Recursion, Iteration and Functional Languages

John E. Howland
Department of Computer Science
Trinity University
715 Stadium Drive
San Antonio, Texas 78212-7200
Voice: (210) 736-7480
Fax: (210) 736-7477
Internet: jhowland@ariel.cs.trinity.edu

Abstract

Functional programming languages are shown to be useful in the teaching of the concepts of recursion and iteration. The functional language approach presented in this paper has advantages over imperative languages in the area of analysis of recursive and iterative algorithms. Examples using the J and Scheme programming languages, with emphasis on the use of functional programming notation in exposition are given. [1]

Subject Areas: Computer Science Education, J, Scheme, Exposition.
Keywords: computer science introductory course, J, Scheme, exposition.

1 Introduction

Functional languages provide a computational environment where functions are applied to arguments producing results. Once an item is created in memory it is never altered. Function application occurs without side effects. Algorithms involve sequences of function applications (functional composition). Most functional language environments automatically reclaim (garbage collection) items which are no longer needed.

[1]This paper appears in the Journal of Computing in Small Colleges, Volume 13, Number 4, Pages 86-97, March 1998. The paper was presented at the CCSC South-Central Conference, April 18, 1998, Millsaps College, Jackson, Mississippi.

Imperative languages use a state model of computation wherein procedures modify the state of items stored in memory as a computation proceeds from beginning to end.

Functional languages provide somewhat different view of program design which can be useful in the teaching of introductory computer science topics. This paper discusses the use of functional programming techniques in the teaching of recursion and iteration.

The topic of recursion is usually not taught in close combination with iteration in the CS I-CS II course sequence. Sometimes these topics are presented as being unrelated to one another with recursion being treated as a more advanced topic and sometimes less efficient approach to problem solving which should be avoided unless absolutely necessary.

An important problem solving strategy, sometimes called divide and conquer, involves sub-dividing a problem into parts. At least one part, often called a base case, is easily handled and one or more other parts are identical, except smaller, to the original problem. Each of the parts may be handled by the solution or subdivided into parts which may be handled by the solution.

In the following sections, programming examples are given in the Scheme [Har 94, Man 95, Spr 89] and J [Ive 95] programming languages. J is a pure functional language, however, Scheme is not. A subset of Scheme, which omits any Scheme function which mutates an existent Scheme item, is used for the examples in this paper.

The choice of programming language used to teach computer science topics has been widely discussed in the computer science education literature. In particular, [Kon 74, Kon 94, Rie 93, How 94, How 95, How 96, How 97] advocate the use of functional languages, such as J and Scheme, in the teaching of many introductory computer science topics. This paper considers the use of Scheme or J when teaching recursion and iteration.

2 Recursion

A definition which refers to itself is a *recursive* definition. We consider several examples of recursive definitions which illustrate our approach to teaching recursion. The first of these examples is a trivial problem which is used so that the problem being solved does not interfere with understanding the approach being used to teach recursion.

2.1 Summing the First n Positive Integers

The first example uses a recursive definition to sum the integers 1 to k. The divide and conquer solution has two cases.

1. *Base case*.

   When there are no integers to be added, the result should be zero.

2. *Smaller, but identical, problem.*

   When there are `k (k>0)` integers, the solution can be written as `k+sum(k-1)`.

Following are the Scheme and J programs for summing the integers 1 to k.

```
(define sum
  (lambda(n)
    (if (= n 0)
      0
      (+ n (sum (- n 1))))))
```

Program `sum` (Scheme Version)

```
sum =: monad define script
if. y. = 0
  do. 0
  else. y. + sum y. - 1
end.
)
```

Program `sum` (J Version)

2.2 Tracing the Execution of a Recursive Definition

Most functional programming environments support traced execution of definitions. Next, we show the traced output of the Scheme version of `sum`.

```
> (trace sum)
#<unspecified>
> (sum 5)
"CALLED" sum 5
 "CALLED" sum 4
  "CALLED" sum 3
   "CALLED" sum 2
    "CALLED" sum 1
     "CALLED" sum 0
     "RETURNED" sum 0
    "RETURNED" sum 1
   "RETURNED" sum 3
  "RETURNED" sum 6
 "RETURNED" sum 10
"RETURNED" sum 15
15
```

If a programming environment does not support traced evaluation, it is straight forward to implement traced versions of recursive definitions. This is illustrated for the J version of `sum`.

```
traced_sum =: monad define script
entering y.
if. y. = 0
  do. leaving 0
  else. leaving y. + traced_sum y. - 1
end.
)
```

Program `traced_sum` (J Version)

The functions `entering` and `leaving` are defined as shown below.

```
entering =: 'Entering, input = '&trace
leaving =: 'Leaving, result = '&trace

trace =:  monad define script
display y.
:
display (format x.),format y.
y.
)

display =: 1 !: 2 & 2
```

Execution of `traced_sum` gives the following output.

```
   traced_sum 5
Entering, input = 5
Entering, input = 4
Entering, input = 3
Entering, input = 2
Entering, input = 1
Entering, input = 0
Leaving, result = 0
Leaving, result = 1
Leaving, result = 3
Leaving, result = 6
Leaving, result = 10
Leaving, result = 15
15
```

2.3 The Factorial Function

To compute the product of the integers 1 to k using the divide and conquer approach, we have, as in Section 2.1, two cases.

1. *Base case*.

   When there are no integers to be multiplied, the result should be one.

2. *Smaller, but identical, problem.*

   When there are `k (k>0)` integers, the solution may be written as `k*factorial(k-1)`.

Following are the Scheme and J programs for computing the product of the integers 1 to k.

```
(define factorial
  (lambda(n)
    (if (= n 0)
      1
      (* n (factorial (- n 1))))))
```

Program `factorial` (Scheme Version)

```
factorial =: monad define script
if. y. = 0
  do. 1
  else. y. * factorial y. - 1
end.
)
```

Program `factorial` (J Version)

As in Section 2.1, we can trace the execution of `factorial`.

```
> (trace factorial)
#<unspecified>
> (factorial 6)
"CALLED" factorial 6
 "CALLED" factorial 5
  "CALLED" factorial 4
   "CALLED" factorial 3
    "CALLED" factorial 2
     "CALLED" factorial 1
      "CALLED" factorial 0
      "RETURNED" factorial 1
     "RETURNED" factorial 1
    "RETURNED" factorial 2
   "RETURNED" factorial 6
  "RETURNED" factorial 24
 "RETURNED" factorial 120
"RETURNED" factorial 720
720
```

3 Continuations

In Sections 2.2 and 2.3 we notice that the functions `sum` and `factorial` are called repeatedly until the problem has been reduced to a problem of size zero. No results are returned by any of these calls until a call is made for a problem of size zero. For each of the calls made for problems of size greater than zero, a record of the computation remaining to be done must be saved. We formalize the concept of representing the remaining computation as a monad (function of one argument) with the following definition.[2]

Given a compound expression e and a subexpression f of e, the *continuation* of f in e is the computation in e, written as a monad, which remains to be done after first evaluating f. When the continuation of f in e is applied to the result of evaluating f, the result is the same as evaluating the expression e. Let c be the continuation of f in e. The expression e may then be written as $c\ f$.

Continuations provide a "factorization" of expressions into two parts; f which is evaluated first and c which is later applied to the result of f. Continuations are helpful in the analysis of algorithms.

3.1 Scheme Example

Suppose e is the expression `(* 2 (+ 3 4))` and f is the subexpression `(+ 3 4)`, then the continuation of f in e is

```
(lambda(n) (* 2 n))
```

and

```
((lambda(n) (* 2 n)) (+ 3 4))
```

produces the same result of 14 as does

```
(* 2 (+ 3 4))
```

3.2 J Example

Suppose e is the expression `5 * 4 + 5` and f is the subexpression `4 + 5`, then the continuation of f in e is

```
monad define '5 * y.'
```

and

```
(monad define '5 * y.')  4 + 5
```

produces the same result of 45 as does

```
5 * 4 + 5
```

[2] This definition of continuation should not be confused with the definition of the Scheme `call-with-current-continuation`[Spr 89].

4 Expressing Recursion as Functional Composition

Consider the function `sum` of Section 2.1 and evaluate the sum of the integers 1 to 5. We next write out the 5 continuations which must be formed to complete this evaluation.

4.1 Using Scheme Notation

The five continuations are:

```
(define c1
  (lambda(n) (+ 5 n)))
(define c2
  (lambda(n) (+ 4 n)))
(define c3
  (lambda(n) (+ 3 n)))
(define c4
  (lambda(n) (+ 2 n)))
(define c5
  (lambda(n) (+ 1 n)))
```

Then the value of `(sum 5)` may be written as

```
(c1 (c2 (c3 (c4 (c5 0)))))
```

4.2 Using J Notation

Consider the `factorial` function defined in Section 2.3 and evaluate `factorial 5`. Five continuations must be formed during this evaluation. They are:

```
c1 =: monad define'5 * y.'
c2 =: monad define'4 * y.'
c3 =: monad define'3 * y.'
c4 =: monad define'2 * y.'
c5 =: monad define'1 * y.'
```

Then the value of `factorial 5` may be written as

```
c1 c2 c3 c4 c5 1
```

5 Iteration

We next consider an alternate solution to the problem of summing the integers from 1 to k. This solution uses a recursive definition but does not use the divide and conquer strategy.

5.1 Scheme Example

The Scheme definition

```
(define sum-iter
  (lambda(n acc i)
    (if (> i n)
      acc
      (sum-iter n (+ acc i) (+ i 1)))))
```

solves the problem of summing the integers 1 to 5 when applied to the arguments 5 0 1. We can create a new definition `sum1` which solves the problem of summing the integers 1 to k, given the size of the problem k, with the definition

```
(define sum1
  (lambda(k)
    (sum-iter k 0 1)))
```

Analysis of `sum1` involves analyzing `sum-iter` since `sum1` makes a single call to `sum-iter`. The definition of `sum-iter` is recursive. Next, using the definition of a continuation in Section 3, we write the continuation of each call to `sum-iter` inside the definition of `sum-iter`. This definition consists of a single `if` expression. The only time recursive calls are made to `sum-iter` is when `i` is less than or equal to `n`. The continuation of the call to `sum-iter` may be written as

```
(lambda(n) n)
```

This is the identity function. Since each continuation simply returns its argument, there is no need to form the continuations in the first place and it is possible for an optimizing compiler or interpreter to derive an equivalent program which replaces the recursive calls to `sum-iter` with an iteration which directly forms the sum of $0 + 1 + \ldots + k$ with a single call to `sum-iter`.

5.2 J Example

Next we consider an alternate solution to the problem from Section 2.3 of computing the product of the integers 1 to k which does not use the divide and conquer strategy.

The definition

```
factorial_iter =: monad define script
('n' ; 'acc' ; 'i') =. y.
if. i > n
  do. acc
  else. factorial_iter n , (i * acc) , i + 1
end.
)
```

computes the product of the integers 1 to 5 when applied to the argument 5 1 1.

A cover function for `factorial_iter` may be defined as

```
factorial1 =: monad define 'factorial_iter y. , 1 1'
```

so that `factorial1 5` produces a result of 120.

The analysis of `factorial_iter` is similar to the analysis of `sum-iter` in Section 5.1. The continuation of the recursive call to `factorial_iter` in the definition of `factorial_iter` is

```
monad define 'y.'
```

Hence, no continuations need be saved and an optimizing compiler or interpreter will replace the recursion with iteration which directly computes the product of the integers 1 to k using a single call to `factorial_iter`.

5.3 Tail Recursion

A recursive definition, *f*, is said to be *tail recursive* if the continuation of each recursive call to *f* is the identity function. As long as the definition is not mutually recursive, it is an easy exercise for students to examine the source code of a definition and explicitly write the continuations to determine whether or not the definition is tail recursive. As many compilers and interpreters automatically recognize tail recursion and optimize these definitions as loops (gcc, for example), students quickly learn that they can write efficient iteration loops in a functional style.

The analysis of mutually recursive definitions is done in a similar fashion by hand, but is problematic for compilers and interpreters, particularly if the definitions are compiled separately.

6 Analyzing Algorithms

Students find that it is convenient to think of recursion in terms of composition of functions. They are explicitly aware of the order in which operations are done. For example, in Section 4.1, it is clear that the sum is accumulated in the order $0 + 1 + \ldots + 5$. They are also aware that recursive definitions may be used to provide efficient iterative programs as a result of analysis to determine whether or not a definition is tail recursive.

6.1 Recursive Fibonacci

The recursive definitions mentioned above all involve trivial linear recursive processes. The next example illustrates the kind of analysis a student might perform on a less trivial problem. Consider the standard recursive definition of the `fibonacci` function.

```
fibonacci =: monad define script
if. y. < 2
  do. y.
  else. (fibonacci y. - 1) + fibonacci y. - 2
end.
)
```

Program `fibonacci` (J Version)

Applying `fibonacci` to the argument 5 produces a result of 5. Tracing `fibonacci` produces the output

```
   traced_fibonacci 5
Entering, input = 5
Entering, input = 4
Entering, input = 3
Entering, input = 2
Entering, input = 1
Leaving, result = 1
Entering, input = 0
Leaving, result = 0
Leaving, result = 1
Entering, input = 1
Leaving, result = 1
Leaving, result = 2
Entering, input = 2
Entering, input = 1
Leaving, result = 1
Entering, input = 0
Leaving, result = 0
Leaving, result = 1
Leaving, result = 3
Entering, input = 3
Entering, input = 2
Entering, input = 1
Leaving, result = 1
Entering, input = 0
Leaving, result = 0
Leaving, result = 1
Entering, input = 1
Leaving, result = 1
Leaving, result = 2
Leaving, result = 5
5
```

Analyzing the `fibonacci` definition, we notice that there are two recursive calls to `fibonacci` inside this definition. We next write the continuations of each of these calls.

```
monad define 'y. + fibonacci n - 2'

monad define '(fibonacci n - 1) + y.'
```

`fibonacci` is not tail recursive. In fact, each continuation contains a recursive call to `fibonacci`. We also notice, from the traced output, that `fibonacci` makes applications of `fibonacci` to the same argument.

Consider the problem of evaluating `fibonacci 3`. Two continuations must be formed.

```
c1 =: monad define'y. + fibonacci 0'
c2 =: monad define'y. + fibonacci 1'
```

The value of `fibonacci 3` is represented by the expression `c2 c1 1`. Next, consider the problem of evaluating `fibonacci 4`. Three continuations must be formed.

```
c1 =: monad define'y. + fibonacci 0'
c2 =: monad define'y. + fibonacci 1'
c3 =: monad define'y. + fibonacci 2'
```

The value of `fibonacci 4` is represented by the expression `c3 c2 c1 1`.

Next we consider the number of times `fibonacci` is called while evaluating `fibonacci`. Define `fib_work` to be the number of times `fibonacci` is called while evaluating `fibonacci`. We see that:

- `fib_work 0 = 1`
- `fib_work 1 = 1`
- `fib_work 2 = 3`
- `fib_work 3 = 5`
- `fib_work 4 = 9`
- `fib_work 5 = 15`

It is easy to establish the recurrence formula for `fib_work`

```
fib_work n = 1 + (fib_work n - 1) + fib_work n - 2
```

Assuming that the execution time of `fibonacci` is proportional to `fib_work`, then the order of `fibonacci` is `fib_work` which is, itself, a `fibonacci` function. This analysis leads to a laboratory experiment [How 94] in which students conduct timing measurements of the number of recursive calls per second a workstation can make to `fibonacci`. Since `fibonacci` would make 1146295688027634168201 recursive calls while evaluating `fibonacci 100`, a workstation which can perform 1,000,000 recursive calls per second would require approximately 1146295688027634 seconds (more than 363487 centuries!) to evaluate `fibonacci 100`. This laboratory provides the opportunity for students to deal with formal analysis, experimental measurements, recursion and iteration.

6.2 Tail-Recursive Fibonacci

Introductory computer science texts [Abe 85, Spr 89] give tail-recursive definitions for the `fibonacci` function.

```
fib_iter =: monad define 'fib_iter_helper 1 0 , y.'

fib_iter_helper =: monad define script
('a' ; 'b' ; 'count') =. y.
if. count = 0
  do. b
  else. fib_iter_helper (a + b) , a , count - 1
end.
)
```

`fib_iter` makes a single call to `fib_iter_helper` to compute `fibonacci`. `fib_iter_helper` is tail-recursive since the continuation of the one recursive call is the identity function.

7 Iteration and Recursion Operators

Functional languages often have functional abstractions for iteration and recursion. For example, Scheme has a mapping (iteration abstraction) function named `map` which applies a function to each item of a list.

```
(map (lambda(x) (* x x x)) '(2 3 4 5))
```

produces the result

```
(8 27 64 125)
```

This same example is easily expressed using J. In J, the *power* function, expressed as `^`, may be used as the argument of the bond conjunction, `&`, producing the cube function `^&3`. This monad may be applied as:

```
(^&3) 2 3 4 5
```

producing the result

```
8 27 64 125
```

The J programming notation has a rich collection of abstractions for dealing with iteration over the items in lists and arrays. For example, the *insert* adverb, `/`, allows a dyad to be inserted between the items of a list or array.

```
+/ 1 2 3 4 5
```

evaluates the expression

```
1 + 2 + 3 + 4 + 5
```

while

```
*/ 1 2 3 4 5 6
```

evaluates the expression

```
1 * 2 * 3 * 4 * 5 * 6
```

Scheme and J both support an extensive family of abstractions for recursion and iteration which are beyond the scope of this paper. Such features are important in the exposition of a more advanced treatment of recursion and iteration. For example, Iverson [Iv 95] shows extensive use of J to construct mathematical proofs of correctness of algorithms.

8 Summary

Functional languages such as Scheme and J are useful in teaching recursion and iteration to introductory students. The exercise of writing the continuation of each recursive call in a definition forces students to think about the definition. Students also find that the alternate view of recursion as a composition of continuation functions gives a new perspective on recursive definitions. Identifying tail-recursive definitions or transforming non tail-recursive definitions into tail-recursive definitions is a useful exercise which helps enhance understanding of the algorithm. This author has found Scheme and J to be equally effective in the teaching of recursion and iteration. Both languages have significant advantages, particularly when used for exposition, over imperative languages.

References

[Abe 85] Abelson, Harold and Sussman, Gerald with Sussman, Julie., *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[Har 94] Harvey, Brian and Wright, Matthew, *Simply Scheme: Introducing Computer Science*, MIT Press, Cambridge, MA, 1994.

[How 94] Howland, John, "Lecture Notes for Great Ideas in Computer Science", Trinity University Computer Science Department Lecture Notes, http://www.cs.trinity.edu/About/The_Courses/cs301/

[How 95] Howland, John, " A Laboratory Computer Science Course for Liberal Arts Students", The Journal of Computing in Small Colleges, Volume 10, Number 5, May 1995.

[How 96] Howland, John, " Using J as an Expository Language in the Teaching of Computer Science to Liberal Arts Students", ACM APL96 Conference Proceedings, Lancaster University, England, August, 1996.

[How 97] Howland, John, " It's All in the Language (Yet Another Look at the Choice of Programming Language for Teaching Computer Science)", The Journal of Computing in Small Colleges, Volume 12, Number 4, March 1997.

[Iv 95] Iverson, Kenneth, *Concrete Math Companion*, Iverson Software, Toronto, Canada, 1995.

[Ive 95] Iverson, Kenneth E., *J Introduction and Dictionary*, Iverson Software, 1995.

[Kon 74] Konstam, Aaron and Howland, John, "APL as a Lingua Franca in the Computer Science Curriculum", SIGCSE Bulletin, Volume 6, Number 1, February 1974.

[Kon 94] Konstam, Aaron and Howland, John, "Teaching Computer Science Principles to Liberal Arts Students Using Scheme", SIGCSE Bulletin, Volume 26, Number 4, December 1994.

[Man 95] Manis, Vincent S. and Little, James J., *The Schematics of Computation*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[Rie 93] Riehl, Arthur, moderator, "Using Scheme in the Introductory Computer Science Curriculum", Panel, SIGCSE Bulletin, Volume 25, Number 1, March 1993.

[Spr 89] Springer, George and Friedman, Daniel, *Scheme and the Art of Programming*, MIT Press, 1989.