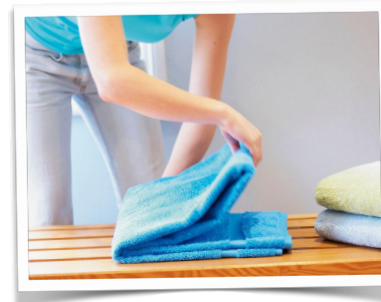
## fold

Intuition:



## fold left

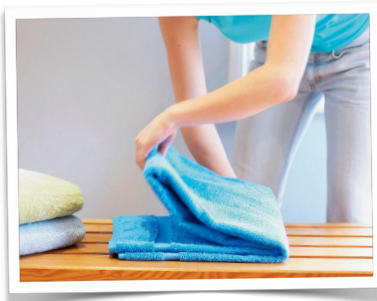
```
(reduce #' + '(1 2 3) :initial-value 0)
```



```
acc = 0, \ (1 2 3)  
acc = 0+1, \ (2 3)  
acc = 1+2, \ (3)  
acc = 3+3, nil  
returns acc = 6
```

## fold right

```
(reduce #' + '(1 2 3) :initial-value 0  
:from-end t)
```



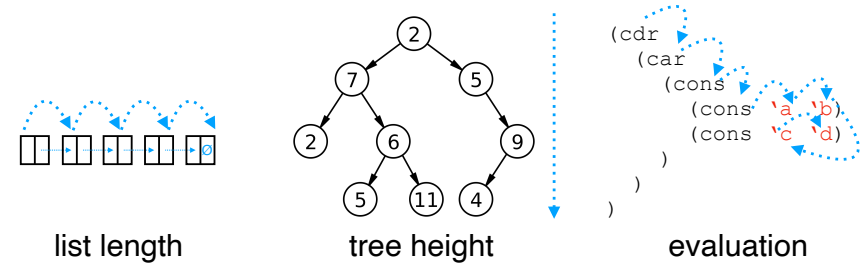
```
\ (1 2 3), acc = 0  
\ (1 2), acc = 0+3  
\ (1), acc = 2+3  
nil acc = 5+1  
returns acc = 6
```

## what does this print?

```
(reduce #' append '((2) (2))  
:initial-value '(w i l l i a m s))
```

## how about?

```
(reduce #'append '((2) (2))  
  :initial-value '(w i l l i a m s)  
  :from-end t)
```



## Activity

### list length using reduce

```
(length-list '(1 2 3 4 5 6)) → 6
```

## Activity

### list length using reduce

```
(defun mycount (acc x) (+ acc 1))  
(defun length-list (xs)  
  (reduce #'mycount xs  
    :initial-value 0)  
)
```

That's pretty much it!

- See “LISP Notes” for all the syntax you need to know on course webpage

## Automatic Memory Management

## Memory management

- C:  
When you want to use a variable, you have to *allocate* it first, then *deallocate* it when done.  

```
MyObject *m = malloc(sizeof(MyObject));  
m->foo = 2;  
m->bar = 3;  
... do stuff with m ...  
free(m);
```

## Memory management

- Java:  
You barely need to think about this at all.  

```
MyObject m = new MyObject(2,3);  
... do stuff with m ...
```
- Same with LISP!  

```
(cons 2 3)
```

## Lisp memory model

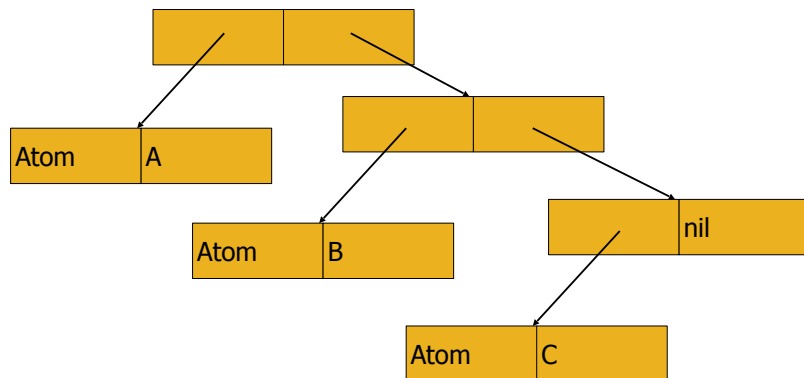
Cons cell: 

Address	Decrement
---------	-----------

Atom: 

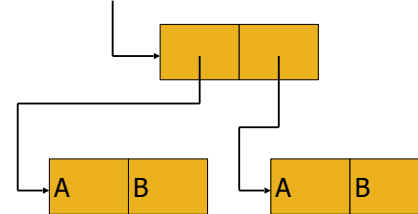
Atom	value
------	-------

```
(cons 'A (cons 'B (cons 'C nil)))
```

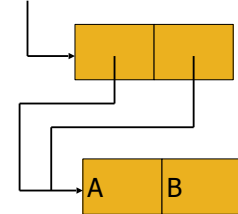


## Sharing data

(a)

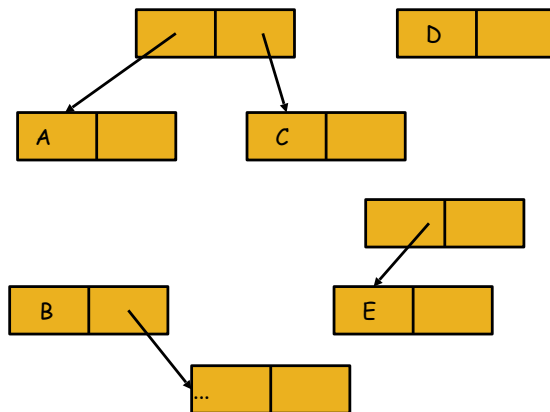


(b)



- Which is the result of evaluating  
`(cons (cons 'A 'B) (cons 'A 'B))` ?

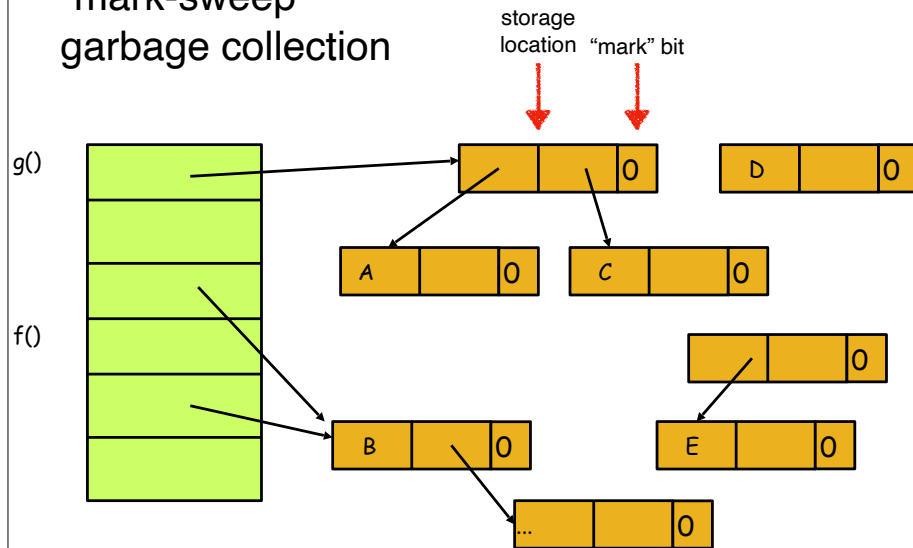
## Garbage collection



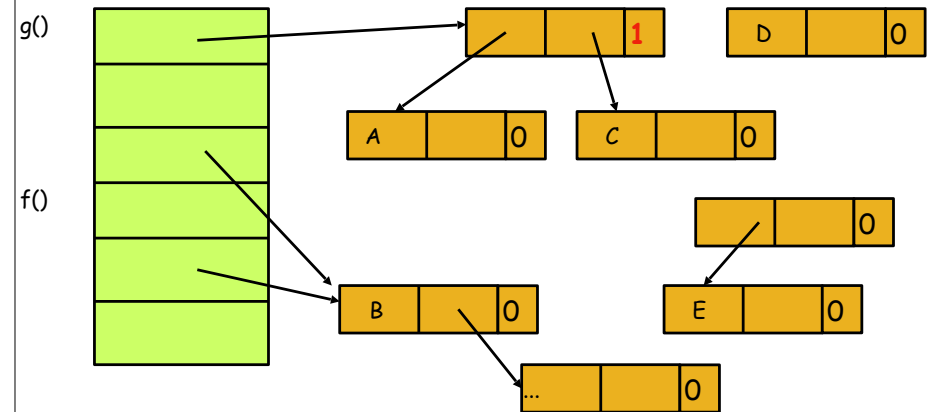
## Garbage collection

A **garbage collection algorithm** is an algorithm that determines whether the storage, occupied by a value used in a program, **can be reclaimed for future use**. Garbage collection algorithms are often tightly integrated into a programming language **runtime**.

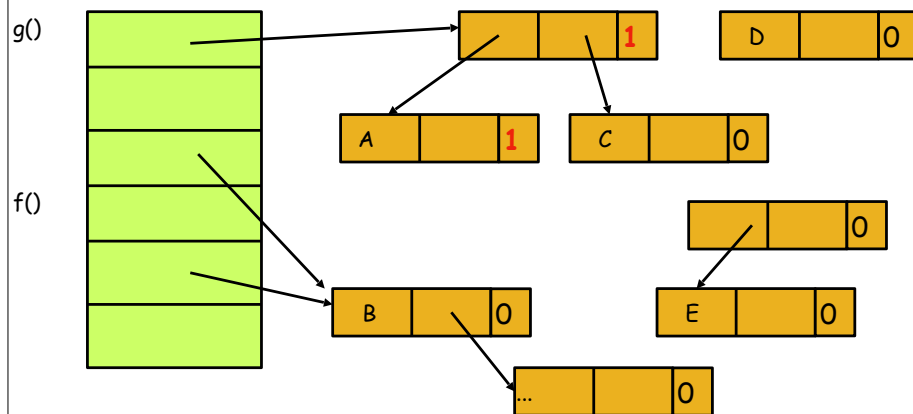
## "mark-sweep" garbage collection



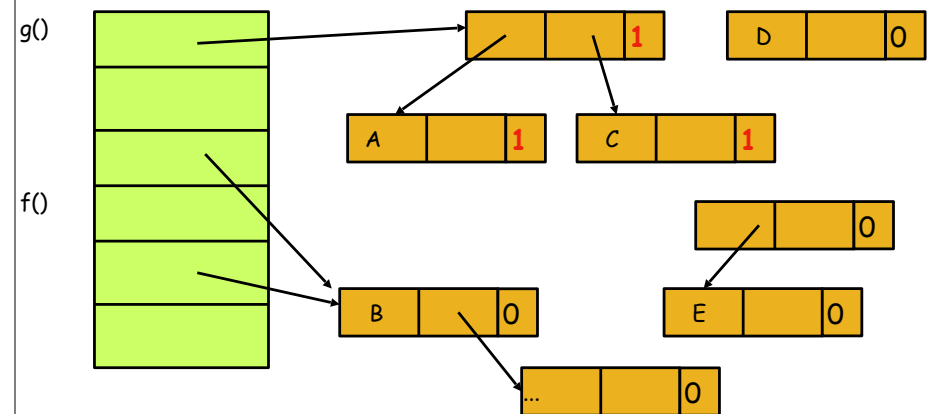
### 1. Mark reachable cells



### 1. Mark reachable cells

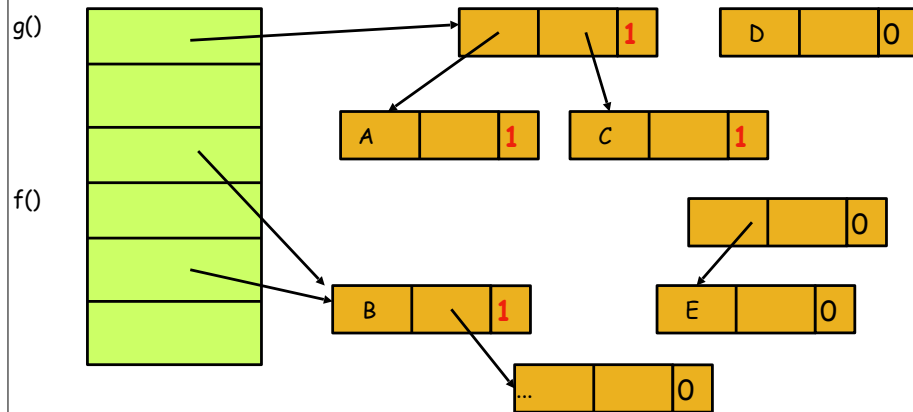


### 1. Mark reachable cells

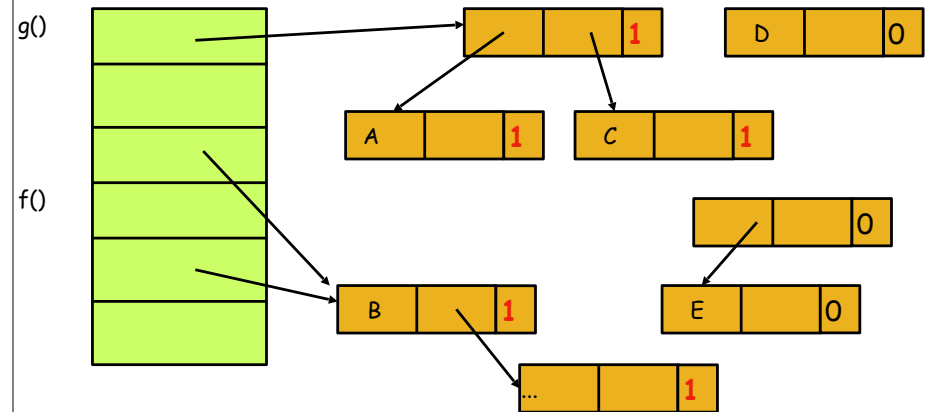




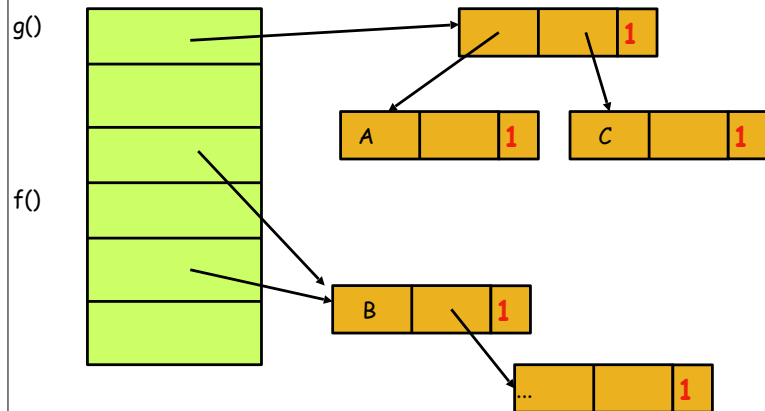
## 1. Mark reachable cells



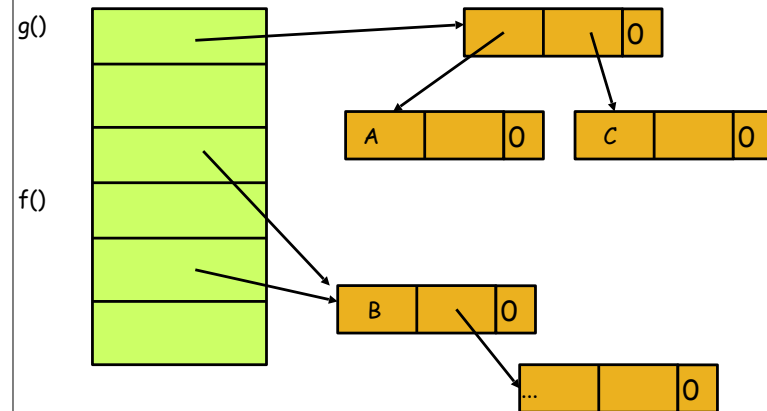
## 1. Mark reachable cells



## 2. Free ("sweep") unreachable cells



## 3. Clear tags



## Computability



## Computability

i.e., what can and cannot  
be done with a computer

A function  $f$  is **computable** if there  
is a program  $P$  that computes  $f$ .

In other words, for **any** (valid) input  $x$ , the  
computation  $P(x)$  **halts** with output  $f(x)$ .

## Computability

example

valid inputs are **integers**

$P(x)$  is:

$$f(x) = x + 5$$

computable?

yes.

## Computability

example

valid inputs are **integers**

$P(x)$  is:

$$f(x) = 5/x$$

computable?

yes, *partially*.

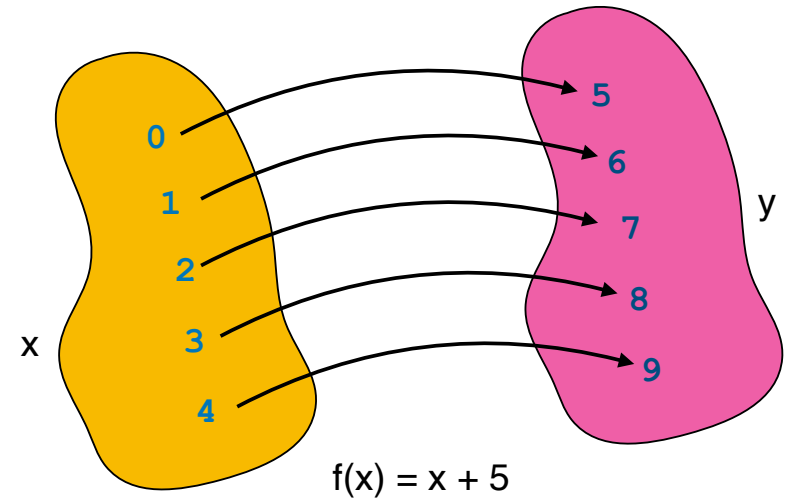
## Total Function

$f: A \rightarrow B$  is a subset  $f \subseteq A \times B$  subject to

1. for every  $a \in A$ , there is a  $b \in B$  with  $\langle a, b \rangle \in f$  **totality**
2. if  $\langle a, b \rangle \in f$  and  $\langle a, c \rangle \in f$  then  $b = c$  **single valued**

e.g,  
 $f(x) = x + 5$

## Intuition: total function



For every element in  $x$ , there is a corresponding element in  $y$ .  $x$  maps to at most one element in  $y$ .

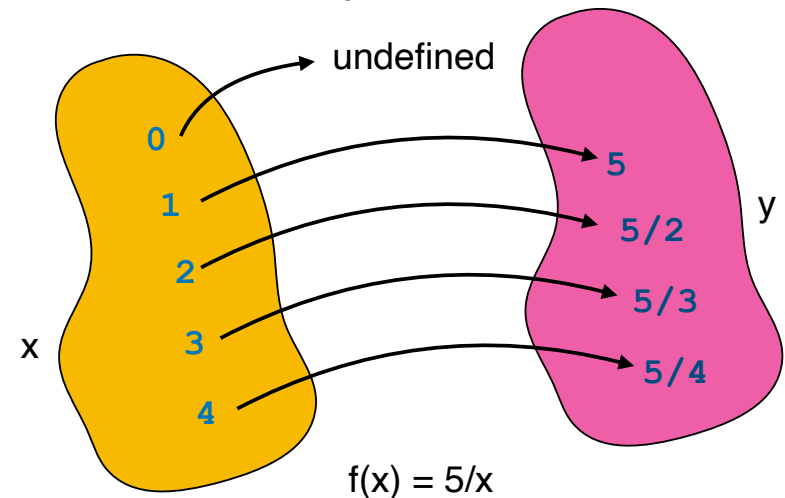
## Partial Function

$f: A \rightarrow B$  is a subset  $f \subseteq A \times B$  subject to

- ~~1. for every  $a \in A$ , there is a  $b \in B$  with  $\langle a, b \rangle \in f$  **totality**~~
2. if  $\langle a, b \rangle \in f$  and  $\langle a, c \rangle \in f$  then  $b = c$  **single valued**

e.g,  
 $f(x) = 5/x$

## Intuition: partial function



$x$  still maps to at most one element in  $y$ , however, there is not a  $y$  for every  $x$ .

The **graph** of a function

$$f(x) = x + 5$$

$$\{ \langle x, x+5 \rangle \mid x \in \mathbb{Z} \}$$

$$\{ \langle x, x+5 \rangle \mid x \text{ is an integer} \}$$

The graph is **not a picture!**

## Recap & Next Class

### Today:

Higher order functions

Computability, part 1

### Next class:

More computability