Lisp – legal move generators & search

objectives

1. development of legal move generators
2. experimentation with search
3. exposure to pattern matching
4. exposure to new Lisp forms

The work described here will lead you through different strategies for developing legal move generator functions and give you some experience of using search mechanisms. You should not expect to complete this work in a single session, in order to complete it you will have to experiment and (probably) check out notes and on-line Lisp guides.

outline

You have been introduced to the role of legal move generators (LMGs) in lectures. You have also seen an example of a LMG in Lisp.

Reminder: a LMG is a function which takes a single state as its argument and returns a list of valid successor states.

The most important activity in developing a solution using search is the design of a representation to capture state descriptions. Often there are no "*right*" and "*wrong*" ways to specify state descriptions but a set of alternatives with some better than others.

Consider the jugs problem introduced in lectures. It is possible to specify state descriptions in various ways, eg:

1. as text: *jug1 holds five pints, jug2 holds three pints, jug3 is empty*
2. as a set of associations: ((jug1 . 5) (jug2 . 3) (jug3 . 0))
3. as a list of numbers: (5 3 0)
4. as a single number: 530

...etc...

The style of state description obviously influences the design of a LMG. A good style of state description will the LMG much easier to develop. In the example above...

option-1 a very poor choice,

option-2 ok assuming the LMG is to be based on adding & subtracting volumes but probably by using the symbols *jug1,2,3* and the association nesting it becomes unnecessarily detailed and thereby more error prone,

option-3 better than option-2, can be used as a style of state description for LMGs based on adding & subtracting volumes and also for an LMG which explicitly maps states onto successor states (see below);

option-4 not such a good choice as option-3 for an LMG based on numeric manipulation (adding & subtracting volumes) but good for an LMG which explicitly maps states onto their successor states (see below).

Example of a LMG which explicitly maps states to successor states

This LMG (which is very restricted) is for a word puzzle game where the goal is to get from one word to another through a series of intermediate steps such that (i) only one letter may be changed at each step and (ii) each step must spell a legal word. For example the word "BOAT" can be transformed into the word "LAST" in 4 steps as follows...

Boat - Coat - Cost - Lost - Last

A LMG for this puzzle could be written either as a collection of defmatch forms or using an association list with a small lookup function.

Example1: using defmatch

```
(defmatch lmg1 (coat)
  '((boat moat cost)))

(defmatch lmg1 (cost)
   '((most lost coat cast)))

(defmatch lmg1 (boat)
   '((moat coat boot)))
```

...etc...

Example 2: using an association list

```
(defparameter *words*
  '((coat boat moat cost)
    (cost most lost coat cast)
    (boat moat coat boot)
    (moat moot most boat)
    (moot soot boot loot)
    (lost last cast loot)
    ;;...etc...
   ))

(defun lmg2 (word)
   (-> *words* word))
```

NB: the "`->`" operator in Utils.lsp, you will need to load this before using "`->`". You will also need to load search.cl to use the search routine. The search mechanism is called *breadth-search*, it's called with 3 arguments: start-state, goal-state and the function body of the LMG (use the abbreviation #' when you pass functions as arguments).

Using best-search...

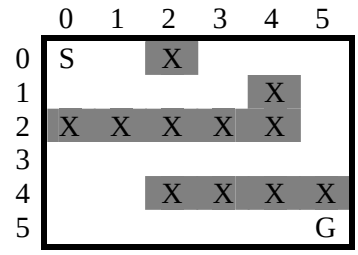
```
lisp prompt>  (breadth-search 'boat 'last #'lmg2)
result>       (boat coat cost lost last)
```

task 1

Think about the farmer-fox-goose-grain problem, design an appropriate state description and build a LMG for it. Try your LMG with the best-search mechanism provided.

task 2

This problem concentrates on the problem of navigating a robot around a maze (a simple example is shown below).



The start position is marked S & the goal position marked G. Shaded squares are illegal but we will slowly build up to avoiding these positions.

Each state descriptions for this problem is a pair of coordinates, to pull each coordinate pair into **x** & **y** parts you will need to write the legal move generator. Use a defmatch form with the first line looking something like this...

```
(defmatch maze-lmg ( (?x ?y) )
    ...)
```

you may then refer to then match variables in the body of the defmatch form as **#?x** and **#?y**. Your defmatch form should return all the points you can get to from any (x,y) point don't worry about the shaded squares or those outside of the grid just yet (don't allow diagonal moves though). When I developed this I wrote a small *helper* function – a function that doesn't really do anything but help readability. My helper function worked like **list** but I used it to make new points...

```
(defun point (a b)
    (list a b))

> (point 3 4)
(3 4)
```

Try out your legal move generator with the breadth first search and trace its action by setting the debug argument of breadth-search to **t**.

```
> (breadth-search '(0 0) '(5 5) #'maze-lmg)
...
```

NB: if your call to *breadth-search* generates too much volume of output Lisp will abbreviate it by cutting it short. To get full output, surround your call to *breadth-search* with a **pprint** call. **pprint** is pretty-print...

```
(pprint (breadth-search '(0 0) '(5 5) #'maze-lmg))
```

task 3

When you read the output of the search you ran at the end of task 2, you'll notice that the search considers moving to points that are off the grid. To prevent with this we need to do two things (i) find which points are off the grid (ii) remove them from the list of points.

Dealing with issue (ii) first... there are a suite of useful Lisp functions which prune different types of values from lists. An interesting pair of functions are called **remove-if** and **remove-if-not**. They both take two arguments, a predicate (a function returning true or false) and a list of values.

For example: **evenp** is a predicate which returns true if its numeric argument is even.

```
> (evenp 5)
nil

> (evenp 6)
t
```

remove-if returns a list after removing any items which satisfy the predicate, **remove-if-not** removes items which do not satisfy the predicate.

```
> (remove-if #'evenp '(1 2 3 4 5 6 7 8))
(1 3 5 7)

> (remove-if-not #'evenp '(1 2 3 4 5 6 7 8))
(2 4 6 8)
```

evenp is not much use for our purposes since it doesn't help determine which points are off the grid so the next step is to develop a predicate which does this. For the grid shown above this predicate needs to report if an x or y coordinate falls outside the range 0..5. We can develop this using the less than or equal operator...

```
(setf x-ok 3)
3
> (setf x-bad 7)
7
> (>= 5 x-ok 0)
t
> (>= 5 x-bad 0)
nil
```

A possible *in-bounds* predicate is given below – take time to examine it & understand how it is coded & how it works.

```
(defmatch in-bounds ( (?x ?y) )
  (and (>= 5 #?x 0)
       (>= 5 #?y 0)
       ))

(in-bounds '(3 4))
t
> (in-bounds '(3 7))
nil
> (in-bounds '(-3 4))
nil
```

Use the ideas above to modify your legal move generator so it does not generate points which are off the grid. The best way to do this is to generate all your points then reject (ie: use **remove-if-not** to filter out) all those which are not *in-bounds*.

task 4

The next stage is to remove/reject coordinate points which are blocked-off in the grid (marked as a shaded X in the diagram above). This can be done in a similar way to dealing with points that are out of bounds but developing the predicate requires a different strategy.

The first step is to build a structure containing all illegal (blocked) points...

```
(defvar illegals)
(setf illegals
  '((0 2)(1 4)(2 0)(2 1)...etc...
    ))
```

then we can check to see if a point is a member of this structure. The most obvious way to do this is using the **member** function which returns non-nil if an item is a member of a list (remember that anything not nil is considered to be true in Lisp).

```
> (member 'cat '(dog bat cat rat))
(cat rat)
> (member 'cat '(herring haddock kipper carp))
nil
```

The member function needs a bit of tweaking when we check for membership of structures (like coordinate pairs) rather than simple, atomic symbols...

```
> (member '(2 3) '((1 2)(2 3)(3 4)))
nil
> (member '(2 3) '((1 2)(2 3)(3 4)) :test #'equal)
((2 3) (3 4))
```

Wrap a call to the **member** function in the following function definition & then use it to filter out illegal states in your legal move generator.

```
(defun illegal (p)
  ...)
```

Finally... check your legal move generator by inspecting the results of a search and ensuring its recommended path is valid.

task 5

The final modifications to your maze search will make it recommend the *cheapest* path in mazes where there is a *cost* associated with different cells in the grid. The maze below has some blocked (illegal) squares and other squares have costs associated with them.

	0	1	2	3	4	5	6
0	**S**	4	X	4	X	4	5
1	3	1	1	2	2	1	5
2	X	X	1	X	X	8	X
3	1	1	2	5	9	8	1
4	1	7	X	X	7	X	1
5	1	1	6	6	4	3	3
6	X	1	X	1	1	1	1
7	X	1	1	1	X	5	**G**

If you look back at the code you have developed for the tasks above, you'll notice that some of the data & functions work together to describe the structure of the maze, you will need to modify some of these. Start by changing your code to define the bounds of the grid and the illegal (blocked) squares.

The cost information can be encoded in a variety of ways, the code below deals with costs in a similar way to illegal states, with a data structure and a lookup function. The lookup function assumes any cell which does not have an entry in the data structure has a cost of 1.

```
(defvar costs)
(setf costs
  '((0 1 4) (0 3 4) (0 5 4) (0 6 5)
    (1 0 3) (1 3 2) (1 4 2) (1 6 5)
    (2 5 8)
    (3 2 2) (3 3 5) (3 4 9) (3 5 8)
    (4 1 7) (4 4 7)
    (5 2 6) (5 3 6) (5 4 4) (5 5 3) (5 6 3)
    (7 5 5)
    ))

(defun get-cost (x y)
  (or (is-present (`(,x ,y ?n) costs) #?n)
      1
      ))
```

The most important modification you need to make concerns the structure of state descriptions. With the previous maze (the one with no costs), state descriptions were points like (3 2). State descriptions now need to contain cost information as well. One representation you could use is as follows...

```
(cost  (X-coordinate  Y-coordinate))
```

Write a new legal move generator which takes a state encoded as suggested and generates successor states with updated cost information, eg:

```
> (cost-lmg '(7 (3 2)))
((8 (2 2)) (8 (3 1)) (12 (3 3)))
```

When your LMG is complete you should try it with the search routine. This time your call to *best-search* needs to include a bit of extra information. Your state descriptions now include

cost details as well as the state information. The search routine needs to filter out the cost details when checking to see if it has reached the goal state, etc. Given states represented as...

```
(cost  (X-coordinate  Y-coordinate))
```

the filter will be a function which returns `(X-coordinate Y-coordinate)` –the function *second* will do this, eg:

```
> (second '(12 (5 0)))
(5 0)
```

Using second as a filter your call to best-first should result in something like this...

```
> (pprint (best-search '(0 (0 0)) '(7 6) #'cost-lmg
           :filter #'second))
((41 (7 6)) (40 (7 5)) (35 (6 5)) (34 (6 4)) (33 (5 4))
 (29 (4 4)) (22 (3 4)) (13 (3 3)) (8 (3 2))  (6 (2 2))
 (5 (1 2))  (4 (1 1))  (3 (1 0))  (0 (0 0)))
```

Notice that the route returned is the shortest (in terms of the number of grid-squares) but not the least-cost.

task 6

To obtain the least cost path the search routine must select the least cost node on each cycle. If we want this to happen we must provide it with an appropriate selector function – which selects the least cost node from a collection of nodes.

A typical node (including all the path information) looks something like the one below which shows a path from (4 4) through (5 4) and (6 4) to (6 3) with an accumulated cost of 6.

```
((6 (6 3)) (5 (6 4)) (4 (5 4)) (0 (4 4)))
```

One strategy for writing a selector function is to use the in-built function sort which takes the following arguments:

1. a list to be sorted;
2. a comparison function;
3. an optional function *key* to extract the values to be compared.

see examples below & note the use of the *:key* argument.

```
> (sort '(2 7 5 9 4) #'>)
(9 7 5 4 2)

> (sort '(2 7 5 9 4) #'<)
(2 4 5 7 9)

> (sort '((2 the) (7 a) (5 chased) (9 mouse) (4 cat))
          #'< :key #'first)
((2 the) (4 cat) (5 chased) (7 a) (9 mouse))
```

Write a suitable function *path-cost* to extract the accumulated cost from a path...

```
> (path-cost '((6 (6 3)) (5 (6 4)) (4 (5 4)) (0 (4 4))))
6
```

...check this works with sort...

```
> (sort '(((25 (2 5)) (17 (3 5)) (9 (3 4)) (0 (4 4)))
          ((18 (3 6)) (17 (3 5)) (9 (3 4)) (0 (4 4)))
          ((6 (6 3)) (5 (6 4)) (4 (5 4)) (0 (4 4)))
          ((11 (6 3)) (10 (5 3)) (4 (5 4)) (0 (4 4)))
          ((16 (5 2)) (10 (5 3)) (4 (5 4)) (0 (4 4)))
          ((8 (6 5)) (7 (5 5)) (4 (5 4)) (0 (4 4)))
          ((10 (5 6)) (7 (5 5)) (4 (5 4)) (0 (4 4)))
          ) #'< :key #'path-cost)
(((6 (6 3)) (5 (6 4)) (4 (5 4)) (0 (4 4)))
 ((8 (6 5)) (7 (5 5)) (4 (5 4)) (0 (4 4)))
 ((10 (5 6)) (7 (5 5)) (4 (5 4)) (0 (4 4)))
 ((11 (6 3)) (10 (5 3)) (4 (5 4)) (0 (4 4)))
 ((16 (5 2)) (10 (5 3)) (4 (5 4)) (0 (4 4)))
 ((18 (3 6)) (17 (3 5)) (9 (3 4)) (0 (4 4)))
 ((25 (2 5)) (17 (3 5)) (9 (3 4)) (0 (4 4))))
```

...and use this as a basis for writing a selector function to choose the path with least associated cost from a list of paths...

```
> (selector '(((25 (2 5)) (17 (3 5)) (9 (3 4)) (0 (4 4)))
              ((18 (3 6)) (17 (3 5)) (9 (3 4)) (0 (4 4)))
              ((6  (6 3)) (5 (6 4))  (4 (5 4)) (0 (4 4)))
              ((11 (6 3)) (10 (5 3)) (4 (5 4)) (0 (4 4)))
              ((16 (5 2)) (10 (5 3)) (4 (5 4)) (0 (4 4)))
              ((8  (6 5)) (7 (5 5))  (4 (5 4)) (0 (4 4)))
              ((10 (5 6)) (7 (5 5))  (4 (5 4)) (0 (4 4)))
            ))
((6 (6 3)) (5 (6 4)) (4 (5 4)) (0 (4 4)))
```

Now you are finally ready to run a best-first search...

```
> (pprint (best-search '(0 (0 0)) '(7 6) #'cost-lmg
                       :filter #'second
                       :selector #'selector))

((22 (7 6)) (21 (6 6)) (20 (6 5)) (19 (6 4)) (18 (6 3))
 (17 (7 3)) (16 (7 2)) (15 (7 1)) (14 (6 1)) (13 (5 1))
 (12 (5 0)) (11 (4 0)) (10 (3 0)) (9 (3 1))  (8 (3 2))
 (6 (2 2)) (5 (1 2)) (4 (1 1)) (3 (1 0)) (0 (0 0)))
```

Look at the results & check that the search has now produced the least cost route. Check your solution again by adjusting the costs of the grid and checking the results (changing the two 8 values to 1's works).