Softening the Complexity of Intelligent Systems Programming

Simon Lynch
Department of Computing and Information Systems
Botswana Accountancy College, Botswana
simonl@bac.ac.bw

Keywords: POP-I.B.barriers to programming, POP-III.A.recursion.

Abstract

Intelligent Systems (IS) are often complex to implement by their nature. This presents IS tutors with a problem if they want to encourage their students to explore practical implementation issues. If tutors wish to give students concise, easy to understand, practical examples of IS they are often forced to simplify systems to a point where their functionality is no longer realistic and may additionally hide important practical issues. Alternatively tutors may encourage students to build small but real systems. This requires students to possess advanced programming abilities and takes time, limiting what can be covered in other theoretical aspects of an IS course.

As the nature of computing degrees becomes more diverse, and with it the background of students sitting IS modules, a third alternative is preferred. This paper explores an alternative which provides a suite of programming tools designed to aid students' progress with practical symbolic computation. The paper describes these tools and demonstrates their efficacy in simplifying practical aspects of IS programming.

Introduction

New students often complain that practical, symbolic Artificial Intelligence programming is difficult. However there is nothing fundamental about Symbolic Computation (SC) that should indicate this, indeed symbolism is one of the major features that defines human intelligence and culture (Sinha, 1996). At one level there is little difference between symbolic and non-symbolic computation. A non-symbolic function (like a mean calculator for example) takes numeric data in one end and squirts new, derived numeric data (the mean) out of the other end. Symbolic functions carry out similar processes with symbolic rather than numeric data. The utility of either function is determined by the user who recognises its purpose.

Without examining the experience of students we may predict that they would adapt more readily to describing symbol manipulators than number manipulators. Studies imply that symbolism at an appropriate level should aid problem solving significantly (Dennett, 1995). None the less students appear to find problems adapting to SC. One reason for this may be that a background in mathematics has prepared them for programming activity based on expressing arithmetic and boolean operations on a von Neumann machine, a view which is (typically) reinforced further by introductory courses in computer programming.

This paper considers the reasons why students find SC problematic and investigates the use of specific programming tools to help overcome these difficulties. The paper draws on the experience of using POP-11, Prolog and Lisp as practical languages on Intelligent Systems modules with both undergraduate and postgraduate computing students but the primary focus in this document will be Lisp.

Informal questionnaires and discussion groups involving the author and students (at undergraduate and postgraduate level) have often highlighted similar problems relating to practical Symbolic Computation. These broadly fall into three categories...

1 solving problems at a conceptual level - an extreme example of this could be: how is it possible to accurately derive meaning from natural language utterances;

2 specifying problems and/or solutions in any form

3 using the target programming language

The first two of these are typically the primary focus of theoretical study within IS modules, the last directly relates to the issues under discussion here and has been investigated further within the discussion groups mentioned. These groups focussed on four specific issues relating to programming language use...

1 adapting to new language features - Lisp examples included the Common Lisp Object System (CLOS) and mapping functions;

2 deconstructing (pulling apart) symbolic data-structures;

3 developing recursive solutions -as highlighted in other studies (Anderson et al 1988, Whittle & Cumming 2000);

4 using prior programming knowledge to aid specifying systems and functions in Lisp.

Different students experience these difficulties for different reasons but typically these students are new to IS and are also trying to adapt to a new programming paradigm. This paper outlines strategies which have been used to address the four problems identified above taking account of the typical student profile. It examines software tools that were developed as a result and presents three example problems, considering how the use of these tools alleviate some of the difficulties faced by students.

Strategies

It is important to accept that all problem solving and programming is taking place within a language learning framework where developing marketable programming expertise is a secondary goal to gaining practical experience in IS. Lisp was examined within this context to identify which new programming language features helped to achieve the primary goal and which, on balance, detracted from it. As a result the burden of learning CLOS, for example, was considered to outweigh the advantages offered by its use in single semester modules. In contrast, the use of mapping functions reduces the need for recursion and was therefore considered beneficial. In this way, while keeping to the overall paradigm of Lisp programming, language learners were constrained to a subset of language features. This strategy, to limit the need for recursion, is in contrast to other approaches which deal with teaching recursion in new ways (Ben-Ari, 1997; Glaser *et al.*2000; Bhuiyan *et al.* 1994; etc.).

Typical data deconstruction tasks were identified and tools developed to assist programmers engaged in these tasks. Specific examples discussed below are the *matcher* and *association list functions*. Languages like Lisp are suited to the addition of new tools, since the language can effectively be extended by producing new functions and macros, so adding features dos not detract from the programming paradigm or muddle Lisp's syntax.

IS software often uses collections of data which can be treated as sets in a mathematical sense (membership of the collection is significant but order is not). Most students expressed prior knowledge of basic set operations (set-union, set-difference, etc) and confidence using them. Some set operators exist in Lisp but use irregular names. In response, the tools outlined here include a more complete suite of operators.

Inevitably some IS solutions need to be specified recursively but many can be satisfied by alternative approaches. Two approaches used are (i) mapping functions and (ii) matcher iterators (described below).

Programming tools

This section briefly examines the programming tools use to implement the strategies identified above. These tools are provided as add-ons to Lisp with the exception of the mapping functions which are part of the Lisp language.

set operators - convenience utilities

A suite of set operators are supplied, their names all start with "$" (to be read "set") and are followed by mathematical or boolean symbols chosen to be easily remembered mnemonics. 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 on 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 input to the Lisp system, lines prefixed "→" indicate output from the Lisp system.

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

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

In addition to mapcar, Lisp provides mapping functions like *remove-if-not*, *find-if*, *reduce*, etc., and additionally allows them to use anonymous functions defined in-line (using lambda), for example...

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

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 also 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 have been provided to encourage students 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 . 980))
   (Sri-Lanka (Capital . Colombo) (Population . 15))
  )))

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

the matcher - deconstructing data, reducing recursion, aiding design

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 (Sussman *et al.* 1970; Sacerdoti *et al.* 1976) but despite the optimism of early systems like Eliza (Weizenbaum 1965) pattern matching alone is not considered a successful basis for AI systems. Perhaps because of this there is no standard pattern matcher in Lisp.

Pattern matchers exist in other languages including Prolog, ML and POP-11 which provides a sophisticated matcher as part of the language (Barrett *et al.* 1985) POP-11 was initially designed both as a language for novices and as a language for IS. The matcher was seen as a key tool for both of these user groups and POP-11 programmers use matching for a variety of tasks (examples include Gazdar & Mellish 1989).

The matcher described in this section is influenced by the use of the POP-11 matcher and by the mechanism of clause invocation which occurs in Prolog (Bratko 1990). It allows matcher methods to be specified using *defmatch* forms which superficially resemble Prolog clauses but are constructed and called in the same way as Lisp functions or CLOS methods (see Lynch & Barker 1999 for details).

The following section describes the specification of patterns and their use with matcher methods and other forms.

methods & patterns

The example below 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 1992; (Barrett *et al.*1985). The pattern (?x plus ?y) will match with any three element list containing the symbol "plus" as its second element.

Symbols like "#?x" in the body of method definitions retrieve the value of matcher variables (the #? syntax is reserved for user applications in Common Lisp (Steele1990)).

"Calculate" is used as below (in each case the method applied is that which matches the argument provided).

```
> (calculate '(5 plus 3)) →  8
> (calculate '(5 minus 3)) →  2
```

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
```

a matcher form of LET

In addition to the implicit use of the matcher when matcher methods are invoked it is used explicitly by *mlet* which provides a convenient mechanism for destructuring data. (note that matcher forms other than *defmatch* need their patterns to be quoted).

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

Other matcher tags act as wildcards. "=" and "==" are the tags for 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. One 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 problem solving and/or language learning burden for students new to symbolic computation. The examples are typical of the type that arise on introductory modules in 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, defined in Lisp...

```
(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)
 ))
```

The *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 students to understand a recursive solution or to deal with code which uses a procedural approach to pull apart and rebuild data-structures.

example 2

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

(Rule 32 (has fido hair) => (isa 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 hides important issues in the design of rule-based inference engines. The matcher allows more realistic rules to be developed...

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

The *forevery* form allows rule application to be achieved easily as shown in the following example which uses a rule with multiple antecedents.

```
(defvar 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-bind replaces ?x forms with their
;     appropriate values

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

This approach can be used to develop a general purpose mechanism for updating a set of facts by applying a rule to them. The function below makes further use of the matcher to permit a better structure for rules and also makes use of the set union operator to update facts. 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 application at a more abstract level.

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

This rule application mechanism can be used to underpin further work in IS modules which typically examine rule-based inference engines like forward and backward chaining processes.

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, 1963) 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 could be defined as shown.

```
(defvar blocks
  '((isa b1 block) (isa b2 block) (isa p1 pyramid)
    (on p1 b1)      (on b1 floor)  (on b2 floor)
    (cleartop p1) (cleartop b2) (cleartop floor)
    (holds nil)))

(defvar pick-up       ; pick up the object ?x
  '((pre (holds nil) (cleartop ?x) (on ?x ?y))
    (del (holds nil) (on ?x ?y))
    (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) (on ?x ?y))
```

Given the kind of operator description above, a generalised "apply operator" function can be built around the use of the matcher. Note the use of *all-present*, a predicate which checks the occurrence of a set of patterns within a set of data. *all-present* (like other matcher forms) may optionally be provided with initial bindings for match variables (in the code below #?x is initially bound to the value of *object*).

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

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

The style of simple operator definition used above allows the problems of defining and using operators to become quite abstract and conceptual. The difficulties associated with programming the actions of operators are dealt with by the matcher and set operators. Learners are able to concentrate on the mechanics of applying operators in symbolic world environments without spending time writing low-level functions. Additionally, the use of matcher patterns encourages a more declarative approach to operator definition. The example can be extended by creating more operators and either mechanisms to apply a series of operators or, alternatively, mechanisms to plan a sequence of operator applications.

observations and results

Observation and informal reporting from students demonstrates a strong bias for using pattern matching to deconstruct data in preference to using Lisp primitives. Discussion with students suggests that the reasons for this are (i) that the declarative nature of patterns makes them easier to derive from sample data than a sequence of primitive data-manipulation functions and (ii) the resulting code is easier to read (again because of its declarative nature). For similar reasons, but to a lesser extent, students make use of the association list functions and set operators but tend not to comment on these without specific prompting.

Students favour the use of *foreach* & *forevery* over the construction of recursive forms. As noted here (and by others referenced above) students have problems with recursion. In contrast those familiar with matching patterns do not experience a conceptual barrier to using match iterators.

The use of mapping functions (like *mapcar* and *remove-if-not*) gained a mixed response from students who reported undertanding mappings when presented and explained to them but none-the-less having difficulties with deriving mapping forms for themselves. Students typically had not met mapping functions before and had no analogy for their use.

concluding remarks

Intelligent Systems (IS) software can be complex for new students who do not have previous experience with symbolic computation and therefore have to learn a new programming paradigm at the same time as experimenting practically with complex software.

This paper has identified some specific difficulties students have adapting to symbolic software construction and of problem solving within a symbolic IS framework. Programming tools have been identified which aim to reduce the problem solving and programming burden for learners and three specific programming examples have been briefly investigated to demonstrate the use of these tools.

The examples demonstrate that IS courses can include a practical component without having to over-simplify IS software capabilities and without, alternatively, becoming swamped by the intricacies of IS software construction. The tools used aim to provide the capability for learners to examine/produce software at a level of abstraction which allows them to concentrate more on the conceptual issues of IS software rather than issues relating to the use of a programming language.

References

Anderson, J.R., Pirolli, P., Farrel, R. (1988) Learning to program recursive functions. In Chi, M.T., Glaser, R. Farr, M.J. (eds) *The Nature of Expertise,* P153-183, Hillside: Erlbaum.

Barrett, R., Ramsay, A., Sloman, A. (1985) *POP-11: A Practical Language for Artificial Intelligence*. Ellis Horwood,.

Ben-Ari, M. (1997). Recursion: From Drama to Program. *Journal of Computer Science Education*, 11(3), 9-12.

Bhuiyan, S., Greer, J., McCalla., G. I. (1994) *Supporting the learning of recursive problem solving*. Interactive Learning Environments, 4(2):115-139

Bratko, I. (1990) *Prolog Programming for Atrificial Intelligence (2nd ed)*. Addison Wesley.

Dennett, D. (1995). *Darwins Dangerous Idea*. Penguin. P488-489.

Gazdar, G., Mellish, C. (1989). *Natural Language Processing in POP-11, An Introduction to Computational Linguistics*. Addison Wesley.

Glaser, H., Hartel, P.H., Garratt, P.W. (2000) Programming by Numbers: A Programming Method for Complete Novices. *Computer Journal*, 43 p.252-265.

Lynch, S., Barker, D.J. (1999) Using a Pattern Matcher to Build Adaptable Interfaces for Lisp Modules. *Lisp User Group Meeting*, San Francisco.

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

Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming*. Morgan Kaufman, P.154-162,178-187

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

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

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

Sussman, G., Winograd, T., Charniak, E. (1970). Micro-planner reference manual. *A.I.Memo 203, Artificial Intelligence Laboratory*, MIT.

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

Whittle, J., Cumming, A. (2000) Evaluating Environments for Functional Programming, *International Journal of Human-Computer Studies*, 52, pps. 847-878, Academic Press