PROVIDING TOOLS FOR NOVICE AI PROGRAMMERS

S.C.Lynch
University of Teesside
Middlesbrough
UK TS1 3BA
s.c.lynch@tees.ac.uk
www.agent-domain.org

ABSTRACT

Solutions to Artificial Intelligence (AI) problems are often complex to implement by nature. This presents AI tutors with problems if they want to encourage students to explore practical implementation issues. If tutors wish to give students concise, easy to understand, practical examples of AI solutions they are often forced to simplify systems to a point where their functionality is no longer realistic and may 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 AI course.

As the nature of computing degrees becomes more diverse, and also the background of students sitting AI modules, we suggest a third alternative. This paper describes a suite of programming tools which have successfully helped novice students learn the practical aspects of AI programming.

Keywords

Programming, Lisp, Pattern Matching.

1. INTRODUCTION

The number of students taking traditional computer science courses is falling but there is an increasing number of students electing to study degree programmes which concentrate on computer games, mobile computing and other areas of digital media. These new students have more varied backgrounds in computing and have had a different exposure to computer programming than students who have studied more traditional Computer Science (CS). None the less, many of these students are interested in studying AI, games students hope to engineer more intelligent behaviour into autonomous games characters while other students are interested in user modelling, multimodal dialog and multiagent systems. Unfortunately, we found that our course in AI, which had originally been developed to suit the needs and profiles of students with a CS background was not ideally suited to the these new students. The challenge of teaching AI to students from avenues other than CS has also affected others [16].

In this paper we consider the problems faced by non-traditional students and the approach we have taken to overcome these problems while preserving the primary aims of our AI course which include giving students experience of the practical programming issues involved in building real AI systems. Like other practitioners we believe that it is necessary for students to investigate practical solutions in order to gain proper understanding of the issues involved [7,5] and that hands-on activity also stimulates greater interest [6].

In the past, in our institution, CS students progressed to an AI course only after having some exposure to functional programming and also to formal methods. This gave them an introduction to symbolic computation, discrete maths and the use of tuples as a means of encoding information about the world. In contrast, more recent groups of students have no exposure to these areas of study. Games students tend to possess some skills in C++, students from other routes have either Java or VB but few students have knowledge of more than one language or have programmed in more than one application domain.

New students often suggest that practical symbolic AI programming presents a steep learning curve and that the transition to Symbolic Computation from other paradigms is difficult. Our observations indicate that these barriers are greater for students coming from computing backgrounds with less traditional CS content. However there is nothing fundamental about Symbolic Computation that should indicate this, indeed symbolism is one of the major features that defines human intelligence and culture [12] and our studies indicate that symbolic declarative representations are not intrinsically difficult for novices to understand, if this were true then various formalisms from expert system rules to database query languages should present similar problems.

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 as input and produces new, derived numeric data (the mean) as output. Symbolic functions carry out a similar process with symbolic rather than numeric data. The utility of either function is determined by the programmer/user who recognises its purpose. If

anything we would expect the activity of simple symbolic functions to be more easily recognised from their input/output than a numeric function of similar complexity because they often extract or rearrange easily recognised words (symbols) rather than performing some mathematical task. Mathematical functions can require problem solving on the part of a human reader in order for them to understand what function is being applied. The function *f* for example, is such that *f*(73.6) = 220.8 and *f*(14.79) = 44.37. For most people, it requires some attention to recognise that *f* is simply performing *multiply-by-3*. It is comparatively easy to recognise the activity *g*, a symbolic function where

```
g((the cat)(the dog)(the frog))
   = (cat dog frog)
```

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 [3]. 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. Alternatively it may be because the real utility of functions produced as part of AI systems can only be understood in terms of a wider problem solving context. The purpose of conflict resolution, for example, can only be appreciated when the wider problem domain (expert systems, planning, etc) is known.

In various areas of education, selecting one programming language in preference to others can be controversial, AI is no exception to this. We recognise that AI practitioners use various alternative languages for teaching (including non-symbolic languages like C++ and Java) and we do not engage in *the language debate* in this paper. Our focus is to consider the reasons why students find *symbolic* AI programming problematic and investigate 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 AI modules with both undergraduate and postgraduate computing students but the primary focus in this study is Lisp.

Questionnaires and discussion groups with students often highlight similar problems which broadly fall into three categories...

1. solving AI 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 AI modules, the last directly relates to the issues under discussion here and has been investigated further within discussion groups and by observing students as they attempt different types of programming exercise. This study highlighted four specific issues relating to programming language use...

1. adapting to new and different language features - Lisp examples included the Common Lisp Object System (CLOS) and mapping functions;
2. deconstructing symbolic data-structures to retrieve specific units of data - examples include: pulling apart nested association lists;
3. developing solutions using unfamiliar control structures – notably recursion;
4. using prior programming knowledge to aid specifying systems and functions in Lisp.

Different students experience these difficulties for different reasons but typically they are new to AI and they are also learning a new style of programming. This paper outlines strategies which have been used to address the four problems identified above taking account of the profile of typical students. It examines software tools that were developed as a result and presents three example problems, illustrating how the use of these tools alleviate some of the difficulties faced by students. In addition to the tools described in this paper we have also developed a multiagent platform which allows students to link Lisp code to software written in other languages (including games and graphics engines), this is not discussed here.

2. STRATEGIES

1. In our AI modules we 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 AI. Within this context the new language features were examined to identify which helped to achieve the primary goal and which, on balance, detracted from it. In the approach described here, the burden of learning to use CLOS, for example, was considered to outweigh the advantages its use offered in a single module. In contrast the use of mapping functions so reduces the need to develop recursive solutions that it was considered beneficial to introduce them to students who had no previous experience of them. In this way, while keeping to the overall

paradigm of Lisp programming, language learners could on a subset of language features.

2. Core 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 particularly suited to the addition of new programming tools since the language can effectively be extended by producing new functions and macros.
3. Inevitably some solutions need to be specified recursively but many can be satisfied by alternative approaches that still fit within the Lisp paradigm. Two approaches were used to reduce the need for recursion (i) mapping functions and (ii) matcher iterators (described below).
4. Prior knowledge differs from student to student but typically students have some background in 3GLs and object oriented languages. Additionally many are confident in specifying pre- and post- conditions and using set operations at least at a conceptual level. Some programming features that they are familiar with they none-the-less find confusing or difficult to apply in Lisp. This has lead to the further development of the matcher and a small suite of set operations (set-union, set-difference, etc) some of which exist in Lisp but use irregular names.
5. We limit the structuring of data to two basic types: association lists and tuples (tuples are typically collected into sets of tuples). In practice this imposes almost no restriction on the types of AI systems we investigate since most can have their knowledge base, facts, etc, naturally realised in these forms. Both of data structures are found to be readily accessible by various groups of students and some students state that they are familiar with these data structures from prior work with databases.

3. 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.

3.1 Set Operators

Tuples are used in various problem domains to represent information about worlds. Tuples are typically collected together as sets and these sets are manipulated in various ways often using set operations. A suite of Common Lisp functions provide these set operations but students suggest that the lack of naming convention for these functions makes them hard to remember (*set-difference*, *subsetp*, *union*, etc) and, in addition, the existing set functions use comparisons based, by default, on equality of pointers rather than equality of data. This makes them more efficient but also introduces problems for novice programmers who can experience apparently strange behaviour from set functions like that shown below (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 to the Lisp system are sometimes placed on the same line).

```
> (union '((a cat)(a dog)(a frog))
         '((a frog)(a bat)(a dog)))
→ ((a frog) (a dog) (a cat) (a frog)
   (a bat) (a dog))
```

Experienced Lisp programmers would engineer their preferred behaviour from these functions by providing additional *key* arguments but this increases the learning burden for new programmers. One other problem with the predefined functions is that they only support operations on two sets at a time so cannot be used (without nesting) to take the union of three sets for example.

Our solution is to provide a new suite of set functions/operators whose names follow a simple convention – the names all start with "$", to be read "set", and are followed by mathematical or boolean symbols chosen to be easily remembered mnemonics. These functions carry out equality testing on data (so avoid the problems described above) and take any number of sets as arguments, Examples include:

name	use	example
$-	set difference	($- '(a b c d e) '(b a d)) → (e c)
$+	union	($+ '(a b c d) '(c d e f)) → (b a c d e f)
$*	intersection	($* '(a b c d) '(c d e f)) → (d c)
$<=	is-subset-of	($<= '(b a d) '(a b c d)) → t

3.2 Mapping Functions

In most languages programmers typically develop iterative solutions when working with collections of data, with AI languages recursion is often considered more appropriate. Unfortunately students often report difficulties with recursion and

the Lisp primitives for iteration are also poorly received. Lisp offers an alternative to both iteration and recursion by providing mapping functions which apply some function to each item in a sequence of data items. 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.

```
(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 (third x)
                    (second x)))
       fruit)
→ ((red cherry) (green apple)
    (yellow banana) (green kiwi))
```

Specifying mapping in this way involves some additional syntax and it was not initially clear whether this approach would be preferred by novices. During one phase of our study we asked students to complete small programming tasks using (i) iteration, (ii) recursion and (iii) mapping techniques, to self-mark their work and comment on their results giving opinions about the various methods involved. They expressed an (unexpected) preference for mapping.

mapcar is not the only mapping function, Lisp also provides functions like **remove-if**, **remove-if-not**, **find-if** and **reduce**. To illustrate how these mapping function use can ease logical complexity for programmers the following example shows how a set intersection function could be written from first principles. The first example is of a classically Lisp-like recursive solution which also requires a conditional form, the second example uses remove-if-not to discard elements from the first set which are not members of the second.

```
;; 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))
```

We attempted to reduce the burden of new and unusual syntax (the use of #' and *lambda*) by writing macros to handle filtering functions like the one below (which is equivalent to the *remove-if-not* invocation in the previous example).

```
(filter s1 removing x
              when (member x s2))
```

This approach initially gained favourable student feedback but to accommodate a variety of mapping forms it became necessary to expand the number of keywords that could be used ("*removing*" and "*when*" are keywords in the example above). Increasing the number of keywords required students to memorise different forms of filter expression and this made it unpopular. As a result we have reverted to using mapping.

3.3 Association List Utilities

As stated above we limit the structuring of data to association lists and/or tuples. To encourage students to use symbolic data-structures and wean them off a bias towards numeric indexing we provide a small set of utilities to manipulate association lists. Our primary aim is to help students concentrate on simple knowledge representation schemes at a conceptual level and gain some experience of designing software which uses them without having to develop primitives for their use. The two functions shown here are **->** and **add->** which retrieve and update data in association lists.

```
 (defvar data
    '((africa
       (botswana (capital . gaborone)
                 (polulation . 1.5))
       (zimbabwe (capital . harare)
                 (polulation . 11.2))
      (asia
       (nepal    (capital . kathmandu)
                 (population . 7))
       (sri-lanka (capital . colombo)
                  (polulation . 15)))
      )))
```

```
> (-> data 'africa 'zimbabwe 'capital)
  harare
```

```
> (add-> data
    '(asia sri-lanka climate)
              'tropical)
  ((asia
     (sri-lanka (climate . Tropical)
                (capital . Colombo)
                (polulation . 15))
     (nepal (capital . Kathmandu)
```

```
        (population . 7))
    (africa ....))
```

3.4 Pattern Matching

Students experience various difficulties in deconstructing and accessing sub-parts of symbolic data structures. Consider for example (i) the problem of extracting the antecedents of a rule like...

```
(rule 17 (at ?x ?door)
         (unlocked ?door)
      => (can ?x (open ?door))
```

and (ii) the problem of cross referencing data from the set of tuples below to infer that *agent-5 must be contacted in order to unlock the blue-door*

```
(has agent5 key3)
(unlocks key3 blue-door)
(status blue-door locked)
```

Our study examined students' approach to these types of problem, considering the difficulties they reported and the errors they made in coding solutions. We explored various ways that additional programming facilities could be provided to reduce these problems including providing a suite of functions to specifically handle the data we would use in examples and ways to reshape data so it could be processed by existing Lisp primitives. None of these solutions were both flexible enough and powerful enough to fulfill all our needs. As an alternative we investigated a solution based on pattern matching.

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 [14]) and at least one version of Lisp, Qlisp [11] had matching capabilities. After overcoming the optimism of early language processing systems like Eliza [15] 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 [10].

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 [1]. POP-11 was initially designed both as a language for novices and as a language for implementing AI. 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 [4].

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. 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 but 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.

At a primitive level the pattern matcher is specified as a function which accepted a pattern and some data as its arguments and returned relevant variable-value pairs as its result, eg:

```
> (matches '(the ?n ?v)
           '(the cat sat))
→ ((?n cat)(?v sat))
```

This has been developed to provide a flexible tool which helps with construction and deconstruction tasks and also provides some new iterative forms.

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.

3.4.1 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 [9,1]. 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 [13]).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) matcher variables may be prefixed by "??" in which case they will bind to zero or more list elements. 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. The "??" tag is useful in allowing programmers to define methods in a Prolog-like style which provides an alternative approach to structuring recursive forms. An example of this is a pair of *list length* methods.

```
(defmatch len (nil) 0)

(defmatch len ((= ??rest))
  (1+ (len #?rest)))

> (len '(a b c)) → 3

> (trace len1)
(len1)
> (len '(a b c))
 0: (len (a b c))
   1: (len (b c))
     2: (len (c))
       3: (len nil)
       3: returned 0
     2: returned 1
   1: returned 2
 0: returned 3
→ 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.

3.4.2 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, its layout is as follows...

```
mlet ( pattern matching-list )
    {statement}*
```

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. **match>>** expands a list which contains matcher variables.

```
>(mlet(
    '(the ?obj was eaten by the ?subj)
    '(the rat was eaten by the cat))
  (match>> '(the ?subj ate the ?obj)))
→ (the cat ate the rat)
```

MLET returns nil and does not process its body of statements if its matching fails.

```
>(mlet(
    '(the ?obj was eaten by the ?subj)
    '(the rat was chased by the cat))
  (match>> '(the ?subj ate the ?obj)))
→ nil
```

3.4.3 Foreach & Forevery

Foreach and *forevery* are two iterative constructs which repeatedly pass patterns over data, responding when a match occurs, they are based on the keywords of the same name in POP-11. Their structure is:

foreach (*pattern matching-lists*)
  {*statement*}*

forevery (({*pattern* }*) *matching-lists*)
  {*statement*}*

One obvious use for these forms is with a set of related facts as in the following examples. Both forms have an implicit *progn* in their body of statements. 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.

```
;; a simple set of facts
(defvar db1
 '((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 a single pattern...

```
> (foreach ('(size ?x small) db1)
    (format t "~&~a is small" #?x)
    #?x)

→ b2 is small
→ b3 is small
→ b4 is small
→ (b2 b3 b4)
```

Forevery iterates with multiple patterns...

```
> (forevery ('((isa ?b box)
               (color ?b red)
               (supports ?b ?x)) db1)
```

```
    (format t "~&~a is a red block
         which supports ~a" #?b #?x)
    (list 'red-block #?b))
```

```
→ b1 is a red block which supports b2
→ b2 is a red block which supports b3
→ ((red-block b1) (red-block b2))
```

4. PROGRAMMED EXAMPLES

This section examines three specific AI programming problems and investigates using the software tools introduced above to reduce the difficulty of problem solving and/or language learning burden for students new to symbolic computation. The examples are typical of the type that we have used to introduce practical aspects of symbolic AI.

4.1 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    b1  table) (on    b2  table)
   (on    b5  table)))
```

The matcher's *foreach* form can be used to retrieve the names of objects satisfying a specified relation from this type of structure. The form below matches all triple statements of the *isa-wedge* type and returns the relevant object names.

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

This approach can be used to build a general purpose function lookup which is defined and used as follows...

```
(defun lookup (pair tuples)
 (foreach(
    `(,(first pair) ?x ,(second pair))
           tuples)
    #?x))
```

```
> (lookup '(isa cube) *blocks*)
→ (b1 b3 b5)
> (lookup '(on table) *blocks*)
→ (b1 b2 b5)
```

Lookup returns a set of object names so results of different lookup operations can be combined with set operators like $* (set intersection) and $+ (set union). This allows multiple queries to be satisfied.

```
> ($* (lookup '(isa cube) *blocks*)
      (lookup '(on table) *blocks*))
→ (b5 b1)
```

A generalised query function can be built which maps pairs like (isa cube) and (color blue) 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)(on table))
                 *blocks*)
→ (b5 b1)
```

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)
```

4.2 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 matcher's forevery form...

(setf family

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

; applying...
;    ((parent ?a ?b)(parent ?b ?c))
;               => (grandparent ?a ?c)

> (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 returns an updated set of facts. This 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 ?a ?b) (parent ?b ?c)
              => (grandparent ?a ?c))
    family)

→ ((grandparent rob sam)
    (grandparent ellen tom)
    (parent sarah tom)
    (parent ellen sarah)
    (parent 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 Lisps 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
    (loop
      (setf old-facts facts)
      (setf facts
        (reduce #'$+
          (mapcar
             #'(lambda (r)
                 (apply-rule r facts))
             rules)))
      (if ($= old-facts facts)
          (return facts))
  )))

; another set of facts & rules

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

(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))
  ))

> (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))
```

4.3 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 [8] 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) (supports b1 p1)
   (supports floor b1)
   (supports floor b2)
   (cleartop p1) (cleartop b2)
   (cleartop floor) (holds nil)))

(defvar 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
; so its parts are accessed 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.

```
(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))
```

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 abstract and conceptual. The difficulties associated with programming the *actions* of operators have largely been removed and are dealt with by the matcher. Learners 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 obj on another
   (pre (holds ?obj) (cleartop ?x))
   (del (holds ?obj) (cleartop ?x))
   (add (holds nil)(supports ?x ?obj))
  )))

;arm-controller 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
```

5. CONCLUSION

AI software can be complex. Additionally many AI students do not have previous experience with symbolic computation and are therefore having to learn a new programming paradigm at the same time as experimenting practically with complex software. This is especially true for students who are studying subjects like computer games and mobile computing compared to those studying traditional computer science.

This paper has identified some specific difficulties students have in programming solutions for AI problems. These were obtained through discussion

groups with students and from observing different aspects of their practical work. To address these difficulties we have provided students with various programming tools and have examined how these effect their performance. The use of these tools has been successful, students from mixed academic backgrounds sit the same course with the same problem-based learning requirements and we can no longer differentiate between different types of student from their results.

In the paper we outlined three specific examples of AI programming problems and investigated how the tools we provide can produce simple solutions to these problems. These examples demonstrate that AI courses can include a practical component without having to over-simplify AI software capabilities and without, alternatively, becoming swamped by the intricacies of AI software construction. The tools used 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 AI software rather than issues relating to the use of a programming language.

We have monitored student performance over 3-4 years. Students take summative assessment towards the end of their course and undertake formative problem-based learning exercises throughout their course. These exercises each concentrate on a single topic area (search, planning, etc) and are designed to take 4-5 weeks to complete. They are divided into 10-15 steps which progress from introductory practical exercises to more advanced material. Earlier, groups of students used standard programming constructs and learning exercises designed around using these. More recently students have used the tools described above and learning exercises have investigated more advanced concepts. Students have been monitored through learning exercises and, over three years, we have seen an increase in the proportion of students who complete the final stages of these exercises with reasonable success (from approximately one third of students in the first group to over two thirds in the last group) and an improvement in final assessment performance (a rise from approximately 50% to 60%).

6. REFERENCES

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

[2] R.M. Crespo Garcia, J.V. Roman and A. Pardo 'Peer Review to Improve AI Teaching' *Frontiers in Education*, 36, 3-8, (2006)

[3] D.Dennett.. *Darwins Dangerous Idea*. Penguin. P488-489. (1995).

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

[5] J.M.D. Hill and K.L. Alford *A Distributed Task Environment for Teaching AI with Agents* ACM SIGCSE, 36, 224-228. (2004).

[6] J.Kerins 'Teaching AI and Intelligent Agents: Challenges and Perspectives' *Italics* 4,3. (2005).

[7] T.L. McCluskey and R.M. Simpson 'The use of an integrated tool to support teaching and learning in artificial intelligence' *Italics* 4,3. (2005).

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

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

[10] S. Russell and P. Norvig AI, A Modern Approach, Prentice Hall, (2003).

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

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

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

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

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

[16] G.S.Young and S.E.Haupt, 'Teaching AI to Meteorology Undergraduates' AI Applications to Environmental Science, 4 2005