

Graph Reachability – Part 1

CS 5010 Program Design Paradigms

“Bootcamp”

Lesson 7.1

Goals of this lesson

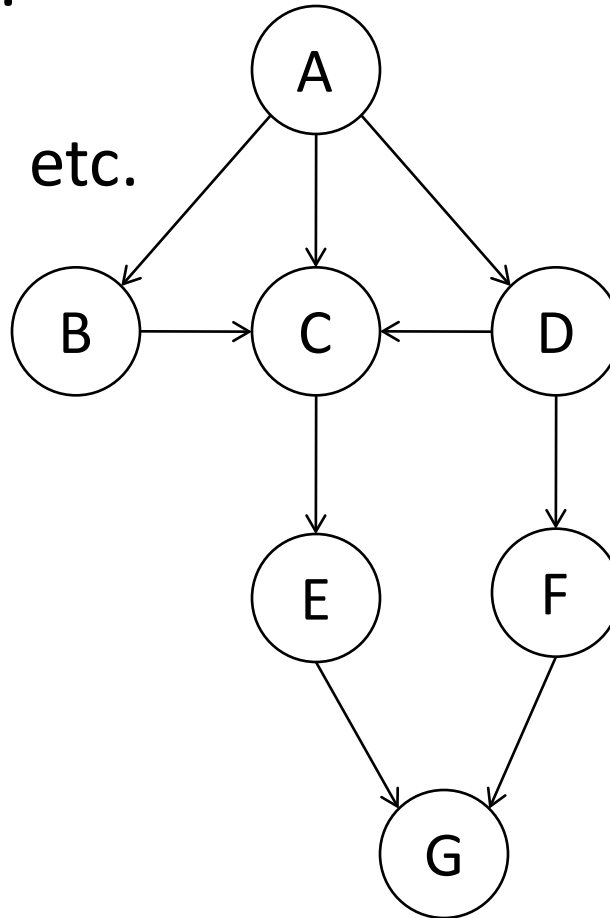
- Learn about an important application of general recursion
- Learn about reachability in a graph
 - think about alternative data representations
- Limits of termination arguments

What's a graph?

nodes: A, B, C, etc.

edges:

(A,B), (A,C),(A,D), etc.



paths:

(A,C,E)

(B,C,E,G)

(A,D,C,E)

(A)

reachability:
is there a path?

from	to	
A	E	Yes
D	G	Yes
C	G	Yes
E	D	No
C	F	No

How to represent a graph as data?

- What information needs to be represented?
 - nodes
 - edges
- What operations do we need to support?
 - **node=? : Node Node -> Boolean**
 - **successors**
: Node Graph -> ListOf<Node>

List-of-Edges Representation

;; A Node is a Symbol

(define-struct edge (from to))

;; An Edge is a (make-edge Node Node)

;; A Graph is a ListOf<Edge>

Graph Reachability via General recursion

- Livecoding: 07-1-graph-reachability.rkt

What if my function doesn't always halt?

You must deliver a termination argument for each function that uses general recursion. This has one of two forms:

1. The function produces a solution for all problems because *[show halting function]* is always non-negative and gets smaller at every recursive call
2. The function does not terminate on some input problems, example: *[put here a specific example that does not terminate]*.

Which of these applies to our example?

Summary

- We've applied General Recursion to an important problem: graph reachability
- We considered the functions we needed to write on graphs in order to choose our representation(s).
- We used general recursion with a *list* of subproblems
- We used list abstractions to make our program easier to write

Graph Reachability – Part 2

CS 5010 Program Design Paradigms
“Bootcamp”
Lesson 7.2

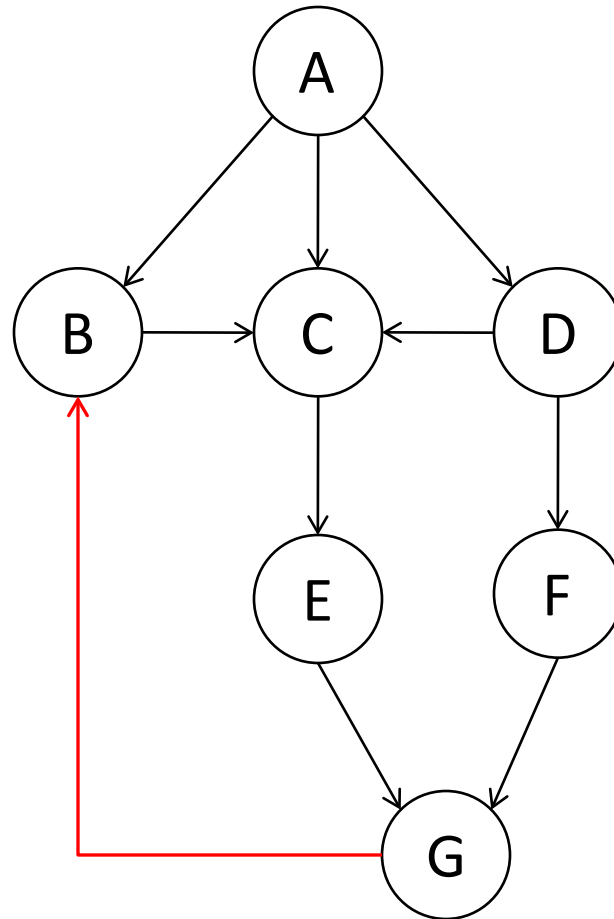
Goals of this lesson

- Learn how to use accumulators with generative recursion
- Example: graph reachability in cyclic graphs

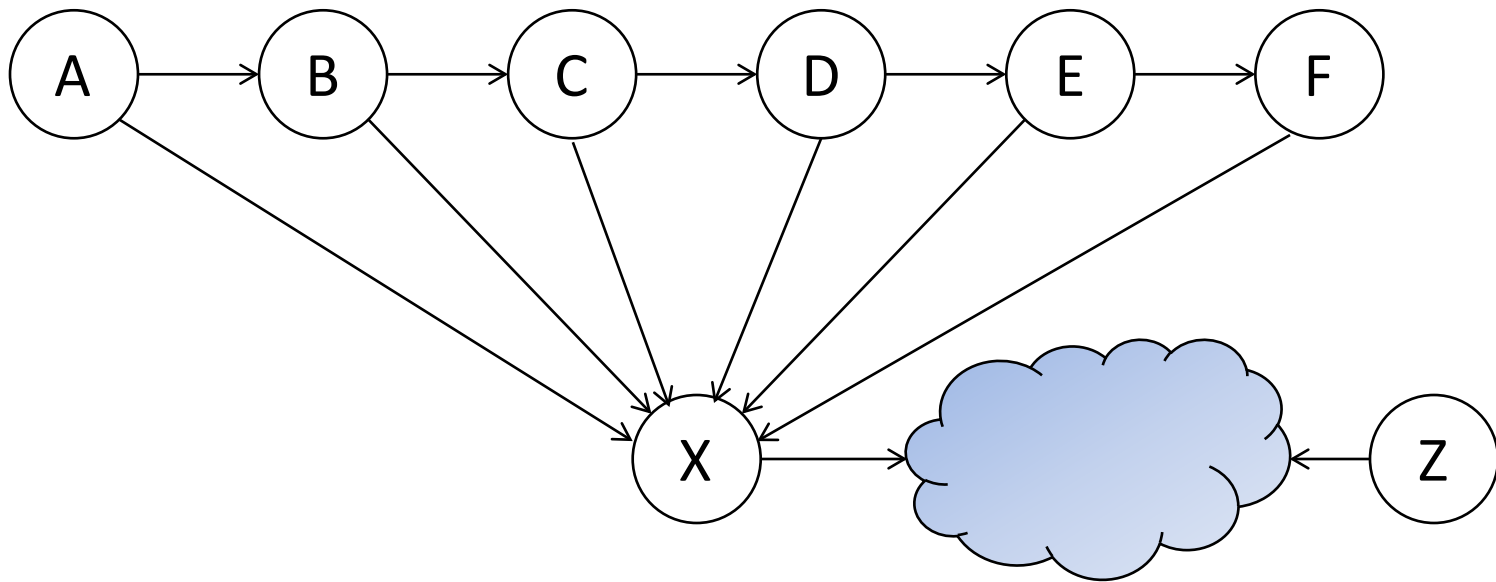
Review: Graph Reachability via generative recursion

```
;; reachable? : Node Node Graph -> Boolean
;; is there a path from src to tgt in g?
(define (reachable? src tgt g)
  (cond
    [(node=? src tgt) true]
    [else
     (ormap
      (lambda (m) (reachable? m tgt g))
      (successors src g))]))
```

A Cyclic Graph



Our function doesn't even work well
for acyclic graphs:



Searching from A to Z, how many times
will we explore paths starting at X?
Let's find out...

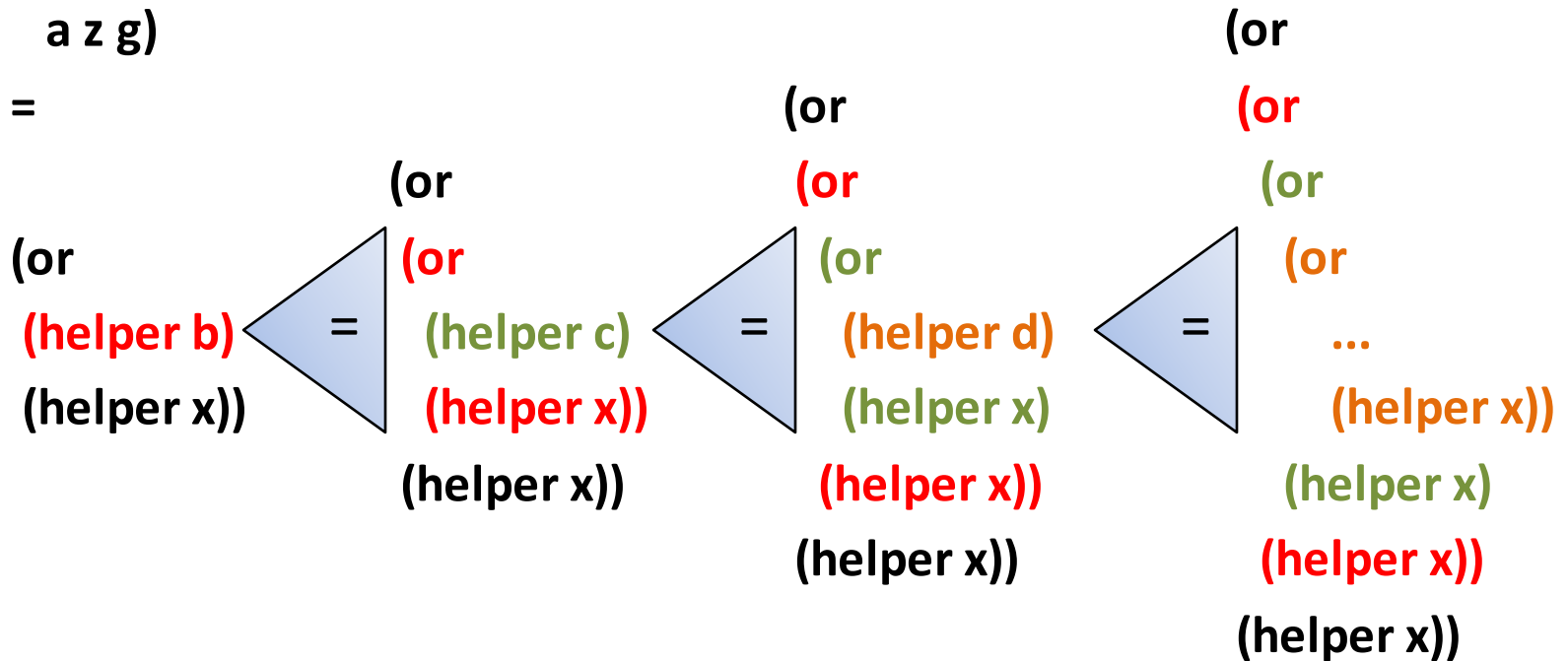
Let's give the lambda a name...

```
;; reachable? : Node Node Graph -> Boolean
;; is there a path from src to tgt in g?
(define (reachable? src tgt g)
  (local
    ((define (helper m) (reachable? m tgt g)))
    (cond
      [(node=? src tgt) true]
      [else
       (ormap
        helper
        (successors src g))])))
```

Let's watch this work

(reachable
a z g)

=



Solving this with an accumulator

- Keep track of the list of nodes to be explored, so you don't add an element to the list that's already there.
- [livecoding]

Can we generalize?

- We're always looking at a list of nodes and asking whether any of them have a path to the target.
- So let's generalize.

Function Definition

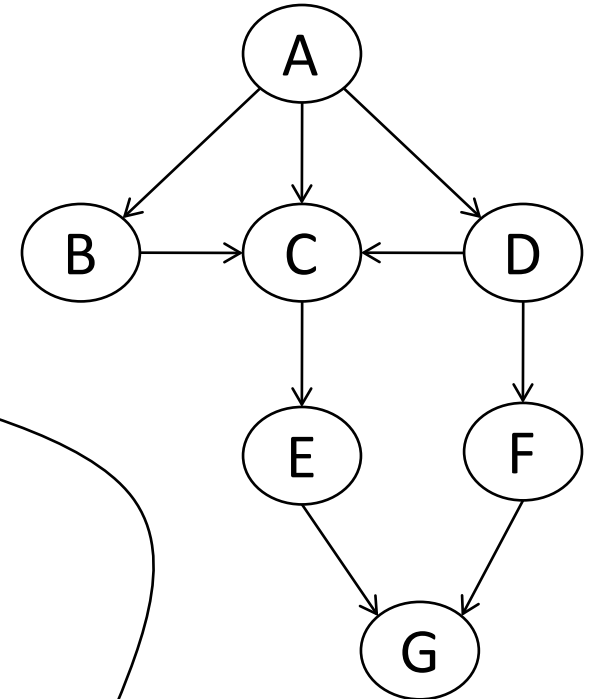
```
;; Graph Node ListOf<Node> -> Boolean
;; is there a path from any node in srcs to the target?
;; strategy: general recursion
(define (path-from-any? g tgt srcs)
  (cond
    [(empty? srcs) false]
    [(node=? (first srcs) tgt) true]
    [else (path-from-any? g tgt
      (append
        (filter
          (lambda (n) (not (member? n srcs)))
          (successors (first srcs) g))
        (rest srcs)))]))
```

Retrieving the original

```
;; Graph Node Node -> Boolean
;; strategy: function composition
(define (path? g src tgt)
  (path-from-any? g tgt (list src)))
```

Does this do the job?

```
(helper (list 'a))  
= (helper (list 'b 'c 'd))  
= (helper (list 'c 'd))  
= (helper (list 'e 'd))  
= (helper (list 'g 'd))  
= (helper (list 'd))  
= (helper (list 'c 'f))
```



B's only successor is C, which is already on the list

Oops, we're back at C again!

Does this do the job?

- No: ☹️
- How can we solve this?
- What information have we lost?
- We've lost track of the nodes that have already been removed from **srcs** (i.e. the nodes that have been explored).

Solution: add another accumulator!

- **seen** keeps track of all the nodes we've already explored. Don't add a node to **srcs** if it's already on **srcs** or on **seen**.

[look at code]

Function Definition

```
;; Graph ListOf<Node> Node ListOf<Node> -> Boolean
;; Is there a path from g from any of the srcs to the tgt,
;; not passing through seen?
;; INVARIANT: srcs and tgt are disjoint
(define (path-from-any? g tgt srcs seen)
  (cond
    [(empty? srcs) false]
    [(node=? (first srcs) tgt) true]
    [else (path-from-any? g tgt
                          (append (rest srcs)
                                   (filter
                                    (lambda (n) (and (not (member? n srcs))
                                                         (not (member? n seen))))
                                    (successors (first srcs) g)))
                          (cons (first srcs) seen))]))
```


Recovering the original

```
;; Graph Node Node -> Boolean  
(define (path? g src tgt)  
  (path-from-any?  
    g tgt (list src) empty))
```

What about the termination argument?

- Yes: every time we call path-from-any?, seen increases by one node.
- Therefore the number of nodes not in seen decreases by one node.
- Hence path-from-any? can't be called more times than there are nodes in the graph!

Halting Measure

*Termination Argument: the halting measure is the number of nodes NOT in **seen**.*

So this will work for cyclic graphs, too!

- termination doesn't depend on there being no cycles.

What about bigger graphs?

- LOE is only one representation of a graph.
- All we need is **node=?** and **successors**.
- So we can represent a graph any way we like, so long as we can write these two functions for that representation.
 - eg: Rubik's cube, tic-tac-toe, etc.

Summary

- We've applied accumulators twice
 - once to keep track of candidates
 - (one kind of context!)
 - once to keep track of nodes we've seen
 - (another kind of context!)
- We recorded in the invariant what each accumulator means
- We checked that at each call, the invariant was true.
- Whew!

General Recursion and Invariants

CS 5010 Program Design Paradigms

“Bootcamp”

Lesson 7.3

Goals

- Learn to use general recursion and invariants to solve problems involving numbers
- Learn to use the linear search abstraction to solve problems when appropriate
- Learn to use accumulators and invariants to collect partial information about numerical problems

Example: function-sum

function-sum :

Nat Nat (Nat Nat -> Number)

-> Number

**Given natural numbers $lo \leq hi$ and
a function f , find**

$SUM\{f(i) \mid lo \leq i \leq hi\}$

Examples/Tests

```
(define-test-suite function-sum-tests
```

```
  (check-equal?
```

```
    (function-sum 1 3 (lambda (i) i))
```

```
    (+ 1 2 3)))
```

```
(check-equal?
```

```
  (function-sum 1 3 (lambda (i) (+ i 10))))
```

```
  (+ 11 12 13)))
```

Function Definition

```
;; Strategy: general recursion + accumulator [sofar]
(define (function-sum lo hi f)
  (local
    ((define (helper i sofar)
      ;; PURPOSE: compute  $\text{SUM}\{f(i) \mid lo \leq i \leq hi\}$ 
      ;; WHERE  $lo \leq i \leq hi$ 
      ;; AND  $sofar = \text{SUM}\{f(i) \mid lo \leq i\}$ 
      (cond
        [(= i hi) sofar]
        [else (helper
                  (+ i 1)
                  (+ sofar (f (+ i 1))))]))
      (helper lo (f lo))))
```

the helper finds the answer for the whole function

invariant

update accumulators to maintain invariant

initialize accumulators to make invariant true

Halting measure

- halting measure is $(-h_i \ i)$
- invariant ensures this is always non-negative
- decreases at every recursive call.

Linear Search

linear-search :

Nat [Nat -> Bool] -> Maybe<Nat>

PURPOSE: given a number N and a predicate p, return the smallest number in $[0, N)$ that satisfies p, or false if there is none.

STRATEGY: generative recursion
+ accumulator

Function Definition

the helper finds the answer for the whole function

```
(define (linear-search N p)
  (local
```

```
    ;; PURPOSE: given a number N and a predicate p, return the
    ;; smallest number in [0,N) that satisfies p, or false if
    ;; there is none.
```

```
    ;; WHERE  $0 \leq i < N$ 
```

```
    ;; AND  $p(j)$  is false for  $0 \leq j < i$ 
```

```
    ;; HALTING MEASURE:  $(N - i)$ 
```

```
    (define (helper i)
```

```
      (cond
```

```
        [(p i) i]
```

```
        [(= i (- N 1)) false]
```

```
        [else (helper (+ 1 i))])))
```

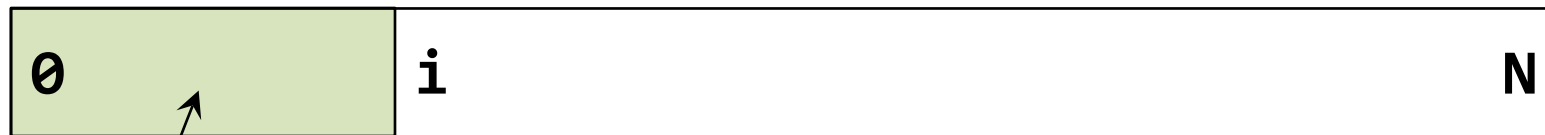
```
  (helper 0)))
```

invariant

update accumulator to maintain invariant

initialize accumulator to make invariant true

A picture of this invariant



INVARIANT: $p(j)$ is false for j in $[1, i)$

$p(j)$ is false in this region

Integer Square Root

int-sqrt : Nat -> Nat

Given n, find z such that

$$z^2 \leq N < (z+1)^2$$

examples:

$$(\text{int-sqrt } 25) = 5$$

$$(\text{int-sqrt } 26) = 5 \dots$$

$$(\text{int-sqrt } 35) = 5$$

$$(\text{int-sqrt } 36) = 6$$

int-sqrt.v0

;; STRATEGY: HOFc

(define (int-sqrt.v0 n)

(linear-search n

(lambda (z) (< n (* (+ z 1) (+ z 1))))))

int-sqrt.v1

```
(define (int-sqrt.v1 n)
  (local
    ((define (helper z)
      ;; PURPOSE: Compute int-sqrt(n)
      ;; WHERE  $z^2 \leq n$ 
      ;; HALTING MEASURE  $(- n z)$ 
      (cond
        [(< n (sqr (+ z 1))) z]
        [else (helper (+ z 1))])))
    (helper 0)))
```

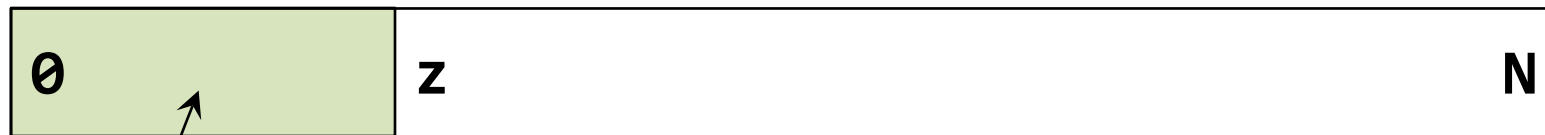
the helper finds the answer
for the whole function

invariant guarantees
that the halting
measure is non-
negative

update accumulator to
maintain invariant

initialize accumulator
to make invariant true

A picture of this invariant



INVARIANT: $z^2 \leq n$

all these numbers also have squares that are $\leq N$

Improving this code

don't like to do `sqr` at every step, so let's keep the value of

`(sqr (+ z 1))`

in an accumulator, which we'll call `u`.

Compute new value of `u` as follows:

$$z' = (z+1)$$

$$\begin{aligned} u' &= (z'+1)*(z'+1) \\ &= ((z+1)+1)*((z+1)+1) \\ &= (z+1)^2 + 2(z+1) + 1 \\ &= u + 2z + 3 \end{aligned}$$

Function Definition

```
(define (int-sqrt.v2 n)
  (local
    ((define (helper z u)
      ;; PURPOSE: Compute int-sqrt(n)
      ;; WHERE  $z^2 \leq n$ 
      ;; AND  $u = (z+1)^2$ 
      ;; HALTING MEASURE  $(- n z)$ 
      (cond
        [(< n u) z]
        [else (helper
                (+ 1 z)
                (+ u (* 2 z) 3))]))
      (helper 0 1)))
```

the helper finds the answer
for the whole function

$u = (z+1)^2$

update accumulators to
maintain invariant

initialize accumulators
to make invariant true

Let's do it one more time

Add invariant: $v = 2*z+3$

$$z' = z+1$$

$$v' = 2*z' + 3$$

$$= 2*(z+1) + 3$$


$$= 2*z + 2 + 3$$

$$= (2*z + 3) + 2$$

$$= v + 2$$

Function Definition

```
(define (int-sqrt.v3 n)
  (local
    ((define (helper z u v)
      ;; PURPOSE: Compute int-sqrt(n)
      ;; WHERE  $z^2 \leq n$ 
      ;; AND  $u = (z+1)^2$ 
      ;; AND  $v = 2z+3$ 
      ;; HALTING MEASURE  $(- n z)$ 
      (cond
        [(< n u) z]
        [else (helper
                  (+ 1 z)
                  (+ u v)
                  (+ v 2))]))))
    (helper 0 1 3)))
```


$$u = (z+1)^2$$


$$v = 2z+3$$

Summary

- We've seen how generative recursion can deal with problems involving numerical values
- We've seen how accumulators and invariants can help avoid recalculating expensive values
- We've seen how invariants can be an invaluable aid in understanding programs

Testing and Debugging

CS 5010 Program Design Paradigms

“Bootcamp”

Lesson 7.4

Goals of Testing

- *Acceptance Testing*: Make sure program works on the given examples or use cases
- *Requirements Testing*: Make sure program works as intended on other examples
- *Regression Testing*: Make sure that a change hasn't broken anything.
- *Stress Testing*: How does the program perform for large inputs, heavy loads, etc.?
- *Usability Testing*: Is the program usable by its intended audience?

Acceptance Testing

- The requirements probably give some examples. Be sure to test them!
- Sometimes the requirements are more complicated:

Test against requirements

- Your task is to create a traffic simulation. The simulation involves vehicles on a street. You will create a top view of the street. Your vehicles are cars, trucks, and tractor-trailers.
- The street is 1000 pixels long. The street runs horizontally across the middle of your canvas. This is a two-way street. In one lane, vehicles enter at the left end of the street and proceed to the right end, where they travel off the screen. In the other lane, vehicles enter at the right and proceed to the left. There is a gap of 5 pixels between the lanes.
- We don't want the vehicles to collide, so a vehicle may not get closer than 5 pixels to the rear bumper of the vehicle ahead of it.
- A car is a rectangle 20 pixels long and 10 pixels wide. A truck is 30 pixels long and 10 pixels wide, and a tractor-trailer is 40 pixels long and 10 pixels wide
- Each car travels at 8 pixels per tick on the first half of the street, and 4 pixels per tick for the second half of the street, unless it needs to move more slowly to avoid collisions.
- Trucks travel at a maximum speed of 8 pixels per tick, the same as cars.
- Tractor-trailers also travel at a maximum speed of 8 pixels per tick. But, as in the real world, they accelerate slowly from a stop. Their maximum acceleration is 2 pixels/tick². That is, their maximum speed after 1 tick is 2 pixels/tick; after 2 ticks it is 4 pixels/tick, after 3 ticks, it is 6 pixels/tick, etc.

Requirements Testing

- You'll need to turn these into testable examples.
- Example: go down the list of vehicles and compare the position of the front bumper of each car with the position of the rear bumper of the car ahead of it.
 - except for the first car, of course 😊

Kinds of Testing

- *Black-box testing*: Tests where we don't know anything about the internals of the program
- *White-box testing*: Tests where we take advantage of what we know about the program or the requirements
 - Example: our tests for f2c took advantage of the fact that f2c was a linear relationship: if the program worked for two values, we can be confident that it will work for others
 - Except for overflow, etc. 😊
- We will do mostly white-box testing.

Test Coverage

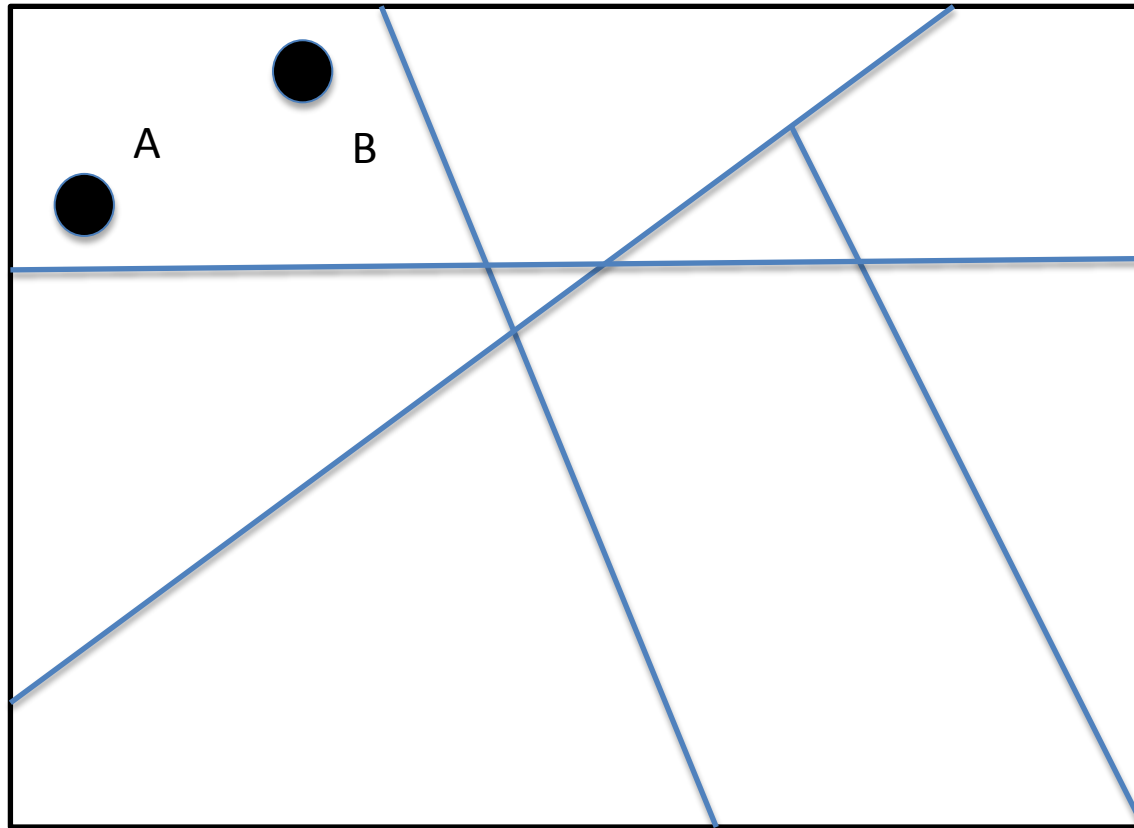
- How much of the possible behaviors have we tested?
- Want every line in the program exercised. This is called *100% expression coverage*.
- This is our **minimum** testing requirement.

Equivalence Partitioning

- Possible arguments to your function typically fall into classes for which the program yields similar results.
- Example: f2c had only 1 partition.
- Example: ball-after-mouse depends on
 - Which mouse event we're dealing with
 - Whether the mouse event is inside or outside the ball
 - Whether the ball is selected
- So we need $3 \times 2 \times 2 = 12$ tests to cover all these combinations.

Equivalence Partitioning

Regions of similar behavior



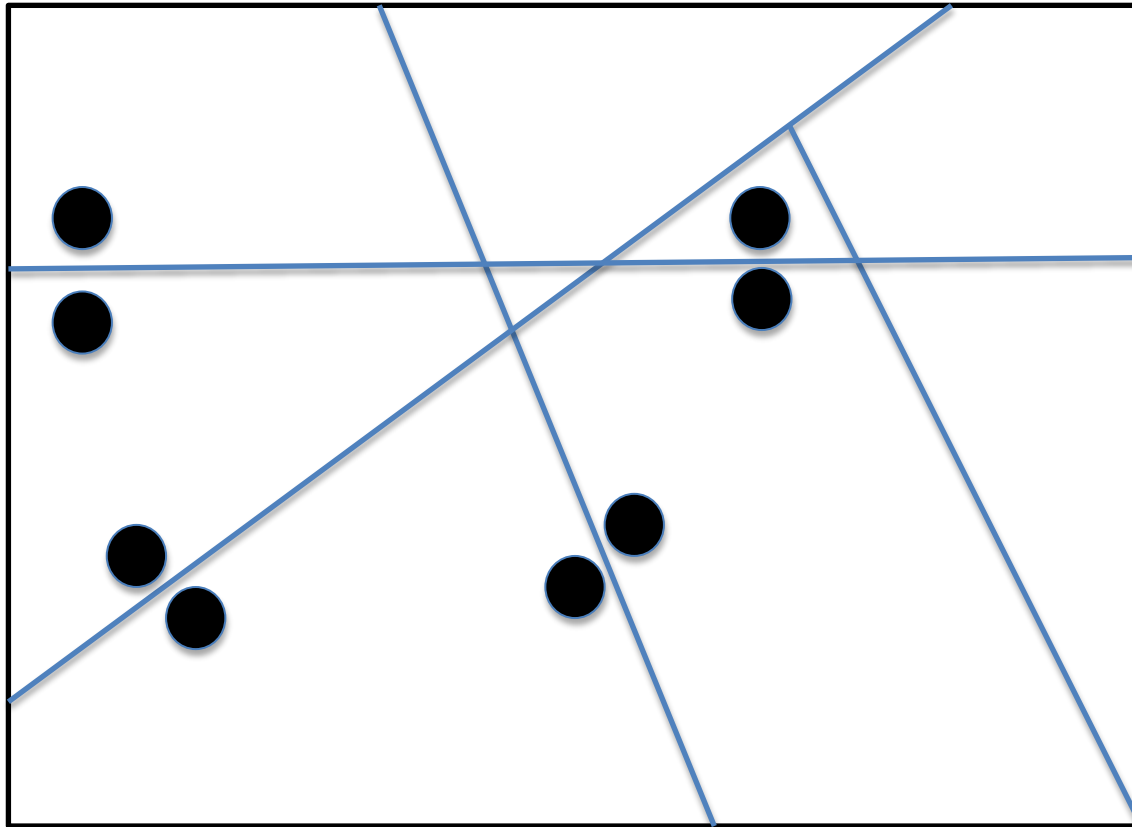
If the program works for input A, it will probably work for input B

Boundary Testing

- Want to make sure that the boundaries between these equivalence classes are in the right place.
- Example: int-square-root
 - Need to test 24, 25, 26

Boundary Testing

Regions of similar behavior



Mechanics of Testing

- Near the top of your file, write
`(require rackunit)`
`(require rackunit/text-ui)`
to load our testing framework.
- Tests live in the file with the code
- That way they get run every time the code is loaded
 - This accomplishes regression testing.

Choosing test cases

- We'll concentrate on the equivalence partitioning.
- Choose mnemonic names for the input and output values in each partition.

Example

```
(define ball-unselected (make-ball 20 30 10 false))
```

```
(define ball-selected (make-ball 20 30 10 true))
```

```
(define point-inside-x 22)
```

```
(define point-inside-y 28)
```

```
(define point-outside-x 31) ;; 20+10 = 30, so 31 is outside
```

```
(define point-outside-y 19) ;; 30-10 = 20, so 19 is outside
```

```
(define ball-moved-to-point-inside
```

```
  (make-ball point-inside-x point-inside-y 10 true))
```

```
(define ball-moved-to-point-outside
```

```
  (make-ball point-outside-x point-outside-y 10 true))
```

Example

```
(check-equal?  
  (ball-after-mouse  
    ball-unselected  
    point-inside-x point-inside-y  
    "button-down")  
  ball-selected  
  "button-down inside the ball should select it")  
(check-equal?  
  (ball-after-mouse  
    ball-unselected  
    point-outside-x point-outside-y  
    "button-down")  
  ball-unselected  
  "button-down outside the ball should leave it unchanged")
```

Turning Your Tests into Test Suites

- If you have many tests, or several groups of tests, turn them into test suites using **define-test-suite**, and run them using **run-tests**.
- This gets you:
 - Better visual reporting
 - More control over when tests are run
- Try running
ball-after-mouse-with-tests.rkt

Using Tests

- Run your program with its tests
- Debug so that all your tests pass
- If you didn't achieve 100% expression coverage, go back and add more tests.

Testing Pitfalls

- DON'T just paste in the actual results of your function.
- Some functions may have more than one correct answer;
 - your tests should accept *any* correct answer, not just the one your solution happens to produce
 - use (**check pred**)

Testing Pitfalls (2)

- Avoid coincidences in your tests

- Bad:

```
(check-equal?  
  (book-profit-margin  
    (make-book "Little Lisper" "Friedman" 2.00 4.00))  
  2.00)
```

- Better:

```
(check-equal?  
  (book-profit-margin  
    (make-book "Little Lisper" "Friedman" 2.00 5.00))  
  3.00)
```

What could go wrong?

- Program fails to load
 - unbalanced parens? The unmatched paren is highlighted in the interaction window.
 - missing function?
 - forgot to write definition
 - misspelled function name
 - forgot to **require** the library module
 - misspelled library name

What could go wrong? (2)

- Test fails
 - identify the test that failed
 - did the test ask for the right answer?
 - if not, fix the test
 - DON'T just paste in the actual results of your function.
 - if test is right:
 - did your function call the right helper?
 - yes: test the helper
 - no: test the predicate

Did your function call the right helper?

Code:

```
(define (ball-after-mouse b mx my mev)
  (cond
    [(mouse=? mev "button-down")
     (ball-after-button-down b mx my)]
    [(mouse=? mev "drag") (ball-after-drag b mx my)]
    [(mouse=? mev "button-up") (ball-after-button-up b mx my)]
    [else b]))
```

Failing Test:

```
(check-equal?
  (ball-after-mouse
    ball-unselected point-inside-x point-inside-y
    "button-down")
  ball-selected)
```

Did your function call the right helper? (2)

```
(check-equal?  
  (ball-after-mouse  
    ball-unselected  
    point-inside-x point-inside-y  
    "button-down")  
  (ball-after-button-down  
    ball-unselected  
    point-inside-x point-inside-y))
```

Test succeeds: problem is in **ball-after-mouse**

Test fails: problem is in **ball-after-button-down**

Tracking down your bug

```
(define (ball-after-button-down b mx my)
  (if (inside-ball? mx my b)
      (ball-make-selected b)
      b))
```

```
(check-equal?
  (ball-after-button-down
    ball-unselected
    point-inside-x point-inside-y)
  (ball-make-selected ball-unselected))
```

Test succeeds: problem is in **ball-make-selected**

Test fails: problem is in **inside-ball?**

Keep your bug from re-appearing

- Leave the extra tests in your file
- Regression testing FTW!
- That way if the bug reappears you will have the detective work all set up.

What else could go wrong?

- You could have called your help function with the wrong arguments
- You could get an error rather than a failure
 - eg: "can't apply string=? to 1"
 - You've probably violated your help function's contract
 - Use same techniques as before to locate the problem

Disclaimer

- Our presentation has been specific to Racket and to this course, but the ideas and techniques are adaptable to other settings and other languages.
- Your employer may have different conventions for managing tests.
- If your employer does not have conventions for systematic testing, you should urge him (or her) to introduce one.

How to choose a design strategy

CS 5010 Program Design Paradigms

“Bootcamp”

Lesson 7.5

Design functions systematically

- Follow the recipe!
- Structure of data tells you the structure of the program...
 - Or at least gives you good hints!
 - Data Definition → Template → Code
 - The data definitions structure your wishlist, too.
- Examples make you clarify your thinking
 - Be sure to cover corner cases

The Design Recipe

The Design Recipe

1. Information Analysis and Data Design
2. Contract and Purpose Statement
3. Examples
4. Design Strategy
5. Code
6. Tests

Four Important Design Strategies

- Domain Knowledge
- Function Composition
- Structural Decomposition
- General Recursion

Domain Knowledge

- Inputs are (usually) *atomic data*;
- To design the function's entire body, the programmer uses knowledge about some "real-world" domain
- There's always domain knowledge going on; we only call it "domain knowledge" when there's nothing else going on.

Function Composition

- the function doesn't decompose its arguments
- typical shape:

```
(define (f x y z) (g x (h y z)))
```

where g and h are defined elsewhere

- A simple "if" is OK, so is calling a fold, map, etc.
- You may need a new data def for intermediate results. These will be recorded in the contracts for g and h.
- There's always function composition going on; we only call it "function composition" when there's nothing else going on.

Structural Decomposition

- inputs are always *structured* (compound or mixed) *data*;
- the function's organization is based on the *data definition* for one (or more) of the function's parameters
- one function per interconnected data definition
- *recursions in the functions follow recursions in the data definitions.*
- are some of the decisions or transformations complicated? Then introduce helper functions
 - There's a reason for that ugly little thing— document it and test it.

The Recursion Recipe

Recursion and Self-Reference

Represent arbitrary-sized information using a *self-referential* (or *recursive*) data definition.

Self-reference in the data definition leads to self-reference in the template

Self-reference in the template leads to self-reference in the code.

General Recursion

- Inputs encode problems from a *class of problems*
- Recursion solves *a related problem* from the same class (“*subgoal*” or “*subproblem*”)
 - requires ad hoc insight to find a useful subproblem.
- Termination argument is required:
 - *how are each of the subproblems easier* than the original problem?

General Recursion vs. Structural Decomposition

- Structural decomposition is a special case: it's a standard recipe for finding subproblems that are guaranteed to be easier.
 - A field is always smaller than the structure it's contained in.
- In the definition of function **f** :
 - `(... (f (rest lst)))` is structural
 - `(... (f (... (rest lst)))` is general recursion

Variations

- Structural decomposition using abstraction
 - Use of an abstraction (foldr, map, etc.)
 - if you have your own data structure, you may want to write your own.
- With accumulators
 - slogan: accumulators represent context. *We accumulate context, not answers.*
 - *accumulator invariant* describes how the accumulator represents the context.

Choosing a Design Strategy

- Trivial problem: trivial solution
- Reduce the problem to one or more simpler problems:
 - Reconstruct solution to your problems from the solutions to the simpler problems

Finding the Simpler Subproblems

- Independent/sequential pieces: functional composition
 - Test: can you give meaningful names & purpose statements to the subproblems?
- Simpler instance of same problem:
 - Structural decomposition (maybe w/ accumulator)
 - General recursion: when none of the above works.

Another way of thinking about this

- Always try to complete the design with one of the 4 basic strategies above; try in the order above.
 - are the parameters atomic?
→ *exploit the domain*
 - does the problem statement suggest separate tasks?
→ *functional composition*
 - is there structure to the 'main' argument?
→ *structural decomposition*
 - can you find some other recursive structure to your problem?
→ *general recursion*

Next Up: OOP

- We'll see how to apply our techniques in the context of object-oriented programming
- Most will be the same, some will be tailored to fit OOP paradigm
- Learn about OOP vs functional approach
- Learn about the true purpose of state.