(define (find-route orig dest G)
  (cond
    ; (symbol=? orig dest) (list dest)
    [(symbol=? orig dest) (list dest)]
    [else
      (local
        [(define nhrs (neighbours orig G))
         (define route (find-route/list nhrs dest G))]
         (cond
           [(false? route) false]
           [else (cons orig route)]))))
;; (find-route/list los dest G) produces route from an element of los to dest in G, if one exists
;; find-route/list: (listof Node) Node Graph ->
;;               (anyof (listof Node) false)
(define (find-route/list los dest G)
  (cond
   [(empty? los) false]
   [else
    (local
     [(define route (find-route (first los) dest G))]
     (cond
      [(false? route)
       (find-route/list (rest los) dest G)]
      [else route]))]))
CQ 2

In find-route and find-route/list, we have used false as a "sentinel value" indicating that no route exists. Which of the following is not an acceptable choice of a sentinel for these functions?

A. true
B. 0
C. empty
D. 'some-symbol
E. All of these are acceptable as sentinel values.
In `find-route` and `find-route/list`, we have used `false` as a “sentinel value” indicating that no route exists. Which of the following is not an acceptable choice of a sentinel for these functions?

A. `true`
B. `0`
C. `empty`
D. `'some-symbol`
E. All of these are acceptable as sentinel values.
If we wish to trace `find-route`, trying to do a linear trace would be very long, both in terms of steps and the size of each step. Our traces also are listed as a linear sequence of steps, but the computation in `find-route` is better visualized as a tree.

We will use an alternate visualization of the potential computation (which could be shortened if a route is found).

The next slide contains the trace tree. We have omitted the arguments `dest` and `G` which never change.
(find-route 'A ...)  
(find-route/list '(C D E) ...)  

(find-route 'C ...)  
(find-route/list empty ...)  
(find-route 'D ...)  
(find-route/list '(F J) ...)  
(find-route 'E ...)  
(find-route/list '(K))  

(find-route 'F ...)  
(find-route/list '(K H) ...)  
(find-route 'J ...)  
(find-route/list '(H) ...)  
(find-route 'K ...)  
(find-route/list empty ...)  

(find-route 'K ...)  
(find-route/list empty ...)  
(find-route 'H ...)  
(find-route/list empty ...)  
(find-route 'H ...)  
(find-route/list empty ...)

A  B  C  D  E  F  K  H  J
Backtracking in implicit graphs

The only places where real computation is done on the graph is in comparing the origin to the destination and in the neighbours function.

Backtracking can be used without having the entire graph available.

Example: nodes represent configurations of a board game (e.g. peg solitaire), edges represent legal moves.

The graph is acyclic if no configuration can occur twice in a game.
In another example, nodes could represent partial solutions of some problem (e.g. a Sudoku puzzle, or the puzzle of putting eight mutually non-attacking queens on a chessboard).

Edges represent ways in which one additional element can be added to a solution.

The graph is naturally acyclic, since a new element is added with every edge.
The find-route functions for implicit backtracking look very similar to those we have developed.

The neighbours function must now generate the set of neighbours of a node based on some description of that node (e.g. the placement of pieces in a game).

This allows backtracking in situations where it would be inefficient to generate and store the entire graph as data.
Backtracking forms the basis of many artificial intelligence programs, though they generally add heuristics to determine which neighbour to explore first, or which ones to skip because they appear unpromising.
Termination
Termination of `find-route` (no cycles)

In a directed acyclic graph, any route with a given origin will recurse on its (finite number) of neighbours by way of `find-route/list`. The origin will never appear in this call or any subsequent calls to `find-route`: if it did, we would have a cycle in our DAG.

Thus, the origin will never be explored in any later call, and thus the sub-problem is smaller. Eventually, we will reach a sub-problem of size 0 (when all reachable nodes are treated as the origin).

Thus `find-route` always terminates for directed acyclic graphs.
Non-termination of find-route(cycles)

It is possible that find-route may not terminate if there is a cycle in the graph.

Consider the graph '(((A (B)) (B (C)) (C (A)) (D ()))'). What if we try to find a route from A to D?
(find-route 'A)  (find-route/list '(B))
=> (find-route 'B)  (find-route/list '(C))
=> (find-route 'C)  (find-route/list '(A))
=> (find-route 'A)  (find-route/list '(B))
=> ...
Improving find-route
Improving find-route

We can use accumulative recursion to solve the problem of find-route possibly not terminating if there are cycles in the graph.

To make backtracking work in the presence of cycles, we need a way of remembering what nodes have been visited (along a given path).

Our accumulator will be a list of visited nodes. We must avoid visiting a node twice.

The simplest way to do this is to add a check in find-route/list.
;; find-route/list:
;;  (listof Node) Node Graph (listof Node) ->
;;                     (anyof (listof Node) false)

(define (find-route/list los dest G visited)
  (cond
   [(empty? los) false]
   [(member? (first los) visited)
    (find-route/list (rest los) dest G visited)]
   [else (local
      [(define route
        (find-route/acc (first los) dest G visited))]
   (cond
    [(false? route)
     (find-route/list (rest los) dest G visited)]
    [else route]))]))
The code for `find-route/list` does not add anything to the accumulator (though it uses the accumulator).

Adding to the accumulator is done in `find-route/acc` which applies `find-route/list` to the list of neighbours of some origin node.

That origin node must be added to the accumulator passed as an argument to `find-route/list`. 
;;; find-route/acc: Node Node Graph (listof Node) ->
  (anyof (listof Node) false)
(define (find-route/acc orig dest G visited)
  (cond
   [(symbol=? orig dest) (list dest)]
   [else
    (local
     [(define nbrs (neighbours orig G))
      (define route (find-route/list nbrs dest G
                      (cons orig visited)))
      (cond
       [(false? route) false]
       [else (cons orig route))])))])
Revisiting our example
(find-route/acc 'A … empty)
(find-route/list '(C D E) … '(A))

(find-route 'C … '(A))
(find-route/acc empty …)
(find-route/list empty …)

(find-route 'D … '(A))
(find-route/list '(F J) … '(D A))

(find-route 'E … '(A))
(find-route/list '(K) … '(E A))

(find-route 'F … '(D A))
(find-route/list '(K H) … '(F D A))

(find-route 'J … '(D A))
(find-route/list '(H) … '(J D A))

(find-route 'K … '(E A))
(find-route/list empty …)

(find-route/list empty …)

(find-route 'K … '(F D A))
(find-route/list empty …)

(find-route 'H … '(F D A))
(find-route/list empty …)

(find-route 'H … '(J D A))
(find-route/list empty …)
This example has no cycles, so the trace only convinces us that we have not broken the function on acyclic graphs, and shows us how the accumulator is working.

But it also works on graphs with cycles.

The accumulator ensures that the depth of recursion is no greater than the number of nodes in the graph, so find-route terminates.
(find-route/acc 'A empty)
=> (find-route-list '(B) '(A))
=> (find-route/acc 'B '(A))
=> (find-route-list '(C) '(B A))
=> (find-route/acc 'C '(B A))
=> (find-route-list '(A) '(C B A))
no further recursive calls
CQ 3:

Which of the following would be possible values of visited after (find-route/acc 'A 'Z G '()) has found node 'Z, given graph G below?

A. '(A Z)
B. '(W A)
C. '(Y X P A)
D. '(Y W Y P X A)
E. none of the above
CQ 3:

Which of the following would be possible values of visited after (find-route/acc 'A 'Z G ') has found node 'Z, given graph G below?

A. '(A Z)
B. '(W A)
C. '(Y X P A)
D. '(Y W Y P X A)
E. none of the above
In practice, we would write a wrapper function for users which would avoid their having to specify the initial value of the accumulator.

Backtracking now works on graphs with cycles, but it can be inefficient, even if the graph has no cycles.

If there is no path from the origin to the destination, then find-route will explore every path from the origin, and there could be an exponential number of them.
If there are $d$ diamonds, then there are $3d + 2$ nodes in the graph, but $2d$ paths from A to Y, all of which will be explored.
Making **find-route/acc** efficient

Applying **find-route/acc** to origin A results in **find-route/list** being applied to `(B1  B2)`, and then **find-route/acc** being applied to origin B1.

There is no route from B1 to Z, so this will produce `false`, but in the process, it will visit all the other nodes of the graph except B2 and Z.

**find-route/list** will then apply **find-route/acc** to B2, which will visit all the same nodes.
When find-route/list is applied to the list of nodes los, it first applies find-route/acc to (first los) and then, if that fails, it applies itself to (rest los).

To avoid revisiting nodes, the failed computation should pass the list of nodes it has seen on to the next computation.

It will do this by returning the list of visited nodes instead of false as the sentinel value. However, we must be able to distinguish this list from a successfully found route (also a list of nodes).
Remembering what the list of nodes represents

We will make a new type that will store both the list of nodes (either those nodes which have been visited, or the nodes along the successful path) as well as a Boolean value indicating what the list of nodes represents.

```
(define-struct routepair (valid? nodes))
;; a RoutePair is a (make-routepair Bool (listof Node))
```
(define (find-route/list los dest G visited)
  (cond
   [(empty? los) (make-routepair false visited)]
   [(member? (first los) visited)
    (find-route/list (rest los) dest G visited)]
   [else
    (local
     [(define route (find-route/acc
                     (first los) dest G visited))]
      (cond
       [(not (routepair-valid? route))
        (find-route/list (rest los) dest G
                         (routepair-nodes route))]
       [else route]))])))
(define (find-route/acc orig dest G visited)
  (cond
    [(symbol=? orig dest)
      (make-routepair true (list dest))]
    [else
      (local
        [(define nbrs (neighbours orig G))
         (define route (find-route/list nbrs dest G
                       (cons orig visited)))]
        (cond
          [(false? (routepair-valid? route)) route]
          [else (make-routepair true
                       (cons orig (routepair-nodes route)))]))])
)
;; find-route: Node Node Node Graph ->
;;                               (anyof (listof Node) false)
(define (find-route orig dest G)
  (local
    [(define route (find-route/acc orig dest G empty))]
  (cond
    [(route-pair-valid-path? route) (route-pair-nodes route)]
    [else false])) )
With these changes, \texttt{find-route} runs much faster on the diamond graph.

In future courses we will see how to make \texttt{find-route} even more efficient and how to formalize our analyses.

Knowledge of efficient algorithms, and the data structures that they utilize, is an essential part of being able to deal with large amounts of real-world data.

These topics are studied in CS 240 and CS 341 (for majors) and CS 234 (for non-majors).
Goals of this module

• You should understand directed graphs and their representation in Racket.
• You should be able to write functions which consume graphs and compute desired values.
• You should understand and be able to implement backtracking on explicit and implicit graphs.
• You should understand the performance differences in the various versions of find-route.