world states & operators

NB: to run these examples you need to load the definitions of functions associated with applying operators (*ops-search*, *apply-mop*, etc).

These examples follow on from the last set of seminar traces which examined writing LMGs for a simple world involving a cook and a waiter exchanging food through a serving hatch. The last approach concentrated on *states*. We wrote LMGs Which stated how one state could be transformed into others. The approach presented here concentrates instead on the kind of activities that can take place. This approach ends up with neater solutions which are typically shorter and easier to extend. As you will see in the lectures, by developing LMGs based on activities we can also write goal directed problem solvers which offer large performance gains over conventional search algorithms.

A simpler & less error-prone style for representing states is to use sets of tuples

```
(defparameter *start*
  '((in cook kitchen)
    (at cook stove)
    (holds cook nil)
    (in waiter cafe)
    (at waiter table)
    (holds waiter nil)
    (on soup stove)
    (on plates table)
    ))
```

Taking this approach we define operators in terms of their preconditions & effects and we can conveniently describe their effects by specifying which tuples are deleted by operators and what new tuples are added...

```
(defparameter *grab*
  '((pre . ((holds ?person nil)
            (at ?person ?place)
            (on ?thing ?place)
            ))
    (del . ((holds ?person nil)
            (on ?thing ?place)))
    (add . ((holds ?person ?thing)))
    (txt . (?person grabs ?thing off ?place))
    ))
```

The structure chosen above is one that works with our mechanism to apply matching operators, its called *apply-mop*...

```
> (apply-mop *grab* *start*)
(((path (cook grabs soup off stove)) (holds cook soup)
  (on plates table) (holds waiter nil) (at waiter table)
  (in waiter cafe) (at cook stove) (in cook kitchen))
 ((path (waiter grabs plates off table))
  (holds waiter plates) (on soup stove) (at waiter table)
  (in waiter cafe) (holds cook nil) (at cook stove)
  (in cook kitchen)))
```

We can define a suite of operators in this form...

```
(defparameter *drop*
  '((txt . (?person puts ?thing on ?place))
    (pre . ((holds ?person ?thing)
            (at ?person ?place)))
    (del . ((holds ?person ?thing)))
    (add . ((holds ?person nil)
            (on ?thing ?place)))
    ))
```

move operators are often a little more complex & require more information about the world domain. For various reasons (both functional & performance) we choose to split dynamic state information from static world information. Below is one (of many) possible approaches to dealing with movement around the world...

```
(defparameter *world*
  '((in stove kitchen)
    (in hatch kitchen)
    (next-to hatch stove)
    (next-to stove hatch)
    (in table cafe)
    (in hatch cafe)
    (next-to hatch table)
    (next-to table hatch)
    ))

(defparameter *move*
  '((txt . (?person moves from ?place1 to ?place2))
    (pre . ((at ?person ?place1)
            (in ?person ?room)
            (next-to ?place1 ?place2)
            (in ?place2 ?room)))
    (del . ((at ?person ?place1)))
    (add . ((at ?person ?place2)))
    ))

> (apply-mop *move* *start* :world *world*)
(((path (cook moves from stove to hatch)) (at cook hatch)
  (on plates table) (on soup stove) (holds waiter nil)
  (at waiter table) (in waiter cafe) (holds cook nil)
  (in cook kitchen))
 ((path (waiter moves from table to hatch))
  (at waiter hatch) (on plates table) (on soup stove)
  (holds waiter nil) (in waiter cafe) (holds cook nil)
  (at cook stove) (in cook kitchen)))
```

finally we can collect together all the operators & use them with an operator search mechanism, the one you have access to currently is v.slow, we will discuss reasons for this

```
(defparameter *all-ops*
  '((grab (txt . (?person grabs ?thing off ?place))
          (pre . ((holds ?person nil)
                  (at ?person ?place)
                  (on ?thing ?place)
                  ))
          (del . ((holds ?person nil)
                  (on ?thing ?place)))
          (add . ((holds ?person ?thing)))
          )
    (drop (txt . (?person puts ?thing on ?place))
          (pre . ((holds ?person ?thing)
                  (at ?person ?place)))
          (del . ((holds ?person ?thing)))
          (add . ((holds ?person nil)
                  (on ?thing ?place)))
          )
    (move (txt . (?person moves from ?place1 to ?place2))
          (pre . ((at ?person ?place1)
                  (in ?person ?room)
                  (next-to ?place1 ?place2)
                  (in ?place2 ?room)))
          (del . ((at ?person ?place1)))
          (add . ((at ?person ?place2)))
          )
    ))

> (ops-search *start* '((on soup table))
                          *all-ops* :world *world*)
```

... intermediate states deleted from output to aid readability...

```
((path (cook grabs soup off stove)
       (waiter moves from table to hatch)
       (cook moves from stove to hatch)
       (cook puts soup on hatch)
       (waiter grabs soup off hatch)
       (waiter moves from hatch to table)
       (waiter puts soup on table))
  (on soup table) (holds waiter nil) (at cook hatch)
  (on plates table) (in waiter cafe) (in cook kitchen)
  (holds cook nil) (at waiter table)))
```