Intelligent Systems Programming – some essential tools

Simon Lynch

Programming tools

This paper briefly examines some programming tools to help building intelligent systems in Lisp. With the exception of the mapping functions (which are part of the Lisp language) these tools are provided as add-ons to Lisp (either as part of the matcher or defined in utils.cl). basic

set operators - convenience utilities

A suite of set operators are supplied, their names all start with "$" (to be read "set") and are followed by mnemonic mathematical or boolean. Examples include $- (set difference), S+ (set union), $* (intersection) and $<= (is-subset-of).

mapping functions - reducing the need for recursion

Mapping functions in Lisp apply some function to each item in a sequence of data items removing the need for programmers to develop their own iterative or, more usually, recursive mechanisms to work through sequences. The simplest use of mapping functions is with some predefined Lisp function. For example: **mapcar** is a mapping function which collects the results of applying a function to each element in a sequence; **second** is a function which retrieves the second element of a list (note: throughout this paper lines prefixed ">" indicate interaction with the Lisp system, lines prefixed "→" indicate output from the Lisp system. For the sake of readability input & output).

```
(defvar fruit
      '((color cherry red)
        (color apple  green)
        (color banana yellow)
        (color kiwi   green)
        ))

> (mapcar #'second fruit)
→ (CHERRY APPLE BANANA KIWI)
```

Calls to functions like **mapcar** often exploit Lisps ability to define anonymous functions in-line. For example:

```
> (mapcar #'(lambda (x) (list (first x)(third x)))
                fruit)
→ ((COLOR RED) (COLOR GREEN)
    (COLOR YELLOW) (COLOR GREEN))
```

In addition to mapcar, Lisp provides mapping functions like remove-if, remove-if-not, find-if and reduce.

To illustrate how mapping functions can ease logical complexity consider how a set intersection function could be written first using a classically Lisp-like recursive solution and then using remove-if-not to discard elements from the first set which are not members of the second (see below). Not only is the second approach more concise it also handles the conditional recursion/iteration implicitly and, likewise, the deconstruction and reconstruction of the data-structures involved.

```
;; example 1
(defun intersect1 (s1 s2)
  (cond ((null s1)
         nil)
        ((member (first s1) s2)
         (cons (first s1)
               (intersect1 (rest s1) s2)))
        (t
         (intersect1 (rest s1) s2))
        ))

;;example 2
(defun intersect2 (s1 s2)
  (remove-if-not #'(lambda (x) (member x s2)) s1))
```

association list utilities - helping to deconstruct data

A small set of functions which manipulate association lists are provided to encourage new Lisp programmers to use symbolic data-structures and wean them off a bias towards numeric indexing. This helps them to concentrate on simple knowledge representation schemes at a conceptual level without having to develop complicated Lisp code to manipulate them. The function shown here, as an example, is **->**, which retrieves data from an association list.

```
(defvar country
'((Africa
   (Botswana (Capital . Gaborone) (Population . 1.5))
   (Zimbabwe (Capital . Harare)   (Population . 11)))
  (Asia
   (India   (Capital . New-Delhi) (Population . 200))
   (Sri-Lanka (Capital . Colombo)  (Population . 15))
  )))

> (-> country 'Africa 'Zimbabwe 'Capital) → HARARE
```

the matcher - deconstructing data, reducing recursion, aiding design

Background to the Lisp matcher – an optional read

Pattern matching has been employed as a viable tool in Lisp for more than 30 years. Some (now dated) languages built on top of Lisp offered pattern matching (Micro-Planner for example [Sussman et al]) and at least one version of Lisp, Qlisp [Sacerdoti et al.] had matching capabilities. After overcoming the optimism of early language processing systems like Eliza [Weizenbaum] the general view has been that pattern matching alone is not a successful basis for AI systems. Perhaps because of this there is no standard pattern matcher in Lisp, instead the implementation of a pattern matcher is left up to the discretion of programmers. As a result, while pattern matching and unification mechanisms appear in some Lisp applications, they are mostly simple functions for specific and limited use.

Other AI languages take different approaches to pattern matching. Prolog uses a method of matching and unification to invoke its clauses. POP-11 provides a sophisticated pattern matcher, capable of binding values to POP-11 variables, as part of the language [Barrett et al]. POP-11 was initially designed both as a language for novices and as a language for implementing IS. The pattern matcher was seen as a key tool for both of these programmer groups. POP-11 programmers use their matcher for small conveniences which would not, on their own, justify the development of a matcher if one had not been supplied. Many examples of this can be found in POP-11 texts (see [Gazdar & Mellish] for examples).

The matcher provided here is influenced by the use of the POP-11 matcher and by the mechanism of clause invocation which occurs in Prolog. It provides many of the features offered by the POP-11 matcher and also allows matcher methods to be specified using a DEFMATCH form. These superficially resemble Prolog clauses (the term matcher method is used in preference to clause because it is considered less ambiguous). Their design is such that they are called in the same way as Lisp functions (or CLOS methods), they can be traced with a standard Lisp tracer and the body of their definition can contain any statements legal in the body of a Lisp function (see [Lynch & Barker] for implementation details).

The following section describes the specification of patterns and the definition and use of matcher methods. They introduce a new LET form for associating values with matcher variables and describe two constructs, FOREACH and FOREVERY, which iterate patterns over sets of lists.

methods & patterns

The following example demonstrates the definition of pattern matcher methods. The methods, called "calculate", are of little use but highlight the basic syntax of the matcher described here.

```
(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 [Norvig], [Barrett et al]. 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 (this syntax combines the # macro dispatch character with "?", a combination reserved for user applications in Common Lisp [Steele]).

calculate is 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 structuring 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, note that matcher forms other than DEFMATCH need their patterns to be quoted - this allows patterns to be dynamically constructed or stored in variables.

```
> (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 *blocks*
 ;; a simple set of facts concerning 4 boxes
 '((isa   b1 cube)  (isa   b2 wedge) (isa   b3 cube)
   (isa   b4 wedge) (isa   b5 cube)  (isa   b6 wedge)
   (color b1 red)   (color b2 red)   (color b3 red)
   (color b4 blue)  (color b5 blue)  (color b6 blue)
   (on    b4 b1)    (on    b2 b3)    (on    b5 b6)
 ))
```

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

```
> (foreach ('(isa ?b cube) *blocks*)
    (format t "~&~a is a cube" #?b)   ; side effect
    #?b)                              ; result values
→ b1 is a cube
→ b3 is a cube
→ b5 is a cube
→ (b1 b3 b5)

> (forevery ('((isa ?b cube)(color ?b red)(on ?x ?b))
              *blocks*)
    (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 b4
→ b3 is a red block which supports b2
→ ((red-block b1) (red-block b3))
```

programmed examples

This section examines three specific IS programming problems and investigates using the software tools introduced above to reduce the programming burden. These examples are taken from introductory modules to Symbolic Computation and Artificial Intelligence.

example 1

The first example examines how a generalised query mechanism can be built to retrieve information from a sets of statements where each statements is a triple of the form: (relation object value).

For the sake of experimentation this example presents a simple world environment describing a collection of blocks. In Lisp this can be defined as...

```
(defvar *blocks*
 '((isa   b1 cube)  (isa   b2 wedge) (isa   b3 cube)
   (isa   b4 wedge) (isa   b5 cube)  (isa   b6 wedge)
   (color b1 red)   (color b2 red)   (color b3 red)
   (color b4 blue)  (color b5 blue)  (color b6 blue)
   (on    b4 b1)    (on    b2 b3)    (on    b5 b6)
 ))
```

As demonstrated above, the matcher's *foreach* form can be used to retrieve the names of objects satisfying a specified relation from this type of structure.

```
> (foreach ('(isa ?obj wedge) *blocks*) #?obj)
→ (b2 b4 b6)
```

By using the matcher it is it is possible to build a general purpose function *lookup* defined as follows...

```
(defun lookup (pair triples)
  (mlet ('(?relation ?value) pair)
        (foreach ( '(?relation ?obj ?value) triples)
           #?obj)))

> (lookup '(isa cube)  *blocks*) → (b1 b3 b5)
> (lookup '(color red) *blocks*) → (b1 b2 b3)
```

Set operators like $* (set intersection) and $+ (set union) can be used to combine the results of different lookup operations so multiple queries can be satisfied.

```
> ($* (lookup '(isa cube)  *blocks*)
      (lookup '(color red) *blocks*))
→ (b3 b1)
```

Using these ideas a generalised query function can be built which maps pairs like (isa cube) and (color red) over the lookup function and reduces results using $* (in the case of logically ANDed queries).

```
(defun query-and (pairs triples)
  (reduce #'$*
     (mapcar #'(lambda (p) (lookup p triples)) pairs)
    ))

> (query-and '((isa cube) (color red)) *blocks*)
→ (b3 b1)
```

The end result is concise (requiring less than 10 lines of program code) but more importantly it does not require a recursive solution or code which uses a procedural approach to pull apart and rebuild a data-structure

Logically ORed queries can be similarly handled using $+ in place of $*. The final query function shown below can then be used in conjunction with two variables Qand & Qor (introduced for readibility) which deal with ANDed results and ORed results.

```
(defvar Qand #'$*)
(defvar Qor  #'$+)

(defun query (logic-op pairs triples)
  (reduce logic-op
     (mapcar #'(lambda (p) (lookup p triples)) pairs)
          ))

>> (query Qand '((isa cube)(color red)) *blocks*)
→ (b3 b1)
>> (query Qor '((isa cube)(color red)) *blocks*)
→ (b5 b1 b2 b3)
```

example 2

The second example considers the specification of rules and the development of functions to apply them. Some student texts simplify rule application by using rules that have no matching capability. Using a common example rules may be written...

(Rule 32 (has fido hair) => (is fido mammal))

The preconditions of this kind of rule are tested for equality with known facts which means that such a rule could conclude nothing given the fact (has lassie hair). This is an over simplification which can hide important issues in the design of expert systems and other rule-based inference engines. Using the matcher allows more realistic rules to be developed - the rule above would be written...

(Rule 32 (has ?x hair) => (is ?x mammal))

Rule application can be achieved directly using the forevery form...

```
(setf family
   '((parent-of Sarah Tom)  (parent-of Steve Joe)
     (parent-of Sally Sam)  (parent-of Ellen Sarah)
     (parent-of Emma  Bill) (parent-of Rob   Sally)
    ))

; applying ((parent-of ?a ?b) (parent-of ?b ?c))
;               => (grandparent ?a ?c)
; NB: match>> replaces ?x forms with their
;     appropriate values

> (forevery ('((parent-of ?a ?b) (parent-of ?b ?c))
             family)
      (match>> '(grandparent ?a ?c)))
→  ((GRANDPARENT ELLEN TOM) (GRANDPARENT ROB SAM))
```

This approach can be used to develop a general purpose mechanism for applying rules to facts which return an updated set of facts. The function uses the matcher to deconstruct a rule and then forevery to

apply it. Notice that the mechanism for rule deconstruction and the details of repeated rule application are all removed from the programmer who is left to focus on rule antecedents, consequents and facts.

```
(defun apply-rule (r facts)
  (mlet ('(rule ?n ??antecedents => ??consequents) r)
        (forevery (#?antecedents facts)
            (setf facts
              ($+ (match>> #?consequents) facts)))
        facts))

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

→ ((GRANDPARENT ROB SAM) (GRANDPARENT ELLEN TOM)
   (PARENT-OF SARAH TOM) (PARENT-OF ELLEN SARAH)
   (PARENT-OF STEVE JOE)....)
```

The example is completed with a mechanism which repeatedly applies a set of rules to update facts, continuing until the rules are unable to generate any new inference - acting as a forward chaining process. It is only at this final stage that the program code explicitly uses any repetitive construct - in this case an iterative form. Some Lisp programmers may question the structure of the iteration in the function preferring to replace **let** and **loop** with **do** or make more use of the loop macro facilities. However the function is presented as shown since this does not require students to know any details of the loop macro or work with Lisp's do/do* forms since some students report that the unusual structure of do/do* is hard to read.

```
 (defun fwd-chain (rules facts)
  (let (old-facts)            ; setup new variable
    (loop                     ; loop until return
      (setf old-facts facts)  ; save existing facts
      (setf facts             ; do one pass of rules
        (reduce #'$+
            (mapcar #'(lambda (r)
                        (apply-rule r facts))
                    rules)))
      (if ($= old-facts facts) ; are facts updated?
          (return facts))      ; if not finished
      )))                      ; so quit
```

```
; a slightly larger set of facts & rules

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

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

> (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) (ONTOP ELEPHANT MOUSE))
```

example 3

This third example examines the use of General Problem Solver (GPS) style operators used within a blocks world environment. Since first presented by Newell and Simon [Newell & 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 (parts of the world description that the operator deletes) and what becomes true (parts of the world description that the operator adds). In this way simple operators can be described in terms of three sets of facts **pre**conditions **del**etions and **add**itions.

Using the type of world description below, the (pick-up ?x) operator can be defined as shown.

```
(defvar 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)))

(defvar pick-up       ; pick up the 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 so its parts
; can be accessed using the -> function

> (-> pick-up 'del) → ((HOLDS NIL) (SUPPORTS ?Y ?X))
```

Given the kind of operator description above, a generalised apply operator function can be built.

```
(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)...)
```

This example is concluded by creating more GPS operators and a mechanism which will apply a series of commands. Notice below, that with simple operators like the ones used here the problems of defining and using operators have become quite abstract and conceptual. The difficulties associated with programming the *actions* of operators have largely been removed and are dealt with by the matcher. Programmers are able to concentrate on the mechanics of applying these types of operators in symbolic world environments without spending large amounts of time writing Lisp functions. As an additional benefit, the use of matcher patterns encourages a more declarative approach.

```
(defvar 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 object on top of another
     (pre (holds ?obj) (cleartop ?x))
     (del (holds ?obj) (cleartop ?x))
     (add (holds nil) (supports ?x ?obj))  )))

; arm-control accepts a series of commands like
;   (pick-up b1) & provides textual output

(defun arm-control (world commands)
  (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-control 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
```

References

[Barrett et al] R.Barrett, A.Ramsay & A.Sloman. POP-11: A Practical Language for Artificial Intelligence. Ellis Horwood, 1985.

[Dennett] D.Dennett. 1995. Darwins Dangerous Idea. Penguin. P488-489.

[Gazdar & Mellish] G.Gazdar & C.Mellish. 1989. Natural Language Processing in POP-11, An Introduction to Computational Linguistics. Addison Wesley.

[Lynch & Barker] S Lynch, D J Barker. 1999. Using a Pattern Matcher to Build Adaptable Interfaces for Lisp Modules. Conference Paper & Proceedings, Lisp User Group Meeting, San Francisco.

[Newell & Simon] A.Newell & H.A.Simon. 1963. GPS, a program that simulates human thought. Computers and Thought. ed. E.Feigenbaum & J.Feldman. McGraw-Hill, New York.

[Norvig] P.Norvig. 1992. Paradigms of Artificial Intelligence Programming. Morgan Kaufman, P.154-162,178-187

[Sacerdoti et al.] E.Sacerdoti, R.Reboh, D.Sagalowicz, R.Waldinger, B.Wilber. 1976. QLISP-a language for the interactive development of complex systems. AFIPS National Computer Conference, P349-356.

[Sinha] C.G.Sinha. 1996. Theories of symbolization and development. Handbook of Human Symbolic Evolution. Clarendon Press, Oxford. pp204-238.

[Steele] G.Steele. 1990. Common Lisp The Language (2nd Ed). Digital Press, p.531.

[Sussman et al] G.Sussman, T.Winograd & E.Charniak. 1970. Micro-planner reference manual. A.I.Memo 203, Artificial Intelligence Laboratory, MIT.

[Weizenbaum] J.Weizenbaum. 1965. ELIZA - a computer program for the study of natural language communication between man and machine. Communications of the ACM 9(1):36-44.