Functional Languages and Introductory Computer Science

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

The choice of which programming language to use in introductory computer science courses is guaranteed to spark debate in the computer science community. Programming languages used in computer science instruction have followed various trends or fads within the computing industry. The language choice has often been between languages which are currently in wide use by industry for software production. While it is true that computer science education has a responsibility to achieve a balance between providing training in current practices within the field and core concepts and theory, it is felt that computer science education should not be overly influenced by popular trends when choosing a programming language to use in the teaching of introductory computer science. Functional programming languages are shown to be useful in the teaching of the concepts of computer science. The functional language approach presented in this paper has advantages over imperative languages in the areas of model building, exposition, experimentation and analysis of algorithms. Examples using the J and Scheme programming languages, with emphasis on the use of functional programming notation in exposition are given. [1]

[1] The abstract of this paper appears in the Journal of Computing in Small Colleges, Volume 13, Number 4, Page 151, March 1998. Copyright ©1998 by the Consortium for Computing in Small Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing in Small Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission. The paper was presented at the CCSC South-Central Conference, April 18, 1998, Millsaps College, Jackson, Mississippi.

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

1 Introduction

The choice of which programming language to use in introductory computer science courses borders on being a religious issue in which divides computer science departments over issues involving what are believed to be practical skills required by industry and the requirements of pedagogy. More often, in the past, industrial requirements have prevailed as witnessed by the use of languages such as Cobol, FORTRAN, PL/I and more recently C and C++. Languages have been developed which have, in part, focused on education, i.e. Pascal, Modula, etc., but these languages have not become a dominant force in the commercial production of software. At least one computer scientist [Dij 89] has advocated the use of an unimplemented programming language to teach computer science which necessarily forces programming instruction to be purely an intellectual activity. Recent growth in Internet activity has provided stimulus for the development of software systems which may be executed on a variety of different hardware/software environments. One of these, Java, which uses an abstract Java virtual machine to host the software is remarkably similar in overall concept to the Pascal P machine. Because of Internet popularity, Java is now proposed by some as a suitable vehicle for teaching computer science. Trends or fads come and go in computer science education just as in other fields.

In each of the programming languages mentioned above, it is not clear that the choice to use the language for computer science instruction is made primarily for pedagogical reasons. In Section 1.1, criteria are given which are based on requirements of computer science instruction, particularly use of programming notation in an expository fashion in the teaching of introductory computer science.

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.

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.

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 introductory computer science.

1.1 Criteria

- Interactive Environment
- Language Sentence Structure
- Model Building
- Experimentation
- Reasoning About Programs
- Data Abstraction
- Procedure Abstraction
- Functional and Imperative Programming
- Exact and Inexact Arithmetic
- Object Programming
- Recursion
- Iteration
- Algorithm Analysis
- Recursion and Iteration Operators

2 Interactive Environment

The advent of computer equipped classrooms where the instructor and students have workstations and network based systems for using language aware electronic blackboards has increased the importance of an interactive environment for a language. By interactive, we mean systems which operate in a read-evaluate-print loop. One enters an expression which is parsed, analyzed and evaluated in real time and then results are printed and the process is repeated. Of course such systems may be run in batch mode reading from standard input and writing to standard output with input/output redirection. Also, although such systems are often implemented as interpreters, the interactive Scheme and J systems may also have compilers which are capable of generating binary machine language programs.

2.1 Scheme Example

The Scheme system illustrated here prompts the user for input with "> ". System output starts at the left margin.

```
> (map + '(1 2 3) '(10 20 40))
(11 22 43)
>
```

2.2 J Example

The J system illustrated here prompts the user for input with three spaces. System output starts at the left margin.

```
   1 2 3 + 10 20 40
11 22 43
```

3 Language Sentence Structure

When using programming notation in an expository notation for computer science, it is important that the notation have a sentence structure that can be easily verbalized. When reading and thinking in an explicit manner, we actually verbalize our thoughts even though we are not speaking out loud. Conventional programming languages are often difficult to verbalize and because of this are not as suitable for exposition as are languages which are more easily verbalized. Dijkstra [Dij 72], in his Turing lecture, "The Humble Programmer", stated "... that the tool we are trying to use and the language or notation we are using to express or record our thoughts are the major factors determining that we can think or express at all!". The expressive power of a language is one yardstick by which one may measure the relative merits of a programming language.

A programming notation which is used in an expository manner should have a simple syntax which is easy to learn and an easily understood evaluation rule.

3.1 Scheme Sentences

Scheme sentences are sequences of words separated by spaces, preceded and followed by "(" and ")". The first word of a sentence is a verb (or verb like special word) which is applied to the remaining words in the sentence. For example:

```
> (* 2 3)
6
```

is verbalized as times 2 3. Some sentences use special words which are technically not verbs. An example is:

```
(if (< a b)
     a
     b)
```

This sentence may be verbalized as If less than a b, then a, else b. The word if is not a verb which means that an if sentence has a special evaluation rule. There are relatively few special words and hence relatively few exceptions to the normal rule for sentence formation.

Compound sentences may be formed as in:

```
(* (- a b) (- a c))
```

which might be verbalized as Times the quantity minus a b and the quantity minus a c.

3.2 J Sentences

J sentences are sequences of words separated by spaces. Most sentences are limited to one physical line and are read from left to right. The J primitive words are formed from the ASCII character set. Primitive words are represented with a single character or a character followed by a period or colon. For example, ¿ represents larger than, ¿. larger of and ¿: increment. Terminology from English grammar is used to describe J. Functions are referred to as verbs, constants as nouns, names assigned to values as pronouns and names assigned to verbs as proverbs. Infix conventions are used which means that dyads (verbs having two nouns or pronouns as inputs) are written between the nouns while monads (verbs having one noun or pronoun as input) are written before the noun. For example:

```
   2 * 3
6
```

is verbalized as 2 times 3. Compound sentences may be formed as in:

```
   2 * 3 + 4
14
```

which might be verbalized as 2 times, 3 plus 4. With the exception of subordinate clauses enclosed in parentheses, verbs are evaluated in the order from right to left so that the sentences may be read from left to right. The right input to a verb is the value of the entire expression to the right and the left input is the value of the noun immediately to the left of the verb. Most verb symbols have two interpretations (dyad or monad) and the choice between interpretations is determined by the context as illustrated by the sentence:

```
   4 - - 2
6
```

which is verbalized as 4 subtract negate 2. Punctuation (parentheses) may be used to modify the order of evaluation as in:

```
   (2 * 3) + 4
10
```

which is verbalized as The quantity 2 times 3 plus 4.

In J, verbs may be modified by adverbs or conjunctions to form new verbs which are then applied to inputs. For example, in:

```
   +/1 2 3 4
10
   */1 2 3 4
24
```

the first is verbalized as plus insert the list 1 2 3 4 while the second is verbalized as times insert the list 1 2 3 4. In each of these sentences, the adverb insert (spelled "/") modifies the verb (plus or times) producing a new verb which sums or multiplies the elements of the list. Adverbs or conjunctions have precedence over verbs with the left input being the entire verb phrase on the left.

3.3 Programs

In both Scheme and J, programs and data are represented by the same notation; lists in Scheme and lists or arrays in J. This closely models the situation of storing both programs and data in the memory of a computer in numeric form.

Scheme Programs A single sentence is a simple Scheme program. For example:

```
> (* (+ 4 5) (- 3 2))
9
```

Functions may be defined as compound sentences involving the special words define and lambda. For example:

```
(define square
    (lambda (x)
      (* x x)))
```

Then

```
> (square 10)
100
```

Data is described using the special word quote as follows:

```
(define people
    (quote ((Clinton (president United States))
            (Dole (wanted to be President))
            (Perot (also wanted to be President)))))
> (length people)
3
```

J Programs A single J sentence is also a program. For example:

```
   (4 + 5) * (3 - 2)
9
```

The J word "=:" assigns (binds) a name (pronoun) to a value (noun). For example, suppose we have defined the following words:

```
   monad =: 3
   define =: :
```

Then the sentence:

```
   square =: monad define 'y. * y.'
```

is an explicit definition for the square function. The pronoun "y." always refers to the input of a monad. Given this definition, then we may write:

```
   square 10
100
```

The J notation provides great expressive power when defining functions through an alternate method called tacit definition. A feature of tacit definition is there is no reference to the inputs of a definition. For example, square could also be defined as:

```
   square =: ^&2
```

Here we are using a conjunction (a verb producing verb) called bond (spelled "&") which takes a verb input on the left (power function, spelled "^") and a noun input (2) on the right and produces a new verb which squares its argument. Square could also be defined as:

```
   square =: *~
```

Here we use the adverb reflex (spelled "~") which takes a dyad as its left input and converts the dyad into a monad by using its input y. as its left and right input for the dyad. In this case the verb derived from * is y. * y. .

Finally, the square function occurs frequently enough in programs so that it is provided as a primitive function in J, (spelled "*:". So we could also write:

```
   square =: *:
```

4 Model Building

A technique, useful in the teaching of computer science, is to use programming notation to build small working models of the topic being described. A successful notation, in this application, will provide concise, but fully accurate, working models. Both Scheme and J excel in model building. Suppose we wish to use recursive definitions, in a divide and conquer fashion, to model both recursive and iterative processes. This technique is often used when analyzing the fibonacci sequence. We use this example to illustrate not only modeling techniques but also illustrate the expressive power of Scheme and J to describe recursive and iterative processes and continuations.

4.1 Modeling Processes with Scheme

The fibonacci sequence 0 1 1 2 3 5 8 13, ... may be generated by the recursive definition:

```
(define fibonacci
  (lambda (n)
    (if (< n 2)
      n
      (+
        (fibonacci (- n 1))
        (fibonacci (- n 2))))))
```

When analyzing this recursive definition, it is useful to define a related function, fib-work, whose value, given an input n, is the number of times fibonacci is called when evaluating `(fibonacci n)`. It is easy to show that fib-work may be defined as:

```
(define fib-work
  (lambda (n)
    (if (< n 2)
      1
      (+ 1
        (fib-work (- n 1))
        (fib-work (- n 2))))))
```

fib-work, itself, generates the values of a kind of fibonacci sequence. If it is our goal to evaluate either of these functions for inputs greater than 25 to 30, it is necessary to convert these definitions to definitions which result in iterative processes. A recursive definition for fib-work which results in an iterative process is given by the definition:

```
(define fib-work-iter
  (lambda (n) (fib-work-iter-helper 1 1 n)))

(define fib-work-iter-helper
  (lambda (a b count)
    (if (= count 0)
      b
      (fib-work-iter-helper (+ 1 a b) a (- count 1)))))
```

Suppose f is a compound expression and e is a sub expression of f. The continuation of e in f is that function of a single input x, (lambda (x) ...) which contains the execution in f which remains to be done after evaluating the sub expression e. This means that the value of the entire expression f may be obtained by evaluating ((lambda (x) ...) e). Continuations allow a compound expression to be factored into an expression e which is evaluated first and a function which may be called with the resulting value of e as an input.

The idea of a continuation may be used to define tail recursive functions. A function is tail recursive if the continuation of each recursive reference in the definition is the identity function.

Analysis of fib-work-iter reveals that the work of this definition is done by the recursive definition fib-work-iter-helper which has one recursive use of fib- work-iter-helper whose continuation is the identity function. Hence, fib-work- iter is tail recursive which means that its process is iterative. Here we are assuming that any tail recursive definition will be optimized by the Scheme system so that an iterative process will be generated. This will be true of any standard Scheme implementation. The end result of all of this is that fib-work-iter will easily evaluate the fib-work function for the input value 100. Indeed,

```
> (fib-work-iter 100)
1146295688027634168201
```

4.2 Modeling Processes with J

We express the same example used in Section 4.1 using the J notation to show the expressiveness of J for modeling and recursive and iterative processes.

The fibonacci sequence 0 1 1 2 3 5 8 13, ... may be generated by the recursive definition:

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

When analyzing this recursive definition, it is useful to define a related function, fib_work, whose value, given an input n, is the number of times fibonacci is called when evaluating fibonacci n . It is easy to show that fib_work may be defined as:

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

As in Section 4.1, fib_work, itself, generates the values of a kind of fibonacci sequence. If it is our goal to evaluate either of these functions for inputs greater than 25 to 30, it is necessary to convert these definitions to definitions which result in iterative processes. A recursive definition for fib_work which results in an iterative process is given by the definition:

```
fib_work_iter =: monad define 'fib_work_iter_helper 1 1 , y.'

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

Again, as in Section 4.1, we may define the idea of a continuation. Suppose f is a compound expression and e is a sub expression of f. The continuation of e in f is that monad having input y., monad define '... y. ...' which contains the execution in f which remains to be done after evaluating the sub expression e. This means that the value of the entire expression f may be obtained by evaluating:

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

Since the continuation of each recursive use of fib_work_iter_helper in the definition of fib_work_iter_helper is the identity function, fib_work_iter generates a more efficient iterative process so that:

```
   fib_work_iter 100x
1146295688027634168201x
```

The J numeric suffix "x" indicates that an exact numeric representation should be used in this computation.

5 Experimentation

Experimental methods play an important role in computer science and should be a part of the introductory computer science curriculum. Measuring program performance, testing experimental hypotheses are areas where traditional scientific methodology may be used. Scheme and J systems provide facilities for measurements of memory space and execution time of programs. As an example of a simple experiment which might be performed by introductory students, consider the problem of estimating the execution time of the recursive fibonacci function discussed in Section 4. In Section 5.1, a J version of this experiment is described. The Scheme description of this experiment is similar.

5.1 Using J to Estimate the Time to Evaluate fibonacci 100

First define the time verb using the external conjunction. time returns the time in seconds to evaluate the sentence given as its right input.

```
time =: 6 !: 2
```

Next, using the definition of fibonacci given in Section 4.2, determine the speed, in calls per sec to fibonacci, of fibonacci n for values of n not more than about 25. Note that you need to use the fib_work_iter function from Section 4.2 to compute these speeds. For example, on a RISC workstation you might measure this speed as:

```
   (fib_work_iter 20) % 10 time 'fibonacci 20'
2650.24
```

This measurement gives a speed of about 2650 calls per sec as determined by the average of 10 evaluations of fibonacci 20.

The next step of the experiment involves dividing fib_work_iter 100 by 2650 to obtain the estimate of the time in seconds to evaluate fibonacci 100. This division requires exact integer division which is expressed in J as:

```
   0 2650 #: 1146295688027634168201x
432564410576465723x 2251x
```

Ignoring the remainder of 2251 we have a result of 432564410576465723 seconds (the suffix "x" indicates an exact integer). Students performing this lab are always surprised to learn that this time is 13,716,527,478 years, 350 days, 4 hours, 55 minutes and 23 seconds. This result is easily expressed in J as:

```
   0 365 24 60 60 #: 432564410576465723x
13716527478x 350x 4x 55x 23x
```

6 Reasoning About Programs

Myers [My 90] makes compelling arguments that an introduction to formal methods should be a part of introductory computer science courses. Such methods include the topics of proof of program correctness, analytic methods of transformation and simplification of programs, etc. Since both Scheme and J are derived from formal mathematical notation it is not surprising that they may be used to introduce and describe formal methods in computer science. In [Ive 95], Iverson describes J as "... a formal imperative language. Because it is imperative, a sentence in J may also be called an instruction, and may be executed to produce a result. Because it is formal and unambiguous it can be executed mechanically by a computer, and is therefore called a programming language. Because it shares the analytic properties of mathematical notation, it is also called an analytic language."

6.1 Using J for Proofs

Iverson and others have written several books which use J to describe a number of computing related topics. One of these [Iv 95] uses J in a rather formal way to express algorithms and proofs of topics covered in [Gr 89]. Following is an example from the introduction of [Iv 95].

A theorem is an assertion that one expression l is equivalent to another r. We can express this relationship in J as:

```
t=: l -: r
```

This is the same as saying that l must match r, that is, t must be the constant function 1 for all inputs. T is sometimes called a tautology. For example, suppose

```
l =: +/ @ i.     NB. Sum of integers
r =: (] * ] - 1:) % 2:
```

If we define n =:] , the right identity function, the we can rewrite the last equations as:

```
r =: (n * n - 1:) % 2:
```

Next,

```
t =: l -: r
```

Notice that by experimentation, t seems to always be 1 no matter what input argument is used.

```
   t 1 2 3 4 5 6 7 8 9
1 1 1 1 1 1 1 1 1
```

A proof of this theorem is a sequence of equivalent expressions which leads from l to r.

```
l
+/ @ i.                           Definition of l
+/ @ |. i.                        Sum is associative and commutative
                                  (|. is reverse)
((+/ @ i.) + (+/ @ |. @ i.)) % 2: Half sum of equal values
+/ @ (i. + |. @ i.) % 2:          Summation distributes over addition
+/ @ (n # n - 1:) % 2:            Each term is n -1; there are n terms
(n * n - 1:) % 2:                 Definition of multiplication
r                                 Definition of r
```

Of course, each expression in the above proof is a simple program and the proof is a sequence of justifications which allow transformation of one expression to the next.

7 Data Abstraction

Both Scheme and J allow a functional approach to data abstraction which allows data abstractions to be separated from actual representation of abstract data types. This approach provides an interface which defines software layers. A J example of an abstract data type of stack is given.

7.1 J Data Abstraction for Stacks

We define the stack data type to be a collection of J items together with the following operations:

```
make_stack ==> constructs a stack
stackp obj ==> 1 if obj is a stack, else 0
empty_stackp stack ==> 1 if stack empty, else 0
push_stack item ; stack ==> put item on stack
pop_stack stack ==> remove last item pushed on stack
top_stack stack ==> return last item pushed on stack
                      without removing that value from stack
```

We represent a stack as a J boxed list which has a stack "tag" of 'stack' as its first item. First we define the helping words:

```
box =: <
open =: >
match =: -:
first =: {.
append =: ,
drop_last =: _1 & }.
last =: _1 & {

stack_tag =: box 'stack'
the_empty_stack =: box stack_tag
```

The stack operations may be written as:

```
make_stack =: monad define 'the_empty_stack'
stackp =: monad define 'stack_tag match first open y.'
empty_stackp =: monad define 'the_empty_stack match y.'

push_stack =: monad define script
('item' ; 'obj') =. y.
if. not stackp box obj
  do. error 'wrong type second input to push_stack' ; obj
  else. box obj append box item
end.
)

pop_stack =: monad define script
if. not stackp y.
  do. error 'wrong type input to pop_stack' ; y.
  else. box drop_last open y.
end.
)
```

```
top_stack =: monad define script
if. not stackp y.
  do. error 'wrong type input to top_stack' ; y.
  else. open last open y.
end.
)
```

7.2 Using the J stack abstraction

Following is a sample session using the stack abstraction of Section 7.1.

```
   s =: make_stack ''
   stackp s
1               NB. s is a stack
   empty_stackp s
1
   s =: push_stack 1 2 3 ; s  NB. Push the list 1 2 3 on s
   top_stack s
1 2 3
   empty_stackp s
0               NB. s is not empty now
   s =: push_stack 'Some text' ; s  NB. Push a char string on s
   top_stack s
Some text
   s =: pop_stack s
   top_stack s
1 2 3
   s =: pop_stack s
   empty_stackp s
1               NB. s is empty again
```

8 Procedure Abstraction

Procedure abstraction is not easily achieved in languages such as Pascal, C or C++, however, in Scheme and J, functions are first class entities. They may be passed as arguments, assigned names and returned as values. Springer and Friedman [Spr 89] describe procedural abstractions in Scheme which solve classes of problems involving flat recursion of the top level elements of a list or deep recursion on all sub lists of a list.

8.1 Procedural Abstraction using J

In J, functions may be passed as arguments and returned as values. Adverbs are functions whose arguments are functions and results are functions. For example, insert (spelled "/") is an adverb which derives a verb result which is inserted between the items of its argument.

```
   +/ 10 20 50    NB. sum
80
   */ 10 20 50    NB. product
10000
   -/ 10 20 50    NB. difference
40
```

Suppose a is defined by:

```
   a =: i. 2 3
   a
0 1 2
3 4 5

      +/ a
3 5 7
      */ a
0 4 10
```

Rank (spelled ", double quote) is a conjunction (a verb producing dyad) produces a result verb (derived from its left input) which is applied to its argument according to the right input of rank. For example, a (defined above) has two rows and 3 columns, and is said to be of rank 2 (2 dimensions).

```
      +/ " 1 a  NB. plus insert applied to the rank 1 items of a (rows)
3 12  NB. row sum
   */ " 1 a  NB. times insert applied to the rank 1 items of a (rows)
0 60  NB. row product
   +/ " 2 a  NB. plus insert applied to the rank 2 items of a (columns)
3 5 7  NB. column sum
   */ " 2 a  NB. times insert applied to the rank 2 items of a (columns)
0 4 10  NB. column product
```

J supports a number of other abstractions, too numerous to mention in this paper, such as hooks, forks, trains, function arrays, gerunds, agenda, power and inverse (where defined) for primitive functions as well as explicitly defined functions.

9 Functional and Imperative Programming

Scheme supports a functional style of programming (when you restrict use of procedures which alter already existent object, such as set-car!, set-cdr!, vector-set!, vector-fill!, etc.) as well as an imperative style of programming when the above mentioned procedures are used. J is a functional notation where the model of computation consists of application of functions to arguments without side effects (roll and deal have side effects; pseudo random state is modified). Once an item is created in memory it is never modified; functions may be applied to

such items producing new items. Functional composition is the primary model for computation.

10 Exact and Inexact Arithmetic

Scheme and J both support a model of exact integer arithmetic in addition to arithmetic of other numeric types such as complex and inexact (floating point) numbers. Exact values are limited in precision only by available (possibly virtual) memory.

11 Object Programming

Object programming combines data structure and operations on data structure to entities called objects. Objects provide abstraction, encapsulation and inheritance to provide data based modularization for programs. Objects are easily modeled in Scheme using lexical closures [How 94], Chapter 6. Objects may be modeled in J using locales [How 94], Chapter 6.

12 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.

12.1 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)

12.2 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
```

12.3 The Factorial Function

To compute the product of the integers 1 to k using the divide and conquer approach, we have, as in Section 12.1, 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 12.1, 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
```

12.4 Continuations

In Sections 12.2 and 12.3 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

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

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.

12.5 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))
```

12.6 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
```

12.7 Expressing Recursion as Functional Composition

Consider the function `sum` of Section 12.1 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.

12.8 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)))))
```

12.9 Using J Notation

Consider the `factorial` function defined in Section 12.3 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
```

13 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.

13.1 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 12.4, 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`.

13.2 J Example

Next we consider an alternate solution to the problem from Section 12.3 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 13.1. 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`.

13.3 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 convert such definitions to 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.

14 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 12.8, 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.

14.1 Recursive Fibonacci

The recursive definitions mentioned above all involved 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.

14.2 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 computer `fibonacci`. `fib_iter_helper` is tail-recursive since the continuation of the one recursive call is the identity function.

15 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.

16 Available Implementations

There are a number of Scheme and J implementations available for nearly every machine and operating system including Windows, WindowsNT, MacOS, Linux and other varieties of UNIX. Commercial versions of these languages are available as well, but free versions have proven to be more than adequate for laboratory use and students are able to install versions of the software identical to lab systems on their own machines. More information on available Scheme software may be found at:

`ftp://ftp.cs.indiana.edu/pub/scheme-repository/`

Information on J software may be found at:

`http://www.jsoftware.com .`

16.1 Text Materials

Several well known introductory computer science text books which use Scheme are (most notably) [Abe 85, Fr 92, Har 94, Man 95, Spr 89]. Development of the J programming language is relatively recent, with the first papers on J appearing in 1991. To this date, the only J based computer science text materials are [How 94]. However, J has been used in an expository fashion to describe several topics in mathematics [Iv 92, Iv 93, Iv 95, Re 95].

17 Schools Using Scheme or J

A list of colleges and universities using Scheme in various courses is maintained at the Scheme Repository:

`http://www.cs.indiana.edu/scheme-repository/home.html`

Information about the use of the J programming language is maintained at:

`http://www.jsoftware.com/`

18 Summary

Languages which are routinely used by industry, such as Basic, FORTRAN, C, C++, etc., are often not entirely suitable for expository presentation of topics in the introductory computer science curriculum. Frequently, students entering computer science programs have already studied one of these languages and have some programming experience. Choosing a language which is better suited for expository presentation of computer science topics, such as Scheme or J, can have a leveling effect amongst students who have different preparation for college level training in computer science. Moreover, choosing a language which allows expression of powerful ideas helps give the mindset which allows students to think what might otherwise have been "unthinkable" thoughts. Such notation should foster development of formal methods in addition to the practical aspect of design, analysis and programming. An important method of exposition involves building small working models of each topic. Once built, such models provide the basis for laboratory experimentation which can involve measurements, formulation and verification of hypotheses and analysis.

Programming notation becomes a powerful tool of exposition by making an appropriate choice of language. The decision about choice of programming language should be made primarily on the basis of how well key concepts in computer science may be expressed in the language. For these reasons Scheme or J is preferable to other languages commonly used in introductory computer science courses.

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.

[Dij 72] Dijkstra, Edsger, "The Humble Programmer", 1972 ACM Turing lecture, reprinted in ACM Turing Award Lectures, The First Twenty Years, pp 17 - 31, ACM Press, 1987.

[Dij 89] Dijkstra, Edsger, "On The Cruelty of Really Teaching Computing Science", 1989 SIGCSE Award Lecture, SIGCSE Bulletin, Vol. 21, No. 1, February 1989.

[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 Ar ts 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 Curricul um", Panel, SIGCSE Bulletin, Volume 25, Number 1, March 1993.

[Re 95] Reiter, Cliff, *Fractals Visualization and J*, Iverson Software, Toronto, CA, 1995.

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

[Fr 92] Friedman, Daniel, Wand, Mitchell and Haynes, Christopher. *Essentials of Programming Languages*, MIT Press, 1992.

[Gr 89] Graham, Knuth and Patashnik, *Concrete Mathematics*, Addison-Wesley Publishing Company, Reading, MA, 1989.

[Iv 92] Iverson, Kenneth, *Arithmetic*, Iverson Software, Toronto, Canada, 1992.

[Iv 93] Iverson, Kenneth, *Calculus*, Iverson Software, Toronto, Canada, 1993

[My 90] Myers, J. Paul, "The Central Role of Mathematical Logic in Computer Science", SIGCSE Bulletin, Vol. 22, No. 1, February 1990.