The SXL Lisp Matcher - an overview

brief

The SXL Lisp Matcher is a symbolic pattern matcher for Common Lisp which provides macros for iterating patterns over collections of data, a new type of method which specialises on structures of data (think Prolog, Haskell, etc) and a range of other facilities.

This document gives a quick introduction to some of the matcher's features. For a copy of the matcher and for more documentation visit the "Lisp Matcher" downloads page of www.agent-domain.org.

methods & patterns

The following example shows how to define matcher methods. The methods, called "calculate", are of little use but highlight the basic syntax of the matcher...

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

(defmatch calculate ((?x minus ?y))
  (- #?x #?y))
```

The first method is defined to be applicable to an argument matching the pattern (?x plus ?y). The use of the "?" character prefixes the name of a matcher variable in common with other pattern matchers. The pattern (?x plus ?y) will match with any three element list containing the symbol "plus" as its second element.

The use of symbols like "#?x" in the body of the method definition is to retrieve the value of a matcher variable.

calculate can be used as follows:

```
> (calculate '(5 plus 3))   ==>  8

> (calculate '(5 minus 3))   ==>  2
```

In each case the calculate method used is that which matches the argument provided.

The matcher which underpins the use of DEFMATCH provides various matcher directives/tags. In addition to the single "?" prefix for matcher variables (for matching with single list elements). The "??" prefix will bind to zero or more list elements. This allows methods to be defined in a Prolog-like style and provides an alternative approach to specifying recursive forms. An example of this is a pair of list length methods...

```
(defmatch len1 (())  0)

(defmatch len1 ((??x))
  (1+ (len1 (rest #?x))))

> (len1 '(herring haddock hake))   ==>  3
```

Note that where more than one method is applicable (because two or more match the argument provided) the method used is always the one which was defined first, like Prolog. Unlike Prolog there is no backtracking so other methods will never be invoked.

a matcher form of LET

In addition to the implicit use of the matcher when matcher methods are invoked it can be used explicitly through a small number of macro calls. The most basic of these is a LET form called MLET which provides a convenient mechanism for destructuring data.

An example of MLET in use follows, note that matcher forms other than DEFMATCH need their patterns to be quoted - this allows patterns to be dynamically constructed or stored in variables. Note also that the pattern & data are enclosed in additional brackets in the MLET macro, this is discussed in more detail in other documentation.

```
> (mlet ('(the ?subj ate the ?obj)
         '(the mouse ate the cheese))
    (list #?subj #?obj))

==> (MOUSE CHEESE)
```

Two other matcher tags act as wildcards. These are "=" and "==" which are the matching tag for a single element wildcard and multiple element wildcard respectively.

```
> (mlet ('(= ?2nd == ?last) '(a b c d e f g))
    (list #?2nd #?last))

==> (B G)
```

foreach & forevery

FOREACH and FOREVERY are two iterative constructs which work by passing patterns over lists of possibly matching statements, they are based on the keywords of the same name in POP-11. One obvious use for these forms is with a set of related facts as in the following examples. With both FOREACH and FOREVERY forms the result returned is a collection of the results produced by their body of statements for each successful match.

```
(defvar data
  ;; a simple set of facts concerning 4 boxes
  '((isa b1 box) (color b1 red)  (size b1 large)
    (isa b2 box) (color b2 red)  (size b2 small)
    (isa b3 box) (color b3 blue) (size b3 small)
    (isa b4 box) (color b3 blue) (size b4 small)
    (supports b1 b2) (supports b2 b3)
    ))
```

FOREACH iterates with single patterns, FOREVERY with multiple patterns...

```
> (foreach ('(size ?x small) data)
     (format T "~&~a is small" #?x)    ; side effect
     #?x)                              ; result values

==> B2 is small
==> B3 is small
==> B4 is small
==> (B2 B3 B4)


> (forevery ('((isa ?b box)(color ?b red)
                (supports ?b ?x)) data)
     (format T "~&~a is a red block which supports ~a"
                #?b #?x)
     (list 'red-block #?b))          ; value returned

==> B1 is a red block which supports B2
==> B2 is a red block which supports B3
==> ((RED-BLOCK B1) (RED-BLOCK B2))
```

example 1 – rules

In this example we consider how to write a mechanism to apply simple *Expert System* style rules.

Given a set of facts like...

```
  ((parent Sarah Tom)  (parent Steve Joe)  (parent Sally Sam)
   (parent Ellen Sarah) (parent Emma Bill) (parent Rob  Sally)))
```

We want to have the ability to apply a rule like the one below to generate new facts...

```
  (rule 15 (parent ?a ?b) (parent ?b ?c) => (grandparent ?a ?c))
```

We can do this using defmatch (to deconstruct the rule) and forevery (to repeatedly apply it over the facts. Note: in the following code I also use a function $+ which performs a set-union operation. This, and other, set functions/operators are available from my collection of general utilities (check www.agent-domain.org & look at the Lisp resources page).

```
;; apply rule definition

(defmatch apply-rule ((rule ?n ??antecedents => ??consequents) facts)
  (forevery (#?antecedents facts)
      (setf facts ($+ (match>> #?consequents) facts)))
   facts))
```

```
;; apply rule in use

(defparameter family
 '((parent Sarah Tom)  (parent Steve Joe)  (parent Sally Sam)
   (parent Ellen Sarah) (parent Emma Bill) (parent Rob  Sally)))

> (apply-rule
    '(rule 15 (parent ?a ?b) (parent ?b ?c) => (grandparent ?a ?c))
    family)

((grandparent Rob Sam) (grandparent Ellen Tom)
 (parent Sarah Tom) (parent Steve Joe) (parent Sally Sam)
 (parent Ellen Sarah) (parent Emma Bill) (parent Rob Sally))
```

To wrap up this example (eventhough it does not need the matcher) I write a simple forward chainer to exhaustively apply a set of rules to a collection of facts, inferring any new facts that can be generated by the rules. This example uses $+ as above and also uses $= which returns true iff two sets are equal (ie: contain the same facts).

```
;; the forward chainer

(defun fwd-chain (rules facts)
  (let (old-facts)
    (loop (setf old-facts facts)
          (setf facts
            (reduce #'$+
               (mapcar #'(lambda (r) (apply-rule r facts)) rules)))
          (when ($= old-facts facts) (return facts))
        )))


;; another set of facts

(defvar facts1
 '((big elephant) (small mouse) (small sparrow) (big whale)
   (on elephant mouse)))

;; a collection of rules

(defvar rules1
 '((rule 1 (heavy ?x)(small ?y)(on ?x ?y) => (squashed ?y) (sad ?x))
   (rule 2 (big ?x)    => (heavy ?x))
   (rule 3 (light ?x) => (portable ?x))
   (rule 4 (small ?x) => (light ?x))  ))

;; running the rules

> (fwd-chain rules1 facts1)
  ((portable sparrow) (portable mouse) (squashed mouse)
    (sad elephant)  (heavy whale) (heavy elephant)
    (light sparrow) (light mouse) (big elephant) (small mouse)
    (small sparrow) (big whale)   (on elephant mouse))
```

example 2 – state changing operators

This example considers using General Problem Solver (GPS) style operators used within a blocks world environment. Since first presented by Newell and Simon this has become a classic example of symbolic computation.

At one level of abstraction, commands like (pick-up ?x) and (drop-it-on ?y) are issued to a virtual robot existing in a simple blocks-world environment. The robot carries out these commands by effecting changes to the description of its virtual world. At another level, individual operators are defined in terms of their preconditions (what needs to exist in the world for the operator to be used) and their effects.

The effects of operators are defined in two parts: what is no longer true about the world after the operator is applied (facts about the world description that the operator deletes) and what becomes true (facts about the world description that the operator adds)

In this way simple operators can be described in terms of three sets of facts preconditions deletions and additions.

Using the type of world description below...

```
(defparameter blocks
 '((isa b1 block) (isa b2 block)
   (isa p1 pyramid) (supports b1 p1)
   (supports floor b1)
   (supports floor b2)
   (cleartop p1) (cleartop b2)
   (cleartop floor) (holds nil)))
```

...the (pick-up ?x) operator could be defined as...

```
(defparameter pick-up     ; pick up object ?x
  '((pre (holds nil) (cleartop ?x)
         (supports ?y ?x))
    (del (holds nil) (supports ?y ?x))
    (add (holds ?x)  (cleartop ?y))
   ))
```

Note: pick-up is an association list. I use the function -> (pronounced lookup) to access the different part of the operator. "->" is defined in the utils file (check www.agent-domain.org & look at the Lisp resources page) it is a varion of the Common Lisp assoc function. It works as follows...

```
;; using ->

> (-> pick-up 'del)
→ ((holds nil) (supports ?y ?x))
```

Given the kind of operator description above, a generalised apply operator function can be built around the use of the matcher (for more details of the matcher check the user guide).

```
(defun apply-op (op object world)
 (let ((pre (-> op 'pre))
       (del (-> op 'del))
       (add (-> op 'add))
       )
   (all-present
     (pre world `((x ,object)))
       ($+ (match>> add)
           ($- world (match>> del)))
      )))


> (apply-op pick-up 'p1 blocks)
→ ((cleartop b1) (holds p1)
    (cleartop floor) (cleartop b2)
    (cleartop p1) (supports floor b2)
    (supports floor b1)
    (isa p1 pyramid)
    (isa b2 block) (isa b1 block))
```

We can finish-off this example by creating more GPS operators and a mechanism which will apply a series of them. Notice that with simple operators like the ones used here the problems of defining and using operators have become abstract and conceptual. The difficulties associated with programming the *actions* of operators have largely been removed and are dealt with by the matcher.

```
(defparameter ops
 '((pick-up     ; as defined above
    (pre (holds nil) (cleartop ?x)
         (supports ?y ?x))
    (del (holds nil) (supports ?y ?x))
    (add (holds ?x)  (cleartop ?y))
   )
  (drop-on  ; puts one obj on another
   (pre (holds ?obj) (cleartop ?x))
   (del (holds ?obj) (cleartop ?x))
   (add (holds nil)(supports ?x ?obj))
  )))

;; arm-controller is a function which takes a series of commands

(defun arm-controller (world commands)
  ;; this provides textual output
  (dolist (com commands)
    (format t "~2&Applying ~a..." com)
    (setf world
      (apply-op (-> ops (first com))
                  (second com) world))
    (format t "ok~%")
    (pprint world)
    ))
```

```
> (arm-controller blocks
       '((pick-up p1) (drop-on b2)))

→ Applying (pick-up p1)...ok
→ ((cleartop b1) (holds p1)
    (cleartop floor) (cleartop b2)
    (cleartop p1) (supports floor b2)
    (supports floor b1)
    (isa p1 pyramid)
    (isa b2 block) (isa b1 block))

→ applying (drop-on b2)...ok
→ ((supports b2 p1) (holds nil)
    (isa b1 block) (isa b2 block)
    (isa p1 pyramid)
    (supports floor b1)
    (supports floor b2) (cleartop p1)
    (cleartop floor) (cleartop b1))
nil
```