

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@Ariel.CS.Trinity.Edu
Web: <http://www.cs.trinity.edu/~jhowland/>



Trinity University Computer Science

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.



Overview of Presentation

- Define Software Modeling
- Criteria for a Software Modeling Language
- J as a Software Modeling Language
- Examples
- Conclusions



Software Modeling

The term *modeling* is used in the context of software modeling, as in models of computer science principles implemented as programs (or program fragments) in some programming language.



Software Modeling

Successful model building depends on:

- Choice of Programming Language
- 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.



Attributes of a Good Modeling Language

- 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 Execution Environment
- Freely Available on a Variety of Computing Systems



Software Modeling Language

A software modeling language needs an extensive function repertoire.

- Classical Functions of Mathematics
- Functions to Create and Manipulate Data Structures
- Exact Arithmetic
- Functions are First Class



Software Modeling Language

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



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.



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.



Experimentation with Models

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



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 for a modeling language.

- J is a functional language using infix notation.
- Primitive functions are 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).



J Vocabulary

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
~ <i>Reflex</i> * <i>Passive</i> / 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
/ <i>Insert</i> * <i>Table</i>	/ . <i>Oblique</i> * <i>Key</i>	/ : Grade Up * Sort
\ <i>Prefix</i> * <i>Infix</i>	\ . <i>Suffix</i> * <i>Outfix</i>	\ : Grade Down * Sort

Figure 1: J Vocabulary, Part 1



[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)	@. Agenda	` : Evoke Gerund
@ Atop	&. Under (Dual)	@ : At
& Bond / Compose	? . Roll * Deal (fixed seed)	& : Appose
? Roll * Deal		
a. <i>Alphabet</i>	a: <i>Ace</i> (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. γ . Explicit Arguments
x: Extended Precision	_9: to 9: Constant Functions	

Figure 2: J Vocabulary, Part 2



Evaluation of J Sentences

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.



Evaluation of J Sentences

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).

- Example: The adverb insert (/) produces a verb which inserts the adverb's argument between the items of the derived verb's argument (+/ produces a verb which sums the items of its argument).
- 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).



Tacit Programs

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*.
- Example: `+/ % #` is a fork which computes the average.



Functional Programs

- Items, once created from notation for constants or function application, are never altered.
- Items possess attributes such as *type* (number 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.



Example Models

A few sample models used in the teaching of various computer science topics are given in the following slides. 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



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
```



Fibonacci Analysis

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.



Fibonacci Analysis

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:



Fibonacci Analysis

```
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
```



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.  
)
```



Experimentation

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
```



Experimentation

Next, 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
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!



Experimentation

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
```



Experimentation

```
(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.



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.



Conclusions

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.
-

