Building Models: A Direct but Neglected Approach to Teaching Computer Science *

John E. Howland
Department of Computer Science
Trinity University
715 Stadium Drive
San Antonio, Texas 78212-7200
Voice: (210) 999-7364
Fax: (210) 999-7477
E-mail: jhowland@Trinity.Edu
Web: http://www.cs.trinity.edu/~jhowland/

December 20, 2003

Abstract

The use of software models for teaching a variety of computer science topics is a valuable technique. Such models may be studied by reading and examining each model itself. Additionally, the models form the basis for experimentation. The J language is particularly well suited for modeling. It is not necessary that students be proficient in J programming to make effective use of J models and experiments with models are easily devised so that laboratory measurements may be taken. Example models for a number of computer science topics are given. [1]

Subject Areas: Computer Science Education, Computer Science Curriculum Computer Science Laboratories.
Keywords: Modeling, J Programming Language.

1 Introduction

In this paper, the term *modeling* is used in the context of software modeling, as in models of computer science entities implemented as programs (or program fragments) in some programming language. The extent to which model building is useful depends on the choice of programming language as well as the skill of the model builder. Since much of what we teach in computer science is based in part on mathematics it is useful for the modeling language to be a reasonable substitute for ordinary mathematical notation.

A good model building language should possess the following attributes:

*With apologies to Berry, Falkoff and Iverson who used a similar title [Berry 1970] for a paper on teaching mathematics which was inspiration, in part. for this paper.

[1]This paper was published in the Journal of Computing in Small Colleges, Volume 17, Number 5, Pages 110–124, April 2002. Copyright ©2002 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 copright 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.

1.1 Attributes for Software Modeling

- notation for mathematics
- rich set of primitive operations
- rich set of primitive data structures
- exact and inexact arithmetic
- higher level functions
- concise expressive power
- interactive environment
- freely available on a variety of computing systems

A modeling language needs a wide variety of functions. These include the classical functions of mathematics as well as functions for creating and manipulating data structures and functions which perform exact as well as inexact arithmetic. A successful modeling language will allow the expression of higher level functions (operators defined on function domains which produce function results) and treat functions as first class data items.

Ordinary mathematical notation provides economical expression of powerful ideas such as $\int_a^b f(x)dx$, $\lim_{n\to\infty} x_i$ or $\sum_{i=1}^n x_i$. A successful modeling language should be able to express powerful abstractions of mathematics and computer science.

Finally, modern computer science lecture halls and laboratory rooms have elaborate computer driven, large-screen displays. A modeling language should allow an interactive environment so that an instructor can write the language as one would write equations or diagrams on a white board.

1.2 Software Models

Software models, described in precise notation, serve an expository purpose. A student's reading of the model gives insight as to the form, structure and function of the entity being modeled. The model may be inspected and abstracted to be used as a building block for a more complex entity.

1.3 Reading Models

A reading of the source code for the model gives the student a precise description of the entity being modeled. Expository use of notation serves to remove the ambiguity and imprecision of natural language descriptions of a computing concept.

1.4 Experimentation with Models

Software models, being executable programs, have the potential of providing experimental apparatus. Experimentation with a model often provides insight and ocasionally uncovers model behavior which is counter-intuitive.

2 J as a Modeling Notation

The J programming language [Berry 1970, Burk 2001, Bur 2001, Hui 2001] is, perhaps, the only programming language which satisfies the criteria of Section 1.1. J uses infix notation with primitive functions denoted by a special symbol, such as `+` or `%`, or a special symbol or word followed by the suffix of `.` or `:` . Each function name may be used as a *monad* (one argument, written to the right) or as a *dyad* (two arguments, one on the left, the other on the right).

The J vocabulary of primitive (built-in) functions is shown in Figures 1 and 2. These figures show the monadic definition of a function on the left of the `*` and the dyadic definition on the right. For example, the function symbol `+:` represents the monad `double` and the dyad `not-or` (`nor`).

`=`	Self-Classify * Equal	`=.`	Is (Local)	`=:`	Is (Global)
`<`	Box * Less Than	`<.`	Floor * Lesser Of (Min)	`<:`	Decrement * Less Or Equal
`>`	Open * Larger Than	`>.`	Ceiling * Larger of (Max)	`>:`	Increment * Larger Or Equal
`_`	Negative Sign / Infinity	`_.`	Indeterminate	`_:`	Infinity
`+`	Conjugate * Plus	`+.`	Real / Imaginary * GCD (Or)	`+:`	Double * Not-Or
`*`	Signum * Times	`*.`	Length/Angle * LCM (And)	`*:`	Square * Not-And
`-`	Negate * Minus	`-.`	Not * Less	`-:`	Halve * Match
`%`	Reciprocal * Divide	`%.`	Matrix Inverse * Matrix Divide	`%:`	Square Root * Root
`^`	Exponential * Power	`^.`	Natural Log * Logarithm	`^:`	**Power**
`$`	Shape Of * Shape	`$.`	Sparse	`$:`	Self-Reference
`~`	***Reflex*** * ***Passive*** / EVOKE	`~.`	Nub *	`~:`	Nub Sieve * Not-Equal
`|`	Magnitude * Residue	`|.`	Reverse * Rotate (Shift)	`|:`	Transpose
`.`	**Determinant** * **Dot Product**	`..`	**Even**	`.:`	**Odd**
`:`	**Explicit** / **Monad-Dyad**	`:.`	**Obverse**	`::`	**Adverse**
`,`	Ravel * Append	`,.`	Ravel Items * Stitch	`,:`	Itemize * Laminate
`;`	Raze * Link	`;.`	**Cut**	`;:`	Word Formation *
`#`	Tally * Copy	`#.`	Base 2 * Base	`#:`	Antibase 2 * Antibase
`!`	Factorial * Out Of	`!.`	**Fit** (**Customize**)	`!:`	**Foreign**
`/`	***Insert*** * ***Table***	`/.`	***Oblique*** * ***Key***	`/:`	Grade Up * Sort
`\`	***Prefix*** * ***Infix***	`\.`	***Suffix*** * ***Outfix***	`\:`	Grade Down * Sort

Figure 1: J Vocabulary, Part 1

J uses a simple rule to determine the order of evaluation of functions in expressions. The argument of a monad or the right argument of a dyad is the value of the entire expression on the right. The value of the left argument of a dyad is the first item written to the left of the dyad. Parentheses are used in a conventional manner as punctuation which alters the order of evaluation. For example, the expression `3*4+5` produces the value `27`, whereas `(3*4)+5` produces the value `17`.

The evaluation of higher level functions (function producing functions) must be done (of course) before any functions are applied. Two types of higher level functions exist; *adverbs* (higher level monads) and *conjunctions* (higher level dyads). Figures 1 and 2 show adverbs in bold italic face and conjunctions in bold face. For example, the conjunction bond (Curry) binds an argument of a dyad to a fixed value producing a monad function as a result (`4&*` produces a monad which multiplies by `4`).

J is a functional programming language which uses functional composition to model computational processes. J supports a form of programming known as *tacit*. Tacit programs have no reference to their arguments and often use special composition rules known as *hooks* and *forks*. Explicit programs with traditional control structures may also be written. Inside an explicit definition, the left argument of a dyad is always named `x.` and the argument of a monad (as well as the right argument of a dyad) is always named `y.` .

J supports a powerful set of primitive data structures for lists and arrays. Data (recall that functions

[	Same * Left	[.	**Lev**	[:	Cap
]	Same * Right	].	**Dex**	]:	***Identity***
{	Catalogue * From	{.	Head * Take	{:	Tail * {:: Map * Fetch
}	***Item Amend*** * ***Amend***	}.	Behead * Drop	}:	Curtail *
"	**Rank**	".	Do * Numbers	":	Default Format * Format
‘	**Tie** (**Gerund**)			‘:	**Evoke Gerund**
@	**Atop**	@.	**Agenda**	@:	**At**
&	**Bond / Compose**	&.	**Under** (**Dual**)	&:	**Appose**
?	Roll * Deal	?.	Roll * Deal (fixed seed)		
a.	*Alphabet*	a:	*Ace* (Boxed Empty)	A.	Anagram Index * Anagram
b.	***Boolean / Basic***	c.	Characteristic Values	C.	Cycle-Direct * Permute
d.	**Derivative**	D.	**Derivative**	D:	**Secant Slope**
e.	Raze In * Member (In)	E.	* Member of Interval	f.	***Fix***
H.	**Hypergeometric**	i.	Integers * Index Of	i:	Integers * Index Of Last
j.	Imaginary * Complex	L.	Level Of	L:	**Level At**
m. n.	Explicit Noun Args	NB.	Comment	o.	Pi Times * Circle Function
p.	Polynomial	p:	Primes *	q:	Prime Factors * Prime Exponents
r.	Angle * Polar	s:	Symbol	S:	**Spread**
t.	***Taylor Coefficient***	t:	***Weighted Taylor***	T.	**Taylor Approximation**
u. v.	Explicit Verb Args	u:	Unicode	x. y.	Explicit Arguments
x:	Extended Precision	_9: to 9:	Constant Functions		

Figure 2: J Vocabulary, Part 2

have first-class status in J), once created from notation for constants or function application, is never altered. Data items possess several attributes such as *type* (numeric or character, exact or inexact, etc.) *shape* (a list of the sizes of each of its axes) and *rank* (the number of axes). Names are an abstraction tool (not memory cells) which are assigned (or re-assigned) to data or functions.

3 Example Models

A small sample of models used in the teaching of various computer science topics is given in Section 3. When reading each of the example models, remember that they are used for the following purposes:

- To give a precise specification of the topic
- To allow examination of the properties of the topic
- To use the model for experimentation

In Section 3.1 some detail about the model and experimental approach is given. The remaining examples in Sections 3.2 to 3.7 only give a representative J model and leave out most experimental details due to space constraints. When J is used interactively, inputs to a J session are shown indented by 3 spaces while responses begin at the left margin.

3.1 Algorithms and their Processes

Howland [How 1998] used the often studied recursive Fibonacci function to describe recursive and iterative processes. In J, the recursive Fibonacci function is defined as:

```
fibonacci =. monad define
if. y. < 2
```

```
  do. y.
  else. (fibonacci y. - 1) + fibonacci y. - 2
end.
)
```

Applying fibonacci to the integers 0 through 10 gives:

```
   fibonacci "0 i.11
0 1 1 2 3 5 8 13 21 34 55
```

Howland [How 1998] also introduced the idea of a continuation; a monad representing the computation remaining in an expression after evaluating a sub-expression.

> Given a compound expression e and a sub-expression 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.

Analysis of the recursive `fibonacci` definition reveals that each continuation of `fibonacci` in `fibonacci` contains an application of `fibonacci`. Hence, since at least one continuation of a recursive application of `fibonacci` is not the identity monad, the execution of fibonacci results in a recursive process.

Define a monad, `fib_work`, to be the number of times `fibonacci` is applied to evaluate `fibonacci`. `fib_work` is, itself, a fibonacci sequence generated by the J definition:

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

Applying `fib_work` to the integers 0 through 10 gives:

```
   fib_work "0 i.11
1 1 3 5 9 15 25 41 67 109 177
```

3.1.1 Experimentation

Consider the experiment of estimating how long it would take to evaluate `fibonacci` on a workstation. First evaluate `fib_work 100`. Since the definition given above results in a recursive process, it is necessary to create a definition which results in an iterative process when evaluated. Consider the following definitions:

```
fib_work_iter =: monad def 'fib_iter 1 1 , y.'

fib_iter =: monad define
('a' ; 'b' ; 'count') =. y.
if. count = 0
```

```
  do. b
  else. fib_iter (1 + a + b) , a , count - 1
end.
)
```

Applying `fib_work_iter` to the integers 0 through 10 gives the same result as applying `fib_work`:

```
   fib_work_iter "0 i. 11
1 1 3 5 9 15 25 41 67 109 177
```

Next, use `fib_work_iter` to compute `fib_work 100` (exactly).

```
   fib_iter 100x
57887932245395525494200
```

Finally, time the recursive `fibonacci` definition on arguments not much larger than 20 to get an estimate of the number of applications/sec the workstation can perform.

```
   (fib_work_iter ("0) 20 21 22 23) % time'fibonacci ("0) 20 21 22 23'
845.138 1367.49 2212.66 3580.19
```

Using 3500 applications/sec as an estimate we have:

```
   0 3500 #: 57887932245395525494200x
16539409212970150141 700
   0 100 365 24 60 60 #: 16539409212970150141x
5244612256 77 234 16 49 1
```

which is (approximately) 5244612256 centuries!

An alternate experimental approach to solve this problem is to time the recursive `fibonacci` definition and look for patterns in the ratios of successive times.

```
   experiment =: (4 10 $'fibonacci ') ,. ": 4 1 $ 20 21 22 23
   experiment
fibonacci 20
fibonacci 21
fibonacci 22
fibonacci 23
   t =: time "1 experiment
   t
2.75291 4.42869 7.15818 11.5908
   (1 }. t) % _1 }. t
1.60873 1.61632 1.61924
   ratio =: (+/ % #) (1 }. t) % _1 }. t
   ratio
1.61476
   0 100 365 24 60 60 rep x: ratio^100
205174677357 86 306 9 14 40
```

This experimental approach produces a somewhat larger estimate of more than 205174677357 centuries. Students should be cautioned about certain flaws in either experimental design.

3.2 Computer Arithmetic

Arithmetic representations are easily modeled. Blaauw[Blaa 1976], one of the designers of the IBM Stretch and System/360 computers, recognized the importance of software modeling in the design process. Blaauw used *APL* for his software models. Following are models of IEEE 754 single precision inexact representations. `fs2bin` gives the binary representation of an inexact value.

```
fs2bin =: monad define
NB. check for infinities and nan's
if. _ = y.
   do. 0 1 1 1 1 1 1 1 1 , 23 copy 0
       return.
  elseif. __ = y.
   do. 1 1 1 1 1 1 1 1 1 , 23 copy 0
       return.
NB. this case is tricky since (for some reason)
NB. _. = _. is false.  Perhaps this is because
NB. IEEE 754 specifies _1 + 2 ^ 23 different
NB. representations for nan.
  elseif. 0 not_equal y. - y.
   do. 0 1 1 1 1 1 1 1 1 , 23 copy 1
       return.
end.
NB. compute the characteristic
e =. 1 + floor log2 | y. + y. = 0
NB. now the mantissa
f =. 1 drop (24 copy 2) rep floor (2 ^ 24 - e) * | y.
NB. finally get the sign and exponent (binary form)
se =. (9 copy 2) rep (y. not_equal 0) * (256 * y. < 0) + e + 126
se , f
)

bin2fs =: monad define
s =. 0 from y.
e =. base 1 2 3 4 5 6 7 8 from y.
f =. base 9 drop y.
if. (0 = e) and 0 = f
  do. 0
elseif. (255 = e) and 0 = f
  do. _ * _1 ^ s
elseif. (255 = e) and 0 not_equal f
  do. _.
elseif. (0 = e) and 0 not_equal f
  do. (_1 ^ s) * f * 2 ^ e - 126
elseif. 1
  do. (_1 ^ s) * (2 ^ e - 127) * (base 1 , 9 drop y.) % 2 ^ 23
end.
)

   fs2bin 0.01
```

```
0 0 1 1 1 1 0 0 0 0 1 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 0 0 1 0 1 0
   bin2fs 0 0 1 1 1 1 0 0 0 0 1 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 0 0 1 0 1 0
0.01
```

3.3 Computer Circuits

```
or =: +.
and =: *.
not =: -.
bitOr =: monad define
('a' ; 'b') =. y.
a or b
)
bitAnd =: monad define
('a' ; 'b') =. y.
a and b
)
bitNot =: not
bitHalfAdder =: monad define
('a' ; 'b') =. y.
bitOr (bitAnd a , bitNot b) , bitAnd (bitNot a) , b
)
bitXor =: bitHalfAdder
wireOutput =: monad define
('pin' ; 'outputs') =. y.
pin from outputs
)

bitAdder =: monad define
('a' ; 'b' ; 'cin') =. y.
t =. bitHalfAdder a , b
g =. bitAnd a , b
p =. bitAnd t , cin
(bitOr g , p) , bitHalfAdder t , cin
)

fourBitAlu =: monad define
('a3' ; 'a2' ; 'a1' ; 'a0' ; 'b3' ; 'b2' ; 'b1' ; 'b0' ; 'sub') =. y.
t0 =. bitAdder a0 , (bitXor b0 , sub) , sub
t1 =. bitAdder a1 , (bitXor b1 , sub) , wireOutput 0 ; t0
t2 =. bitAdder a2 , (bitXor b2 , sub) , wireOutput 0 ; t1
t3 =. bitAdder a3 , (bitXor b3 , sub) , wireOutput 0 ; t2
(wireOutput 0 1 ; t3) , (wireOutput 1 ; t2) , (wireOutput 1 ; t1) , wireOutput 1 ; t0
)
```

Below we show the sum and difference of `_1` and `1` (ignoring the carry) producing results of `0 0 0 0` and `1 1 1 0` (0 and _2).

```
   fourBitAlu 1 1 1 1  0 0 0 1 0
1 0 0 0 0
```

```
   fourBitAlu 1 1 1 1  0 0 0 1 1
1 1 1 1 0
```

3.4 Computer Organization

Following are J models of the multiplier discussed in Section 4.6 of Patterson and Hennessy [Patt 1998]. First, an architectural model of a 32-bit ALU.

```
alu_32 =: monad define
NB. a and b are each 32-bit summands
NB. the result is a 32-bit sum
('a' ; 'b') =. y.
(32#2) #: (#. x: a) + #. x: b
)
```

Next, the multiplier:

```
mult3 =: monad define
('multiplicand' ; 'multiplier') =. y.
product=. (32#0) , multiplier
count=.32
while. 0 ~: count
  do. control =. (_1&{) product
      if. control
        do. product =. (alu_32 (32 {. product) ; multiplicand) , 32 }. product
        end.
      product =. 0 , _1 }. product
      count =. <: count
  end.
product
)
```

Finally, we use the multiplier to form the product 3 * 2.

```
   mult3 ((30#0), 1 1); (30#0),1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
```

3.5 Object Programming

After taking a course in object programming, students sometimes lack a conceptual understanding of object programming which is independent from the syntax of the object programming language used in the course. The following model has been used to teach object programming concepts. J locales are used to build objects which combine both data structure and code. The J object programming system has a similar conceptual basis.

```
make_z_ =: 0 !: 0 @ <
for_effect_only_z_ =: monad def '''unspecified'''
invalid_method_name_indicator_z_ =: 'unknown'
root_object_z_ =: monad define
```

```
('method' ; 'value') =. 2 take y.
if. method -: 'type'
  do. 'root object'
  else. 'In root object: ',method,': Invalid method name.'
end.
)
```

The following file is a definition of a stack object which inherits its printing method from a printing object.

```
NB. The stack object template which inherits a print method

data =: ''

stack_z_ =: monad def 0
('method' ; 'value') =. 2 take y.
if. method match 'type'
    do. 'stack'
  elseif. method match 'emptyp'
    do. 0 = tally data
  elseif. method match 'push'
    do. for_effect_only data =: (box value) , data
  elseif. method match 'top'
    do. if. 0 = tally data
          do. 'top:  stack is empty'
          else. open first data
        end.
  elseif. method match 'pop'
    do. if. 0 = tally data
          do. 'pop:  stack is empty'
          else. for_effect_only data =: rest data
        end.
  elseif. method match 'size'
    do. tally data
  elseif. method match 'print'
    do. if. 0 = tally data
          do. 'print: stack is empty'
          else. for_effect_only display 'top of stack'
                print 'print' ; box data
        end.
  elseif. 1
    do. root_object method ; value
end.
)
```

The Printing object:

```
NB. The stack and queue print object

print_z_ =: monad define
```

```
('method' ; 'value') =. 2 take y.
if. method match 'type'
    do. 'print'
  elseif. method match 'print'
    do.
    while. 0 < tally value
      do. for_effect_only display open first value
          value =. rest value
    end.
  elseif. 1
    do. base method ; value
end.
)
```

Following is an interactive session which makes a stack object, named s, and then pushes and pops items on the stack.

```
   make_s_ 'stack1.object.ijs'
   stack_s_ <'type'
stack
   stack_s_ <'size'
0
   stack_s_ <'age'
In root object: age: Invalid method name.
   stack_s_ 'push' ; i. 10
unspecified
   stack_s_ <'size'
1
   stack_s_ 'push' ; 'Some text'
unspecified
   stack_s_ <'size'
2
   stack_s_ <'top'
Some text
   stack_s_ <'print'
top of stack
Some text
0 1 2 3 4 5 6 7 8 9
   stack_s_ <'pop'
unspecified
   stack_s_ <'top'
0 1 2 3 4 5 6 7 8 9
```

3.6 Computer Graphics

The J programming language provides several facilities for various kinds of graphics programming including an interface to the OpenGL libraries if available on the host system. Models can be built for a variety of graphics topics. Included are models of the 2D transformations (using homogeneous coordinates) scale, rotate and translate.

```
mat_product =: +/ . *
scale =: monad def '3 3 reshape (0 from y.), 0 0 0 , (1 from y.), 0 0 0 1'
translate =: monad def '3 3 reshape 1 0 0  0 1 0  , y. , 1'
rotate =: monad def '((2 2 reshape 1 1 _1 1 * 2 1 1 2 o. (o. y.) % 180),.0),0 0 1'

   NB. A square data object
   square =: 5 2 $ 0 0 10 0 10 10 0 10 0 0
   square
 0  0
10  0
10 10
 0 10
 0  0
   translate 10 _10
 1   0 0
 0   1 0
10 _10 1
   (square,.1) mat_product translate 10 _10
10 _10 1
20 _10 1
20   0 1
10   0 1
10 _10 1
   NB. Don't do unnecessary multiplications
   (square,.1) mat_product 3 2 {. translate 10 _10
10 _10
20 _10
20   0
10   0
10 _10
   rotate 180
_1  0 0
 0 _1 0
 0  0 1
   (square,.1) mat_product 3 2 {. rotate 180
  0   0
_10   0
_10 _10
  0 _10
  0   0
   new_square =: (square,.1) mat_product 3 2 {. translate 10 10
   new_square
10 10
20 10
20 20
10 20
10 10
   NB. Rotate this square 90 degrees about the point 10 10
   xform =: (translate _10 _10)mat_product (rotate 90) mat_product translate 10 10
   xform
 0 1 0
_1 0 0
20 0 1
   (new_square,. 1) mat_product 3 2 {. xform
10 10
10 20
 0 20
 0 10
10 10
```

3.7 Computer Networking

The final example is a model of the crc16 which is often built into data communications hardware.

```
crc16 =. monad define
bcc =. 16 $ 0
while. 0 ~: $ y. do.
  ser_quo =. (1 {. bcc) ~: 1 {. y.
  bcc =. (0 { bcc),(ser_quo ~: 1 { bcc),(2 3 4 5 6 7 8 9 10 11 12 13 { bcc),(ser_quo ~: 14 { bcc),15 {
  bcc =. (1 }. bcc), ser_quo
  y. =. 1 }. y.
end.
bcc
)

   msg =: 1 1 0 1 0 1 1 0 1 1 0 0 0 0
   crc16 msg
0 0 1 1 1 1 0 1 1 0 1 0 0 0 1 1
      crc16 msg,crc16 msg
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

4 Conclusions

The author has used J model building in most computer science courses taught during the last ten years. J has proved invaluable in aiding student comprehension of difficult material as well as providing many opportunities for laboratory experimentation which otherwise would not have been possible.

The author has tried to use, with some success, J language based models in courses where students have no prior experience reading or writing J. It has been possible to use J with little formal J instruction beyond learning the right-to-left evaluation rule. Since the notation may be used interactively, students can begin using models provided by the instructor to perform experiments before they have acquired sufficient J reading skills so that the models provide an expository function.

As an added bonus, J Software, Incorporated, now makes the full J system, on-line documentation and HTML versions of three books, *J Dictionary*, *J User Manual* and *J Phrases* freely available to anyone who wishes to access the J Web site, http://www.jsoftware.com/ .

References

[Berry 1970] Berry, P. C., Falkoff, A. D., Iverson, K. E., "Using the Computer to Compute: A Direct but Neglected Approach to Teaching Mathematics", Technical Report Number 320-2988, IBM New York Scientific Center, May 1970.

[Blaa 1976] Blaauw, Gerrit, *Digital System Implementation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[Burk 2001] Burke, Chris, *J User Manual*, J Software, Toronto, Canada, May 2001.

[Bur 2001] Burke, Chris, Hui, Roger K. W., Iverson, Kenneth E., McDonnell, Eugene, E., McIntyre, Donald B., *J Phrases*, J Software, Toronto, Canada, March 2001.

[Patt 1998] Patterson, David A. and Hennessy, John L., *Computer Organization & Design, The Hardware Software Interface*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998.

[How 1998] Howland, John E.," Recursion, Iteration and Functional Languages", Journal for Computing in Small Colleges, Volume 13, Number 4, April, 1998.

[Hui 2001] Hui, Roger K. W., Iverson, Kenneth E., *J Dictionary*, J Software, Toronto, Canada, May 2001.