Ensuring a Rigorous Curriculum: Practices and Goals

Allen B. Tucker
Bowdoin College

www.bowdoin.edu/~allen
April 12, 2002

Goals of an Undergraduate Program

To prepare graduates for the computing profession and for postgraduate study.

In the profession, they 1) develop reliable software and hardware, and 2) learn new methods and technologies.

How well do (we and) our graduates do?

Often not well: Only 9% of all IT projects are delivered on time and on budget [*IEEE Software*; April 1998].
And occasionally very poorly: In 1996, the Ariane 5 launcher crashed on take-off, at a cost of $500 million, due to insufficient software specifications [Meyer 1997].

... so maybe something's missing.

Practical Need for Rigor in the Curriculum

In practice, the use of formal methods leads to:

- Cost-effective software designs [King 2000]
- Code correctness
- Code readability and efficiency

Results of a Controlled Experiment [Sobol 2002]

- Elevator scheduling problem
- formal methods (6 teams) vs
  traditional design (13 teams)
- all 6 formal designs were correct for all test data;
  6 out of 13 traditional designs were correct
- formal designs' code was more compact
- formal designs' code was more readable

Computer science is a rigorous discipline:

E.g., [ACM/IEEE 2001, section 9.1.1 Mathematical Rigor]:

"*Mathematics techniques and formal mathematical reasoning are integral to most areas of computer science...*

... Given the pervasive role of mathematics within computer science, the CS curriculum must include mathematical concepts early and often."

"Rigor" = the careful, thorough, systematic, and precise process of developing correct, efficient, and robust solutions to computational problems.

How can we ensure rigor in the CS curriculum?

A *rigorous CS curriculum* meets the following goals:

Goal 1: ensure that students can *use precise mathematical ideas and notations* in all subject areas.

Goal 2: ensure that students can *use formal methods* in all *software designs*.

Goal 3: ensure that students can *demonstrate the correctness of their solutions to problems* in all subject areas.

How Can a Curriculum Meet these Goals?

Introduce rigor early
(give the *discrete mathematics course equal status with CS1*, and integrate its principles into CS1)

Use rigor in CS2
(confirm that *principles of mathematical logic are integral to good programming*)

Continue using rigor in every core and elective course
(*integrate the theory with the practice*)

Here are some examples...

Example 1: The CS1 Course - introduce rigor early

... the idea that logic is related to the implementation of programs.

```
boolean search (Argument x, List L) {
   int i = 1;
   while (¬ found(x) ∧ ¬exhausted(L))
     i = i + 1;
   //  found(x) ∨ exhausted(L)
   return found(x);
}
```

... the idea that correct programs are related to their specifications.

```
boolean search (Argument x, List L) {
   pre:  L = {e1, e2, ..., en} ∧ n ≥ 0
   int i = 1;
   while (¬ found(x) ∧ ¬exhausted(L))
     i = i + 1;
   //  found(x) ∨ exhausted(L)
   return found(x);
   post:  ∃i ∈ {1, ..., n}: x = ei ∧ found(x) ∨ ¬ found(x)
}
```

Example 2: The CS2 Course - integrate rigor

Principles:

- Data structure and class design: use formal specifications
- Large program design: use formal specifications
- Implementation: verify code against specifications

Design by contract [Meyer 1997]:

The relation between a software module and its clients is a *formal agreement*, each with rights and obligations that

- are explicitly stated (via *assertions* and *invariants*), and
- are supported by the language (e.g., JJ or Java 1.4).

Assertions can be written only from fluency with logic, so:

- *Discrete math must be a prerequisite*

E.g., specifying part of a Tic-Tac-Toe game:

The game is over if either the player has won or the board has no open cells.

$GameOver: Grid \times Char \to \mathbf{B}$

$GameOver(board, player) =$

$true$	if $Winner(board, player) \vee \forall i, j \in \{0, 1, 2\}: board_{ij} \neq$ OFF
$false$	otherwise

The player has won if the board contains three instances of the player in a row, a column, or a diagonal.

$Winner: Grid \times Char \to \mathbf{B}$

$Winner(board, player) =$

$(\exists i \in \{0, 1, 2\}:$

$(\forall j \in \{0, 1, 2\}: board_{ij} = player \vee \forall j \in \{0, 1, 2\}: board_{ji} = player)) \vee$

$board_{00} = board_{11} = board_{22} = player \vee$

$board_{20} = board_{11} = board_{02} = player$

... and writing the code:

$GameOver(Grid\ board, Char\ player) \leftarrow$

if $Winner(board, player)$

$true$

else for $i \leftarrow 0$ to 2

if $board_{i0} = OFF \vee board_{i1} = OFF \vee board_{i2} = OFF$

$false$

$true$

$Winner(Grid\ board, Char\ player) \leftarrow$

for $i \leftarrow 0$ to 2

if $board_{i0} = board_{i1} = board_{i2} = player$

$true$

if $board_{0i} = board_{1i} = board_{2i} = player$

$true$

if $(board_{00} = board_{11} = board_{22} = player) \vee (board_{20} = board_{11} = board_{02} = player)$

$true$

$false$

... which would have looked like this in Java:

```
public boolean GameOver(Grid board, char player) {
  if (Winner(board, player))
    return true;
  for (int i=0; i<3; i++)
    if (board.get(i,0)==Cell.OFF ||
        board.get(i,1)==Cell.OFF || board.get(i,2)==Cell.OFF)
      return false;
  return true;
}
public boolean Winner(Grid board, char player) {
  for (int i=0; i<3; i++) {
   if (board.get(i,0)==board.get(i,1) &&
       board.get(i,1)==board.get(i,2) && board.get(i,0)==player)
      return true;
   if (board.get(0,i)==board.get(1,i) &&
       board.get(1,i)==board.get(2,i) && board.get(0,i)==player)
      return true;
  }
  if (board.get(0,0)==board.get(1,1) &&
      board.get(1,1)==board.get(2,2) && board.get(0,0)==player)
     return true;
  if (board.get(2,0)==board.get(1,1) &&
      board.get(1,1)==board.get(0,2) && board.get(2,0)==player)
     return true;
}
```

Example 3: The Programming Languages Course
[Tucker 2002]

Goals:

1. to teach principles of language design
   (Syntax, Type Systems, Semantics)
2. To introduce different paradigms
   (Functional, Object-Oriented, Logic)

These can be achieved by

- A mathematical treatment of the principles (BNF, Denotational Semantics), and
- Coordinated laboratory experiences (using an experimental language "Jay")

Students need to be fluent with functions, sets, and logic:
- *So discrete math must be a prerequisite*

Denotational Semantics of "Jay" Assignments

- Based on a formal abstract syntax,

  Assignment = *Variable* target; *Expression* source
  Expression = *Value* | *Variable* | *Binary*
  Value = *int* intValue | *boolean* boolValue
  Variable = *String* v
  Binary = *Operator* op; *Expression* term1, term2
  Operator = + | - | * | /

- a state $\sigma = \{\langle v_1, val_1\rangle, \langle v_2, val_2\rangle, \ldots, \langle v_n, val_n\rangle\}$,

- and a state-transforming (partial) function:

  M: *Expression* × *State* → *Value*

  M: *Assignment* × *State* → *State*

The meaning of *Assignment* and *Expression*

$M(Assignment\ \text{a}, State\ \sigma) = \sigma\ \bar{\cup}\{\langle \text{a.target}, M(\text{a.source}, \sigma)\rangle\}$

$M(Expression\ \text{e}, State\ \sigma)$

$= e$ if e is a Value

$= \sigma(e)$ if e is a Variable

$= Apply(\text{e.op}, M(\text{e.term1}, \sigma), M(\text{e.term2}, \sigma))$ if e is a Binary

$Apply(Operator\ op, Value\ v1, Value\ v2)$

$= v1 + v2$ if op = +

$= v1 \times v2$ if op = *

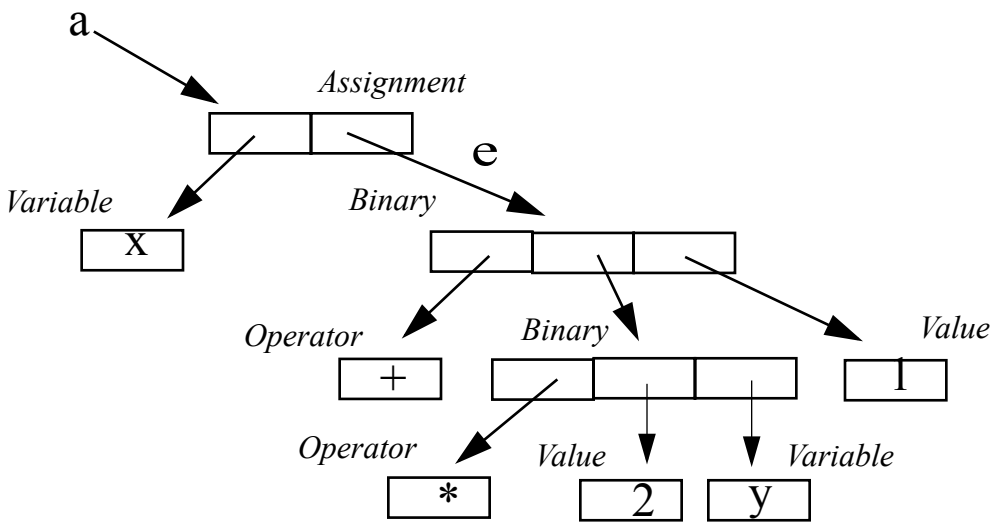
Note: $\sigma_1 \bar{\cup} \sigma_2$ creates a new state σ by replacing every pair $\langle v, val_1\rangle$ in σ_1 for which $v \in \sigma_2$ by $\langle v, val_2\rangle$.

So if state $\sigma = \{\langle x, 1\rangle, \langle y, 5\rangle\}$ then $\sigma\ \bar{\cup}\{\langle x, 11\rangle\} = \{\langle x, 11\rangle, \langle y, 5\rangle\}$

For example, if $\sigma = \{\langle x, 1\rangle, \langle y, 5\rangle\}$, the meaning of the *Assignment* x=2*y+1 is:

$$
\begin{aligned}
M(\text{x=2*y+1}, \sigma) &= \sigma \,\overline{\cup}\{\langle \text{a.target}, M(\text{a.source}, \sigma)\rangle\} \\
&= \sigma \,\overline{\cup}\{\langle x, M(\text{2*y+1}, \sigma)\rangle\} \\
&= \sigma \,\overline{\cup}\{\langle x, \mathit{Apply}(+, M(\text{2*y}, \sigma), M(1, \sigma))\rangle\} \\
&= \sigma \,\overline{\cup}\{\langle x, \mathit{Apply}(+, \mathit{Apply}(*, M(2, \sigma), M(y, \sigma)), M(1, \sigma))\rangle\} \\
&= \ldots = \{\langle x, 1\rangle, \langle y, 5\rangle\} \,\overline{\cup}\{\langle x, 11\rangle\} = \{\langle x, 11\rangle, \langle y, 5\rangle\}
\end{aligned}
$$

Abstract syntax is modeled using Java classes



So for the *Assignment* a = "x = 2*y+1" :

a.target = *Variable* x

a.source = *Expression* e = 2*y+1

e.op = + e.term1 = *Binary* 2*y e.term2 = *Value* 1

...

Laboratory Opportunities

- abstract syntax of Jay => Java classes
- Semantic functions of Jay => Java methods
- OO programming => integrating semantic functions into abstract classes
- Functional programming => Scheme implementation of semantic functions
- Logic programming => Prolog implementation of semantic functions

Students master each paradigm by revisiting the same problem: formal semantics of “Jay.”

In the Java Implementation of Jay semantics:

- the state σ of a program is a `Hashtable`.
- the expression $\sigma(v)$, denoting the value of variable v in state σ, uses `sigma.get(v)`
- implementing functions M follow the math:

```
Value M (Expression e, State sigma) {
    if (e instanceof Value)
       return (Value)e;
    if (e instanceof Variable)
         return (Value)(sigma.get((Variable)e));
    if (e instanceof Binary)
       return apply(((Binary)e).op,
                    M(((Binary)e).term1, sigma),
                    M(((Binary)e).term2, sigma));
    return null;
}
```

The Functional Paradigm

A Jay state is naturally modeled in Scheme as a list of pairs.

E.g., $\{\langle x, 1\rangle, \langle y, 5\rangle\}$ is modeled by `((x 1) (y 5))`

The state access function `get` is:

```
(define (get id sigma)
   (if (equal? id (caar sigma))) (cadar sigma)
       (get  id  (cdr sigma))
))
```

The list representation for expressions is:

```
;;; (value number)
;;; (variable  ident)
;;; (operator term1  term2)
;;;    where operator is one of: plus times
```

The meaning of a Jay abstract expression is:

```
(define (m-expression expr sigma)
  (case (car expr)
     ((value) (cadr expr))
     ((variable) (get (cadr expr) sigma)
   (else (apply (car expr) (cadr expr)
                (caddr expr) sigma))
))
```

The Scheme function `apply` is:

```
(define (apply op term1 term2 sigma)
  (case op
     ((plus)  (+ (m-expression term1 sigma)
                 (m-expression term2 sigma)))
     ((times) (* (m-expression term1 sigma)
                 (m-expression term2 sigma)))
      (else #f)
))
```

The Logic Paradigm (Prolog)

The Jay state $\{\langle x, 1\rangle, \langle y, 5\rangle\}$ is represented by:

```
[[x,1], [y,5]]
```

The state access function `get` is recursive:

```
/* get(var, inState, outValue) */
get(Var, [[Var, Val] | _], Val).
get(Var, [_ | Rest], Val) :- get(Var, Rest, Val).
```

The meaning of a Jay *Expression* uses the following representational convention:

```
value(number)
variable(ident)
operator(term1, term2)
```

where `operator` is one of: `plus times`

The meaning of a Jay *Expression* in Prolog:

```
/* mexpression(expr, state, val) */
mexpression(value(Val), _, Val).
mexpression(variable(Var), State, Val) :-
   get(Var, State, Val).
mexpression(plus(Expr1, Expr2), State, Val) :-
   mexpression(Expr1, State, Val1),
   mexpression(Expr2, State, Val2),
   Val is Val1 + Val2.
```

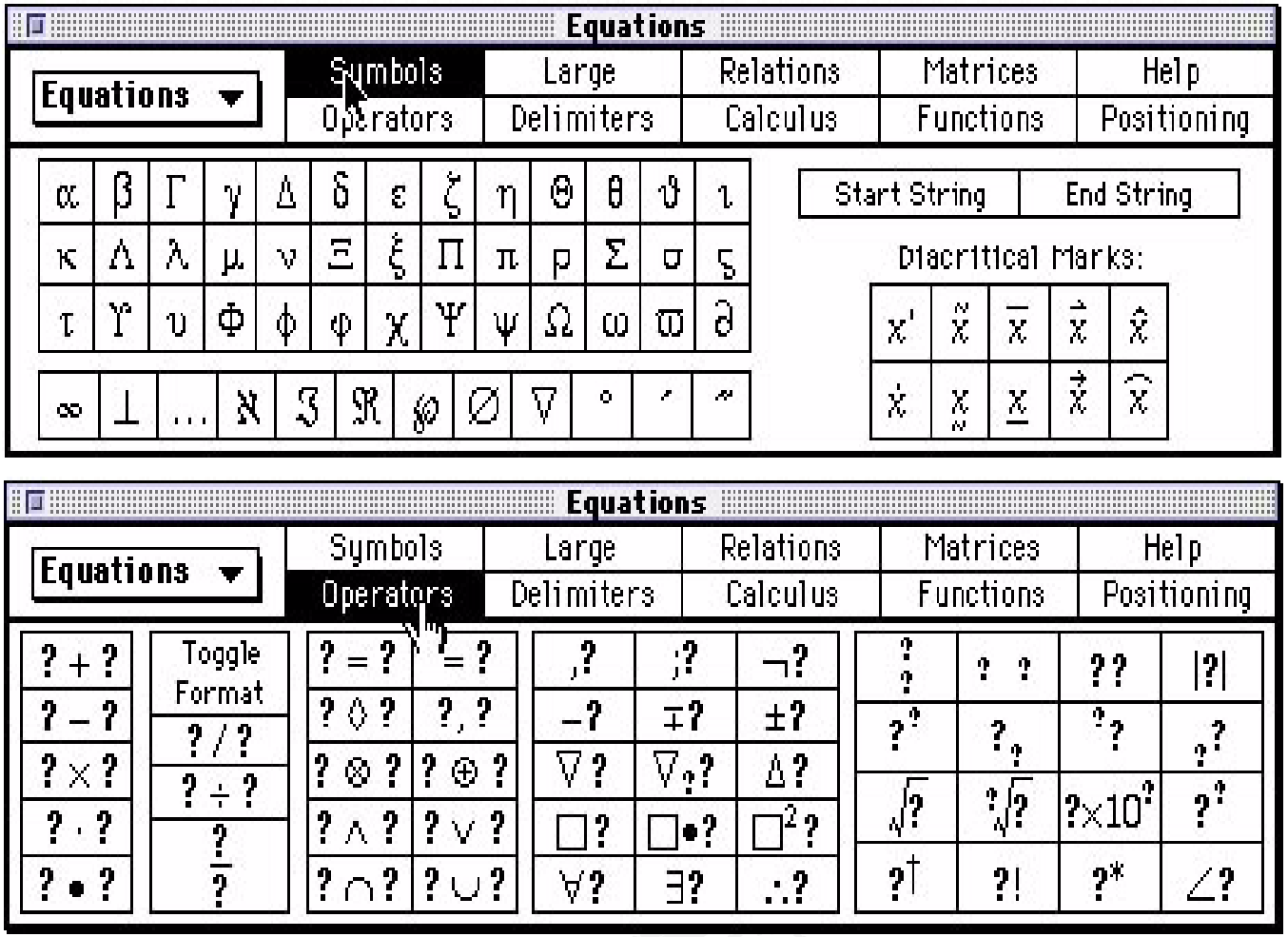
Weaknesses of Current Programming Languages

1. *incomplete character set (the "ASCII bottleneck")* e.g., && for logical conjunction, == for equality
2. *linear style of expression* e.g., `board.get(i,j)` for $board_{ij}$
3. *absence of mathematical notation* e.g., no quantifiers
4. *absence of syntactic and semantic uniformity* e.g., == and `equals` both denote "equality."
5. *limited and overspecified control structures* e.g., forcing the use of `for` statements to ask "Is there any row, column, or diagonal on the board that contains three instances of `player`?"
6. *nonsupport for formal verification* e.g., forcing the use of comments to specify pre- and post-conditions (Java 1.4 now supports these, finally!!!).
7. *absence of integration between code and commentary* e.g., there's no verifiable semantic relation between the comments and the program itself.

A Rigorous Programming Language will:

1. Break the “ASCII Bottleneck”
2. Support a “publication language”
3. Support a rich variety of expressions
4. Enforce a more literate style
5. Unite comments, specifications, and code

Breaking the ASCII bottleneck



Defining a Publication Language [*CACM* 1969 p 562]

procedure *Ising(n,x,t,S)*; **integer** *n, x, t*; **integer array** *S*;

comment *Ising* generates *n*-sequences $(s_1, ..., s_n)$ of zeros and ones where

$x = \sum_{i=1}^{n} s_i$ and $t = \sum_{i=1}^{n-1} |s_{i+1} - s_i|$ are given.

...
begin
  integer *k*; **integer array** *L*, *M*[1 : $t \div 2 + 1$];
  ...
end *Ising*

Enforcing a More Literate Coding Style

$Winner(Grid\ board, Char\ player) \leftarrow$

for $i \leftarrow 0$ to 2

if $board_{i0} = board_{i1} = board_{i2} = player$

true

if $board_{0i} = board_{1i} = board_{2i} = player$

true

if $(board_{00} = board_{11} = board_{22} = player) \vee (board_{20} = board_{11} = board_{02} = player)$

true

false

Uniting Comments, Specifications, and Code

The game is over if either the player has won or the board has no open cells.

$GameOver: Grid \times Char \rightarrow \mathbf{B}$

$GameOver(board, player) =$

$true$	if $Winner(board, player) \vee \forall i, j \in \{0, 1, 2\}: board_{ij} \neq$ OFF
$false$	otherwise

$GameOver(Grid\ board, Char\ player) \leftarrow$

if $Winner(board, player)$

  $true$

else for $i \leftarrow 0$ to 2

  if $board_{i0} = OFF \vee board_{i1} = OFF \vee board_{i2} = OFF$

    $false$

$true$

An Agenda for Curriculum and Language Development

Experiment outside the "current textbook" box

- Create new teaching materials that incorporate and integrate mathematical ideas
- Collaborate with others who are doing the same
- Get NSF and local support for curriculum development

Contribute to language design research

- Extend the alphabet and the primitive functions
- View programming as authorship rather than just coding

Continue to view mathematical thinking as a fundamental paradigm for computer science

References

[ACM/IEEE 2001] ACM/IEEE Joint Curriclum Task Force, *Computing Curricula 2001*, www.acm.org/sigcse/

[King 2000] King, S. et al. "Is Proof More Cost-Effective Than Testing?" IEEE Trans on Software Engineering 26, 8 (August 2000) 675-685.

[Meyer 1997] Meyer, B., *Object-Oriented Software Construction* 2e, Prentice-Hall, 1997.

[Naur 1962] Naur, P (ed), "Revised Report on the Algorithmic Language Algol 60," IFIP Press 1962.

[Sobol 2002] Sobel, A. and M. Clarkson. "Formal Methods Application: An Empirical Tale of Software Development. IEEE Trans on Software Engineering 28, 3 (March 2002) 308-320.

[Tucker 2002] Tucker, A. and R. Noonan, *Programming Languages: Principles and Paradigms*, McGraw-Hill, 411 pages. www.bowdoin.edu/~allen/pl/

[Walker 1996] Walker, H. and M. Schneider, "Revised Model Curriculum for a Liberal Arts Degree in Computer Science," *Communications of the ACM* 39,12 December 1996.