a simple depth-1st search

NB: some of the examples here may use the matcher and/or utility functions available from www.agent-domain.org

the story so far

Previous examples have used a breadth-1st search which based its state-space exploration on a legal move generator (LMG) to generate new states (remember: LMGs take a state & return all legal successor states, they are specific to individual problems & form a basis for using search with these problems).

One LMG we used carried out some simple mathematical transformation, something like...

```
(defun lmg1 (n)
  (list (+ n 7)
        (- n 3)
        (round (/ n 2))
  ))
```

… we used this kind of LMG with the breadth-search function (available from www.agent-domain.org)...

```
cg-user(33): (breadth-search 5 10 #'lmg1)
(10 19 12 5)
```

This LMG is no good for a depth 1st search - why not?

bounding the lmg

We can sort things out as follows...

```
(let ((min 0)
      (max 25))

  (defun out-of-bounds (n)
    (not (< min n max)))
  )


(defun lmg2 (n)
  (remove-if #'out-of-bounds (lmg1 n)))
```

The problem with out-of-bounds is that to modify the upper & lower bounds we would have to edit the source code. We have come across lambda as a way to build anonymous functions for use in mapping functions. We can also return lambda forms as the results of functions, in this way functions can build other functions.

make-boundary (below) returns a function to check boundary conditions...

```
 (defun make-boundary (min max)
  #'(lambda (x)
      (not (< min x max))))
```

We can then build multiple boundary checking functions...

```
cg-user(1): (setf b12-20 (make-boundary 12 20))
#<Closure (:internal make-boundary 0) @ #x10dd988a>

cg-user(6): (setf b30-35 (make-boundary 30 35))
#<Closure (:internal make-boundary 0) @ #x10f21722>
```

Note though, we have to use funcall (this is explained in lectures)...

```
cg-user(3): (b12-20 15)
Error: attempt to call `b12-20' which is an undefined
       function.


cg-user(4): (funcall b12-20 15)
nil
cg-user(12): (funcall b12-20 31)
t

cg-user(13): (funcall b30-35 15)
t
cg-user(11): (funcall b30-35 31)
nil
```

We can then use this to build our modified lmg...

```
(defun lmg2 (n)
  (remove-if (make-boundary 0 25) (lmg1 n)))

cg-user(14): (lmg1 1)
(8 -2 0)
cg-user(16): (lmg1 21)
(28 18 10)
cg-user(17): (lmg2 1)
(8)
cg-user(18): (lmg2 21)
(18 10)
```

Better still would be something that allows us to make an lmg on the fly...

```
(defun make-bounded-lmg (min max lmg)
  (let ((filter (make-boundary min max)))
    #'(lambda (n)
        (remove-if filter (funcall lmg n)))
    ))

cg-user(19): (setf f (make-bounded-lmg 12 20 #'lmg1))
#<Closure (:internal make-bounded-lmg 0) @ #x10f1ea52>
cg-user(20): (funcall f 18)
(15)
```

Notice I have used let to bind a value to filter. Why might I have done this?
Also see later for another example which return lambda forms.

the depth 1st search

One way to write a depth-1st search (the most obvious?) is as a recursive tree-diver but...

1. we must ensure there are no circular paths by keeping a list of nodes we have already visited. That is the role of the visited arg below;
2. we want to return the path from start to goal. That is the role of the path arg below.

The search returns success when it has found the (generated) start state is the same as the goal, and returns failure (explicitly) when the start state has already been visited or (implicitly) when there are no valid successor states. Note the recursion occurs via the lambda form used in 'some' (for more explanation of this come along to my lecture!).

```
(defun depth-1st (start goal lmg &optional (path nil)
                                           (visited nil))
  (cond ((equal start goal)
         (reverse (cons start path))
        )
        ((member start visited :test #'equal)
         nil
        )
        (t
         (some #'(lambda (x)
                   (depth-1st x goal lmg
                              (cons start path)
                              (cons start visited)))
            (funcall lmg start)
            ))
   ))

cg-user(34): (pprint (depth-1st 5 10 #'lmg2))
(5 12 19 16 23 20 17 24 21 18 15 22 11 8 4 2 9 6 13 10)
```

If you look through our solution you will see a simplification. While path & visited are logically different they are the same in this implementation (though we could maybe improve

things a little if visited became global to any search).

So we can rewrite depth-1st as...

```
(defun depth-1st-2 (start goal lmg &optional (path nil))
  (cond ((equal start goal)
         (reverse (cons start path))
        )
        ((member start path :test #'equal)
         nil
        )
        (t
         (some #'(lambda (x)
                   (depth-1st-2 x goal lmg
                                (cons start path)
                   ))
               (funcall lmg start)
         ))
      ))

cg-user(35): (pprint (depth-1st-2 5 10 (make-boundary 0 25)))
(5 12 19 16 23 20 17 24 21 18 15 22 11 8 4 2 9 6 13 10)
```

a breath 1st search

For the depth 1st search we depended on the *call stack* which holds return points for all the recursive calls of the depth-1st function. To implement a breadth 1st search we need a queue instead of a stack (think about it!).

[this is under construction, for an explanation – attend the lecture]

```
(defun breadth (start goal lmg &key debug)
  (let ((explored (list start))
        (waiting  (list (list start)))
        )
    (do () ((null waiting) nil)
      (let* ((path  (pop waiting))
             (state (first path))
             )

        (when debug
          (pprint `(,state ==> ,waiting))
          (if (> (length waiting) 100)
             (error "state space growth")))

        (push state explored)
        (dolist (s (funcall lmg state))
          (when (equal s goal)
            (return-from breadth (cons s path)))
          (unless (member s explored :test #'equal)
                (setf waiting
                  (append waiting
                          (list (cons s path)))))
              )))
    ))
```

another example using lambda

This is a contrived example to demonstrate a second order higher order function. We can (& maybe will) develop this code in a more readable & maintainable way but what we have here is not without interest.

In this scenario we have a collection of agents roaming a virtual world. Among other things they can turn left & right. Type-1 agents can only turn by 90 degrees, type 2 agents turn by 45 degrees, their directions are represented (symbolically) by compass points.

NB: for a full explanation of this example attend the relevant lecture.

step 1 – build type 1 & 2 compass builders

My compasses are circular so I will define a function to make a list circular, this can be quite a risky thing to do & there are often better approaches but here we go...

```
(defun encircle (list)
  (setf (cdr (last list)) list)
  t     ;; best not return the circular list
  )     ;; it could print forever
```

Now here is the main builder function. A function which returns a lambda form which returns a lambda form...

```
(defun builder (list)
  #'(lambda (offset)
      (let ((offset-list (member offset list)))
        #'(lambda ()
            (pop offset-list))
        )))
```

Build type 1 & 2 compasses...

```
(let ((type1 '(N E S W)))
  (encircle type1)
  (defparameter type1-compass (builder type1))
  )

(let ((type2 '(N NE E SE S SW W NW)))
  (encircle type2)
  (defparameter type2-compass (builder type2))
  )
```

Setup some agent-specific compasses, these are initialised with the initial direction of each agent...

```
cg-user(4): (setf agent-1a (funcall type1-compass 'S))
#<Closure (:internal (:internal builder 0) 0) @ #x10f12ff2>
cg-user(5): (setf agent-2a (funcall type2-compass 'SW))
#<Closure (:internal (:internal builder 0) 0) @ #x10f254b2>
cg-user(6): (setf agent-2b (funcall type2-compass 'NE))
#<Closure (:internal (:internal builder 0) 0) @ #x10f2cf2a>
```

Now we can use these compasses to get the current direction of the agent & turn them right (this is not ideal – see later note).

```
cg-user(8): (funcall agent-1a)
S
cg-user(9): (funcall agent-1a)
W
cg-user(10): (funcall agent-1a)
N
cg-user(11): (funcall agent-2a)
SW
cg-user(12): (funcall agent-2a)
W
cg-user(13): (funcall agent-2b)
NE
cg-user(14): (funcall agent-2b)
E
cg-user(15): (funcall agent-2b)
SE
cg-user(16): (funcall agent-2a)
NW
```

exercises

1. write the code to move left (anticlockwise)
2. the usability of our solution is not so good. It would be preferred if we had 3 mechanisms (i) to return the current direction (ii) to turn right (iii) to turn left. Re-work the solution to achieve this
3. rewrite the solution (ie: start again) to specify the same functionality in a way you think most readable & maintainable.