Using Patterns to Teach Recursion in LISP

Amruth N. Kumar
Assistant Professor, T.A.S.
Ramapo College of New Jersey
505 Ramapo Valley Road
Mahwah, NJ 07430-1680
Email: amruth@ramapo.edu

Abstract

It often becomes necessary to teach LISP as part of the Artificial Intelligence (AI) course. In this paper, we discuss oft-repeated patterns in recursive programs, which can be used by the AI instructor to teach LISP quickly. We relate these patterns to the characteristics of the problems to which they are applicable, in order to help students recall and reuse the right pattern in a programming situation.

1 Introduction

Most students encounter functional programming for the first time in the Artificial Intelligence (AI) course, and few will have programmed extensively using recursion by then. Therefore, while teaching LISP, just describing its syntax is not enough to get them to comfortably program in it. In this paper, we discuss oft-repeated patterns [1] in LISP code for recursive functions, and characteristics of the problems to which they apply. It has been our experience that, discussing these patterns to teach LISP in class not only quickly brings the students up to speed, but also seems to provide them with a firmer foundation in recursive programming in LISP.

We will have already covered at least the following topics before discussing patterns: read-eval-print, quote, the 5 primitives car, cdr, cons, list, and append, the truth values t and (), the basic predicates null and equal, the boolean operators and, or, and not, the selection construct cond, the abstraction construct defun, and side-effects through print.

The template for recursive function definition is first provided as:

```
(defun fname ( input )
  (cond ( end-condition
```
In the rest of the paper, we will refer to components of this template, such as end-condition or end-action. We describe patterns for problems with the following input-output characteristics:

- Scalar input, with
  
  - Output by side-effect - Head Recursion/Tail Recursion
  - Scalar Output
  - Vector Output - Head Recursion/Tail Recursion

- Vector input, with
  
  - Output by side-effect - Head Recursion/Tail Recursion
  - Scalar Output
  - Vector Output - Head Recursion/Tail Recursion

Scalar Input, Output by Side-effect

Problem: Write a function which accepts a numerical argument, say 5, and prints all numbers from it down to 1.

Pattern 1:

```lisp
(defun printall (number)
  (cond ((equal number 1)
     (print 1))
     (t
      (print number)
      (printall (- number 1)))))
```

Analysis

- This is a good starting example to illustrate the form of end-condition, end-action, optional-action and recursive-action. They are all s-expressions, returning values. Since input is scalar, it is referred to by its name (number here) as highlighted. The recursive call nudges the value of number in the direction of its base case value, viz., 1.

- Note that the above function will get into infinite recursion if a negative actual parameter is provided during function call! This could be used as a springboard to discuss robust end conditions. The necessary changes are incorporated below.
• This definition also illustrates the concept of tail recursion. Recursive calls can be described using the metaphor of going down a flight of stairs and coming back up. During tail recursion, all the work ((print number)) is done going down the stairs, and none bounding back up.

In order to illustrate “head” recursion as an alternative to tail recursion, we take up the next problem:

Problem: Write a function which accepts a numerical argument, say 5, and prints all the numbers from 1 up to the number.

Pattern 2:

(defun printall (number)
  (cond ((< number 0)
        0)
        (t
         (printall (- number 1))
         (print number))))

Analysis

• The only difference between this and the previous code (other than making it robust) is that recursive-action comes before optional-action. This is an example of what we henceforth refer to as head recursion, where, using the earlier metaphor, no work is being done on the way down the stairs, and all the work is being done on the way back up. Just exchanging the statements is sufficient to change the order in which numbers are printed!

• The end-condition has now been made robust. This can be used as an illustration of better programming, as it accounts for a larger range of inputs. The idea of a condition doubling up as action can also be illustrated with this example.

Scalar Input, Scalar Output

Problem: Write a function which accepts a numerical argument, say 5, and returns the sum of all numbers from 1 up to it.

Pattern 3:

(defun sumall (number)
  (cond ((< number 0)
        0)
        (t
         (+ number (sumall (- number 1))))))

Analysis
Unlike the previous function which resulted in side-effects, this function is required to return a scalar value. Hence, *end-action* returns a value instead of using a print statement. For the same reason, in *recursive-action*, the *input* is *composed* (such as added, multiplied) with the value returned by the recursive call. Note that factorial can be calculated similarly. The effect of *head* and *tail* recursion can be illustrated in this case too, when non-commutative operations such as subtraction and division are used.

**Scalar Input, Vector Output**

**Problem:** Write a function which accepts a numerical argument, say 5, and returns a list of all numbers down to 1.

**Pattern 4:**

```lisp
(defun listall (number)
  (cond ((<= number 1)
         (list 1))
        (t
         (cons number (listall (- number 1))))))
```

**Analysis**

- Since vector output is desired, *end-action* returns a vector (a list) rather than a scalar. In the recursive case, the *input* is *consed* into the list returned by the recursive call. In particular, the use of *cons* is noteworthy here: it is a classical technique to build a vector from its elements.

- In this definition, *input* is inserted at the head of the vector returned by *recursive action*, resulting in a descending order of elements in the final list. By exchanging the textual positions of *input* and recursive call, the order of elements in the list can be reversed. This is illustrated by the next problem.

**Problem:** Write a function which accepts a numerical argument, say 5, and returns a list of all numbers from 1 up to it.

**Pattern 5:**

```lisp
(defun listall (number)
  (cond ((<= number 0)
         ()
        (t
         (append (listall (- number 1)) (list number)))))
```

**Analysis**
In the recursive case, append is used, along with list function applied to input. This is another classical technique to build a vector from its elements. It differs from the previous one in the order in which it puts the elements together into the final list: in this case, the list is in ascending order.

Considering the result of using cons instead of append in the above function is another way to discuss the difference between the two techniques: this would result in a list nested to several levels, actually, as many levels as there are recursive calls!

Vector Input, Output by Side-effect

Problem: Given a list of numbers, print all the numbers in order.

Pattern 6:

(defun printall (lst)
  (cond ((null lst))
        (t
         (print (car lst))
         (printall (cdr lst))))
)

Analysis

- Since input is a vector, end-condition involves checking to see if the vector is empty, using the null predicate. The car of input is referred to in optional-action, and the cdr of input is referred to in recursive-action. These changes are universal to vector inputs. Further, since output is by side-effect, end-action is to do nothing.

- Once again, the above pattern is an example of tail recursion. By using head recursion, the order in which the elements of the list are printed, can be changed. This involves just textually exchanging optional-action and recursive-action.

Vector Input, Scalar Output

Problem: Given a list of numbers, return the sum of the numbers.

Pattern 7:

(defun sumall (lst)
  (cond ((null lst) 0)
        (t
         (+ (car lst) (sumall (cdr lst))))))

Analysis
Since a scalar output is expected, \textit{end-action} returns a value, and in the recursive case, \textit{car of input} is composed (added/multiplied etc.) with the value returned by the recursive call. This is similar to the scalar output case for scalar inputs discussed before.

\textbf{Vector Input, Vector Output}

\textbf{Problem} Given a list of numbers, return a list of +1 numbers.

\textbf{Pattern 8:}

\begin{verbatim}
(defun plus1 (lst)
  (cond ((null lst) ()
        (t
         (cons (+ (car lst) 1) (plus1 (cdr lst))))))
\end{verbatim}

\textbf{Analysis}

- Note that this is very similar to the earlier case. As in the scalar input case, \textit{append} may be used instead of \textit{cons} to reverse the order of the output vector.

\section{Discussion}

We have presented several patterns for writing LISP programs. This set of patterns is not meant to be comprehensive. However, they have been found to be sufficient for most recursive programming in the AI course. We have arranged these patterns according to the characteristics of problems to which they are applicable, an arrangement which helps students recall and reuse the right pattern in a programming situation.

Teaching LISP from scratch, up to and including the above patterns takes us 4-5 hours of classroom instruction. Our experience has been that, in Fall '93 offering of AI, when we did not use these patterns, students felt ill-prepared to attempt anything more than trivial problems. When we did use them for LISP instruction in AI in Fall '94, a much larger percentage of students managed to turn in complete and correct programs on significant problems. Since then, we have been using these patterns to teach Lisp in the Programming Languages course also.

\section*{References}