legal move generators in Lisp – take 2

NB: many of the examples here use the matcher and utility functions 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

```
(defun 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
(defparameter *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)

; a LMG
(defun words-lmg (word)
   (-> *words* word))

> (words-lmg 'cost)
(most lost coat cast)

; using this with search
> (breadth-search 'coat 'loot #'words-lmg)
(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))


(defmatch cafe ( (waiter at hatch holding plates
                  cook at hatch holding nothing) )
  '(waiter at hatch holding nothing
    cook at hatch holding plates))


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

> (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 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:

```
(defparameter *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)))
    ))
```

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 for a general/outline understanding)

```
(defun serve-lmg (state)
  (loop for rule in *cafe-rules*
      when (equal (first rule) state)
      collect (third rule)
        )
  )

> (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)))
```

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

the Lisp matcher allows use of wildcards as follows
- = single symbol, discarded
- == multiple symbols, discarded
- ? single symbol, bound to matcher variable
- ?? multiple symbol, bound to matcher variable

NB: use ?x to bind a value to the matcher variable "x" and #?x to retrieve it example

```
(defmatch calc ((?x plus ?y))
  (+ #?x #?y))

(defmatch calc ((?x minus ?y))
  (- #?x #?y))

> (calc '(5 plus 4))
9

> (calc '(5 minus 4))
1
```

note also the use of match>>

```
(defmatch cafe ( (waiter at hatch holding ?thing ??cook) )
  (match>> '(waiter at table holding ?thing ??cook)))

> (cafe '(waiter at hatch holding melon
             cook up tree with an armadillo))
(waiter at table holding melon cook up tree with an ...)

; use pprint to format complete output
> (pprint (cafe '(waiter at hatch holding melon
             cook up tree with an armadillo)))
(waiter at table holding melon cook up tree with an armadillo)
```

the ideas above can be combined to produce a rule based LMG as below...

```
; the rules
 (defparameter *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
(defun apply-rule (rule state)
  (mlet ((first rule) state)
        (match>> (third rule))
        ))
```

```
(defun serve-lmg (state)
  (remove nil
          (loop for rule in *rules*
              collecting (apply-rule rule state)
                )
          ))


> (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)))
```

another (better?) use of loop directives could give us...

```
(defun serve-lmg2 (state)
  (loop for rule in *rules*
      when (apply-rule rule state)
      collect it
        ))

> (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)))


> (breadth-search '((waiter table nil)(cook stove nil))
                  '((waiter table food)(cook hatch nil))
                  #'serve-lmg)
(((waiter table food) (cook hatch nil))
 ((waiter hatch food) (cook hatch nil))
 ((waiter hatch nil) (cook hatch food))
 ((waiter hatch nil) (cook stove food))
 ((waiter hatch nil) (cook stove nil))
 ((waiter table nil) (cook stove nil)))
```

NB: you could also write apply-rule with the matcher, like this...

```
;; an alternative apply rule
(defmatch apply-rule2 ((?ante => ?conseq) state)
  (mlet (#?ante state)
        (match>> #?conseq)
        ))
```

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?