Brandeis University

Computer Science 21b: Structure and Interpretation of Computer Programs (Spring Term, 2016)

“There are only five ideas in Computer Science, and by the end of this course, you will know three of them.”

--Harry Mairson
<table>
<thead>
<tr>
<th>DATE</th>
<th>CLASS</th>
<th>RATING</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/6/11</td>
<td>COSI190</td>
<td>Average Quality</td>
<td>This class would be both excellent and challenging if the professor weren't a complete jerk. He tells you you won't be able to succeed on the first day of class. He is belittling and uninterested in students. While the lectures are quality, having to interact with this man will make you want to stab yourself in the eye.</td>
</tr>
<tr>
<td>12/14/10</td>
<td>CS21b</td>
<td>Average Quality</td>
<td>An &quot;old school&quot; teacher who needs to upgrade his lectures, which consisted of him verbally tracing lines and lines of code and cracking the occasional joke. Tests consisted of mostly esoteric problems - so do the work and know the material well!</td>
</tr>
<tr>
<td>7/11/10</td>
<td>CS21b</td>
<td>Good Quality</td>
<td>By far the best CS professor I had at Brandeis. Made what could be a dry topic clear and very interesting. He's clearly passionate about what he teaches, and it really makes a difference.</td>
</tr>
</tbody>
</table>
5 Change it up. It's very clear that each lecture is exactly the same as it has been for the past twenty years. Mix things up, make it more interesting and engaging. It's very interesting material, but the professor makes it quite boring. Also, the grading system is bizarre.
Software for Violin Makers

Posted on January 9, 2015 by admin

On January 8 and 9, computer science professor Harry Mairson ran a workshop for a group of luthiers (string instrument makers) on using software he is developing for the design and historical analysis of instruments from the violin family. His software is based on the extraordinary research and book of French luthier François Denis, “Traité de lutherie”, which explains the historical methods of design, founded on classical ideas from architecture and geometry in the tradition of Vitruvius, Alberti, and Palladio. Denis’s text is a tour de force synthesis of practical lutherie with the history and philosophy of both science and art.
From the course review archives...

15 Slower pace, less cultural stuff

7 On the first day the professor said the only way to teach this course is by reading lines and lines of code, but I find that very untrue. He could have spent more time doing live coding of the "lines and lines" of code, or somehow made it more digestible than how it currently was. The TAs were also not very helpful at times, and it seemed they didn't understand the problem sets either.

12 Ahh, Professor Marison. The degree to which I enjoyed lecture was completely contingent upon my mood. Some days I loved the degree of whimsy and distraction thrown into classes, and on other days, I simply wanted to be taught about Lisp variants. Time will tell whether I become one of the many students who idolize him for his passion for the subject material and his hilarious personality, or one of the others who finds his flippant about the content of lectures frustrating and unprofessional. Make no mistake. This course served its purpose, I have gained a superb understanding of Functional Programming, and feel that it has positively impacted my confidence and understanding of computer science as a whole. However, at times that process was frustrating.
Grades: How hard is this course?

2015
A+ A+ A+ A A A  
B+ B+ B+ B+ B+ B+ B+  
B- B- B- B- B- B- B- B- B- B- B- B-  
C+ C+ C+ C+ C+  
C C C C C  
D D

2014
A+ A A A A A A A- A- A- A-  
B- B- B- B- B- B- B- B- B- B- B- B- B- B- B- B-  
B- B- B- B-  
C C C C C C C C C  
D

2013
B+ B+ B B B B B- B- B- B-  
C+ C+ C+ C+ C C C C C C C C C-  

2012
A+ A A A A A- A-  
C C
Structure and Interpretation of Computer Programs

Structure: How we use a computer language to express (through complex interaction of programs and data) complicated "computational thoughts"---using "engineering design" principles to decompose large, incomprehensible systems into comprehensible and manageable parts.

Interpretation: How we develop a computational model for understanding what our programs mean -- what is the mechanism by which programs and data interact to invoke the computational process we desire to construct and control?
In other words:

First, we’re going to learn Scheme, a “really simple” programming language with almost no syntax, and precious little else, and what its programming idiom is---namely, how to structure our computational ideas.

Then we’re going to learn how to interpret it, and how to compile it.

To make the course self-contained, we’ll interpret and compile it in Scheme---though our last compiler will have all the essence of a machine language (i.e., restricted Scheme).
A credo (one, among many):

Programming languages don’t exist simply to tell machines what to do. Well recognized as the engineering vernacular of software, they are more importantly the collective mother tongue of algorithmic ideas.

This point of view is a virtual credo among computer scientists, and reiterated in public emphases on “computational thinking” as a way of understanding complex, constructional processes.

In other words, programming languages---a form of writing---provide a medium for us to express to each other what we know how to do.
Syntax versus semantics

What’s the difference between learning a foreign language (say, French) and learning a computer language (say, Lisp) for the first time?

Syntax: say something that’s meaningful.

> )3 square(
[error]

Semantics: say what you meant:

> (define (square x) (+ x x))

square
> (square 5)

10

NB: The read-eval-print loop...
Invoking Computational Processes: The Gospel According to Abelson and Sussman

How does a powerful programming language let us create processes, and also structure our computational ideas?

PRIMITIVE EXPRESSIONS represent the simplest entities -- our “atomic building blocks”
MEANS OF COMBINATION allow compound expressions to be built up from smaller ones
MEANS OF ABSTRACTION allow compound objects to be named and manipulated almost as if they were primitives

For CS29a ‘Discrete Math’ alumni:
inductive definition + naming
For CS29a ‘Discrete Math’ alumni: Naming and Necessity, or

*inductive definition* + *naming*

---

**Inductive definition of integers:**
0 is a number.
If x is a number, so is x+1.
That’s all.

---

A *googol* is $10^{100}$.
(That’s $0+1+1+1+...+1$ [10100 times]).
Names come in useful...

$10^{100} \approx (1 \cdot 0)^{1 \cdot (1 \cdot 0) \cdot 0}$ (or something like that...)
Computational processes...

...evolve from the interaction of **PROGRAMS** (rules for manipulating data) and **DATA** (objects to be manipulated)

We’ll see soon that the distinction between the two is a bit arbitrary -- in fact, they have a lot in common.

[ Masochist: “Hit me!”  //  Sadist: “No!” ]

So much for philosophy -- now for some real life...
Now **forget** about the “cash register” model of computing, and think instead about **evolving processes** -- or a mathematical model where a program is like a *math expression* that gets **simplified** according to some fixed set of rules...
Primitive expressions

> 325
325
> +
# [procedure +]
> coffee

Why italics?
What distinguishes 3 2 5 from 3 2 5?
...Mythology?
Philosophy?

> cookie

(The last examples represent an ongoing research project...)
1. "When they (my elders) named some object, and accordingly moved towards something, I saw this and I grasped that that the thing was called by the sound they uttered when they meant to point it out. Their intention was shown by their bodily movements, as it were the natural language of all peoples; the expression of the face, the play of the eyes, the movement of other parts of the body, and the tone of the voice which expresses our state of mind in seeking, having, rejecting, or avoiding something. Thus, as I heard words repeatedly used in their proper places in various sentences, I gradually learnt to understand what objects they signified; and after I had trained my mouth to form these signs, I used them to express my own desires."

These words, it seems to me, give us a particular picture of the essence of human language. It is this: the **individual words in language name objects**—sentences are combinations of such names.--In this picture of language we find the roots of the following idea: Every word has a meaning. The meaning is correlated with the word. It is the object for which the word stands...

Now think of the following use of language: I send someone shopping. I give him a slip marked 'five red apples'. He takes the slip to the shopkeeper, who opens the drawer marked 'apples', then he looks up the word 'red' in a table and finds a colour sample opposite it; then he says the series of cardinal numbers--I assume that he knows them by heart--up to the word 'five' and for each number he takes an apple of the same colour as the sample out of the drawer.--It is in this and similar ways that one operates with words--"But how does he know where and how he is to look up the word 'red' and what he is to do with the word 'five'?" --Well, I assume that he 'acts' as I have described. Explanations come to an end somewhere.--But what is the meaning of the word 'five'? --No such thing was in question here, only how the word ‘five’ is used.

*St. Augustine [3rd c.] (Confessions, I.8.), quoted at the beginning of Ludwig Wittgenstein’s *Philosophical Investigations.*
Compound expressions

> (+ 137 349)
486
> (* 5 99)
495
> (/ 10 5)
2
> (/ 10 6)
1.6666666667
> (* 25 4 12)
1200
> (+ (* 3 5) (- 10 6))
19
...

Note:

* Read-eval-print loop
* Parentheses not used for grouping (as in math), but to incur evaluation.
* Prefix operator syntax:
  (&lt;operator&gt; &lt;arg&gt; ... &lt;arg&gt;)
* Nesting of compound expressions.
Evaluating compound expressions

> \((+ (* 3 5) (- (/ 30 3) 6)))\)

19

1. Recursively evaluate each of the subexpressions (what are these?) -- the first is the \textit{operator}, the remainder are the \textit{operands}.

2. Apply the evaluated operator to the values given by evaluation of the operands.
Naming and the environment

> (define size 2)
size
> (define pi 3.14)
pi
> (* size pi)
6.28
> (define radius 10)
radius
> (define circumference
  (* 2 pi radius))
circumference
> circumference
62.8

Expressions are evaluated in the context of an environment: an association of name-thing pairs (not memory registers!)
Procedure definition

> (define (square x) (* x x))
   "to square something, multiply it by itself"

square
> (square 5)
25
> (square (* 3 (square 2)))
144
More procedure definitions...

> (define (average x y) (/ (+ x y) 2))
average
> (define (mean-squares x y)
   (average (square x) (square y)))
mean-squares
> (average (* 2 2) (square 5))
14.5

To apply a compound procedure:

1. **Evaluate** operator and operands (recursively);
2. **Substitute** the values of the arguments for the formal parameters in the body of the procedure;
3. **Evaluate (recursively)** the body of the procedure (with substitutions).
Applying a compound procedure: substitution model

> (define (square x) (* x x))
  square
> (define (average x y) (/ (+ x y) 2))
  average
> (average (* 2 2) (square 5))

(average 4 (* 5 5))
(average 4 25)
(/ (+ 4 25) 2)
(/ 29 2)
14.5

This is called the *applicative order* substitution model
(evaluate arguments before substitution)

-- *is there another way?*
Normal order evaluation
(Scheme doesn’t work this way!)

> (define (square x) (* x x))
square
> (define (average x y) (/ (+ x y) 2))
average
> (average (* 2 2) (square 5))

(/ (+ (* 2 2) (square 5)) 2)
(/ (+ 4 (* 5 5)) 2)
(/ (+ 4 25) 2)
(/ 29 2)
14.5

The *Church-Rosser theorem* says (subject to fine print!) that applicative order and normal order always give the same answer (we assume something like this all the time in mathematical calculations: “any” order is OK and produces the same answer...