legal move generators in Clojure – take 2

NB: many of the examples here use the matcher available from www.agent-domain.org

LMGs – the story so far

Reminder: LMGs take a state & return all legal successor states, they are specific to individual problems & form a basis for using search with these problems.

We have built LMGs in a couple of different ways but only to work on fairly simple problem states. For example...

LMG from simple functions

a simple maths puzzle- get from start no. to goal as quickly as possible using only the following simple rules

1. add 1
2. subtract 1
3. double

```
(def math-lmg [n]
  (list (+ 1 n)
        (- n 3)
        (* n 2)
        ))

> (math-lmg 12)
(13 9 24)
```

using this with search (remember to load the search mechanism)

```
(breadth-search 5 37 math-lmg)
(37 40 20 10 5)
```

explicit successors using associations

this approach is declarative, the successors are written explicitly & (with this example) held in an association list

```
; the explicit successors
(def words
  '{coat (boat moat cost)
    cost (most lost coat cast)
    boat (moat coat boot)
    moat (moot most boat)
    moot (soot boot loot)
    ...etc...
   })
```

```
> (words 'boat)
(moat coat boot)

; using this with search
> (breadth-search 'coat 'loot words)
(loot moot moat coat)
```

using the matcher to build LMGs

In this section we investigate using matcher facilities to build LMGs. We are working towards defining rules and operators, these can both be used in LMGs as well as in other areas of AI.

explicit successors using defmatch

In the lectures we have/will consider a simple *cafe* environment with a cook who works in a kitchen and a waiter who serves the tables. The kitchen & the main room are connected by a serving hatch,

Using this environment, an example move could be...

```
waiter at hatch holding nothing
cook at hatch holding food
=> waiter at hatch holding food
   cook at hatch holding nothing
```

two rules using defmatch

```
(defmatch cafe []
  ([waiter at hatch holding nothing cook at hatch holding food]
    :=> '(waiter at hatch holding food
          cook at hatch holding nothing))
  ([waiter at hatch holding plates cook at hatch holding nothing]
    :=> '(waiter at hatch holding nothing
          cook at hatch holding plates)))

user=> (cafe '(waiter at hatch holding nothing
                     cook at hatch holding food))
(waiter at hatch holding food cook at hatch holding nothing)

user=> (cafe '(waiter at hatch holding plates
                     cook at hatch holding nothing))
(waiter at hatch holding nothing cook at hatch holding plates)
```

a briefer (& slightly better) structure for state descriptions is...

```
([waiter hatch :nil] [cook hatch food])
```

the rest of this sub-section uses this structure...

Note the use of :nil which is not the same as nil. :nil is a keyword we use in our tuples. :nil implies "nothing" but is still treated as a valid symbol.

note that many states have multiple successor states which complicate the rules a little, eg:

```
(defmatch cafe []
  (([waiter hatch :nil][cook hatch food])
    :=>
    '(([waiter hatch food][cook hatch :nil])
      ([waiter table :nil] [cook hatch food])
      ([waiter hatch :nil] [cook stove food])
       )))
```

it is easier to design rules as single mappings rather than as groups, eg:

```
(def cafe-rules
  '([([waiter hatch :nil][cook hatch food])
      :=> ([waiter hatch food][cook hatch :nil])]
    [([waiter hatch :nil][cook hatch food])
       :=> ([waiter table :nil][cook hatch food])]
    [([waiter hatch :nil][cook hatch food])
       :=> ([waiter hatch :nil][cook stove food])]
    ;etc
    ))
```

using rules to drive a LMG is a common & useful approach but we need to redesign our LMG to take advantage of this approach (read through the code and check the matcher documentation for a fuller understanding). Note also that we can sometimes achieve similar results with a map.

```
(defn serve-lmg [state]
  (mfor [ '[?s :=> ?succ] cafe-rules ]
    (when (= (? s) state) (? succ))
    ))

user=> (serve-lmg '([waiter hatch :nil][cook hatch food]))
(([waiter hatch food] [cook hatch :nil])
 ([waiter table :nil] [cook hatch food])
 ([waiter hatch :nil] [cook stove food]))

user=> (serve-lmg cafe-rules
             '([aardvaark tree machete][cook hatch food]))
()
```

better still would be an approach which allowed us to specify *wild-card* symbols so we can specify a rule like...

> if the waiter is at the hatch, regardless of whether s/he is holding anything & regardless of what the cook is doing, the waiter can move to the table

using matcher forms we can produce a rule based LMG as below...

```
(def rules
  '(; cook moves
     ((?w [cook stove ?x]) :=> (?w [cook hatch ?x]))
     ((?w [cook hatch ?x]) :=> (?w [cook stove ?x]))

     ; waiter moves
     (([waiter hatch ?x] ?c) :=> ([waiter table ?x] ?c))
     (([waiter table ?x] ?c) :=> ([waiter hatch ?x] ?c))

     ; cook gets food
     ((?w [cook stove :nil]) :=> (?w [cook stove food]))

     ; waiter gives food
     (([waiter table food] ?c) :=> ([waiter table :nil] ?c))

     ; cook & waiter exchange
     (([waiter ?p :nil][cook ?p ?x])
       :=> ([waiter ?p ?x][cook ?p :nil]))
     ))

; two functions to run the rules
(defn apply-rule [rule state]
  (mlet [ '[?ante :=> ?consq] rule]
    (mif [(? ante) state]
      (mout (? consq))
    )))

;; version #1
(defn serve-lmg [state]
  (for [rule rules]
      (apply-rule rule state)
      ))

user=> (serve-lmg '([waiter table :nil][cook stove :nil]))
(([waiter table :nil] [cook hatch :nil]) nil nil
 ([waiter hatch :nil] [cook stove :nil]) ([waiter table :nil] [cook
stove food]) nil nil)
```

notice version#1 is scatterered with nil's for the rules which do not fire, these need cleaning up...

```
;; version #2
(defn serve-lmg [state]
  (remove nil?
    (for [rule rules]
      (apply-rule rule state)
      )))
```

```
user=> (serve-lmg '([waiter table :nil][cook stove :nil]))
(([waiter table :nil] [cook hatch :nil]) ([waiter hatch :nil] [cook
stove :nil]) ([waiter table :nil] [cook stove food]))

user=> (serve-lmg '([waiter table bucket][cook stove vodka]))
(([waiter table bucket] [cook hatch vodka]) ([waiter hatch bucket]
[cook stove vodka]))
```

we can also do this by using some of the for modifiers...

```
;; or version #3
(defn serve-lmg [state]
  (for [rule rules
        :let [succ (apply-rule rule state)]
        :when succ
        ]
    succ))

user=> (serve-lmg '([waiter table bucket][cook stove vodka]))
(([waiter table bucket] [cook hatch vodka])
 ([waiter hatch bucket] [cook stove vodka]))
```

finally we can run a search...

```
user=> (breadth-search
           '([waiter table :nil][cook stove :nil])
           '([waiter table food][cook hatch :nil]) serve-lmg)
(([waiter table :nil] [cook stove :nil])
 ([waiter hatch :nil] [cook stove :nil])
 ([waiter hatch :nil] [cook stove food])
 ([waiter hatch :nil] [cook hatch food])
 ([waiter hatch food] [cook hatch :nil])
 ([waiter table food] [cook hatch :nil]))
```

review

The rules we defined earlier allow us to deal with complexities that we could not handle before. Some of the rules introduce generalisations that make them more useful than our earlier attempts.

Some rules abstract away the circumstances of all but one of the agents...

```
((?w [cook stove ?x]) :=> (?w [cook hatch ?x]))
((?w [cook hatch ?x]) :=> (?w [cook stove ?x]))
```

Others generalise objects and/or locations...

```
(([waiter ?p nil)[cook ?p ?x])
       :=> ([waiter ?p ?x][cook ?p :nil)))
```

This is ok as far as it goes but as we extend our world (& our state descriptions) our rules will need to change. Consider: does the current rule structure extend/scale ok under the following situations...

- more agents are added to the world
- more objects are added
- more locations
- different object types (eg: carryable, static, climbable, etc)
- different agent capabilities (pick up objects, open doors, climb on tables, etc)

If so... how would you modify the rules?

If not... what do you suggest?