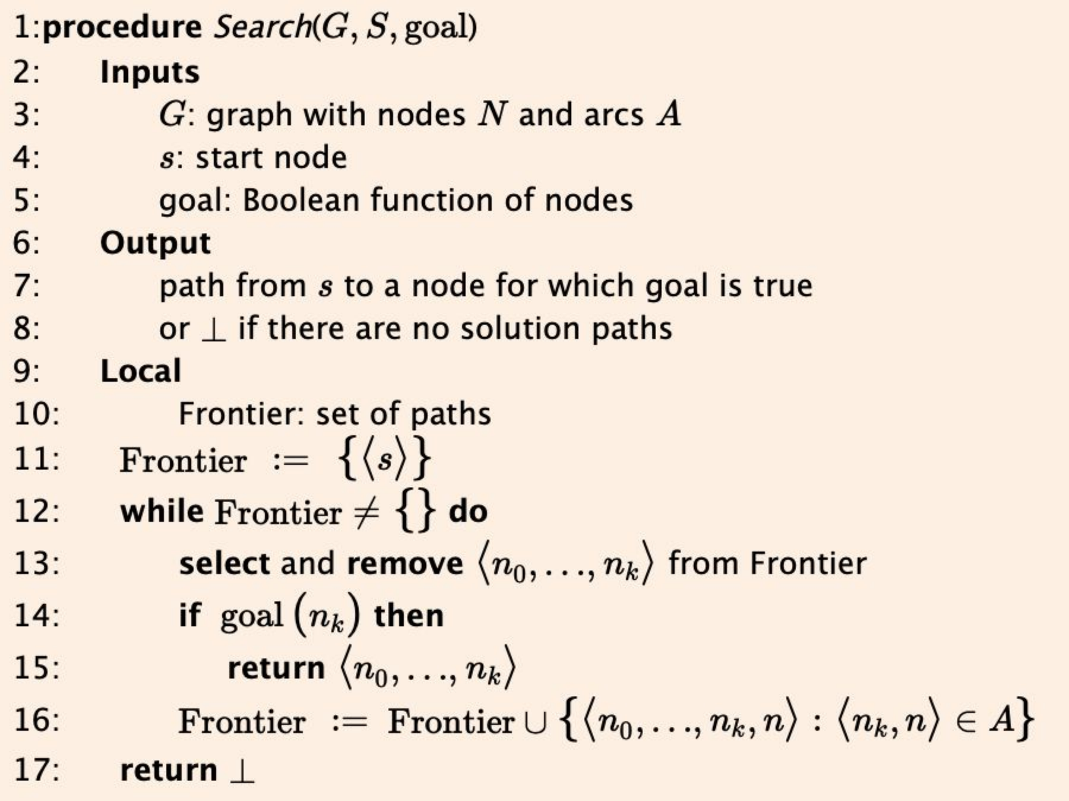
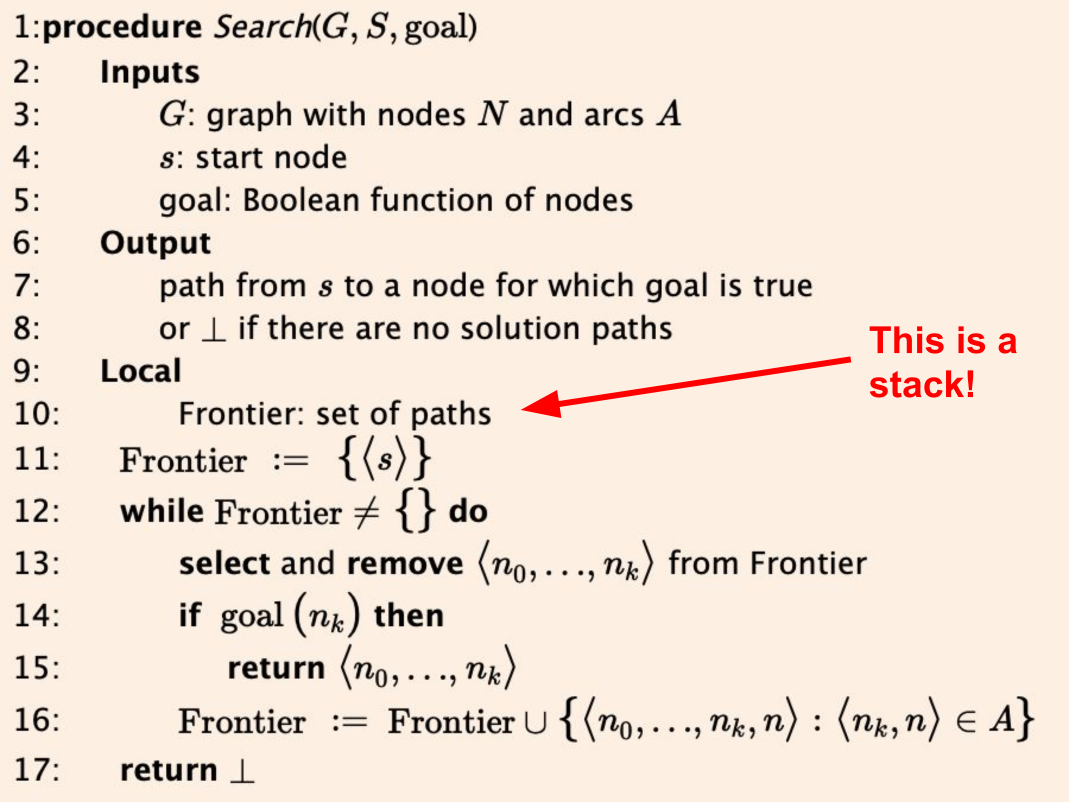
Search Algorithms

Generic Search Covered in Class



```
procedure Search($G, S, \mathrm{goal}$)
    Inputs
        $G$: graph with nodes $N$ and arcs $A$
        $s$: start node
        goal: Boolean function of nodes
    Output
        path from $s$ to a node for which goal is true
        or $\perp$ if there are no solution paths
    Local
        Frontier: set of paths
    $\mathrm{Frontier} := \{\langle s \rangle\}$
    while $\mathrm{Frontier} \neq \{\}$ do
        select and remove $\langle n_0, \ldots, n_k \rangle$ from Frontier
        if $\mathrm{goal}(n_k)$ then
            return $\langle n_0, \ldots, n_k \rangle$
        $\mathrm{Frontier} := \mathrm{Frontier} \cup \{\langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A\}$
    return $\perp$
```

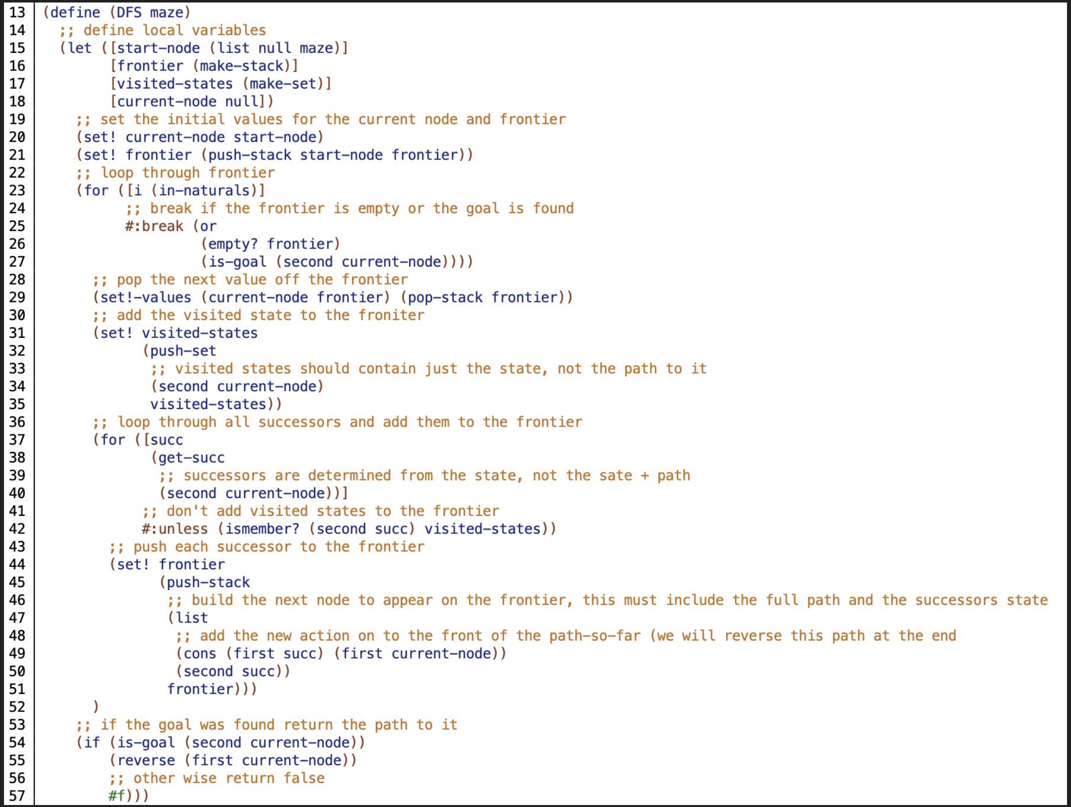
DFS Version



```
procedure Search($G, S, goal$)
    Inputs
        $G$: graph with nodes $N$ and arcs $A$
        $s$: start node
        goal: Boolean function of nodes
    Output
        path from $s$ to a node for which goal is true
        or $\perp$ if there are no solution paths
    Local
        Frontier: set of paths
    Frontier := $\{\langle s \rangle\}$
    while Frontier $\neq \{\}$ do
        select and remove $\langle n_0, \ldots, n_k \rangle$ from Frontier
        if goal$(n_k)$ then
            return $\langle n_0, \ldots, n_k \rangle$
        Frontier := Frontier $\cup \{\langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A\}$
    return $\perp$
```

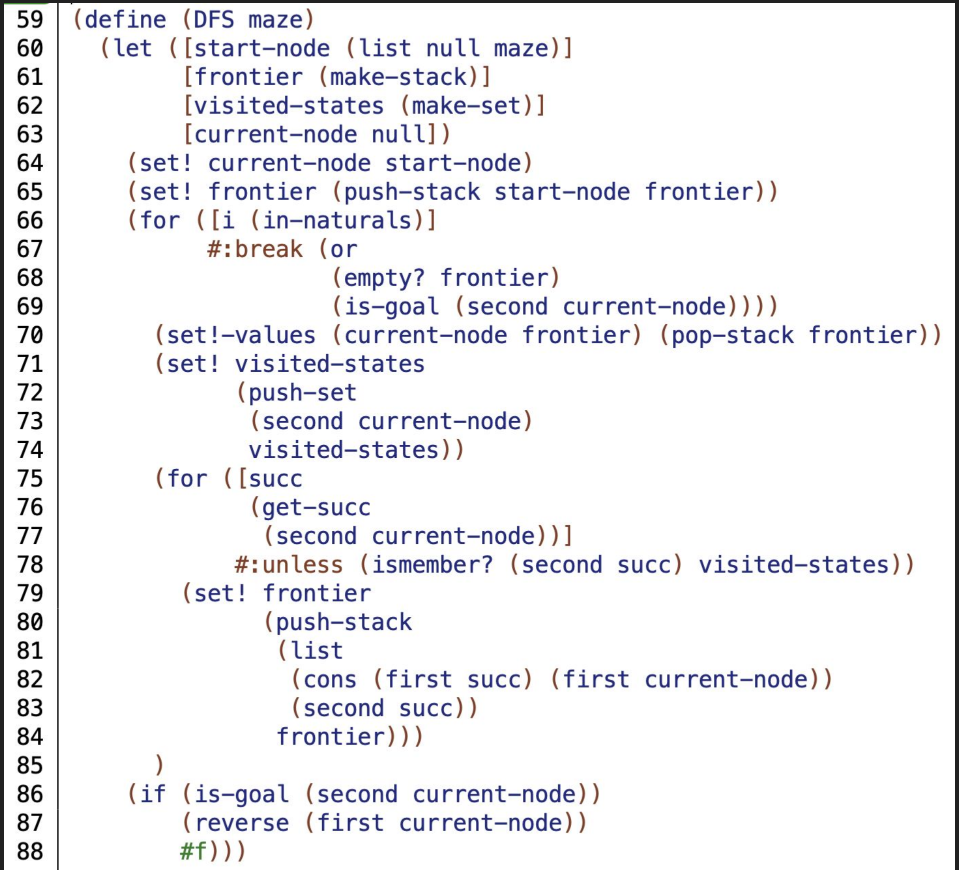
This is a stack!

Racket Code



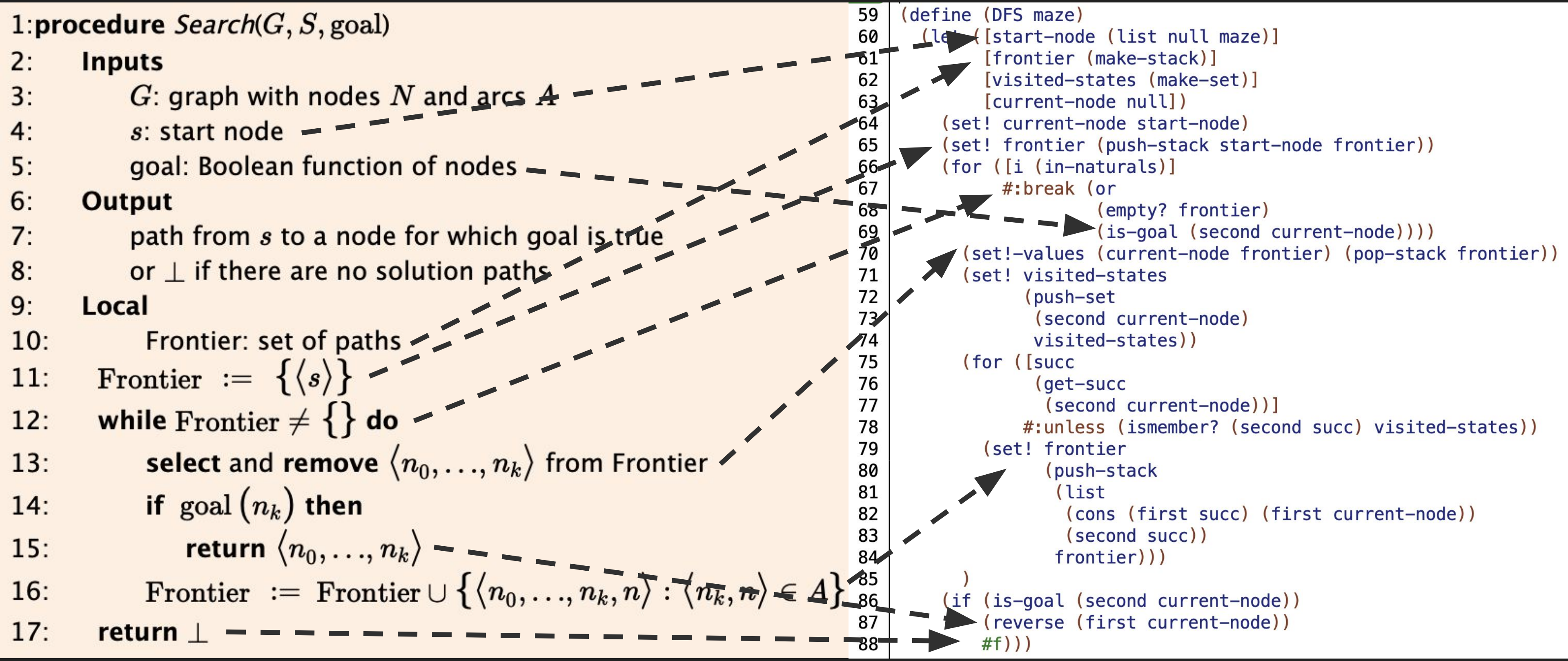
```racket
(define (DFS maze)
  ;; define local variables
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    ;; set the initial values for the current node and frontier
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    ;; loop through frontier
    (for ([i (in-naturals)]
          ;; break if the frontier is empty or the goal is found
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      ;; pop the next value off the frontier
      (set!-values (current-node frontier) (pop-stack frontier))
      ;; add the visited state to the froniter
      (set! visited-states
            (push-set
             ;; visited states should contain just the state, not the path to it
             (second current-node)
             visited-states))
      ;; loop through all successors and add them to the frontier
      (for ([succ
             (get-succ
              ;; successors are determined from the state, not the sate + path
              (second current-node))]
            ;; don't add visited states to the frontier
            #:unless (ismember? (second succ) visited-states))
        ;; push each successor to the frontier
        (set! frontier
              (push-stack
               ;; build the next node to appear on the frontier, this must include the full path and the successors state
               (list
                ;; add the new action on to the front of the path-so-far (we will reverse this path at the end
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    ;; if the goal was found return the path to it
    (if (is-goal (second current-node))
        (reverse (first current-node))
        ;; other wise return false
        #f)))
```

Same Thing Without the Comments



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Side-by-side



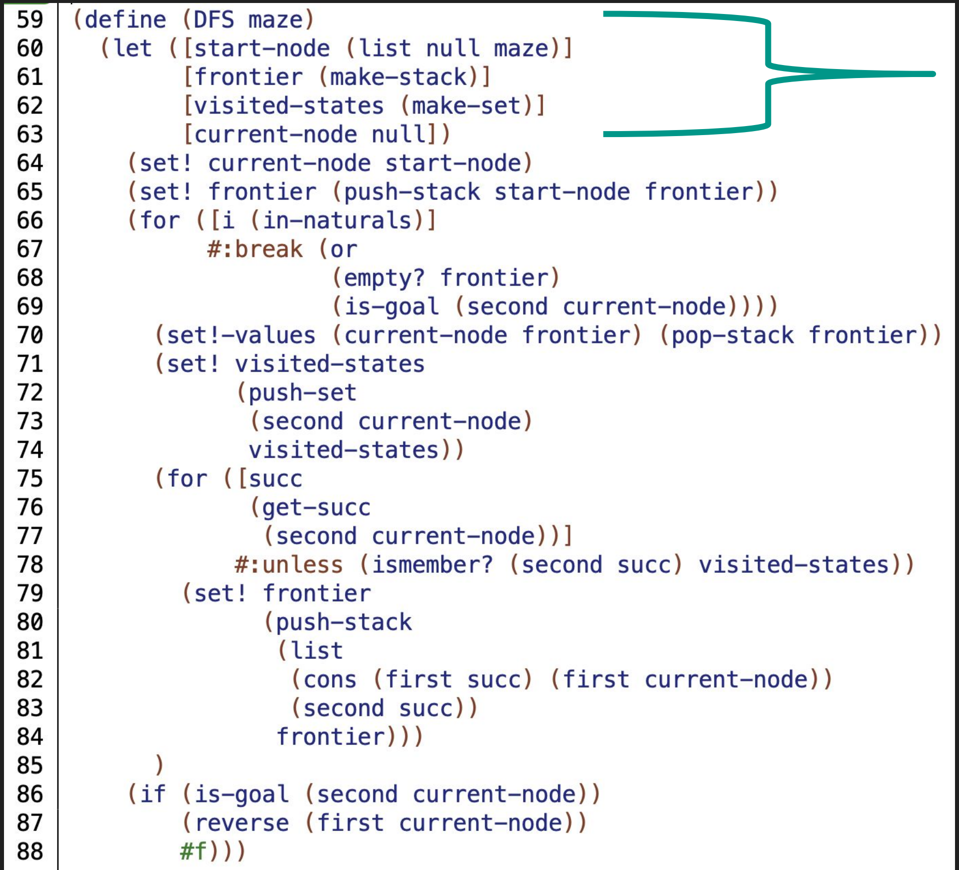
```
procedure Search($G$, $S$, goal)
    Inputs
        $G$: graph with nodes $N$ and arcs $A$
        $s$: start node
        goal: Boolean function of nodes
    Output
        path from $s$ to a node for which goal is true
        or $\perp$ if there are no solution paths
    Local
        Frontier: set of paths
    $\text{Frontier} := \{\langle s \rangle\}$
    while $\text{Frontier} \neq \{\}$ do
        select and remove $\langle n_0, \ldots, n_k \rangle$ from Frontier
        if $\text{goal}(n_k)$ then
            return $\langle n_0, \ldots, n_k \rangle$
        $\text{Frontier} := \text{Frontier} \cup \{\langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A\}$
    return $\perp$
```

```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Some Things not Addressed in the Pseudo Code

- Loops and tracking visited states
  - The pseudo code does not try to avoid loops, but exploring loops can easily blow up your frontier
- Separate action and state representations
  - In the racket code the frontier must store a representation of the final state in addition to paths so successors can be generated
- What is pushed to the frontier must be a deep copy of the path so far
  - The problem of mutability doesn’t come up much in functional languages like Racket, but this can cause a lot of bugs in languages with more mutability like C, Java and Python

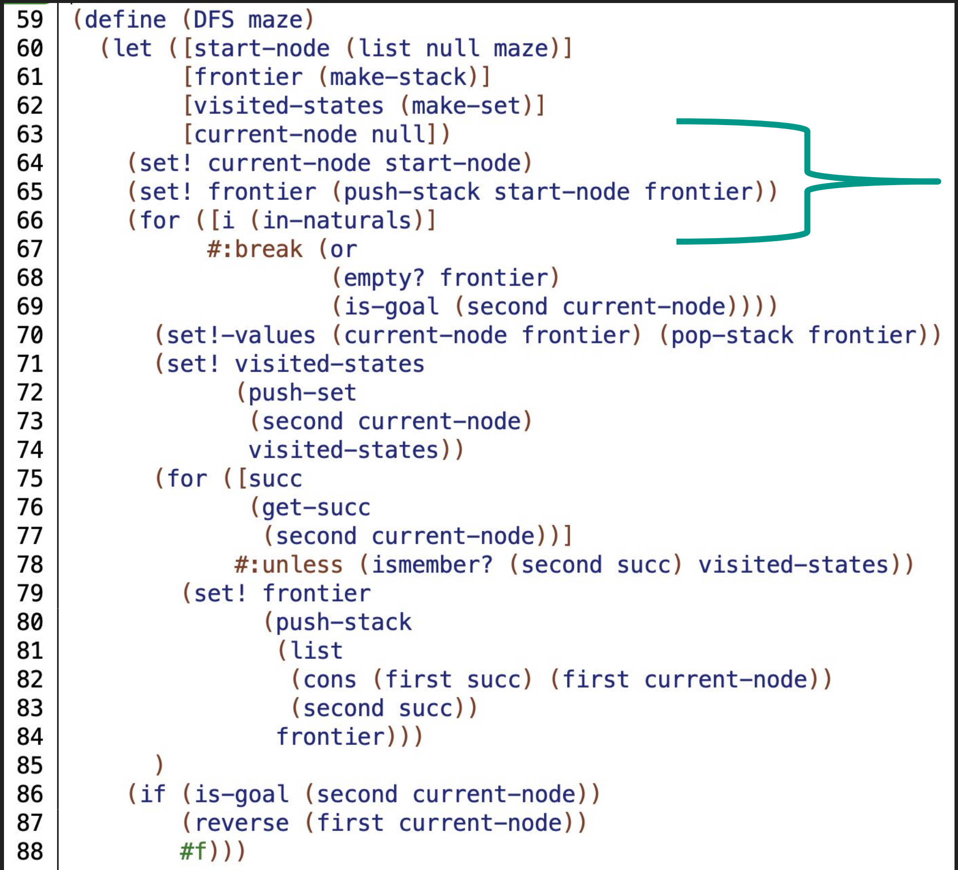
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Define Local Variables

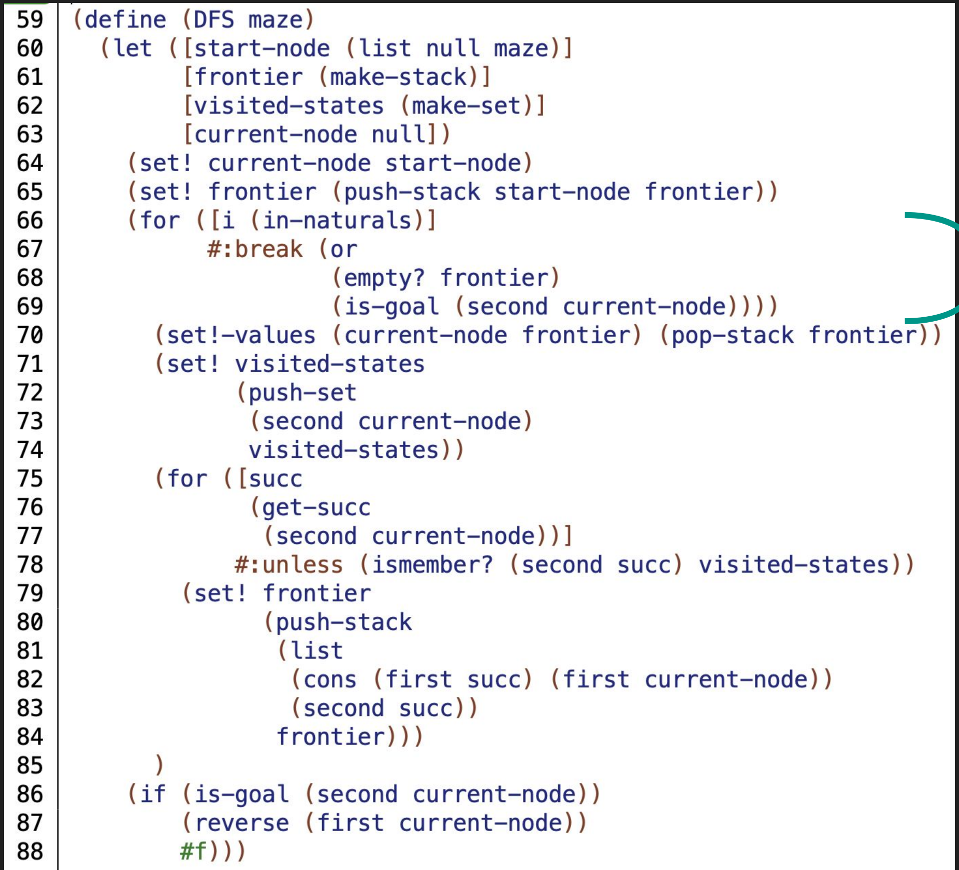
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Set Initial Values

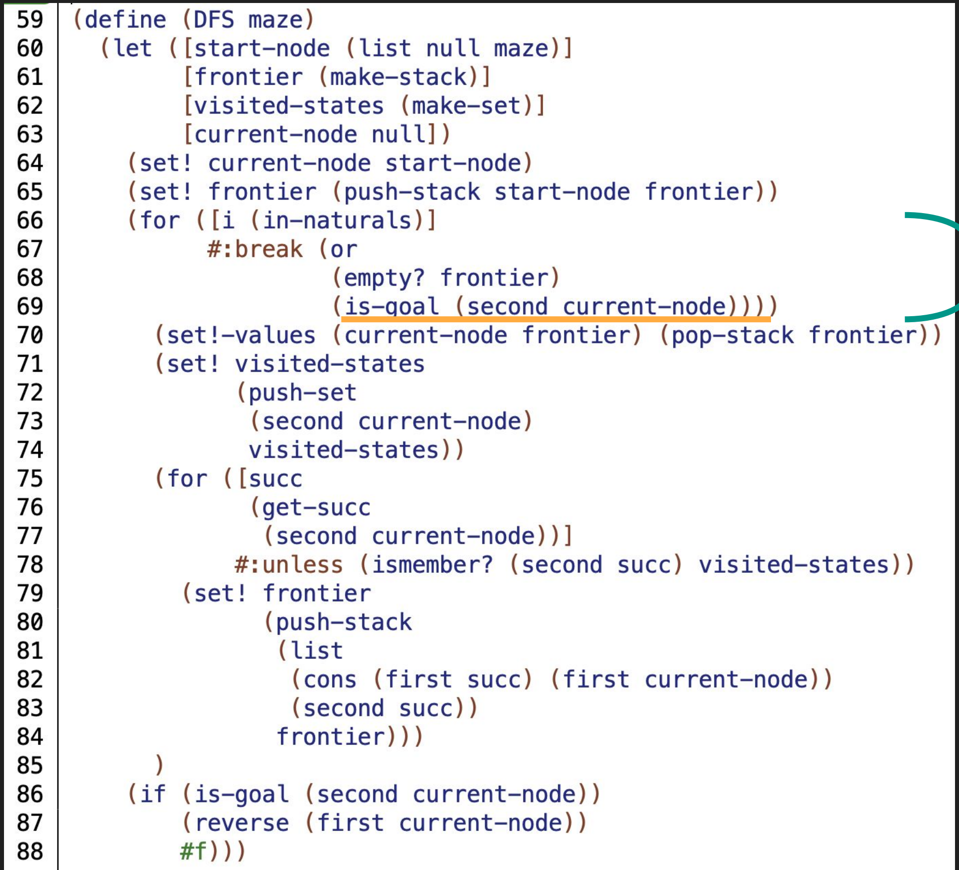
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Loop through the frontier

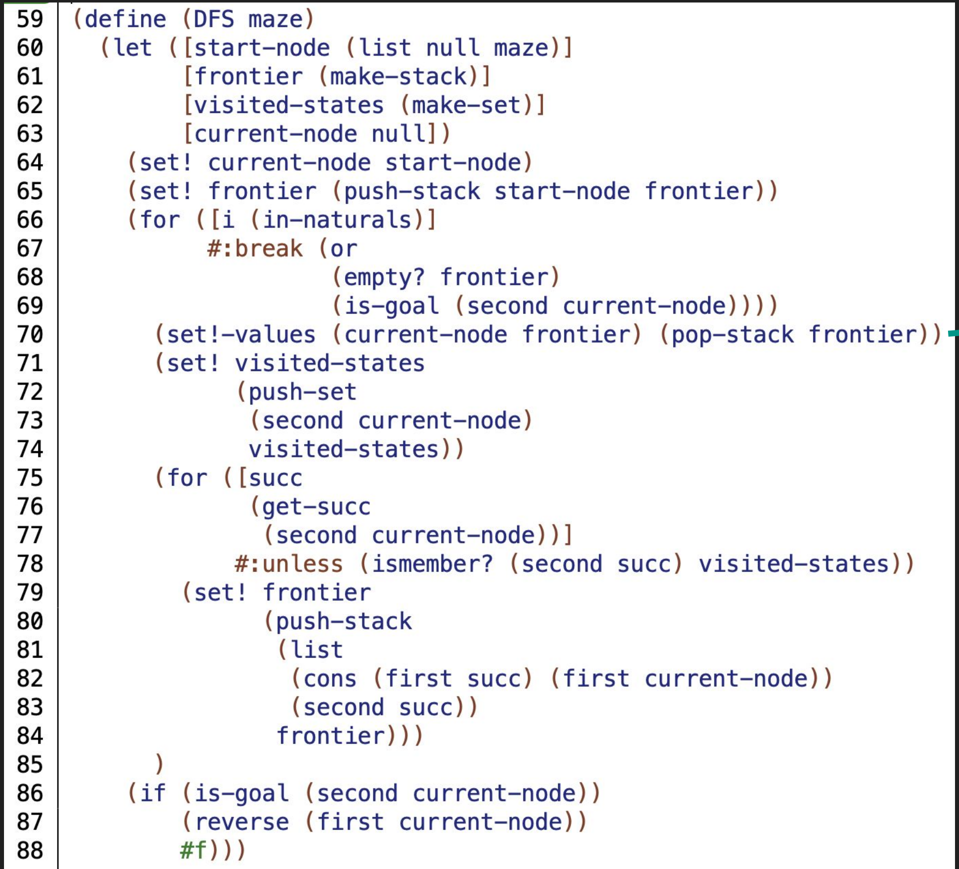
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

We need to test both break conditions here since unlike the break statement in the pseudocode, we don't have a mid-loop break statement

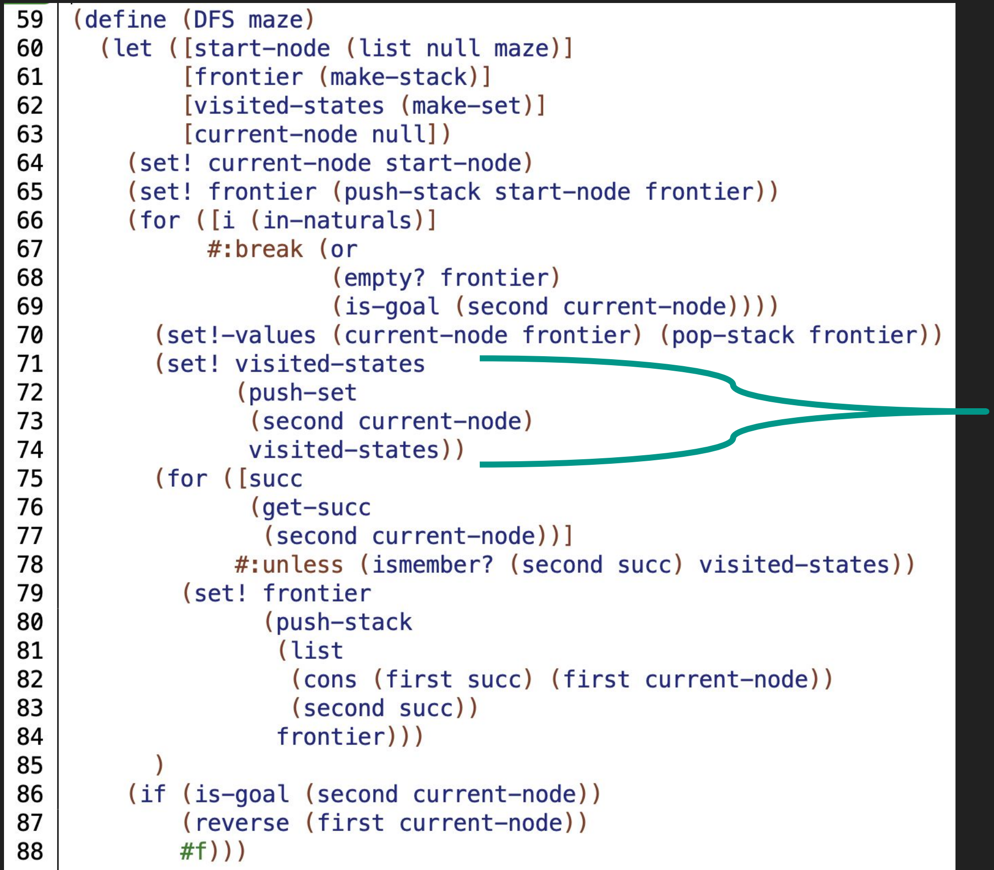
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Pop the next value off the frontier

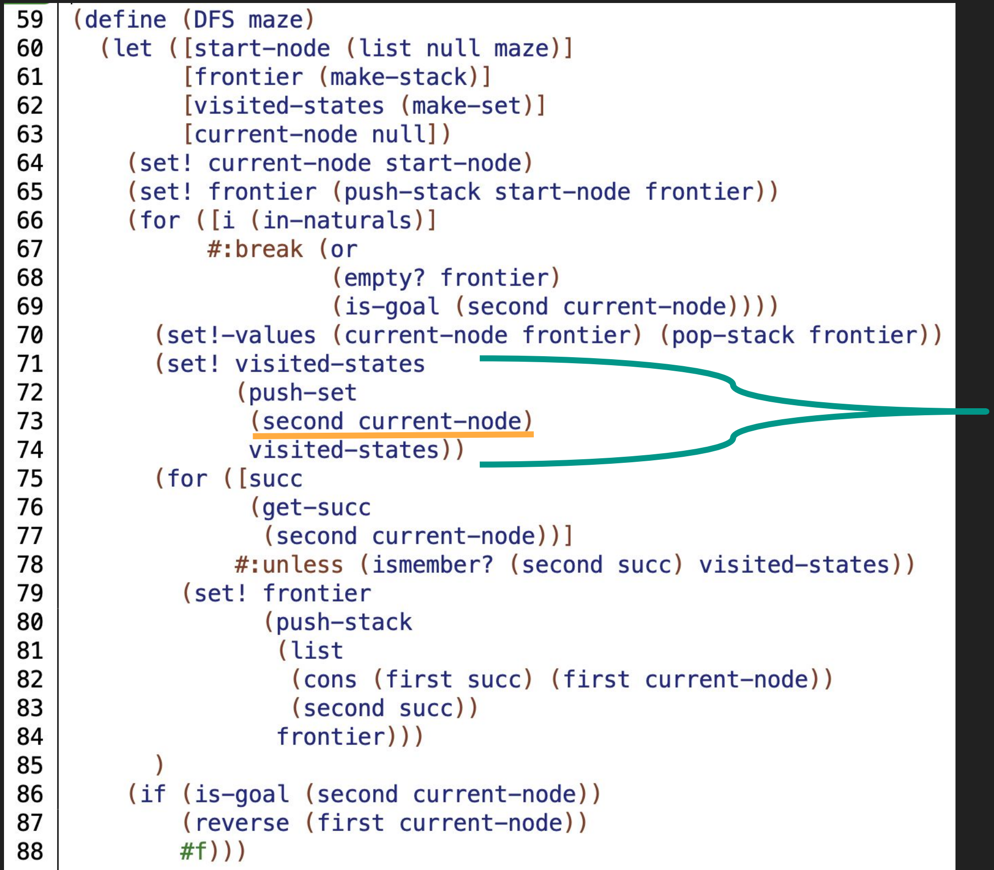
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Push the expanded state onto the set of visited states

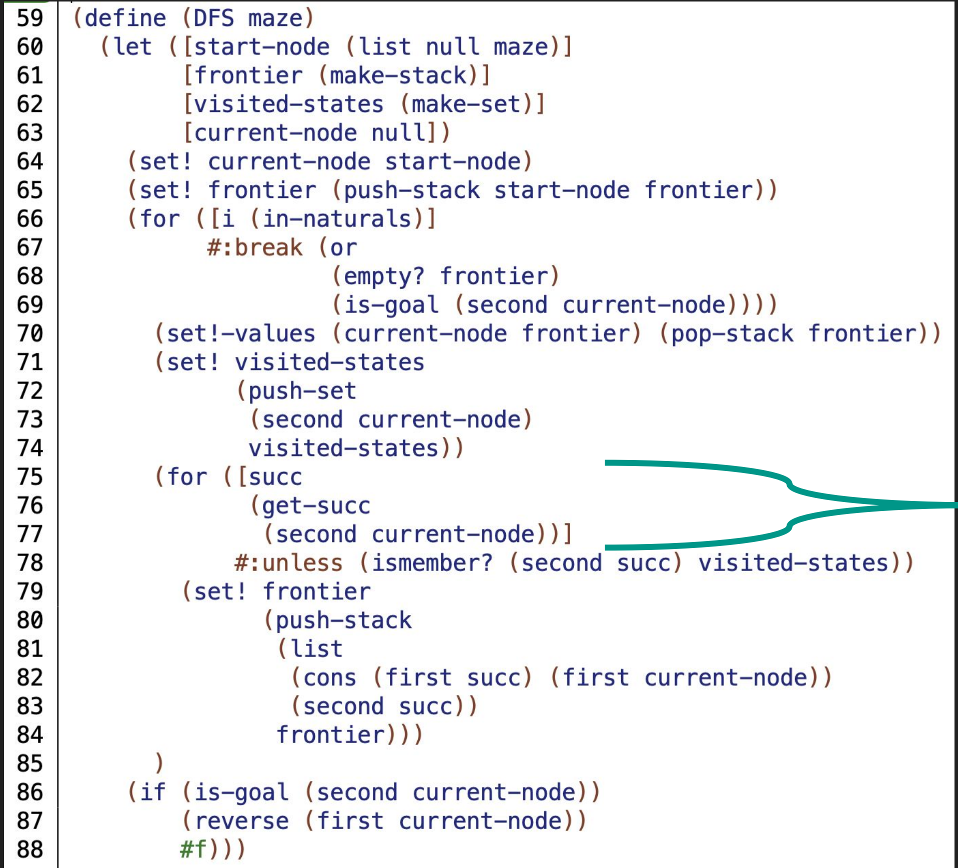
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

A state is not the same as a frontier node which is a path + a state, so we only add the state to the visited states

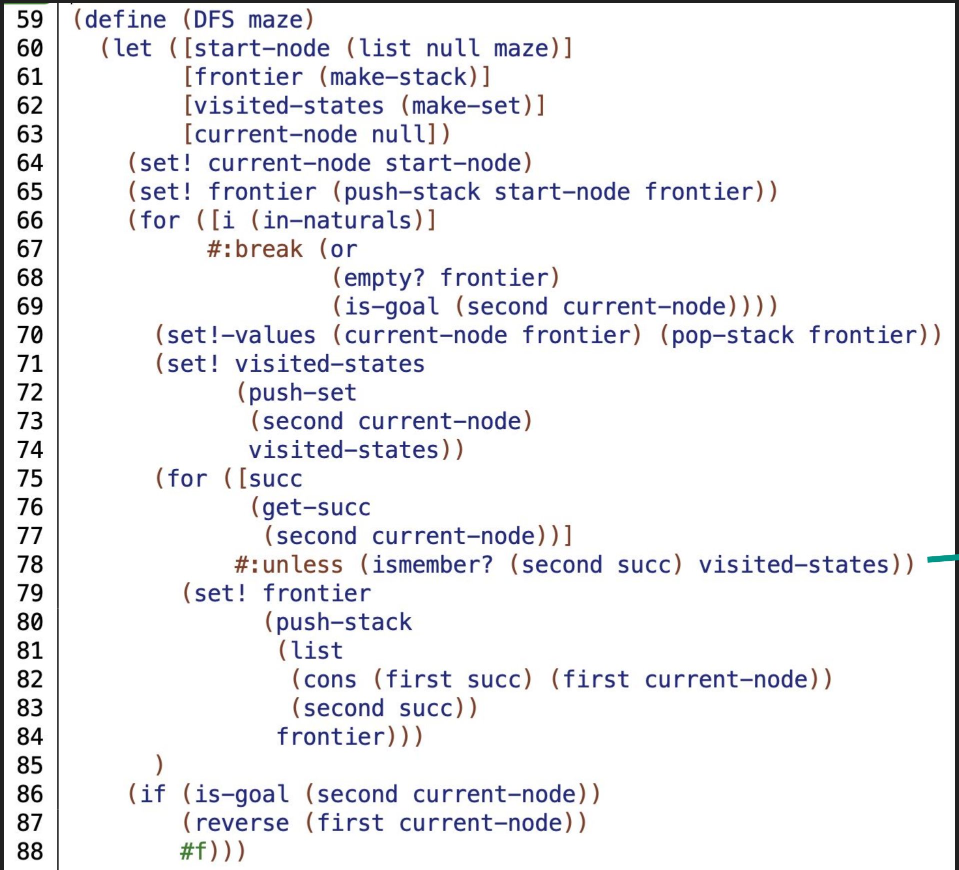
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Loop through successors to add them to the frontier

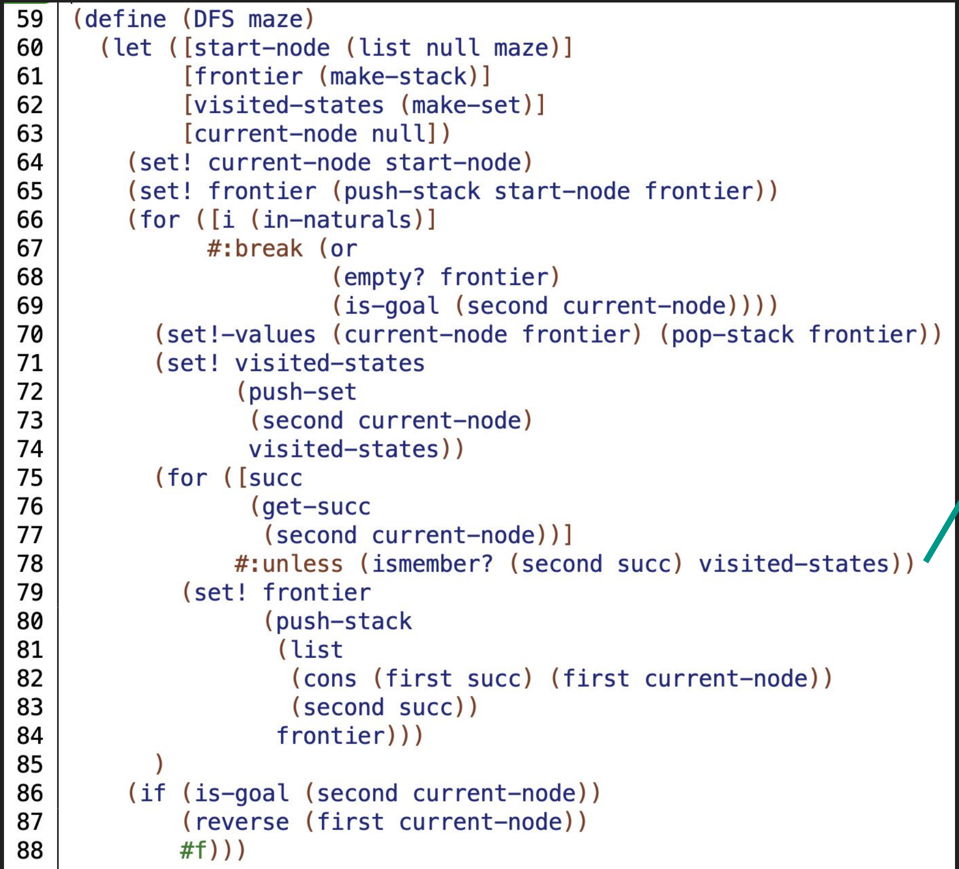
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Do not add previously visited states to the frontier

Stepping Through the Solution

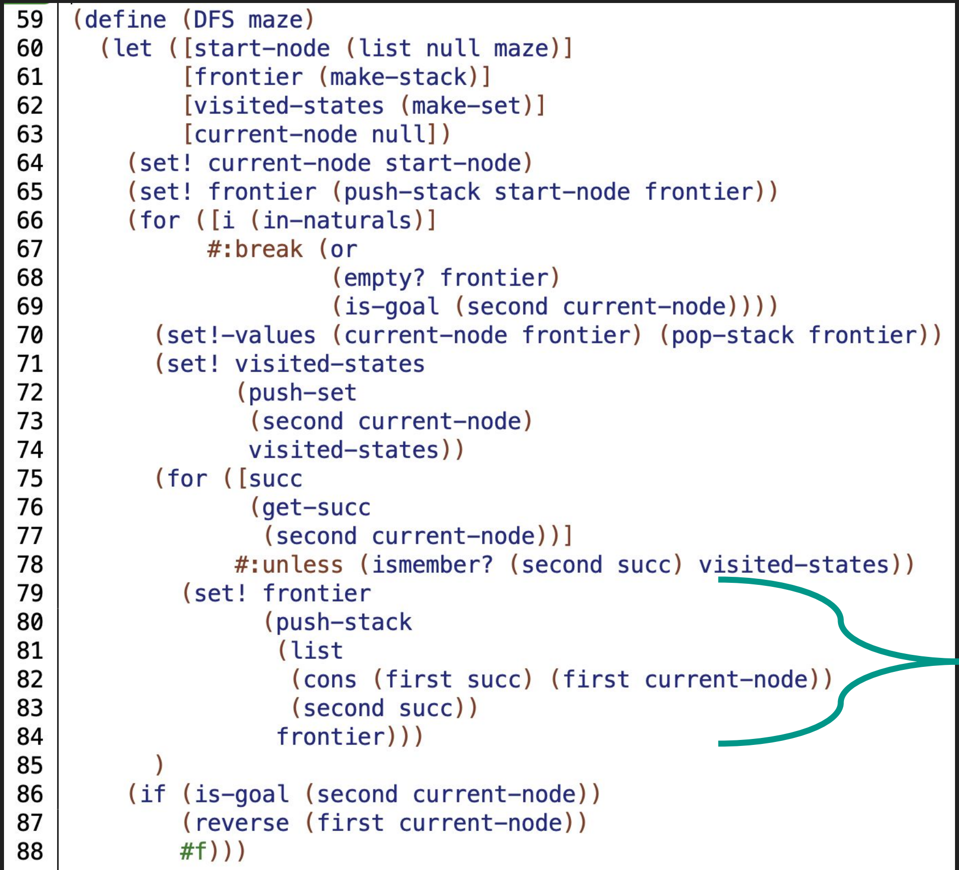


```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Astute students will notice that if a state got added to the frontier twice before it was expanded we aren't filtering it.

This is not enough of a problem in pacman to blow up the frontier, but a better solution would solve it. Perhaps implement visited-states as a hash table

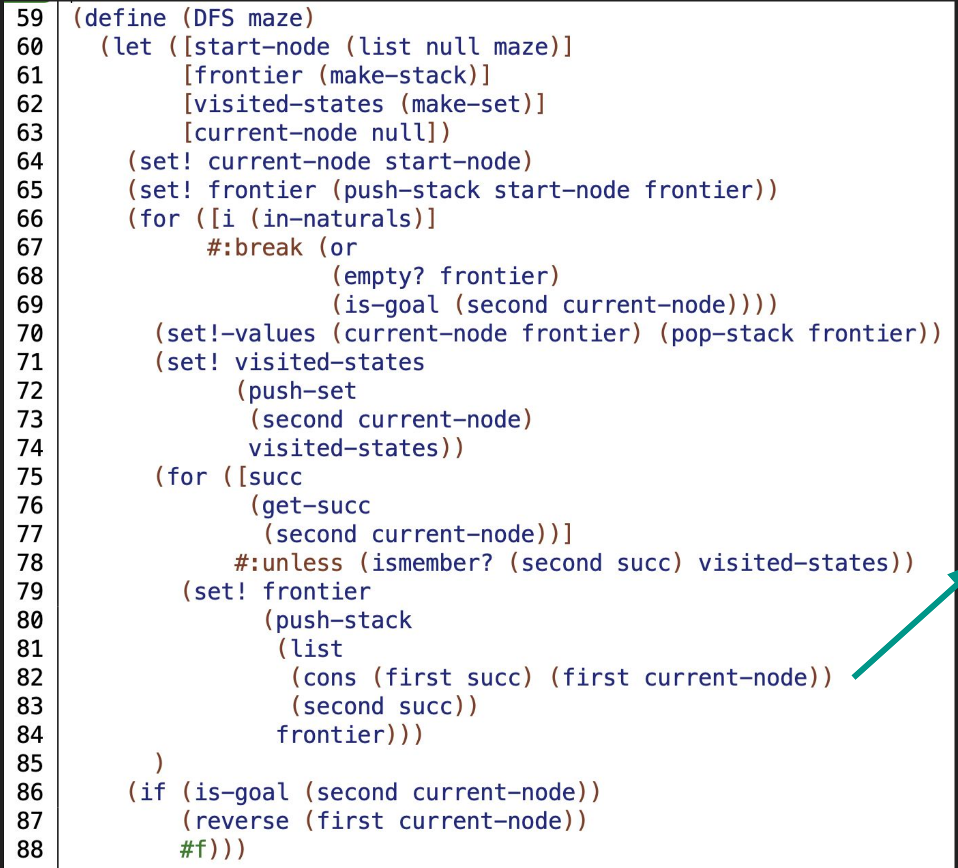
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Push the successor's state plus it's full path onto the frontier

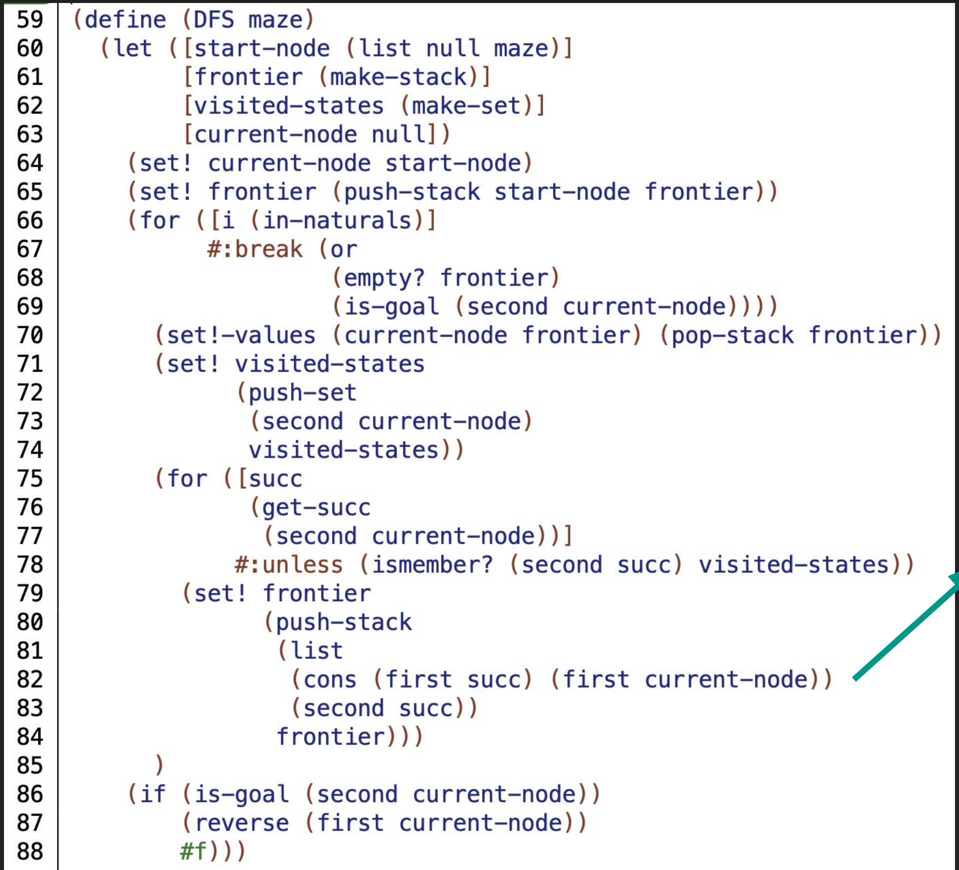
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

The full path is built by adding the action to get from the current state to the successor to the path to get to the current state

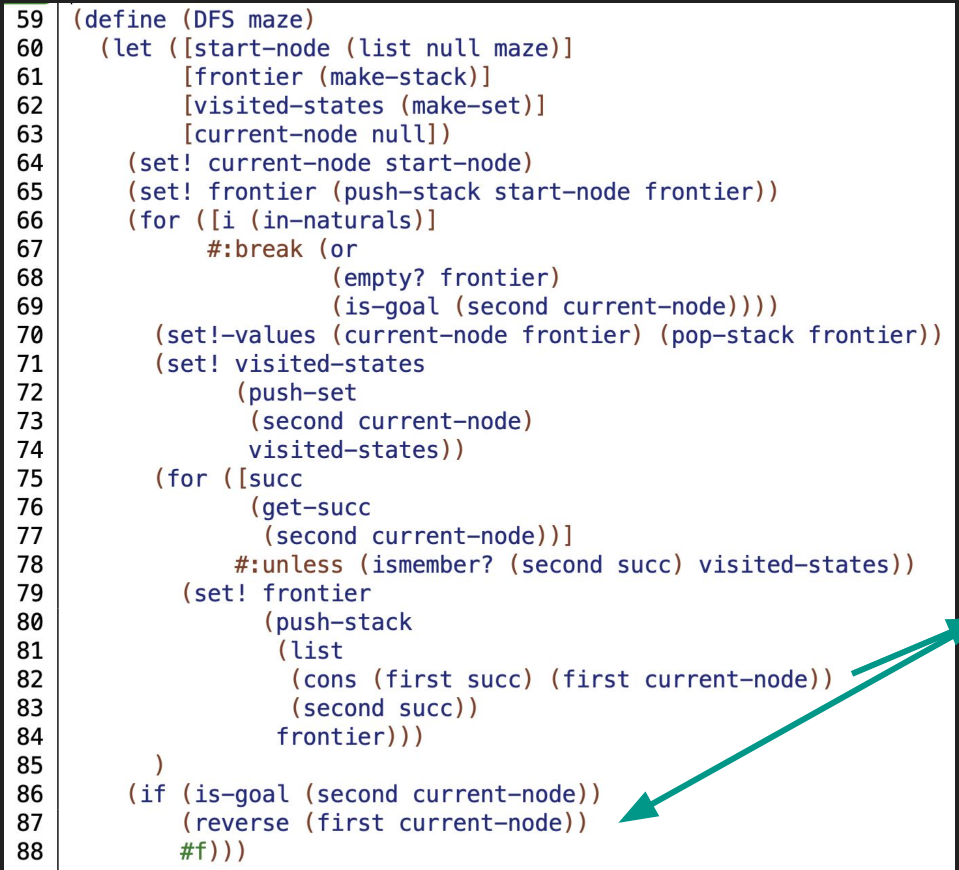
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

In Racket, as in many functional languages, adding an element to the front of a list may be easier than adding it to the back of a list

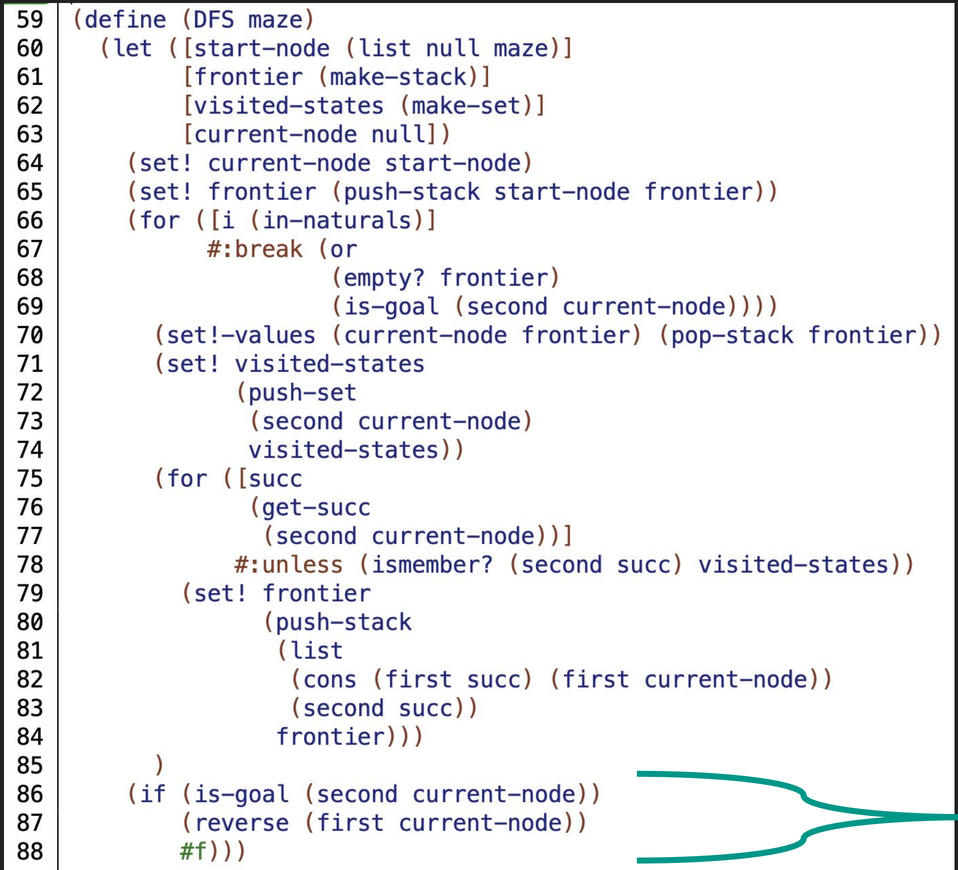
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

For this reason, this solution adds the actions in reversed order, and then reverses the path at the end.

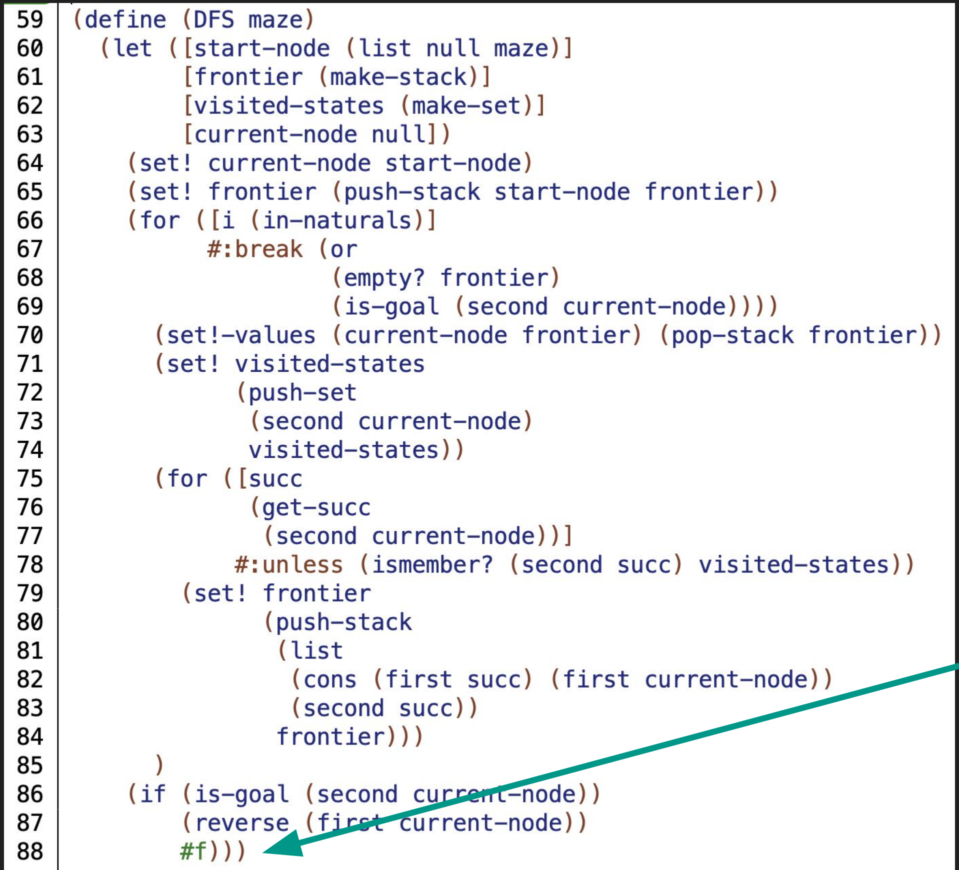
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Once the loop is broken, return the appropriate output.

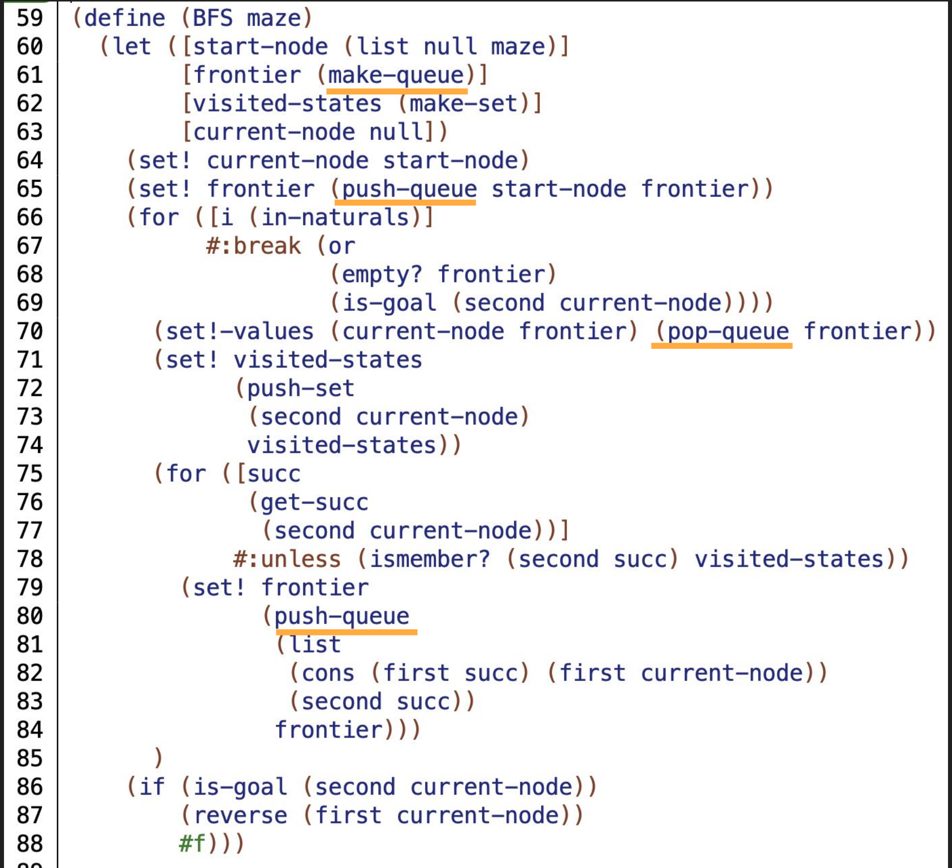
Stepping Through the Solution



```
(define (DFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-stack)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-stack start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-stack frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-stack
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

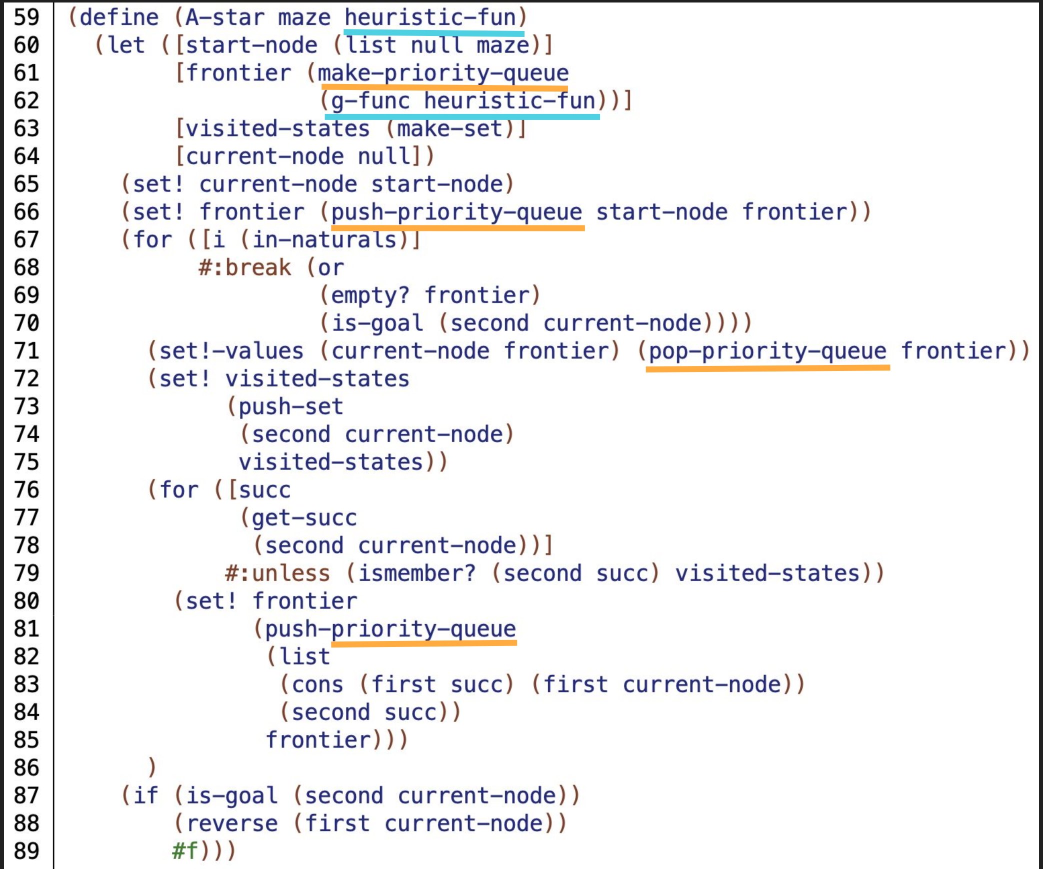
No mazes without solutions were given as examples, so we will not test what you return if the search fails.

BFS Version



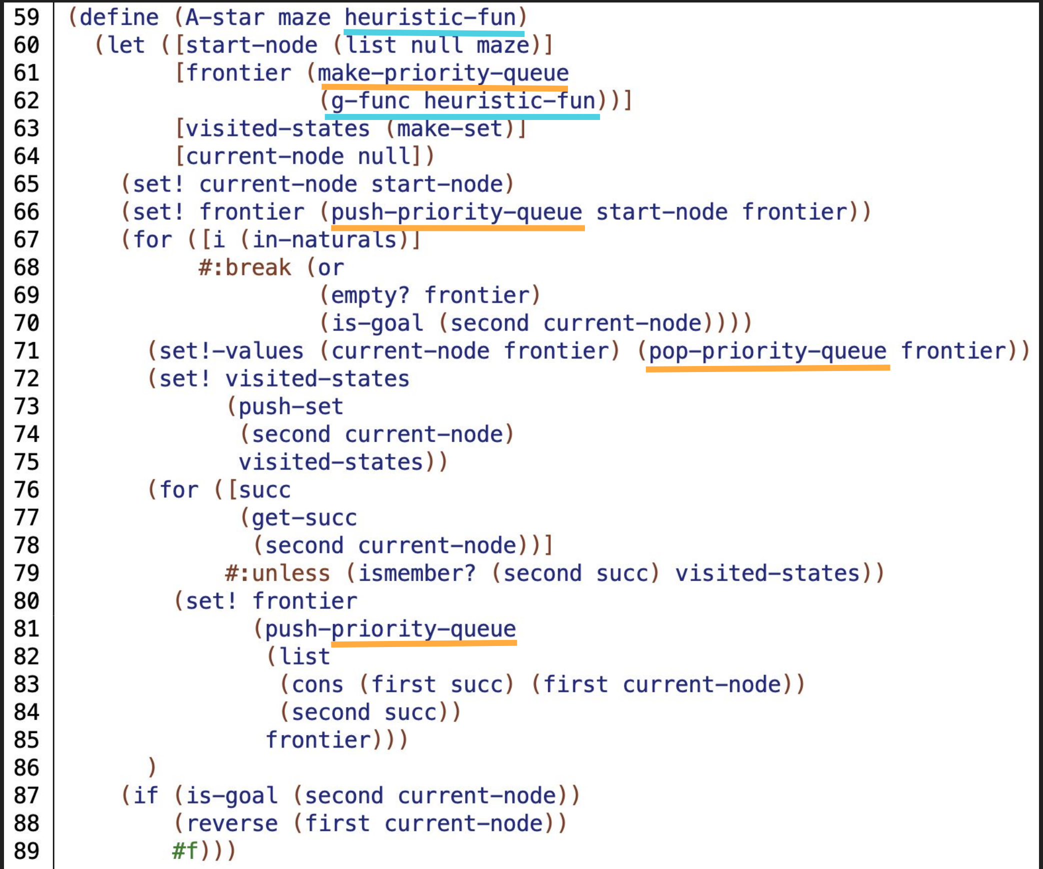
```
(define (BFS maze)
  (let ([start-node (list null maze)]
        [frontier (make-queue)]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-queue start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-queue frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-queue
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

A* Version



```
(define (A-star maze heuristic-fun)
  (let ([start-node (list null maze)]
        [frontier (make-priority-queue
                   (g-func heuristic-fun))]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-priority-queue start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-priority-queue frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-priority-queue
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

A* Version



```
(define (A-star maze heuristic-fun)
  (let ([start-node (list null maze)]
        [frontier (make-priority-queue
                   (g-func heuristic-fun))]
        [visited-states (make-set)]
        [current-node null])
    (set! current-node start-node)
    (set! frontier (push-priority-queue start-node frontier))
    (for ([i (in-naturals)]
          #:break (or
                   (empty? frontier)
                   (is-goal (second current-node))))
      (set!-values (current-node frontier) (pop-priority-queue frontier))
      (set! visited-states
            (push-set
             (second current-node)
             visited-states))
      (for ([succ
             (get-succ
              (second current-node))]
            #:unless (ismember? (second succ) visited-states))
        (set! frontier
              (push-priority-queue
               (list
                (cons (first succ) (first current-node))
                (second succ))
               frontier)))
      )
    (if (is-goal (second current-node))
        (reverse (first current-node))
        #f)))
```

Note that the function used to prioritize nodes in the priority queue is not the heuristic function, but the heuristic function plus the cost function (this sum gives the f function, which is used by A*).

A note on state representations

The solution shown here uses the state representation given to the problem, which is a matrix representing the positions of everything in the maze.

Other representations can be used, but to solve the search for more than one food and avoid loops a representation must be able to tell that: