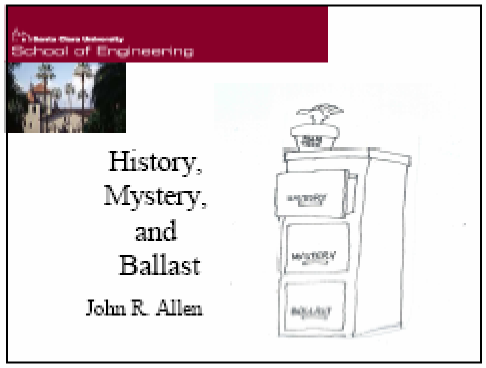
"More Ballast!"
The International Lisp Conference Keynote

For those of you who haven't read the paper, the title "History, Mystery, and Ballast" refers to the labels on Steve Russell's three-drawer file cabinet that I first saw here at Stanford in late 1964.

And a slight correction: the conference that Ruth Davis and I put together in 1980 was the first "Lisp and Functional Programming Conference"; we wanted a larger tent than just Lisp. As far as I know the first Lisp Conference was in Mexico City at the beginning of 1964. And actually JonL called it a "seminal" conference. I thought they were a native-American tribe in Florida.

I'm assuming that all of you —who wish to— can read the paper. The paper contains the words; today I'd like to explain a bit of the music behind those words, and so will proceed through most of the paper in a manner that Dana Scott calls "a hovercraft lecture," skimming over the surface at very high speed.

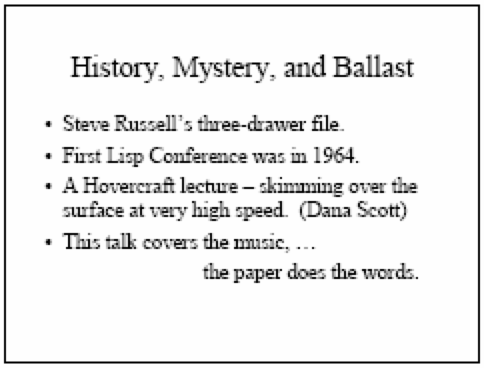


School of Engineering

History, Mystery, and Ballast

John R. Allen

HISTORY
MYSTERY
BALLAST



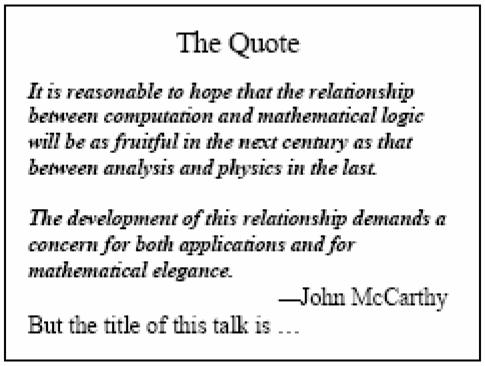
History, Mystery, and Ballast

- Steve Russell's three-drawer file.
- First Lisp Conference was in 1964.
- A Hovercraft lecture – skimming over the surface at very high speed. (Dana Scott)
- This talk covers the music, …
  the paper does the words.

The substantive part of the paper I'll deal with today is based on a quote from John McCarthy:

It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance.

A. This *is* the next century! and …
B. It will be *this* quote that drives the discussion.



The Quote

It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last.

The development of this relationship demands a concern for both applications and for mathematical elegance.

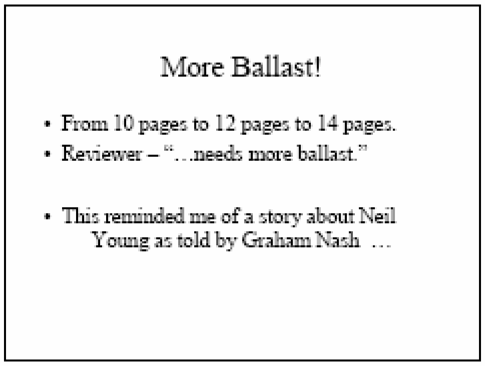
—John McCarthy

But the title of this talk is …

But the title of this talk is "More Ballast!"

It comes from a reviewer's comment about the paper. After sweating the paper into 12 pages, and getting a reluctant approval from the long-suffering JonL for the two-page overage, I noticed that the ACM template had supplied a 9-point font where JonL had asked for a minimum of 10-point. And changing the text from 9-point to 10-point added two *more* pages to the paper. Ugh. After a few tense days, the14-page version was accepted with a comment to the effect that it needed "more ballast." So that's what you get.

More Ballast!

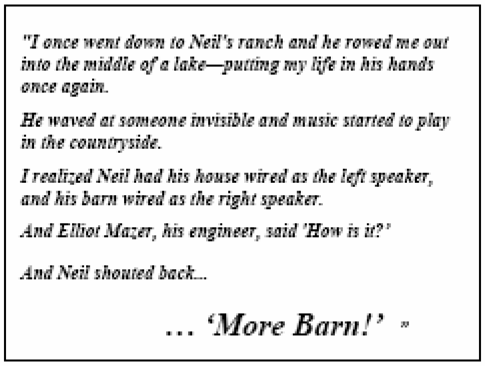


More Ballast!

- From 10 pages to 12 pages to 14 pages.
- Reviewer – "…needs more ballast."
- This reminded me of a story about Neil Young as told by Graham Nash …

But "More Ballast" reminded me of a story about Neil Young as told by Graham Nash:

"I once went down to Neil's ranch and he rowed me out into the middle of the lake—putting my life in his hands once again. He waved at someone invisible, and music started to play in the countryside. I realized Neil had his house wired as the left speaker, and his barn wired as the right speaker. And Elliot Mazer, his engineer, said 'How is it?' And Neil shouted back...
… 'More Barn!' "



"I once went down to Neil's ranch and he rowed me out into the middle of a lake—putting my life in his hands once again.

He waved at someone invisible and music started to play in the countryside.

I realized Neil had his house wired as the left speaker, and his barn wired as the right speaker.

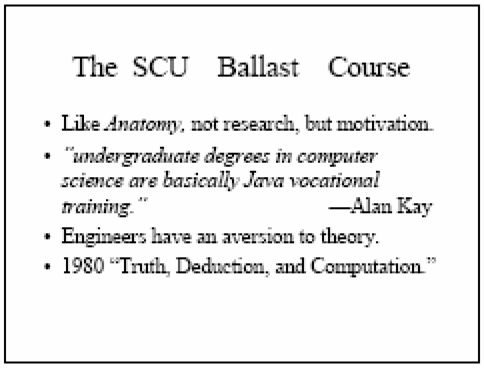
And Elliot Mazer, his engineer, said 'How is it?'

And Neil shouted back...

… 'More Barn!' "

Now a bit about the motivation for the paper and talk. To be clear, neither is research; both are motivation. Motivation in the sense that "Anatomy of Lisp" was motivation, not research. The modest goal in all three cases was (and is) to make some interesting and important ideas accessible to a wider audience.

Some years ago, a novelist wrote that the point of education was to teach kids not to eat spinach with their fingers. Far too many programmers think of spinach as finger-food. We want to change that.



The SCU Ballast Course

- Like *Anatomy*, not research, but motivation.
- *"undergraduate degrees in computer science are basically Java vocational training."* —Alan Kay
- Engineers have an aversion to theory.
- 1980 "Truth, Deduction, and Computation."

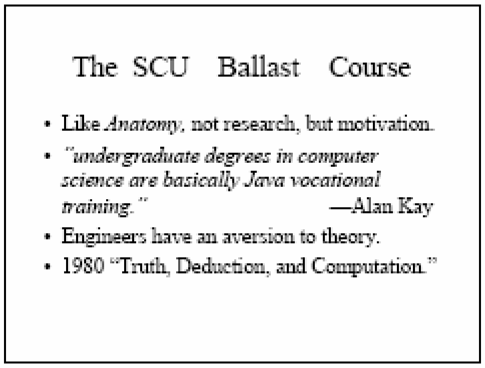
The particular case-in-point that motivated this outburst is the experience Ruth Davis and I have had in the Computer Engineering Department at Santa Clara University—specifically, the program in Software Engineering.

From what I read, the attitude at SCU isn't all that different from other universities. In fact, Alan Kay remarked "*undergraduate degrees in computer science are basically Java vocational training*." The only difference is that most of the people we see at SCU are graduate students wishing to upgrade their undergraduate vocational training.

But to some extent, this makes our situation worse. The SCU people have been making very good money—at least until recently—in an environment that typically pays little attention to a theory-based view of programming. And being an engineering environment, there's a high premium on demonstrable benefit when introducing new material. When theory is involved, the premium is particularly high.

The course that makes up the "Ballast" was begun by Ruth in 1980, and developed into her book "Truth, Deduction, and Computation" which was published in 1989 ... with her publishing party two days before the Loma Prieta Earthquake. I'm sure the juxtaposition of events was coincidental.

All of which brings me to the content of our SCU course.



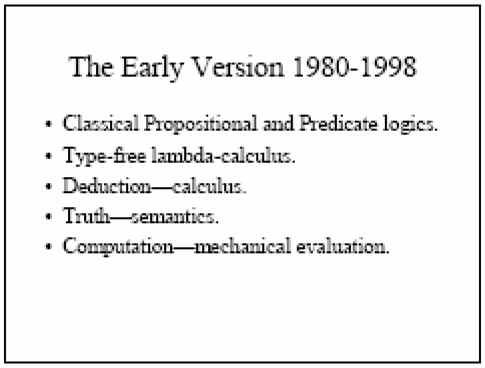
The SCU Ballast Course

- Like *Anatomy*, not research, but motivation.
- *"undergraduate degrees in computer science are basically Java vocational training."* —Alan Kay
- Engineers have an aversion to theory.
- 1980 "Truth, Deduction, and Computation."

The early incarnation *a la* 1980 investigated three formalisms —propositional, predicate, and functional languages—covering their deductive, computational, and semantic properties. Propositional and predicate calculi were done classically with Hilbert-style axioms and the usual truth table and Tarski semantics. Computation was introduced with ground- and general resolution.

The functional language of choice was the type-free lambda calculus, with Scott's first models for the truth discussion. Deduction was the typical lambda-calculus treatment—Church-Rosser, Normalization, etc., with computation based on reduction orders: applicative- and normal-order.

One striking impression remains from that period: my shock that the majority of students did not see the connection between the mechanisms of the lambda-calculus, and what they did in their work. This goes back to the "vocational training" mentioned by Alan Kay: for example, far too many of them did not see a relationship between "call-by-value" evaluation and applicative order reduction.



The Early Version 1980-1998

- Classical Propositional and Predicate logics.
- Type-free lambda-calculus.
- Deduction—calculus.
- Truth—semantics.
- Computation—mechanical evaluation.

The remodeled version began in 1998. It's still a "work in progress," but currently follows the same languages and divisions, but takes account of progress of the intervening 20 years.

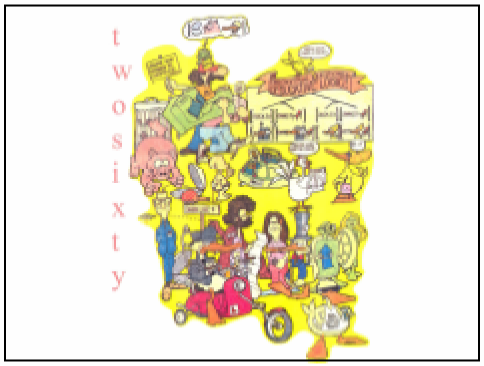
Since we believe that constructive logic is a better model for the logic of computation, constructive counterparts replaced the classical logics. In the deductive discussions we use Gentzen's natural deduction and sequent calculi, rather than Hilbert formulations. In the semantics discussions, we also bring the algebraic versions of semantics into play, using Boolean, cylindric, Heyting, and λ-algebras. Semantic Tableau is explored as well.

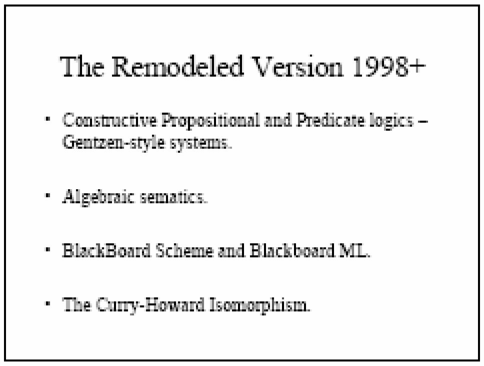
And since engineers, rather than mathematicians, are our audience, I recast the computational part from type-free lambda calculus to what I called BlackBoard Scheme—an M-expression dialect of Scheme, based on McCarthy's BlackBoard Lisp I'd seen him use some 30+ years ago. To that I added a simple, explicitly typed version of ML that I called BlackBoard ML. These two languages become an introduction to the more spartan lambda-calculi.

As to the earlier problem of the disconnect between the computational theory and the students' experience with programming languages, it's not yet clear that the new approach solves the problem.

But a clear benefit of the new approach is the possibility to call upon the Curry-Howard Isomorphism to ease the transition between logics and computation. We will discuss this in detail, but the essential idea is that logic and computation are two sides of the same coin. In the world of sensation and denotation, computation is the sensational aspect of the denotational object we call constructive proofs.

Sometimes we begin with computation and move to the logics; sometimes the other way 'round. We'll sort that out this year and get this thing published … and retire.

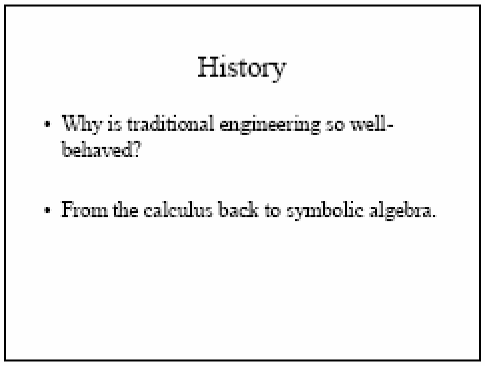




The Remodeled Version 1998+

- Constructive Propositional and Predicate logics – Gentzen-style systems.
- Algebraic sematics.
- BlackBoard Scheme and Blackboard ML.
- The Curry-Howard Isomorphism.

Finally, given the students' reluctance to accept theory as a viable part of "vocational training," it occurred to me to examine the history of traditional engineering. After all, those disciplines are theory- and science-based; and undergraduates in those disciplines accept the necessity of theoretical work. Could we not make the case that software was destined to follow suit? So it became important to me to motivate, motivate, and then motivate some more. Thus there is a front-loading in the course (and the paper) on the history of traditional engineering and the mathematics that supports it. If you can convince people of the efficacy of your quest and they will follow. In traditional engineering, engineers and architects are constrained by physics and became susceptible to science because they wanted their constructions to succeed. Though computation is not constrained by physics, it *is* constrained by logic … believe it or not.

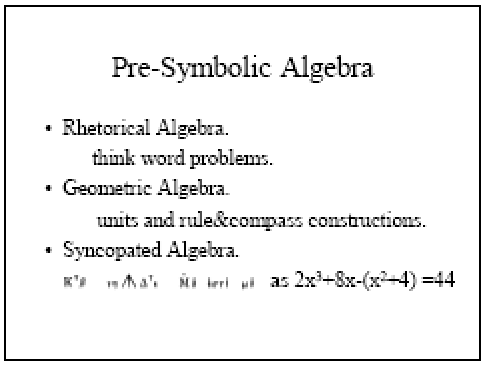


History

- Why is traditional engineering so well-behaved?
- From the calculus back to symbolic algebra.

Sometime around 2000, I began working back from engineering mathematics to the calculus, and somewhere in the process discovered that the symbolism that supported the calculus was—at the time—a recent invention. That was a surprise; I'd always believed that algebra was ancient. Not really true; "algebra" as some kind of generalized arithmetic was around, but even this was hampered by a lack of symbolic notation.

Until the end of the 16th century, there were three basic styles of algebraic language: geometric, syncopated, and rhetorical.



Pre-Symbolic Algebra

- Rhetorical Algebra.
  think word problems.
- Geometric Algebra.
  units and rule&compass constructions.
- Syncopated Algebra.
  [illegible] as $2x^3+8x-(x^2+4)=44$

Rhetorical algebra is what it sounds like: essentially what we call today "algebra word problems." The important distinction between then and now is that *we* recast the problem in symbolic notation and perform transformations on that symbolism. This path from "word problem" to symbolic language did not exist until the very late 16th century. Bummer.

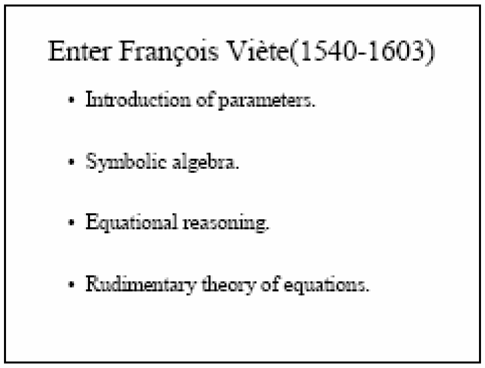
Geometric algebra involved solving arithmetic problems by manipulation of geometrical arguments—like ruler-and-compass. Thus the name "square" for what we write as a^2. Obviously a^2 meant "a square with side of length a." Of course there is bleed-over between the geometric and rhetorical approaches. The notion of "units" can limit the number of solutions to a rhetorical question. Even after the rise of symbolic algebra, geometric interpretation—and issues of units, in general—held sway.

Syncopated algebra appears to be the product of Diophantus' fertile mind around the third century, AD. It was a very rudimentary symbolic form of algebra. For example, where we write $4x^3-6x^2 = 2x+3$ a syncopated scholar would write something like $K^{\gamma}4\Delta^{\gamma}6\iota\zeta 2M3$. This—like Roman numerals—is hardly amenable to manipulation.

The breakthrough came late in the 1500s. François Viète turned symbolic arithmetic into a true algebra in several ways.

Viète's most obvious innovation was his introduction of parameters; so instead of solving specific equations, his notation illustrated general solutions. And since the algebraic formulations involved symbolic combinations of parameters, he also had to express the transformations of these combinations as, what were essentially, rules for equational reasoning.

Now that the specifics were exposed, he could couch this work in a more general setting, exemplifying what was to become the paradigm of modern mathematical science and engineering.



Enter François Viète(1540-1603)

- Introduction of parameters.
- Symbolic algebra.
- Equational reasoning.
- Rudimentary theory of equations.

In Viète's words:

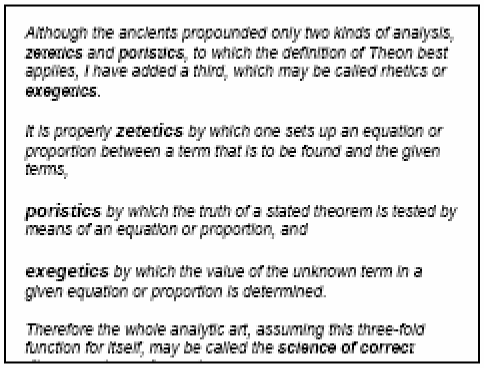
Although the ancients propounded only two kinds of analysis, zetetics and poristics, to which the definition of Theon best applies, I have added a third, which may be called rhetics or ***exegetics****.*

It is properly ***zetetics*** *by which one sets up an equation or proportion between a term that is to be found and the given terms,*

poristics *by which the truth of a stated theorem is tested by means of an equation or proportion, and*

exegetics *by which the value of the unknown term in a given equation or proportion is determined.*

Therefore the whole analytic art, assuming this three-fold function for itself, may be called the ***science of correct discovery in mathematics****.*



Although the ancients propounded only two kinds of analysis, ***zetetics*** *and* ***poristics****, to which the definition of Theon best applies, I have added a third, which may be called rhetics or* ***exegetics****.*

It is properly ***zetetics*** *by which one sets up an equation or proportion between a term that is to be found and the given terms,*

poristics *by which the truth of a stated theorem is tested by means of an equation or proportion, and*

exegetics *by which the value of the unknown term in a given equation or proportion is determined.*

Therefore the whole analytic art, assuming this three-fold function for itself, may be called the ***science of correct***

His use of "zetetics" is reasonably clear, but what of "poristics?"

He wrote little about poristics, but one definition of "porism" says it's a "a result that follows from the argument of a previous proposition;" or "an inference or deduction." Given that his topic was the manipulation of equations, I'm willing to tag poristics in this case as the transformations of an equation by equational logic.

Exegetics, as described by Viète seems pretty straightforward: the theory of equations. But dictionaries define "exegetics" as "the science of interpretation," and so I would stretch the Viète trio a bit further:

$$\begin{tikzcd}
\text{problem} \arrow[r, "\text{Zetetics}"'] \arrow[d, dashed] & \text{(symbolic algebra)} \arrow[d, "\text{Poristics (equational transformation)}"] \\
\text{solution} & \text{(symbolically isolated variable)} \arrow[l, "\text{Exegetics}"']
\end{tikzcd}$$

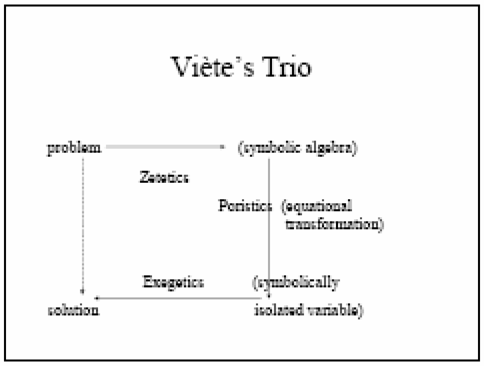
Viète's use of the words, zetetics and poristics, came from Pappus but in the process changed their meanings so I may do likewise:

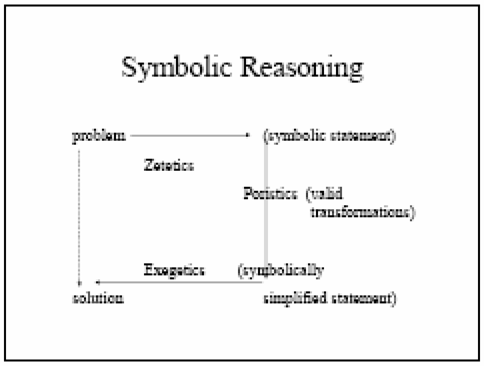
$$\begin{tikzcd}
\text{problem} \arrow[r, "\text{Zetetics}"'] \arrow[d, dashed] & \text{(symbolic statement)} \arrow[d, "\text{Poristics (valid transformations)}"] \\
\text{solution} & \text{(symbolically simplified statement)} \arrow[l, "\text{Exegetics}"']
\end{tikzcd}$$

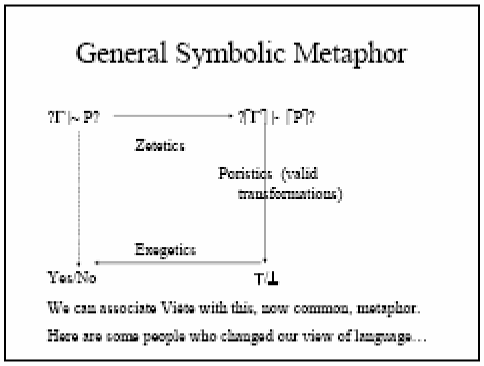
This diagram generalizes to the ubiquitous one that defines all modern symbolic investigations:

$$\begin{tikzcd}
?\ \Gamma \mid\!\sim P? \arrow[r, "\text{Zetetics}"'] \arrow[d, dashed] & ?\ulcorner\Gamma\urcorner \vdash \ulcorner P\urcorner? \arrow[d, "\text{Poristics (transformations by deduction or computation)}"] \\
\text{yes/no} & \top / \bot \arrow[l, "\text{Exegetics}"']
\end{tikzcd}$$

The bottom line: without Viète's work Newton and Leibniz would not have had the tools needed to express their calculi. Furthermore, Viète's analysis of zetetics, porsitics, and exegetics set the stage for generalized symbolic science.


Viète's Trio
problem
(symbolic algebra)
Zetetics
Poristics (equational transformation)
Exegetics
(symbolically isolated variable)
solution





General Symbolic Metaphor

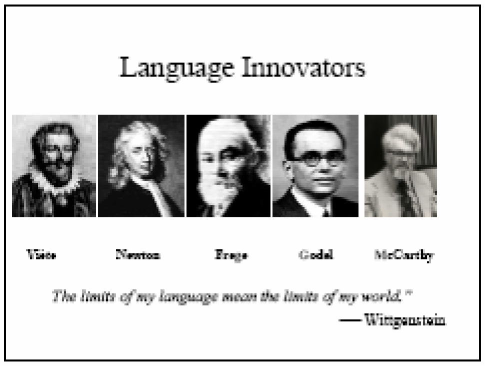
$$\begin{tikzcd}
?\Gamma \mid\sim P? \arrow[r, "\text{Zetetics}"'] \arrow[d, dashed] & ?\lceil\Gamma\rceil \vdash \lceil P\rceil? \arrow[d, "\text{Poristics (valid transformations)}"] \\
\text{Yes/No} & \top/\perp \arrow[l, "\text{Exegetics}"']
\end{tikzcd}$$

We can associate Viète with this, now common, metaphor.

Here are some people who changed our view of language…

This view of using symbol systems to model various realities is so in-grained now that it's taken for granted; it's nice to be able to attach a name and face to a major intellectual idea.

Here are some pictures of people who introduced major intellectual ideas in symbolic languages and their usage:



Language Innovators

The limits of my language mean the limits of my world."

—— Wittgenstein

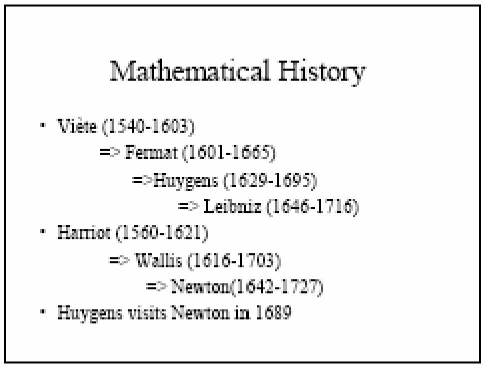
"The limit of my language means the limit of my world"

----Wittgenstein

Here's a little historical back-and-forth—the stuff that makes history intriguing; and why it's important we do something to preserve computational history—like "remembering" that the first PC was invented by IBM. **;-)**

The symbolic algebra of the Frenchman, Viète (1540-1603), was closely followed by that of England's Thomas Harriot (1560-1621). The work of Harriot and John Wallis (1616-1703) was read and appreciated by Newton (1642-1727) who published *Principia* in 1687.

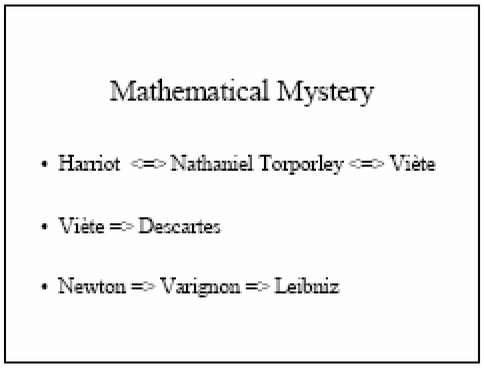
On the Continent, Viète's work strongly influenced Pierre Fermat (1601-1665). Fermat, in turn influenced Christiaan Huygens (1629-1695). Leibniz (1646-1716) studied with Huygens in Paris between 1672 and 1676. It's known that Leibniz' formulation of the calculus was well underway by 1675. And finally Huygens traveled to England in 1689 to meet Newton.



Mathematical History

- Viète (1540-1603)
  => Fermat (1601-1665)
  =>Huygens (1629-1695)
  => Leibniz (1646-1716)
- Harriot (1560-1621)
  => Wallis (1616-1703)
  => Newton(1642-1727)
- Huygens visits Newton in 1689

But there's a more intriguing part: the man named Nathaniel Torporley. We discovered that Harriot and Torporley were both at Oxford at the same time. Torporley, described as "a friend and pupil" of Harriot, later spent two years as secretary to Viète before returning to Harriot's patronage. How much cross-fertilization happened through Torporley is still a matter of conjecture.



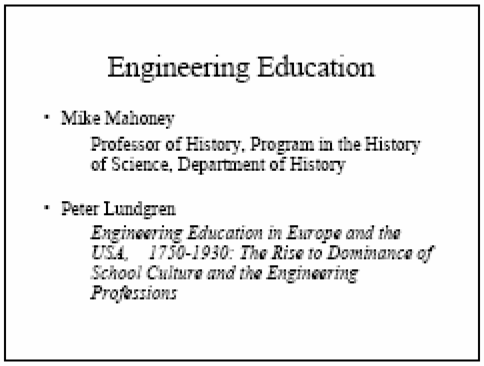
Mathematical Mystery

- Harriot <=> Nathaniel Torporley <=> Viète
- Viète => Descartes
- Newton => Varignon => Leibniz

We *do* know that René Descartes (1596-1650) read Viète's work; Descartes claimed to have done so *after* he'd completed his own work, but hunoz.

And we've all heard about the he-said/he-said between Newton and Leibniz, but one thing I noticed while looking at *Principia* was the lack of abstraction in Newton's work. Particularly in the early sections of *Principia*, Newton's arguments relied heavily on geometric justification and motivation. It was Pierre Varignon who translated much of Newton into the more algebraic approach of Leibniz.

So the technical side was taking shape courtesy of some rather straightforward reading of mathematical history. But what about the "cultural" aspects—the movement of mathematics and science into engineering? That was different.



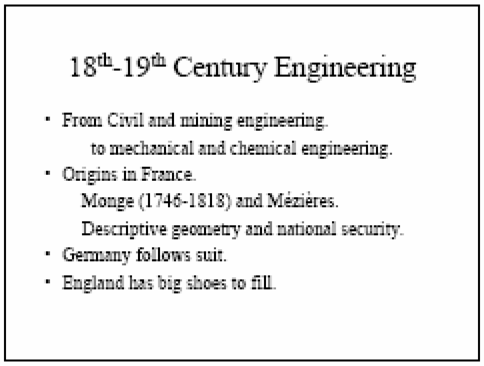
Engineering Education

- Mike Mahoney

  Professor of History, Program in the History of Science, Department of History

- Peter Lundgren

  Engineering Education in Europe and the USA, 1750-1930: The Rise to Dominance of School Culture and the Engineering Professions



18th-19th Century Engineering

- From Civil and mining engineering.

  to mechanical and chemical engineering.
- Origins in France.

  Monge (1746-1818) and Mézières.

  Descriptive geometry and national security.
- Germany follows suit.
- England has big shoes to fill.

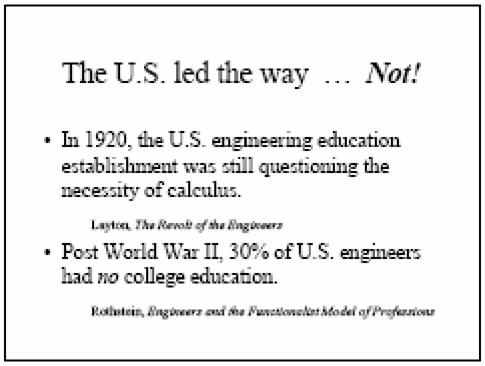
Fortunately I ran across Mike Mahoney's site at Princeton. His wealth of knowledge in the history of mathematics, science, and engineering was overwhelming. From his web site I started tracing the history of engineering.

One paper's title was particularly intriguing: *Engineering Education in Europe and the USA, 1750-1930: The Rise to Dominance of School Culture and the Engineering Professions* by Peter Lundgren, who kindly sent me a copy.

Lundgren's encyclopedic paper was a gold mine. From Lundgren's 40+ page paper I learned that France led the way with a scientific approach to Civil and Mining Engineering. A little reading in the history of mathematics revealed that Gaspard Monge (1746-1818) taught at the artillery school at Mézières; that as a teen-ager he invented descriptive geometry; and that the military immediately classified the technique a state secret until 1794; and that the Mézières school supplied the best scientific education of the 18th century. Not surprisingly that school produced many fine mathematical physicists and engineers.

Lundgren traces spread of the new style through France, to Germany, and then later in the 19th century had taken firm root England. He also details the expanding disciplines: from Civil, to Mechanical, to Electrical.

But then we come to the United States.



The U.S. led the way … *Not!*

- In 1920, the U.S. engineering education establishment was still questioning the necessity of calculus.

  Layton, *The Revolt of the Engineers*
- Post World War II, 30% of U.S. engineers had *no* college education.

  Rothstein, *Engineers and the Functionalist Model of Professions*

From Lundgren's paper I learned that in 1920 the U.S. engineering education establishment was still debating whether students needed to learn calculus [Layton, *The Revolt of the Engineers*]. Sounds like the resistance to software logics in the current debates.

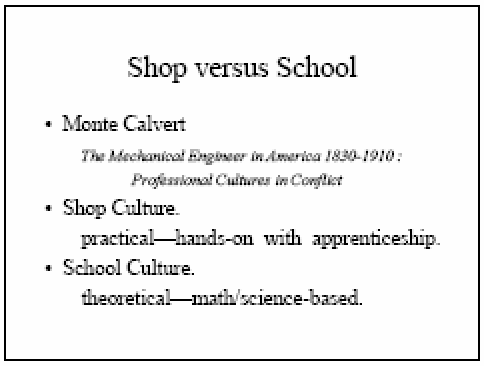
The resistance to theory was a shock. *Principia* was published in 1687 and yet something we take for granted now was still controversial in 1920, some 233 years later. Of course there were far-sighted institutions who had seen the benefits before the turn-of-the century: Rensaeler

Polytechnic Institute and MIT for two, and West Point, perhaps following Monge's lead for a third..

But between 1920 and the post World War II years, calculus *had* become required—perhaps because of World War II and its increasing dependence on technology.

But the U.S. workforce was still populated with old-school engineers. Lundgren also noted that in post World War II, 30% of American engineers had no college whatsoever. [Rothstein, *Engineers and the Functionalist Model of Professions*].

At a meta-level, Lundgren explored Monte Calvert's notions of the "shop-culture" versus the "school-culture" in engineering education. These terms give names to the two distinct views of engineering—one based on hands-on, apprenticeship-driven, training, while the latter relies on theory and mathematical physics. The latter is the stuff of real, long-term, education.



Shop versus School

- Monte Calvert
  The Mechanical Engineer in America 1830-1910 : Professional Cultures in Conflict
- Shop Culture.
  practical—hands-on with apprenticeship.
- School Culture.
  theoretical—math/science-based.

But there *is* a point to shop-culture that's minimized by the school-culture: the apprenticeship. There's more to apprenticing that just being aware of laboratory techniques. Apprenticeship should instill a sense of responsibility, conscientious behavior, and—dare I say it—ethics.

Furthermore, there are things that "book larnin'" is hard-pressed to duplicate. The "finger" metaphor I introduced earlier reminded me of a paper by Derek de Solla Price, called "Of Sealing Wax and String." Here's an excerpt:

"*During the golden age of experimental physics early in this century, all progress seemed to depend in a band of ingenious craftsmen with brains in their fingertips, who exploited a great many little-known properties of materials and other tricks of the trade. These tricks not only made all the difference in what could or could not be done in the laboratory; to a large extent, they determined what was discovered …*"

In 1984, I used that quote in a paper about Steve Russell and his influence in the development of Lisp. I stand by that paper; I only recommend that those of us who are not as smart as John McCarthy and Steve Russell, expend a substantial effort to bring a formal and logical discipline to software so that those who are working in the trenches have something more than the "vocational training" to which Alan Kay alluded.

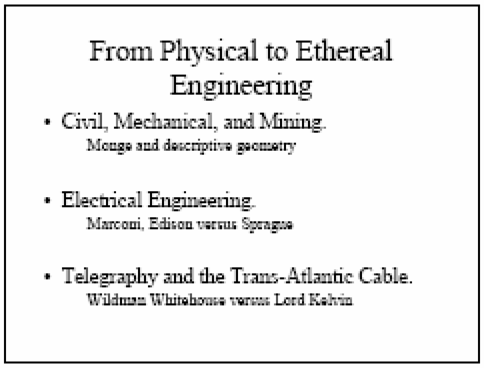
And in that same paper I quoted Stephen Jay Gould:

I have often been amused by the vulgar tendency of the human mind to take complex issues, with solutions at neither extreme of a continuum of possibilities, and break them into dichotomies, assigning one group to one pole and the other to the opposite end, with no acknowledgment of *subtleties and intermediate positions--and nearly always with moral opprobrium attached to opponents.*

We can see that vulgarity in politics (well, duh!) and in the theory/practice, school/shop dichotomies. And we can see the Gould-like reaction to it in the quotes from John McCarthy and Christopher Strachey in "History, Mystery, and Ballast."

But returning to history, traditional engineering in the U.S. *did* extricated itself from the shop-culture (even if it was long after Europe had done so), and this raised a larger question: even with the advanced perspective in the best Continental schools, the shop-culture had persisted in Europe into the 19th century. When did Europe transform, and why? Was there a recognizable "forcing event" that made the Europeans change to a school-culture? And can we foresee, or at least conjecture, a similar forcing function for software? For it's clear that the spinach-eaters are firmly in charge of software construction now.

Regardless, somewhere in all the reading, it occurred to me that electrical engineering represented a change: civil, mining, mechanical, and even chemical engineering, offered a physical immediacy that's absent in electrical engineering. Even so, a great deal of early electrical experimentation was seat-of-the-pants. But then came the Trans-Atlantic cable.



From Physical to Ethereal Engineering

- Civil, Mechanical, and Mining.
  Monge and descriptive geometry
- Electrical Engineering.
  Marconi, Edison versus Sprague
- Telegraphy and the Trans-Atlantic Cable.
  Wildman Whitehouse versus Lord Kelvin

In the paper, I talk about the back-and-forth between Dr. Wildman Whitehouse —the practical man— and William Thomson (1824-1907) —the mathematical physicist. Benjamin Disraeli said it well: "*The practical man is one who repeats the mistakes of his forefathers*." Wildman made more than his share of mistakes, and by some accounts, some were more deceptions than honest mistakes.

The bottom line was that seat-of-pants engineering failed at operating an intact cable, and the problem of discovering faults in a sunken cable was totally beyond any *ad hoc* techniques. Thus I came to believe that the Trans-Atlantic cable was the forcing-event, requiring a shift from shop- to school-culture.

** needs the failure analysis here, but ***

(Aside: I was at this stage in early September of 2001. I remember that on the night of September 10, I had read Postman's "…the United States is not a culture, but merely an economy, which is the last refuge of an exhausted philosophy of education."

On 9/11 I quit. Period. I only returned to computation in 2003 and only returned to this material when JonL sent me e-mail in January 2005.)

Now we move to computation. I believe there's an inevitable progression from shop-culture to school-culture in computational engineering.

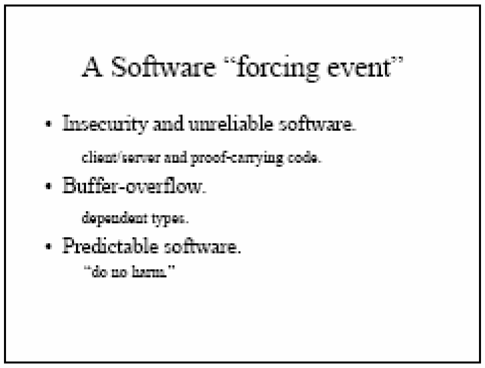
We can already see that in the hardware arena. Purveyors of hardware are well-ensconced on the logical bandwagon. Why? Because they have a huge financial incentive to ship error-free products. They cannot ship patches. But when it comes to software we have a different situation; for some reason we still buy "as is" products, knowing full well that they have bugs. Until customers (or their lawyers) put financial pressure on irresponsible software purveyors, they will continue to produce junk. There is no incentive for them improve—unless we want to talk about intangibles like self-respect, responsibility and ethics. Not likely.

But on a technical level, it is a hopeful sign to see that such questions were being asked as early as McCarthy's quote that begins "History, Mystery, and Ballast." Though it took almost 250 years for the U.S. mainstream to "get it" in traditional engineering, perhaps frustration after 50 years wandering in the software wasteland is excessively pessimistic.

The notion of typed languages, and the interplay between static specification and dynamic computation, is one technique that gives credence to John McCarthy's forty-year old appeal. We'll come back to this later. But now we'd like to scan the horizon for a software "forcing event." Is there some aspect of contemporary software that differs in a substantial way from the last 50 years of "spinach-stained finger" programming? —Something comparable to locating failures in a cable two miles below the surface of the Atlantic Ocean.

I believe there is: it's software security … or perhaps better said: software *insecurity*.

The modern world —such as it is—critically depends on fundamentally unreliable software. It's not just an issue of malicious morons. Building predictable code is impossible when there's no well-defined specification against which one can measure success or failure. *Ad hoc* techniques <u>cannot</u> work.



A Software "forcing event"

- Insecurity and unreliable software.
  - client/server and proof-carrying code.
- Buffer-overflow.
  - dependent types.
- Predictable software.
  - "do no harm."

The poster child for bad code is the obnoxious "buffer overflow problem." There's really no excuse for it. If the hardware is too stupid to protect against inappropriate memory references, then at least the software should monitor array calculations. It's very straightforward, but the argument is that such run-time checks carry a performance penalty —as if viruses don't.

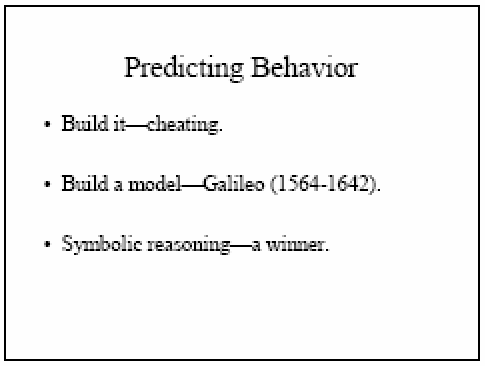
But what if one could be assured *before* execution that none of the program's references exceeded a buffer bound? Then the run-time checks could be removed without jeopardy. And this should be contrasted with what has been called "programming by contract" which essentially involves run-time specifications. Now this was something that might convince a budding software engineer that one's theoretical work was not frivolous.

So I went looking. Of course, the general specification problem is called "program correctness." That's very, very hard unless the program and specification is very, very simple. And showing the correctness of a very, very simple program is very, very unconvincing.

All of this led me to Proof-Carrying Code (PCC) where a client wishes to embed some code in a suspicious server. In PCC there is a sharp repartee and negotiation between the antagonists: the server's statement of its security policy, followed by the client's response in the guise of a proof that its code satisfies the policy, and—in the best of all possible worlds—the server's grudging acceptance of that proof.

I cover some of this in the paper ... but not much. The point of the paper, this talk, and to a large extent the SCU course, is simply to introduce ideas, not give an exhaustive treatment. It's motivation, motivation, and motivation again.

Therefore I opted for a more humble target than correctness. Namely that in the realm of security, we can take comfort in knowing that a program "does no harm." (And thus I came to the term "Predictable Software," with "does no harm" as a possible prediction.)
So I knew where I wanted to go, and knew the tools I wanted to use: logic. But how to convince reluctant engineers?



Predicting Behavior

- Build it—cheating.
- Build a model—Galileo (1564-1642).
- Symbolic reasoning—a winner.

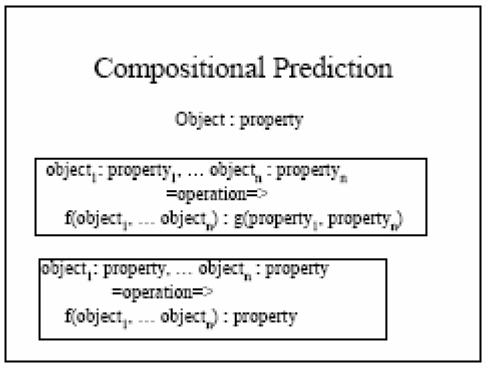
Engineers know there are several ways to address the issue of predictable behavior.

* Build and observe—but that's cheating. We want to *predict* behavior, not to clean up after it. Hammurabi had an effective technique to curb sloppy building habits.

* Build a model— In the paper I mentioned that Edison resorted to this, only to be out-done by Sprague. Engineers even in Galileo's time recognized perils here. In the first few pages of "Two Sciences," Galileo deals with models, and the problems of scale-up.

* A more enlightened approach is to replace simulation with symbolic prediction: to isolate valuable properties of simple objects, and show that combinations of those objects, produces compound object whose properties are predictable combinations of the component's properties.

$$\text{object}_1 \textbf{:} \text{ property}_1, \ldots \text{ object}_n \textbf{ : } \text{property}_n$$
$$=\text{operation}=>$$
$$f(\text{object}_1, \ldots \text{ object}_n) \textbf{ : } g(\text{property}_1, \text{property}_n)$$



Compositional Prediction

Object : property

$$\text{object}_1 : \text{property}_1, \ldots\ \text{object}_n : \text{property}_n \quad \text{=operation=>} \quad f(\text{object}_1, \ldots\ \text{object}_n) : g(\text{property}_1, \text{property}_n)$$

$$\text{object}_1 : \text{property}, \ldots\ \text{object}_n : \text{property} \quad \text{=operation=>} \quad f(\text{object}_1, \ldots\ \text{object}_n) : \text{property}$$

Even more, we'd like:

$$\text{object}_1 \textbf{:} \text{ property}, \ldots\ \text{object}_n \textbf{ :} \text{ property} \quad \text{=operation=>} \quad f(\text{object}_1, \ldots\ \text{object}_n) \textbf{ :} \text{ property}$$

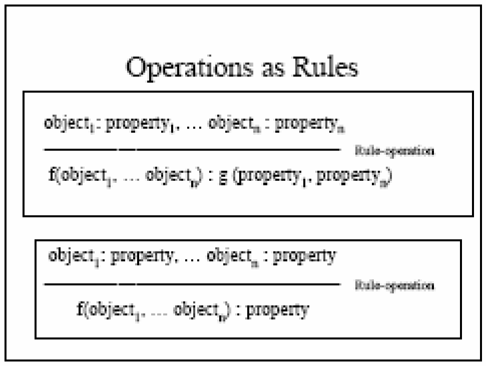
which says that the combining operation preserves a common property of the component objects. And what's a good candidate property? How about "does no harm"?

In the coming computing-as-logic metaphor, it's now natural to equate "=operation=>" with a rule for deduction or computation. With varying degrees of hindsight, we can transform the description into an inference-rule style:

$$\frac{\text{object}_1 \textbf{:} \text{ property}_1, \ldots\ \text{object}_n \textbf{ :} \text{ property}_n}{f\ (\text{object}_1, \ldots\ \text{object}_n) \textbf{ :} \ g\ (\text{property}_1, \text{property}_n)}\ \text{Rule}_{\textbf{operation}}$$

or the simplified form:

$$\frac{\text{object}_1 \textbf{:} \text{ property}, \ldots\ \text{object}_n \textbf{ :} \text{ property}}{f(\text{object}_1, \ldots\ \text{object}_n) \textbf{ :} \text{ property}}\ \text{Rule}_{\textbf{operation}}$$



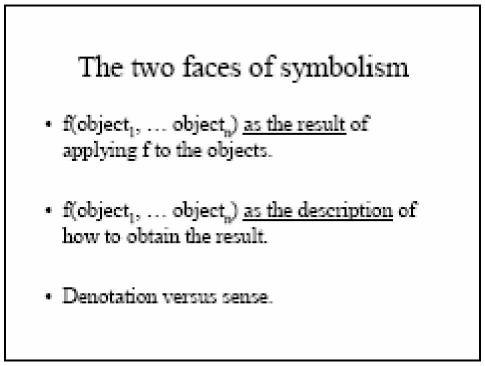
Operations as Rules

$$\frac{\text{object}_1 : \text{property}_1, \ldots\ \text{object}_n : \text{property}_n}{f(\text{object}_1, \ldots\ \text{object}_n) : g\ (\text{property}_1, \text{property}_n)}\ \text{Rule-operation}$$

$$\frac{\text{object}_1 : \text{property}, \ldots\ \text{object}_n : \text{property}}{f(\text{object}_1, \ldots\ \text{object}_n) : \text{property}}\ \text{Rule-operation}$$

where we think of "f ($\text{object}_1, \ldots\ \text{object}_n$)" as the result of applying f to the component objects.

We can also think of " f ($\text{object}_1, \ldots\ \text{object}_n$)" as the description of how to obtain the result.

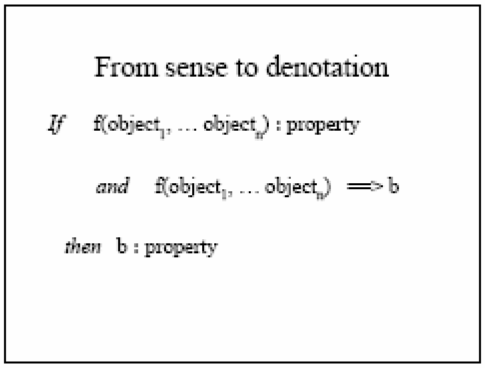


The two faces of symbolism

- f(object$_1$, … object$_n$) as the result of applying f to the objects.
- f(object$_1$, … object$_n$) as the description of how to obtain the result.
- Denotation versus sense.

In this latter reading we expect that

If f(object$_1$, … object$_n$) **:** prop *and* f(object$_1$, … object$_n$) ==> b
then b **:** prop



From sense to denotation

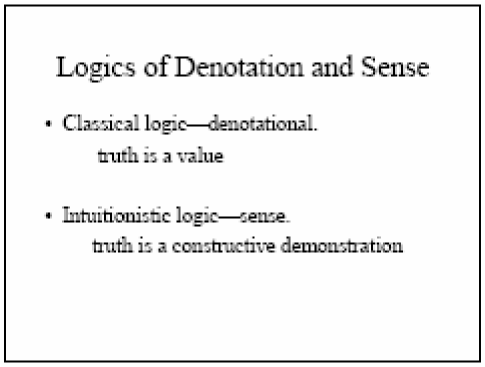
If f(object$_1$, … object$_n$) : property

and f(object$_1$, … object$_n$) ==> b

then b : property

These two flavors of interpretation of " f(object$_1$, … object$_n$)" reflect two separate ideas: denotation and sense. In the paper I use the homely analogy of "sense" as the journey while the "denotation" is the destination. These two flavors give rise to two different views of logic, and rather than go through typed programming languages, which are familiar to you, I'd rather give a taste of the logics.

The typical logics —called classical logics—deal with truth. These are the logics of denotation. These can be contrasted with the logics of sense, where the process used to arrive at the denotation is of importance.



Logics of Denotation and Sense

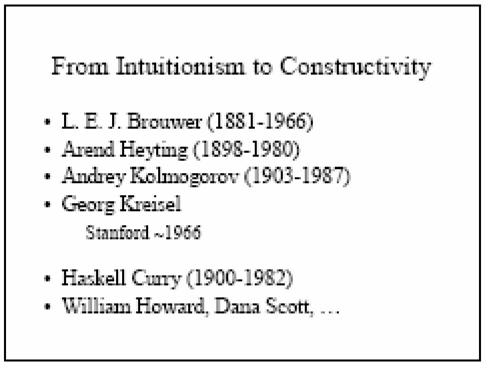
- Classical logic—denotational.
  truth is a value
- Intuitionistic logic—sense.
  truth is a constructive demonstration

Perhaps Viète's introduction of parameters represents the first small taste of sense versus denotation; instead of manipulating specific equations with the corresponding collapse of coefficients by applying arithmetic operations (giving a denotation), the use of parameters requires that the arithmetical manipulation of coefficients remain explicit symbol combinations, thereby exposing their sense.

In the world of logic, the canonical logic of sense is the logic of proof—as compared to the logic of truth. Namely that one should not assert the truth of a proposition without having a convincing demonstration.

The history of such logics really begins in the early part of the 20th century with what is called Intuitionism, the child of L.E.J. Brouwer's (1881-1966) fertile imagination. Brouwer was brought to his ideas by a discomfort with reasoning like that of Zermelo who proved that every set could be well-ordered without giving any indication of what such an order could be in general.

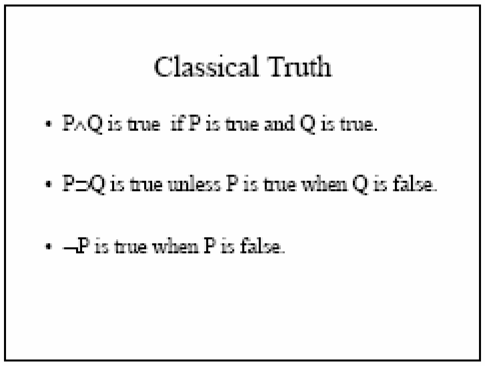
The formalization of Brouwer's ideas was first accomplished independently by Brouwer's student Arend Heyting (1898-1980) and the Russian, Andrey Kolmogorov (1903-1987). The resulting informal semantics, called the BHK interpretation, emphasizes the "sensational" rather than the denotational.



From Intuitionism to Constructivity

- L. E. J. Brouwer (1881-1966)
- Arend Heyting (1898-1980)
- Andrey Kolmogorov (1903-1987)
- Georg Kreisel
  Stanford ~1966
- Haskell Curry (1900-1982)
- William Howard, Dana Scott, …

For example, we know that in a denotational setting the (truth) value of a proposition of the form $P \wedge Q$ is true just in the case that both of P and Q are true; and $P \supset Q$ is true unless P is true when Q is false.

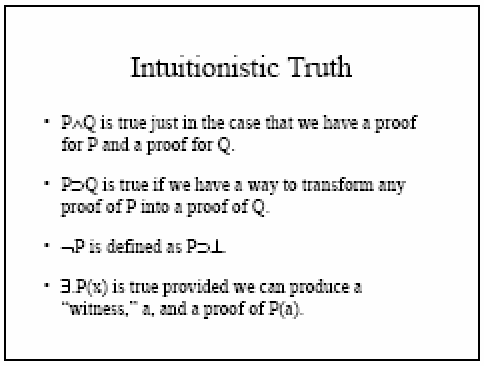


Classical Truth

- $P \wedge Q$ is true if P is true and Q is true.
- $P \supset Q$ is true unless P is true when Q is false.
- $\neg P$ is true when P is false.

Truth in the sensational case is *still* compositional, but is more complex: $P \wedge Q$ is true just in the case we have a proof of P and a proof of Q. Of course this leads to the question "what is a proof?" Informally, we expect a proof to be a construction, a recipe; that is, an algorithm.
So for example, $P \supset Q$ is true if we have a construction that will transform any proof of P into a proof of Q.

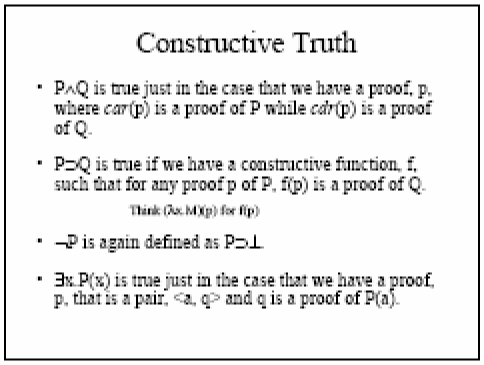
But the signature case for sensational logics arrives with predicates and the interpretation of $\exists x.P(x)$. In fact, the meaning of existence as raised by Zermelo's work on well-orderings, gave rise to Brouwer's initial concerns. So in a constructive setting, the truth of $\exists x.P(x)$ requires two things: the construction of an element called a "witness," and a construction demonstrating that the witness satisfies the predicate P.



Intuitionistic Truth

- P∧Q is true just in the case that we have a proof for P and a proof for Q.
- P⊃Q is true if we have a way to transform any proof of P into a proof of Q.
- ¬P is defined as P⊃⊥
- ∃.P(x) is true provided we can produce a "witness," a, and a proof of P(a).

I still remember taking courses on Intuitionism from Georg Kreisel at Stanford in 1966 or so, trying to (a) read his writing, and (b) understand his ideas and (c) map those ideas into what I knew about Lisp; I wasn't successful. It was a couple years later that Scott, Howard, and others made the appropriate connections between BHK semantics and λ-calculus based languages, leading to the Curry-Howard Isomorphism.

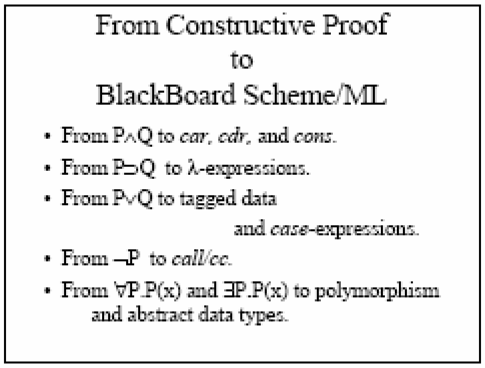


Constructive Truth

- P∧Q is true just in the case that we have a proof, p, where *car*(p) is a proof of P while *cdr*(p) is a proof of Q.
- P⊃Q is true if we have a constructive function, f, such that for any proof p of P, f(p) is a proof of Q.
  Think (λx.M)(p) for f(p)
- ¬P is again defined as P⊃⊥
- ∃x.P(x) is true just in the case that we have a proof, p, that is a pair, <a, q> and q is a proof of P(a).

For example the constructive proof of P∧Q is a pair (think dotted pair) where the left element of the pair (think car) is a proof of P and the right element (think cdr) is a proof of Q. And I *do* mean car and cdr, not first and rest. The treatment of recursive data, like lists, must be handled differently; that treatment makes explicit the relationship between recursion and induction.

In the constructive treatment of P⊃Q, the construction, that maps proofs of P onto proofs of Q, becomes our ubiquitous λ-term. The full-fledged version of this constructive treatment becomes what I call BlackBoard Scheme in the SCU course. So we have two sides: a constructive logic component, and a simple functional programming language component.



From Constructive Proof
to
BlackBoard Scheme/ML

- From P∧Q to *car*, *cdr*, and *cons*.
- From P⊃Q to λ-expressions.
- From P∨Q to tagged data
  and *case*-expressions.
- From ¬P to *call/cc*.
- From ∀P.P(x) and ∃P.P(x) to polymorphism
  and abstract data types.

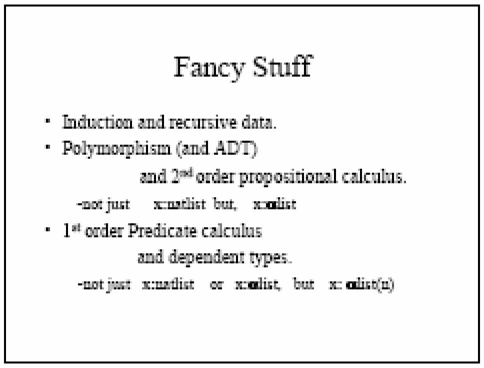
One thing I didn't mention in the paper is the constructive treatment of negation. In constructive logic, ¬P is an abbreviation for P⊃⊥ where ⊥ indicates "contradiction." So a constructive proof of ¬P is a construction that takes any supposed proof of P into ⊥. And ⊥ is an object that has no proof. So what in the functional language corresponds to the constructive logic's treatment of ¬P? Turns out that it's constructs like exceptions and call/cc.

*** *case* and disjoint union***

It might be useful to think of the question behind BBS as "Is proposition, P, constructively provable?" and the question behind BBML as "Is an object, a, a proof of proposition, P?".

The bottom line is that the logic of computation is a constructive/sensational one, not the usual classical/denotational logic.

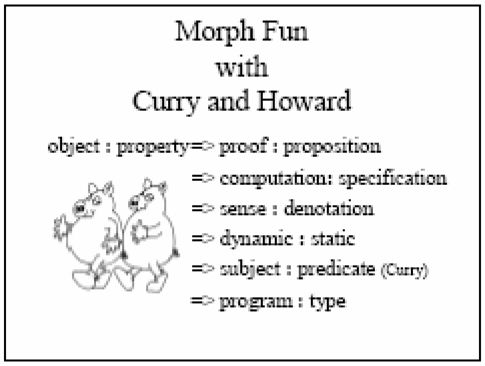
And computation reduces sense to denotation.



Fancy Stuff

- Induction and recursive data.
- Polymorphism (and ADT)
  and 2nd order propositional calculus.
  -not just x:natlist but, x:αlist
- 1st order Predicate calculus
  and dependent types.
  -not just x:natlist or x:αlist, but x: αlist(n)

Back to the issue of security.
Courtesy of the Curry-Howard Isomorphism, we can shape-shift the object**:**property discussion into a proof**:**proposition description of constructive logic; and from that we morph into a computation**:**specification pair. At a more esoteric level we can view this pariring as exemplifying the notions of sense**:**denotation or dynamic**:**static. And returning to Curry's early work, he worked with subject **:** predicate. Finally at a more concrete level, we can map the pair onto the program**:**type structure that we see in typed languages like ML.

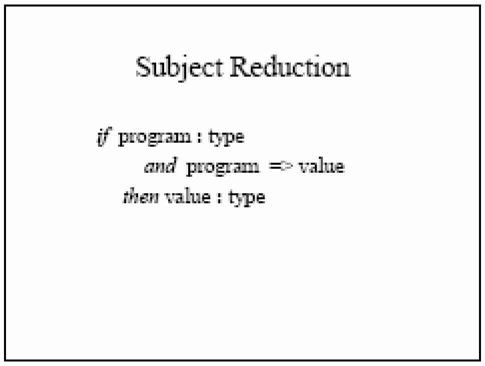


Morph Fun
with
Curry and Howard

object : property=> proof : proposition
=> computation: specification
=> sense : denotation
=> dynamic : static
=> subject : predicate (Curry)
=> program : type

But for the security metaphor to work, we require that the typed language be well behaved in the sense that properties we prove about static text be preserved across execution. This is called "Subject Reduction" as explained in the paper.

Subject Reduction *if* prog**:**type *and* prog => val *then* val**:**type



Subject Reduction

if program : type
and program => value
then value : type

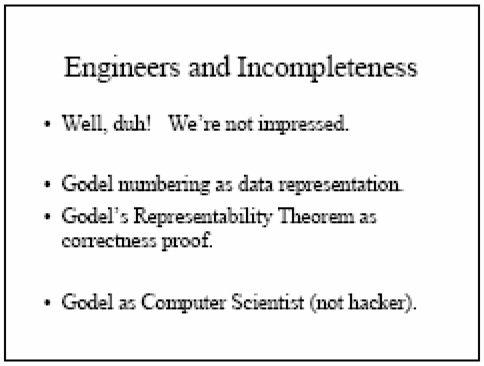
Languages that violate Subject Reduction are to be shunned.

Notice, by the way, that Subject Reduction is *exactly* the same relationship we advertised earlier to reduce sense to denotation.

A final note about types and typed languages: the types induced by constructive logics are not *ad hoc*. They are regular, predictable and consistent.

* * *

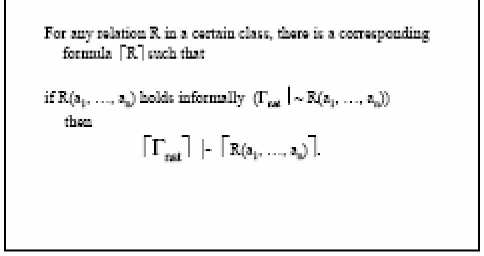
And returning to the issue of motivation, the typical logic and computation course at least mentions Godel's Incompleteness results. We do too, but incompleteness (or completeness) results don't really impress engineers. However there are several parts of the process that *can* resonate.



Engineers and Incompleteness

- Well, duh! We're not impressed.
- Godel numbering as data representation.
- Godel's Representability Theorem as correctness proof.
- Godel as Computer Scientist (not hacker).

First, part of the incompleteness result involves the mapping of language components to data—what's called Godel numbering; it's a particularly nasty case of representation; engineers can relate to that. But the more interesting and more important point is contained in Godel's Representability Theorem. He actually has to prove that his representation is faithful. This is a form of specification-versus-representation proof, making Godel a particularly good example of a Computer Scientist. Maybe the first one.



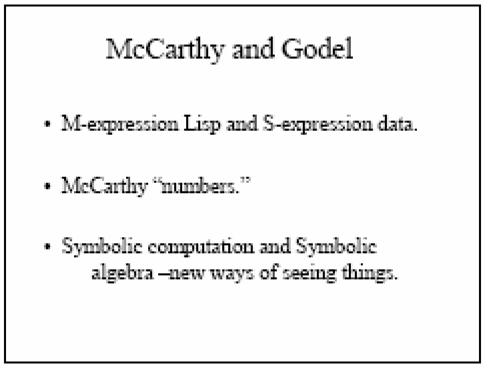
Godel's Representability Theorem

For any relation R in a certain class, there is a corresponding formula $\ulcorner R \urcorner$ such that

if $R(a_1, \ldots, a_n)$ holds informally ($\Gamma_{nat} \mid\!\sim R(a_1, \ldots, a_n)$)
then

$$\ulcorner \Gamma_{nat} \urcorner \vdash \ulcorner R(a_1, \ldots, a_n) \urcorner.$$

And of course in the discussion of our BlackBoard Scheme, we lay bare the origins of Lisp-as-a-programming-language notation. All too often people are unaware of the reasoning behind Lisp's "weird" syntax. In McCarthy's early papers we see Lisp expressed in what he called M-expressions—for "meta-expressions"—with S-expressions confined to the world of data.

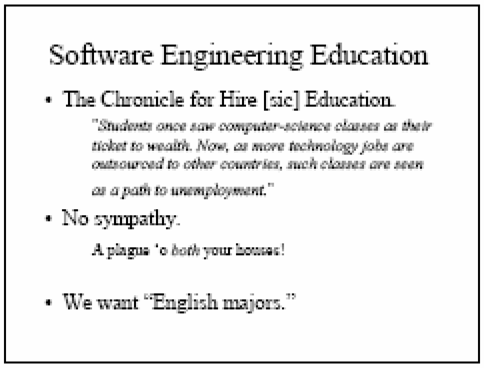


McCarthy and Godel

- M-expression Lisp and S-expression data.
- McCarthy "numbers."
- Symbolic computation and Symbolic algebra –new ways of seeing things.

By comparison, in Godel's work, this is the distinction between the language for talking about numbers and those numbers; or more formally, a language for talking about *numerals* and the *numerals.* Godel's numbering mapped the language of numbers into numbers—and then formalized the business to deal with numerals.

The first part is the path McCarthy took to get the M-expressions represented as S-expressions. While Godel's numeric representation was decodable, but totally opaque, McCarthy's was transparent. One could see the image of the M-expression in the S-expression representation. Thus S-expressions could be finessed into a "language" while Godel's representation could not. One major contribution of McCarthy—one of many—was this new way of symbolizing computation; much like Viète did for symbolic algebra.



Software Engineering Education

- The Chronicle for Hire [sic] Education.
  "Students once saw computer-science classes as their ticket to wealth. Now, as more technology jobs are outsourced to other countries, such classes are seen as a path to unemployment."
- No sympathy.
  A plague 'o *both* your houses!
- We want "English majors."

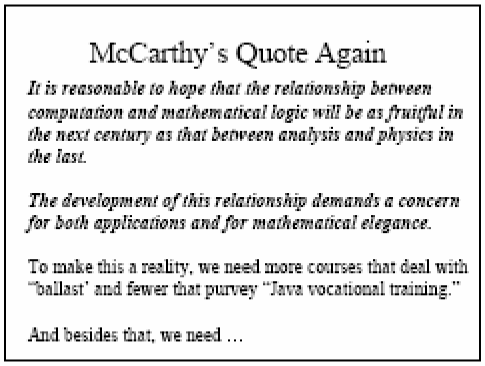
A recent issue of "The Chronicle of Hire [sic] Education" begins, "*Students once saw computer-science classes as their ticket to wealth. Now, as more technology jobs are outsourced to other countries, such classes are seen as a path to unemployment.*"

Frankly I don't have much sympathy. Much sympathy for either side of the issue: for either the students who look upon education as a meal ticket, or for schools who pander to them by teaching technology courses. As Shakespeare said "A plague o' *both* your houses!"

Would you have sympathy for one who chose to major in English because they saw technical writing as a path to wealth? I wouldn't.

So the point of our Ballast course at SCU is to attract *real* "English majors." We want them to understand and appreciate the intellectual content with the hope that they're smart enough to put what they've learned to good use *if/when* the occasion arises.

It is this kind of education, not Java vocational training, that will bring McCarthy's 40+ year old quote to life:
It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance.



McCarthy's Quote Again

It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last.

The development of this relationship demands a concern for both applications and for mathematical elegance.

To make this a reality, we need more courses that deal with "ballast' and fewer that purvey "Java vocational training."

And besides that, we need …

… and to do that we need "More Ballast!"
… and more Bombast. ☺

Thank you.