What Does a Computer Program Mean?
An Introduction to Denotational Semantics

Dr. Gene B. Chase, Messiah College, Grantham, PA 17027

I. Who should read this paper?

This paper is for mathematicians who are curious about how topology is being used to prove computer programs correct. Those advanced parts have been limited to Sections III, V, and VI, and they are marked by a ⌚. By contrast, sections II, IV, and VII are suitable as a companion to existing textbooks in a Computer Science course such as Organization of Programming Languages, the course CS 8 as described in Curriculum [1979]. Perhaps in a first reading you might read just those sections.

Among many books and articles on the semantics, or meaning, of computer languages, several now claim to be easy introductions to the topic. For example, the review by Kitchen [1989] of a book by Hennessy [1988] says that the book is “a very readable introduction ... my wholehearted nominee for this field's principal introductory text.” Instead, I find it terribly dense and unmotivated. It does not answer the most basic questions: Why does anyone care what a computer program means? What are the options for semantics (a formal meaning) from other disciplines that inform our choice?

And what about professional mathematicians now pressed into service teaching Computer Science courses? Sections II, IV, and VII are meant to be an easy introduction to the semantics of computer programs, with the harder details clearly marked. I mostly wanted to say the things that I didn't find in books, so this paper is meant to supplement printed textbooks, not to replace them.

II. The practical perspective: Why should I care what a program means?

As Lewis Carroll has Humpty Dumpty say, “When I use a word, it means whatever I want it to mean, nothing more and nothing less. It's just a matter of who's master.”

Lewis Carroll might say that a computer program can mean anything that we want it to mean, nothing more and nothing less. I am going to justify in as simple a way as I possibly can this definition: The meaning of a computer program is a mathematical function. How might we might arrive at that conclusion, and what does it mean in practical terms to a computer scientist earning her living writing programs?

In order to decide what a computer program should mean, what we call its `semantics,' we should examine the properties that a semantic representation should have. There are both practical and historical
motivations for these properties. We shall examine each of those motivations in turn.

Having a precise meaning for a computer program is important. Peer-reviewed authors even get the meaning of computer programs wrong. That would be less likely to happen if more attention were paid to semantics from the outset. More importantly, if a program does not match its intended meaning (as provided for example by a set of specifications stated declaratively in English), then it might fail in a mission-critical application.

Much attention is paid to semantics in the Computer Science curriculum, but under the guise of programming style. One practical goal of this paper will be to help you to program with good style by showing why bad style is not merely a matter of bad taste but a matter of resolving ambiguities which prevent a program from being clear and correct.

I believe that MIT educator Seymour Papert is right that the fundamental notions of algorithms and of recursion are available to the concrete operational child (about age 6) as primitive notions. Why then should we try to explain them in terms of something else, like mathematics, which has no claim at being any more fundamental? The answer is simply that mathematics provides a precision of thought that analogical thinking cannot.

Various authors (Chase [1979], Mayer [1979], Shneiderman [1979]) have tellingly argued that students of Computer Science need conceptual models with which to explain the workings of a computer program. For example, variables are "boxes" into which values are put in this sort of analogy. But variables on the left of an assignment statement are meant to name the boxes; variables on the right hand side of an assignment statement are meant to name the values in the boxes. To make all of this more precise, a mathematical approach is necessary. That should not be surprising, since one good definition of mathematics is the science of precise analogy.

A. Operational semantics was motivated historically by the desire to write error-free programs.

If operational semantics were to be applied to English, the meaning of the word `dog' would be `perro.' That is, operational semantics gives the meaning of the constructions of one computer language in a supposedly simpler computer language, just as a Spanish speaker would understand the meaning of `dog' if it were translated for her.

Schneider [1989] in his (otherwise) excellent introduction to QuickBasic says that a student should be careful to use `if (...) and (...) then ...` instead of `if (...) then if (...) then ...` because the first is clearer than the second. Clearer it may be, but it can be wrong in a situation like the following:

```
if (B <> 0) and (A/B > Error) then writeln ('Fraction too large');
```

I have used Pascal syntax because `and` in Pascal according to the document that is usually taken to specify Pascal, Jensen [1978], does not say what the order of evaluation of the conjuncts should be. Presumably, different compilers could do the following:

1. Evaluate all conjuncts in an order than cannot be predicted from run to run.
   This might happen, for example, on a parallelizing compiler, where the separate conjuncts are sent to separate processors to be evaluated.

2. Evaluate all conjuncts from left to right.
   This is what most introductory Computer Science students think is reasonable.

3. Evaluate all conjuncts from right to left.
   This is the easiest course of action for a compiler using a stack on which to store expressions.

4. Evaluate conjuncts from left to right, stopping as soon as the truth value of the whole conjunct can be predicted according to the rules of two-valued truth tables.
   This is sometimes called "McCarthy AND" because it is what the programming language Lisp is specified to do. Lisp was invented by John McCarthy, one of the first computer scientists to give serious thought to the semantics of programming languages.

The issue is a semantic one, not a syntactic one.

There are so many features of a language. How can we hope to think of all issues such as this one? The answer is to create a formal way in which we can specify a program, and to prove (which

suggests some mathematics) that the program meets the specifications. This is being done now for programs even as large as the size of a compiler. See Lee & Pleban [1986]. To put it briefly so as to highlight how to prove that a compiler works:

(*) **We first compile a program into mathematics, where proving is well-understood, before compiling it into hardware.**

The job is uninviting and even nearly unmanageable for a human alone, but the computer can assist in the housekeeping details. (Please don't ask how we know that the program that helps us is in turn correct!)

This style of semantics, in which we transform a program into another one about whose behavior we can reason, is sometimes called **operational semantics** because we are still thinking of a program as meaning how it behaves. The first appearance of an operational semantics of a practical language was Vienna Definition Language (VDL) in 1968 [Lucas, 1969], by which IBM Vienna attempted to put the PL/1 language on a sound semantic foundation. VDL is a "mathematical" or idealized computer on which the ideas for PL/1 were tested conceptually. It looked very much like the register-level description of any modern computer. VDL had been successful in describing Algol 60 after the fact. It was put to use in the *design* of PL/1.

B. Axiomatic semantics was motivated historically by the desire to prove programs correct.

If axiomatic semantics were to be applied to English, the meaning of the word `dog' would be `household pet, *Canis familiaris.*' That is, axiomatic semantics gives the meaning of the constructions of one computer language in a mixture of the same computer language and some other more formal language, just as a biologist would understand the meaning of `dog' more precisely if it were translated into both other words and the more precise Latin genus and species.

More generally, we not only want a compiled program to "do the same thing as" the original program, but we want the program to match its specifications. I use quotation marks to remind you that only the compiled version is executed (unless you have a language which can both be compiled or interpreted at request). So, often the machine only executes the translated program. To execute the original program we would have to work by hand. (I'm overlooking the fabrication of a chip which executes Pascal in the hardware, which Pascal's inventor, Niklaus Wirth, has done.)

Since specifications are stated declaratively, there must be some way to relate procedural code to declarative statements of that code's outcomes, like "the array will be sorted."

One possible solution to this dilemma is to restrict people to write in a language which has both declarative and procedural readings of the same statements. Or along the same thread of ideas,

We first write a program's specifications, which we then transform into the program.

That turns out to be very hard to do. So far it has only been done for programs of the size of a single function, such as calculating the quotient q and remainder r upon division of a by b given the specification that $a = qb + r$ and $0 <= r < b$. For larger program, the best that has been done is partially automated, partially manual. For example, there are compiler-compilers, whose job it is to craft a compiler out of specifications for syntax, but the code generation part is still manually input, an art more than a science.

This style of semantics, which is concerned with a program matching its specifications, is often called **axiomatic semantics**, or Floyd-Hoare semantics in honor of Robert Floyd and Tony Hoare, the two main instigators of the ideas. The specifications are declarative statements and the goal is to prove (usually by mathematical induction) that the computer program transforms correct

declarative statements into correct declarative statements, based on a set of axioms related to the constructions of the language. I'll display an example below.

Having a semantics which is already very close to mathematics would make easy the job of reading procedures declaratively. That's the main reason that **functional** languages are receiving widespread attention today, where every step in the program is a function. With the exception of input and output functions, we make those functions to be free of side-effects (that is, of potential changes of global variables). Such a functional program can be proven mathematically to do what it claims to be doing. For that matter, changing global variables threatens the *programmer's* ability to understand what the code is supposed to do. Pure functional languages like John Backus's FP [1978] or a subset of some dialects of Lisp like Scheme or TLC Lisp are not yet widely used, but they are a step in the right direction.

Another way to make this job easy would be to use a **logic** language. A pure logic language also has a semantics, based on relations, which is very close to mathematics. This was one motivation for the Japanese Fifth Generation computing project to choose Prolog as the foundational language.

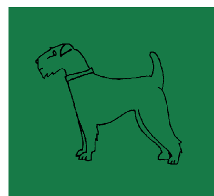
But can we do without global assignment statements, the so-called “go to” of data structures, as required by a functional language? We can. We can even do without assignment statements at all, but that's a little radical for an introduction. To do so we must think in new ways about global assignment, just as we now avoid `go to` by using block-structured constructs. We might even need new hardware which is optimized for the new constructions, just as current hardware is optimized for sequential Pascal-like languages.

The “good” subset of current functional or logic programming languages has not proved to be useful in practice to date. In Prolog for example, recursion relies on the correct ordering of statements (some compilers to be cooperative in this respect reorder statements for you—ugh!), side effects are produced by several of its primitives, and the cut operator prevents a statement from having a purely declarative reading.

Using the mathematics which I shall describe below, not just compilers but also sorting algorithms, a unification algorithm, theorem-provers, and even hardware have been proved correct; that is, to validly match their specifications. Hardware proved correct has included an associative memory unit, a CMOS invertor, a sequential multiplier, and one entire microprocessor—the Viper—according to Larry Paulson [1987]. The Defense Department [1983] reserves its highest level of trust in a computer program for those for which a proof that it does as claimed has been provided.

C. Denotational semantics is motivated by the desire to automate program transformation.

If denotational semantics were to be applied to English, the meaning of the word `dog' would be



. That is, denotational semantics gives the meaning of a computer language construction in terms of a mathematical thing to which that construction refers.

Suppose you had the job of converting thousands of lines of Fortran 77 code to PL/1 code. How would you do it? By hand, no doubt. Why not just write a program to convert Fortran programs to PL/1 programs? Can that be done? Certainly 90% of it ought to be able to be done automatically. Why not 100%? The answer is obvious if we go from PL/1 to Fortran: Fortran 77 doesn't support records, or recursion, or a host of other PL/1 constructions, so we are left stranded when those constructions are found in a

PL/1 program.[1]

Even in the other direction, there are problems. Is the value of a loop control variable being used outside of the loop? If so, what value will it have: the test value of the loop, or one increment more? Is the automatic conversion of an integer to a real in an assignment statement intentional or accidental? We could tell the Fortran programmer not to use any "tricks," but that's just another way of saying use a semantically well-behaved subset of Fortran. Remember that Fortran even allows you to change the value of constants by passing them as (reference) parameters!

One personal anecdote here will help you to see what I mean about tricks. When I first programmed for money at Grumman Engineering Corporation, converting programs from Fortran II to Fortran IV, I ran into an assignment statement that looked like this:

```
I = I
```

I asked a more experienced colleague why the statement was in the program. He answered that the compiler always kept the index of a DO loop in a register (which is why I still call registers "index registers"), so the index could not be used outside the loop. Fortran II had a good idea semantically. But the author of the program wanted to use this index upon exit from the loop. His line of code copied the index in a register on the right hand side into an addressable location in memory on the left hand side! You can see that the semantics of the Fortran II program depended on undocumented knowledge about the compiler's behavior.

If we had a formal way to capture the meaning of a program, then we could say that the translation of a program from Fortran to PL/1 must preserve meaning. One possible strategy that is being tried by various people, at MIT in their automated programming project (see Rich & Wills [1990]) for example, is to create a library of idioms in some kind of "canonical form" and to go from Fortran to the canonical form and then from the canonical form to PL/1 code. That means in particular that the mapping from code to canonical form must be reversible. That form is usually something resembling functional programming: data-flow, or John Backus's FP [1978] for example. The language ML is becoming increasingly popular for this task because it is strongly typed like Pascal.

Such research is in its early stages, but as with any "expert systems" approach, which depends on querying perhaps many human experts to provide such a library, we can never guarantee 100% conversion, since we may always encounter a piece of code that's not in our library of idioms. For a simple example, if the library recognizes a SORT when it sees a Bubble Sort, it will not thereby recognize a Quicksort. But there are an unlimited number of sorting methods, so that plan cannot succeed 100%.

But the late Christopher Strachey came up with an idea to represent the meaning of computer programs as mathematical objects. (And to represent data too, but that's the easy part.) Strachey's approach is called **denotational semantics** because the mathematical objects denote the program or the data. The problem was that his system was contradictory, for it required that a set of elements be equivalent to a set of functions from those elements to themselves. It's easy to see by a counting argument that that's impossible: there are always more functions from D to D than there are elements in D. To put it more precisely, Strachey's models used something called lambda calculus, but there were no consistent models of lambda calculus at the time that Strachey began his work. Until Dana Scott, now of Carnegie Mellon University, came up with (several in fact) models of the lambda

---

[1] Of course, both PL/1 and Fortran 77 are equivalent in a strong way, both being equivalent to a Turing machine. However, this equivalence is not easily recognized by a human. As we shall see in the historical section below, the meaning of a computer program ought to be understandable.

calculus, Strachey had nothing other than a potentially contradictory set of axioms and a great intuition. Scott put Strachey's work on a firm foundation. Denotational semantics represents both data and programs as mathematical objects in a uniform way.

Program transformation then proceeds by transforming a program to its denotation, then translating the denotation into a program in the new language. Much harder to do than to say! But that is what I would like to give you the flavor of in what follows.

⏲ III. A mathematical model: an information system

This section should be skipped on a first reading. It assumes a familiarity with general topology, and it is presented in telegraphic summary style, not an expository style. Any notes that depend on this section will be marked with ⏲ below.

A. Definitions

1. Lattice. A lattice is a set for which every pair of elements has a least upper bound (supremum, lub, or $\sqcup$) and a greatest lower bound (infimum, glb, $\sqcap$).

> **Example 1.** An example from Numerical Analysis. Consider closed intervals of rational numbers, and the empty inverval $\varnothing$ which we write $\top$. We define
>
> $$I_1 \sqcup I_2 \equiv I_1 \cap I_2$$
>
> giving a `semi-lattice.' If we were to define $I_1 \sqcap I_2 \equiv I_1 \cup I_2$, we would have a lattice. (This is the so-called dual of the familiar lattice of subsets. The dual of a lattice is the result of exchanging lub and glb everywhere.) Instead for this example we define
>
> $$I_1 \sqcap I_2 \equiv I_3 \text{ where } [a_1,b_1] \sqcap [a_2,b_2] \equiv [\min(a_1,a_2), \max(b_1,b_2)].$$
>
> This turns out to be a special kind of lattice that Scott called an *information system* (see definition 5 below). It's a lattice used in Numerical Analysis, where the intervals are ranges of uncertainty in measurement.

A lattice is a special case of a partially ordered set, for we can define a partial order $\sqsubseteq$ by the equation $x \sqsubseteq y \equiv x = x \sqcup y$. We say that I_2 is *more informative* than I_1.

> **Example 1, continued.** In our example of rational intervals we can show that $I_1 \sqsubseteq I_2 \equiv I_1 \supseteq I_2$. Knowing that a point is in a smaller interval is more informative than knowing that it is in a larger interval.

2. Complete lattice. A lattice is complete if there is a least element, $\perp$, which we read `bottom,' and the least upper bound of every set of lattice elements is again an element of the lattice. Any lattice can be completed by the simple device of adding another element, $\top$, which we read `top,' which is the lub of any set of otherwise incomparable elements.

> **Example 1, continued.** The single interval Q of all rational numbers will do for $\perp$. It is the least informative of all intervals. The least upper bound in this example is intersection of intervals. Since the intersection of any arbitrary set of intervals of rationals is an interval of real numbers, we can complete the lattice of Example 1 in a familiar way by allowing our intervals to be intervals of real numbers.

Alternatively, we could adjoin in Example 1 a single arbitrary element $\top$ and define the least upper

bound of any set of rational intervals whose intersection is not a rational interval to be the element $\top$. This kind of choice is analogous to the two possibilities in topology of compactifying the real numbers: by adjoining $-\infty$ and ∞ to the reals, or by using a one-point compactification.

We can press the analogy of this example further. Since every partially ordered set has a topology induced by the order, every lattice also has a topology. However, the topology induced by $\sqsubseteq$ is too fine a topology for our purposes. That is, it has too many open sets. Here is the topology that we shall use.

3. The Scott topology. Under $\sqsubseteq$ define basic open sets to be $O_x \equiv \{y: x \sqsubseteq y\}$. We say that O_x is all of the elements of the partially ordered set that are "more informative than x." Now as usual take finite intersections and arbitrary unions of basic open sets to generate all open sets. We shall call this the "Scott topology" on a lattice.

> **Example 2**. Consider the power set of the natural numbers, P(N) as a lattice under set inclusion, $\subseteq$. If we start with $O_{\{n\}} \equiv \{X: n \subseteq X\}$. where $n \in N$, the natural numbers, as a basis, then the topological space P(N) of subsets of N that we get in this way will not be the usual Cantor product topology on 2^N ($=\{0,1\}^N$, which is isomorphic as a set to P(N) via a characteristic function). In fact this Scott topology will be a T_0 separable space but not necessarily even T_1 separable.
>
> This Scott topology on P(N) is determined by finite positive information. But the usual product topology is determined by positive and negative information; for example, $\{X \in P(N): 7 \in X, 13 \notin X\}$ is open in the Cantor topology, but not in the Scott topology. See Barendregt [1985, p. 470] or Rogers [1967, p. 217].

4. Countably generated. A countably generated lattice is one in which there are at most only countably many basic open sets.

> In Example 1, there are uncountably many real intervals in the completed lattice, but it is generated by only countably many intervals of rationals, so it is a countably generated lattice.

In fact the lattices that are needed in what follows are not only countably generated, but every element has only a finite chain of elements below it in the partial order. That is not true of the real intervals in Example 1.

> In Example 2, there are uncountably many sets in P(N), but P(N) is a countably generated lattice since there are only countably many elements in the basis.

5. Information system. An information system is a countably generated complete lattice with a top, $\top$, and a bottom, $\bot$, topologized by the Scott topology. We shall call $\top$ "overdetermined" and call $\bot$ "underdetermined" or "completely uninformative."

> Both Example 1 and Example 2 are information systems. In practice, we shall be interested not only in lattices of sets of numbers, but in lattices of functions with domain and range in such lattices.

6. Continuity in a lattice.

Define a *monotone* function $f: D_1 \longrightarrow D_2$ to be one which preserves the partial order:

$$x \sqsubseteq y \text{ implies } f(x) \sqsubseteq f(y) \quad \forall x,y.$$

The monotone continuous functions on a lattice are well-behaved with respect to directed sets, where continuity is defined in the Scott topology. We have the following

Theorem: f: D_1 ——> D_2 is monotone and continuous if and only if

$$f\left(\bigsqcup_{i=1}^{\infty} x_i\right) = \bigsqcup_{i=1}^{\infty} f(x_i).$$

See Rogers [1967, p. 217].

7. Cartesian products

$D_1 \times D_2$ is defined by $(x_1,x_2) \sqsubseteq (y_1,y_2)$ if and only if $x_1 \sqsubseteq x_2$ & $y_1 \sqsubseteq y_2$

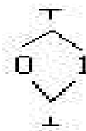
8. Sum (disjoint union)

$D_1 + D_2$ is defined by $(x_1,\bot_2) \cup (\bot_1,x_2)$, factored out by an equivalence relation to assure that the bottom of ordered pairs $\bot_{12} = \bot_1 = \bot_2$ and the top of ordered pairs $\top_{12} = \top_1 = \top_2$.

9. More examples of data structures as lattices. I've named the first two, B for Boolean and Z for Integer, so as to use them in what follows.

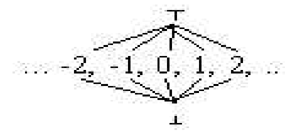
Example 3

B, BOOLEAN



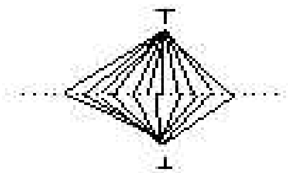
Example 4

Z, INTEGER



Example 5

REAL



(not countably generated, hence not an information system)

In fact, every data structure in every computer language has been modeled by an information system. (The data type REAL in a computer program is really made up of intervals of rationals, and so is more like Example 1 than Example 5.) If they are not recursive data structures, then the lattice is “flat”; that is, only top and bottom are comparable with other elements. Examples 3, 4, and 5 are flat lattices.

The point of what follows is to represent computer *programs* as well as data as elements of a lattice, so that we can leverage the power of mathematics in proving things about them. Here is how we do that.

10. Functions as lattices. The monotone continuous functions themselves form a lattice in our original example, call it D_1——>D_2. The operations are defined pointwise.

g, f: D_1 ——> D_2 is defined as $f \sqcup g\,(x) = f(x) \sqcup g(x)$ and
$f \sqcap g\,(x) = f(x) \sqcap g(x)$

Then “g is more informative than f,” $f \sqsubseteq g$ is defined as $f(x) \sqsubseteq g(x)\ \forall x \in D_2$.

Theorem: D_1——>D_2 is an information system if D_1 and D_2 are.

We must show that D_1——>D_2 is a countably generated complete lattice. The hard part is to show that it's countably generated.

Here's the idea of the proof, based on Example 1: The action of a monotone continuous f on the real numbers in the lattice is determined by its action on the rational intervals; that is, if you know where the rationals go, then you know where the reals go. But it only takes a countable amount of

information to know where the rationals go.

11. Partial recursive functions.

Primitive recursive functions are a class of functions from the natural numbers (non-negative integers) N into N itself. They are defined as the smallest class of functions including constant functions, the identity function, and the successor function, and closed under function composition and under recursion; that is, f is in the class if g and h are where f(0) = g(x) and f(y+1) = h(y, f(y), x). To get the class of *recursive functions,* we also include the function f(y) = the smallest y such that g(x, y) whenever h is in the class.[2]

The *partial recursive functions* are recursive functions defined on only some of N. By letting $\perp$ represent an element which is the value of a partial recursive function when it is given a natural number outside its domain, we can think of partial recursive functions as being (total) functions from N ——> N+{$\perp$}. We say that $\perp$ represents "undefined."

Kleene created the idea of partial recursive functions to present an account of functions that could be computed in a purely mechanical way. That explains our special interest in such functions. I have chosen Kleene's characterization as over against those of others (Church, Markov, Post, or Turing) because it is clearly equational. So it is purely declarative, not procedural. And equations have an important substitution property, called `referential transparency' by linguists: if equals are substituted for equals, the results are equal. See Rogers [1967, p. 18].

Example 6. The fibonacci function is a partial recursive function:

$$f(0) = 1,\ f(1) = 1,$$
$$f(x+2) = f(x+1) + f(x), \text{ for } x > 1.$$

It can be shown to be a partial recursive function by showing separately that the following functions are primitive recursive if j, m, and n are:

MinusOne(x) = x-1
MinusTwo(x) = x-2
IfThenElse (j(x), m(x), n(x)), meaning
m(x) if j(x)=0, otherwise n(x).

Then f can be defined in terms of these for $x \geq 0$, and undefined for $x < 0$.

B. Theorems

1. Partial recursive functions. The partial recursive functions are an information system.

This can be proved directly. There are countably many, so they are certainly countably generated. They form a lattice under pointwise operations. The totally undefined function $f(x) = \perp$ for all x is the bottom function. As usual we assume that a top has been added to complete the lattice.

Partial recursive functions are important because they are a useful model of functions in. say, Pascal, but they are also mathematical objects about which theorems can be proved.

Example 2, continued. If all we care about is partial recursive functions, we do not need the full generality of Scott's definition of an information system in terms of a countable basis. We

---

[2] Using the least number operator, μ, we write $f(x) = \mu y\ g(x,y)$.

can easily restrict our attention to a so-called recursive basis. The reason is contained in the following theorem. Define a recursively enumerable set of natural numbers as a set that is the range of a (total) recursive function. Define a recursive set of natural numbers as one that is recursively enumerable, and its complement is also recursively enumerable. Then, using the countable basis defined in Example 2, $O_{\{n\}}$, if we restrict n to some recursive set, then the monotone continuous functions are all recursive functions.

2. Category theory

An information system is a Cartesian category with sums, and in fact projective (inverse) limits exist in the category sense. Consider for example D_0 = integers. Then $D_1 = D_0$——>D_0 is functions from integers to integers. The sequence of $D_{i+1} = D_i$——>D_i can each be viewed as containing the previous via a projection. For example, identify the integer k with the constant function f(x) = k. The limit D_∞ of the D_i then has the property that $D_\infty = D_\infty$——>D_∞. The beautiful fact is that D_∞ is also a lattice. In fact it is also the direct limit of the D_i, which makes everything about as nice as you could want. In fact $D_\infty = D_\infty$——>D_∞ both topologically and algebraically.

In particular, the partial recursive functions form a Cartesian category.

Scott shows that D_∞ is a model of the lambda calculus—exactly what Strachey needed!

3. Knaster-Tarski theorem.

Any monotone continuous function from a complete lattice D to itself has a **fixed point**. Moreover, there is a continuous functional Fix in the lattice (D——>D) ——> (D——>D) such that for all f in D——>D, Fix(f) is the least fixed point of f.

This is an amazing and powerful uniformity! We shall use only a small part of its power, to guarantee that a function exists even though it is defined only on larger and larger parts of its domain.

IV. The historical perspective: What is semantics?

A. Syntax and semantics may seem arbitrarily chosen at first.

Syntax is the rules for manipulating formal symbols regardless of what they mean. Semantics is an attempt to give them meaning, whether informally in English or formally. When I first heard about formal semantics, I thought that it was an exercise in futility: what point could there be in inventing another formal system to explain a formal system that is at hand? That would mean that the semantics of one formal system would in turn require its own semantics! That could only be helpful if the semantics was somehow clearer or easier to work with.[3] The semantics of a computer program is harder to understand than the program itself. Are there not other criteria for picking a semantics?

Here is a contrast between syntax and semantics to clarify their connection. Syntax is formal and algebraic; semantics is intuitive and geometric. Syntax is finitary, although not necessarily computable; semantics is infinitary, manipulating whole infinite sets at a time. Proof theory is the branch of mathematics that manipulates syntax; model theory is the branch of mathematics that manipulates semantics. It is possible for a formal syntax to have no models. That was the problem with Strachey's use of lambda calculus as a semantics of programming languages.

A map from syntax to semantics is called an `interpretation.' We say that a sentence in a given syntax is `satisfied' in a model if the relationship that it claims to hold is true of the model. For example, Z, the integers, are a model for the axioms of a group. Thus we are assured that the group

---

[3] I have argued that a Christian philosophy of mathematics should have as its first premise that no system is capable of being its own semantics. See Chase [1981] page 84 or Chase [1987] page 238.

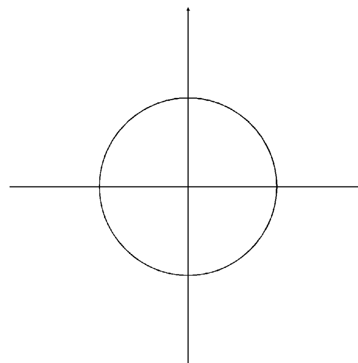
axioms are not contradictory. But the rotations of a square also form a model for group axioms. What criteria do we use for choosing models for syntax? Let me suggest three that have historically been used.

B. Historically there have been three criteria for semantics.

The major contribution of this paper is found right here. No references, even the most elementary for junior Computer Science majors (say, Sebesta [1993]), give even the slightest hint of any criterion for the semantics of a computer program. Even my historically motivated presentation (in Section II above) of the three major[4] kinds of semantics does not provide any overarching criteria for valuing one as over against another. For that, we need to look at the range of semantics that have been proposed in other areas than Computer Science.

To start with a specific example, consider quadratic equations in two unknowns as in high school algebra. We may think of coordinate geometry then as an appropriate semantics. Here is a sample:

Example 7. Algebra and Coordinate Geometry, 17th century

Syntax	**Semantics**
$x^2 + y^2 = 1$. *Equivalently,* $x^2 + y^2 - 1 = 0$	

The meaning of either equation is a circle of radius 1 centered at the origin.
Allowable syntactic rules: the "laws of algebra."

Syntactic rules allow us to transform one equality into another equality. For example, we may add the same quantity to both sides of the equation. We would like to say that the resulting two equations `mean the same thing.' To make that precise, we could say that two equations mean the same thing if their graphs are the same. This small example already suggests two criteria for an appropriate semantics: semantics should be canonical, and semantics should be geometric.

1. Semantics should be Canonical.

By `canonical semantics' I mean that if there are several meanings of an algebraic equation, we should be able to pick one (the `canonical' meaning) and any other possible meanings should be able to be shown to be equivalent to it. One easy but unintuitive way to do that would to be to let the meaning of a quadratic equation in two unknowns be simply the list of numbers A, B, C, D, E, F in

---

[4] I do not deal with algebraic semantics and action semantics. For an overview of them, see the recent graduate text by Slonneger and Kurtz [1995]. The first can be viewed as a variation on denotational semantics; the second can be viewed as a merger of the best features of denotational and operational semantics. Thus neither provides any additional insights relevant to the point of this paper.

some canonical representation of the quadratic, say,

$$Ax^2 + By^2 + Cxy + Dx + Ef + G = 0.$$

In that way, if we wanted to tell whether two equations had the same meaning, we could apply the laws of algebra to transform them both into this canonical form, and then see whether they had the same set of coefficients.

2. Semantics should be Geometric.

An equation in two unknowns represents a restriction on the values that the unknowns can take. We want an equation's meaning not to change if the solution set does not change. Why then don't we just take its solution set to be the meaning of an equation? In that way we have a pictorial or geometric representation of the meaning of the equation.

We are reversing the steps of Fermat and Descartes in the 17th century who sought to reduce to calculation the intuitions of (synthetic) geometry by creating an analytic geometry. We instead are interested in restoring intuition to calculations. But does this criterion of being geometric apply to other examples of semantics? What about set theory as a semantics for logic? What might a semantics for English look like?

Example 8. Set theory as a semantics for logic.

Syntax	**Semantics**	
Propositional Calculus	Boolean Algebra/ Sets	18th century Euler, Venn, Boole
Predicate Calculus	Cylindrical Algebra	20th century Tarski, Herbrand, Halmos

Allowable syntactic rules: “laws of logic,” deduction. For example, modus ponens, universal instantiation, existential generalization, and-elimination, or-introduction.

What happened to the geometry in this example? The semantics still has a geometry, suitably generalized to topology.[5]

⌚ Boolean algebra has a natural lattice topology in the sense described in Section III. Paul Halmos created cylindrical algebra just to prove that predicate calculus can be given a topology, too. I'm rather a crusader for wishing that his work were more widely known. In cylindrical algebra, $\exists$ is a projection operator in the usual geometric sense of projection. I believe that it adds an excellent level of intuition to predicate calculus.

3. Semantics should be Compositional

We would like to be able to determine the meaning of a complicated expression by putting together in a systematic, uniform way the meanings of its pieces and the meaning of the rules for combining the pieces. We say that we want our semantics to be `compositional.' One benefit of a compositional semantics is that if equals are substituted for equals, then the results are still equal. We call this property “referentially transparency.”

⌚ Recall that in Section III.A.11 we argued that referential transparency motivated our preference for an equational rather than a procedural mathematical model of computable functions.

If English were the syntax, we would want the semantics of a sentence to be made up solely

---

[5] Kuyk's [1977] view of the discrete and continuous as fundamental complementarities informs my view philosophically at this point.

of the semantics of the noun phrases, verb phrases, and other pieces making up the sentence. Montague [1950] made some major steps in doing that for English. Here is an example:

Example 9. English and Predicate Calculus

Syntax	**Semantics**
John gives someone the ball. *Equivalently,* The ball is given by John.	(∃x) give (john, x, ball)
Everybody loves somebody.	(∀y)(∃x) loves (x, y)
There is somebody whom everybody loves. *Equivalently,* Somebody is loved by everybody.	(∃x)(∀y) loves (x, y)

Allowable syntactic rules: paraphrasing.

You can see that the semantics of this example is the syntax of the previous example.[6]

What happened to the geometry? This time it is workers in Artificial Intelligence who have geometrized the predicate calculus, using several tricks to create something called a semantic network. Quantifiers (∀, ∃) are eliminated using a trick of Thoralf Skolem's. That two nouns mean the same thing is shown by pointers to unique names instead of duplicating the reference to the name. So for example "John loves himself," which in predicate calculus is *loves(john, john)*, becomes in a semantic network

john ⇄ loves

To achieve referential transparency for English with predicate calculus as its semantics requires that we quantify over functions as well as over variables, however. So-called first-order predicate calculus is not sufficient.[7]

V. How do the three kinds of semantics for computer programs fare under these desiderata?

A. Operational semantics is too low-level.

Operational semantics does not abstract far enough away from a target machine to be useful. But more importantly, none of the three criteria for semantics are met by an operational approach. An operational semantics is not geometric (intuitive), is not canonical (because no unique low-level

---

[6] Actually, Montague used lambda calculus instead of predicate calculus. Their difference is not important for this paper, as van Emden & Kowalski [1976] show, but lambda calculus is to be preferred if you plan to follow up on this paper with readings from the references, because it treats functions as data. I aim by avoiding lambda calculus to make this paper more accessible to the working mathematician.

[7] An example will help here. We know that the heavenly body called "the morning star" and the heavenly body called "the evening star" are both the planet Venus. If you were to try to teach that to someone else you could say, "The morning star is the evening star." But to substitute "the evening star" for "the morning star" would make this sentence say something quite different indeed: "The morning star is the morning star." We have turned an experimental fact into a tautology. This shows (and the example is an old one from W.V.O. Quine) that a purely extensional semantics of English (where noun phrases are equal because they refer to the same object in the real world) is not adequate. Various intensional (that is, non-extensional; not to be confused with intentional) versions of English semantics have been proposed. If we express them in predicate calculus, they require that we quantify over functions as well as over variables. That is why Montague [1950] used lambda calculus instead of predicate calculus in his intensional formulation.

program can lay claim to be *the* meaning of a high-level program), and is not compositional (because local changes can have global effects, about which more below).

The main benefits of operational semantics that other candidates for the semantics of a computer program did not have historically are now true also of denotational semantics: they can be generated automatically by a computer from the syntax, and they can be relatively easy to understand.

B. Axiomatic semantics is too high-level

The main problem with axiomatic semantics is that it is both predicate calculus assertions (specifications) and pieces of code. So it never fully abstracts itself from the code. Although it too can be generated automatically (Loeckx & Sieber [1987]), the resulting solution is inscrutable. Like the proof of the Four Color Theorem, we have computation but no insight. But the *point* of semantics is insight! A final criticism: axiomatic semantics does not handle non-termination (infinite loops) very well.

Axiomatic semantics can be hand-implemented with very intuitive predicates. But it is very hard to use for large programs. The only example of its systematic use that has found its way into the typical undergraduate curriculum is the method of “loop invariants” urged on Computer Science educators by David Gries and others. (See textbooks by Gries [1981], or Dale and Lilly [1988] for example). Even in that case, most curricula deal only with a simpler form of that: requiring that the student be able to answer the question, `What are all the ways in which this loop can exit?' Here is a sample from Dale and Lilly [1988].

Example 10. A comment specifying a “loop invariant”: that is, the axiomatic semantics of the loop.

```
Sum := 0;
Index := 1;
NonZeroCount := 0;
{Loop invariant:  Index may range from 1 .. NumValues AND
            NonZeroCount may range from 0 .. 9 AND
            NonZeroCount is the number of non-zero elements in
                  List[1] .. List[Index-1]}
WHILE (Index <= NumValues)
      AND   (NonZeroCount < 10) DO BEGIN
      IF List [Index] <> 0 THEN BEGIN
            Sum := Sum + List[Index];
            NonZeroCount := NonZeroCount + 1
            END;
      Index := Index + 1
      END;
```

In this example, the specification statements are in English. A proof of invariance of these statements as the loop is executed would proceed informally (that is, by hand rather than by machine verification) using mathematical induction on the length of the list.

Most first-year undergraduate Computer Science majors should be able at least to give all of the reasons that the loop might terminate, and therefore all of the possible states of variables just after the loop is finished. Not all Computer Science curricula require proofs by mathematical induction of loop invariants.

C. Denotational semantics meets all of the criteria mentioned for a good semantics.

Denotational semantics has been proved to be equivalent to both of the other two kinds of semantics; that is, functions have been defined in many useful cases to convert denotations back and forth to the abstract machine code of operational semantics and to the logical specifications of axiomatic semantics. If we want code to tag along, we may have it, and so be closer to axiomatic

semantics; if we want to include things called stores, environments and continuations, we may have something closer to operational semantics. (In fact we must have these things to handle global variables, error handling, and procedures with parameters.)

Denotational Semantics is the representation of computer programs and data by mathematical objects. Apart from a few small problems[8], it seems to be an ideal vehicle for proving programs to be correct. Let's look at several examples to get more insight.

1. The simplest data example: a non-recursive data structure

Consider the following interpretation of integers in Pascal. Read M as "is interpreted as" or "means mathematically."

M (`16`) = 16

This seems trivial and uninteresting. But in Turbo Pascal we would also write the following, where `$` means hexadecimal base.

M (`$10`) = 16

Now it is obvious that we intend the integers on the left to be syntactic objects in the computer programming language. We intend that they mean, on the right, integers, elements of the set Z which can be represented geometrically along the number line.

This simple example suggests two points of view that we could take about representing our mathematical objects. We could purposely choose a notation which always shows that they are different from the syntactic objects that they model, or we could[9] whenever possible choose a notation for the mathematical object that is identical to the syntactic object, treating the confusion as an asset to simplify things, trusting you to be able to supply the information of whether it is the syntax or the mathematical object that is in view. I think that the latter is clearer.

If we could *always* do that, the subject of program semantics would have no substance, but there are many things that computer programs should mean that aren't represented in the syntax (like stores, environments, and continuations). Already with representing simple data structures, there is a problem: the operations on `integer` should be represented modulo `maxint`; the operations on `real` should be appropriately approximate arithmetic. Pascal `integer` is not mathematical integers; Pascal `real` is not mathematical reals.

2. A harder data example: a recursive data structure

At least in defining M on the integers, a finite thing is represented by a finite thing. When we come to describing a linked list, even a finite list must be represented by an infinite object mathematically. Consider the Pascal data type `LinkedList` as follows. What should it mean mathematically?

```
type LinkedList = ^ Node;
     Node = record
          n:    integer;
          link: LinkedList
          end;
var  x: LinkedList;
```

---

[8] It does not handle non-determinism (code which potentially could be executed in parallel) very well yet.

[9] Following Herbrand's model of the predicate calculus.

We describe the meaning of a linked list of integers as a mathematical object L with the following property, where N* = the set of all integers together with the null list.[10]

$$L = N^* + N^* \times L$$

Is there such a mathematical object L? No, if equality is to mean equality as sets. Yes, if it represents an equivalence operation of a certain kind. We call L (and N, too) a *denotational domain.* A list is then interpreted as an element of this domain. What kind of equivalence operation could = be so that domains could be sets? Dana Scott's contribution to the field of denotational semantics is to describe L geometrically.

⌚ More precisely, = is topological equivalence of infinite lattices. More discussion of such lattices in Section VI below.

Equations like $L = A + A \times L$ allow for the possibility of infinite data structures. Of course we would never want to print one out, but using them is easy. For example a circular list `[1, 1, 1, 1, ...]` can be defined in Pascal. Data-driven (so-called "lazy") evaluation of such an expression would only generate as much of the list as needed at a time. Another example of a potentially infinite data structure is an input stream from the keyboard.

The point of this linked list example is that the meaning of even finite data structures, if their definition is given recursively, necessarily require and hence allow for potentially infinite data structures.

3. Comparing an easy and hard program example: recursion versus iteration

As Bohm and Jacopini showed, according to the folk wisdom[11], any computer program can be written using only sequence, `if` and `while` as control structures. Call such programs "flowchart programs," after Scott [1970]. Consider the following Pascal functions. The first is *not* a flowchart program because it involves recursion; the second one is.

Example 11.

```
function factorial1 (n: integer): integer;
begin
      if n = 0     then factorial1 := 1
                   else factorial1 := n * factorial1 (n-1)
end;
```

Example 12.

```
function factorial2 (n: integer): integer;
var   p, i: integer;
begin
      p := 1;
      i := 0;
      while i < n do begin
            i := i + 1;
            p := p * i
            end;
      factorial2 := p
end;
```

Note first that `factorial2` uses assignment in an essential way, but `factorial1` does not. There is an assignment statement, but it wouldn't be necessary if we used an Algol or C language

---

[10] Note the analogy here with a context-free grammar statement.

[11] They don't deserve the credit, claims David Harel [1980] in an entertaining article that is deep both historically and technically.

syntax for `return` instead of Pascal's syntax, and if `if` returned an expression rather than relating statements as it does in some computer languages, so we could write the following as the body of the recursion.

```
return if n = 0 then 1 else n * f(n-1)
```

We want `factorial1` and `factorial2` to have as their meaning elements of N——>N, that is, some mathematical functions from N to N, where we avoid writing the set as N^N to emphasize that not all the uncountable number of functions from N to N are to be selected, but only those that preserve some structure as I described in Section III above.

Define a map M from Pascal to mathematics such that M(`factorial1`) = *factorial1*, the mathematical factorial function. Let *IfThenElse*: B x N x N ——> N be a function of three variables, one Boolean and two Integers with the following property

IfThenElse (true, a, b) = a
IfThenElse (false, a, b) = b.

Then define *factorial1* = *IfThenElse* (n=0, 1, n * *factorial1* (n-1)).

⌚ This is a recursive definition. It is only a slight variation on the definition of Example 6 in Section III: *factorial1* is a partial recursive function.[12]

Here we assume that *factorial1* is well-defined, and that the semantic function M does what we hope it will on the Pascal operators `-`, `*`, `=`, and on function application. Then this definition of the meaning
M (`factorial1`) = *factorial1* will be compositional in the sense described above.

In the case of `factorial2`, the situation is a bit more complicated, since `while` is not a mathematical construction as are equality, multiplication, subtraction, and functional application. But it can be shown that the following equations give a `while` loop a mathematical meaning as a function:

M(⟨A⟩; ⟨B⟩) = M(⟨B⟩) ∘ M(⟨A⟩)
M(`while` ⟨A⟩ `do` ⟨P⟩) = *IfThenElse*(M(⟨A⟩), M(⟨P⟩; `while` ⟨A⟩ `do` ⟨P⟩), id).

Here ⟨A⟩, ⟨B⟩ and ⟨P⟩ are metavariables ranging over pieces of program; the semicolon is the Pascal sequence operator, ∘ is function composition, and id is the identity function id(x) = x. Since the meaning is given recursively, this definition of M on `while` has two problems. First, it violates compositionality. The meaning of a `while` loop is not defined solely in terms of the meanings of its pieces ⟨A⟩ and ⟨P⟩. Second, what right do we have to say that there is any function, call it loop, at all that satisfies this recursion relation, so that M(while ⟨A⟩ `do` ⟨P⟩) = loop, and if so, that there is a unique one?

⌚ That is exactly what is guaranteed by showing that the functions which are the interpretations of Pascal functions form a complete lattice. Then the function loop can be show to be a fixed point of a sequence of functions approximating it topologically, guaranteed to exist by the Knaster-Tarski Theorem. A specific example of such a sequence of functions is given after Example 13 below.

---

[12]Appropriately extending the definition of recursive functions to Boolean domains. The arguments are mostly combinatorial in nature, and straightforward.

In English, these equations say what we know intuitively that there are two cases in executing a while loop. Either the condition is false, so we do nothing, or it is true and we execute the loop once and then ask the question again. We interpret the meaning of an iteration to be a recursive function that does the same thing as the iteration. In this respect, we are working backwards as compared to what a compiler does in replacing recursion with hardware iteration. We are replacing iteration with recursion.

I did not say what the meaning of the assignment statements was because that would be too hard for an introductory paper. Here though is a rough idea. An assignment statement could globally change any memory location in the computer, hence if its meaning is to be regarded as a function, then the function must map not just from integers to integers as the mathematical factorial function does, but it must map from integers and **stores** to integers and stores, where a store is the mathematical representation of the memory locations of the computer. That is why, as I said above in Section II.B, assignment statements are the spaghetti-code of data structures. You probably already knew that about the pointer data type (pointers can point anywhere), but since variables in a Pascal function might be reference (i.e. VAR) or value parameters, the same argument holds for variables in general. (In fact, value parameters give more problems denotationally than reference parameters, but that discussion must await another time.)

⏲ VI. Denotational semantics as an information system

This Section assumes Part III above, and as such should be skipped on a first reading.

A. The functions which are the meanings of computer programs form a lattice.

I have skipped over the fact that the functions which we have been discussing above are not functions whose domain is N, but only some of N. We say that such functions are “partial” functions. It will be useful in what follows to assume that a partial function is a total function from $N+\{\perp\}$ to $N+\{\perp\}$ where $\perp$ represents in this case “divergence.” We have added $\perp$ to the domain, with the stipulation that $f(\perp) = \perp$ for all f.

Now I am prepared to tell you how the meanings of computer programs as functions from $N + \{\perp\} \longrightarrow N + \{\perp\}$ form a lattice. It's enough to define the partial order, $\sqsubseteq$. Consider the Pascal function `mystery` defined as follows:

Example 13.
```
function mystery (n: integer): integer;
begin
      if n = 0     then mystery := 1
                   else mystery := n * f (n-1)
end;
```

where f is a function which computes the same thing that `factorial1` does for n < 100, but for n >= 100, f diverges (loops infinitely).

We shall say that $F \sqsubseteq G$ if G “computes more” than F. That is, $F(x) = G(x)$ for all x such that $F(x) \neq \perp$. Or, F diverges everywhere that G does, and possibly at other places as well. Call the meaning of `mystery` *mystery*; that is, M(`mystery`) = *mystery*. Then *mystery* $\sqsubseteq$ *factorial1*. This reminds me of the room full of monkeys at typewriters one of which typed Hamlet's soliloquy beginning “To be or not to be; that is the igshmifence.” Shakespeare and the monkeys agree for some starting values. *mystery* and *factorial1* agree for some starting values of n, and where they disagree, the function *factorial1* computes more than the function *mystery*.

Now $\sqsubseteq$ is a partial order on a set of mathematical functions from $N+\{\perp\}$ to $N+\{\perp\}$. Is the partial order a lattice? Indeed, it is even an information system. Here I am using $\perp$ in the range to mean that the corresponding Pascal function diverges. The appropriate set of mathematical functions

to model Pascal functions is the set of partial recursive functions.

Define the mathematical function *mystery* as follows. Notice that there is a different function *mystery* for each choice of the function f.

$$mystery\ (n) = IfThenElse(n=0,\ 1,\ n * f(n-1))$$

Define a functional Φ on the domain of partial recursive functions, the domain extended to require that for every recursive function p, $p(\perp) = \perp$,

$$\Phi: (N+\{\perp\} \longrightarrow N+\{\perp\}) \longrightarrow (N+\{\perp\} \longrightarrow N+\{\perp\}),$$

as follows. Φ(f) = *mystery*, where of course *mystery* depends on f. Then Φ(*factorial1*) = *factorial1*, which is to say that *factorial1* is a fixed point of Φ.

I had to include ⊥ in the range because f is not in general defined on all of N, just as factorial is not defined for negative integers. I claim that ⊑ and ⊥ and ⊤[13] together define a lattice to which all of Section III applies.

A Pascal function like `factorial2` that is iterative instead of recursive can be given as a meaning a mathematical function factorial2 which can be defined as a limit of functions like *mystery* which agree with factorial2 on larger and larger initial segments of the natural numbers, just as the data structure [1, 1, 1, 1, ...] is the limit of finite data structures.

Contrast the operational approach to semantics with the denotational approach on this point. In the operational approach, since the hardware with which we are familiar is good at iteration and not designed for recursion, we model recursive operations by iteration (with the help of a stack of activation records to keep track of the recursion) so that they can be executed. Conversely in the denotational approach, we model iterative operations by recursive functions, so that we can apply mathematical techniques to the results.

B. To model computer programs denotationally, certain things are not semantically simple.

1. Functional application must allow for the possibility of a function being in its own domain.

The way to handle this semantically is to treat functions and data in a uniform way, which so-called functional languages do well. In a functional language, functions are “first-class objects.” They can not only be passed as arguments (even Pascal permits that) but they can be returned as values of another function (which Pascal does not permit).

Is this necessary if you don't otherwise want to pass functions as arguments? Yes. Here's an idea why. You have probably been taught as a computer programmer that recursion is inefficient, so you are reluctant to use it. It need not be. A sequence of recursive calls can blow a computer's stack of activation records. But suppose that the recursive call is the last thing in the function. Then a good compiler can convert this so-called tail-recursion into iteration automatically. Then there is no need to return to the calling procedure for further computation, so there is nothing to save on the stack. In `factorial1` of Example 11, the recursion is tail recursion. It can automatically be converted into

---

[13] Which we throw in just so that we have a lattice. The justification for including it is beyond the scope of this paper. You may read it “overdetermined” and assume that it is just tacked on at the top of everything else just so that I can talk about lattices instead of about so-called complete partial orders, which would make ⊤ unnecessary. This is like solving the equation $x^2-2=0$ in the reals to avoid solving it in the smallest possible extension of Q, namely $Q(\sqrt{2})$. Sometimes it's clearer pedagogically to have more structure than you need.

`factorial2` of Example 12.

What if there were a way to make recursion efficient even if it isn't tail recursion? The solution in a language which allows functions as first class data is to pass to a function not only the data needed, but a function, called in the literature a **continuation**, which is the rest of the computation to be done by the calling procedure! Then there will be no need to return to the call point in the original function to continue: we can just continue using by the continuation if it is necessary.

I said “if it is necessary” because there is one case where the continuation is irrelevant, that of the code following a `go to`. So `go to` allows the continuation to be discarded right away.

As a simple special case of a continuation which doesn't even require functions as parameters, consider just saving the resulting computation that the continuation would do in an accumulator. Then factorial becomes a function of two arguments, an integer and an accumulator. Here is what such a version might look like.

Example 14.
```
function factorial3 (n: integer):  integer;
begin
      accumulator := 1;
      factorial3 := factor (n, accumulator);
end;
```

where

```
function factor (var n: integer; var cont: integer): integer;
begin
      if n = 0     then factor   := cont
                   else begin
                         n    := n - 1;
                         cont := n * fact(n,cont)
                         end
end;
```

Continuations show that recursion, even if not tail recursion, can be handled efficiently by current hardware architecture. As Guy Steele says in an MIT memo, “Lambda, the ultimate imperative,” a machine based on recursive functions need not be inefficient.

From Section III above, we see that any information system D can be embedded in a function space D_∞ with the property that $D_\infty = D_\infty \longrightarrow D_\infty$. So a mathematical model is available for such functions as continuations which might have themselves in their domain.

2. Pascal programs are not referentially transparent.

That is, if equals are substituted for equals, the results are not necessarily equal. In Pascal for example,

```
if f(3) = f(3) then writeln ('Here') else writeln ('There')
```

may not write 'Here' every time, should `f` have the side-effect of updating a global variable. As a consequence, Pascal can't be its own semantics if we want our semantics to be compositional.

This frustrates the goal of program transformation, for the following two statements are only equivalent in Pascal if `f` doesn't update `x`:

i.
```
if f(x) then y := g(x) else y := g(x)
```

ii.
```
y := g(x)
```

From a theoretical point of view, the way to handle this problem semantically is to bring the whole **environment** along in the denotation of a Pascal function `f` if necessary.

From a practical point of view, this is the same kind of problem that keeps execution on a parallel machine from being easy.

3. Infinite objects are necessary for functions as well as for data.
How does the modest finite syntax of computer programs get to require such complicated infinite objects to model them as topological spaces of monotone continuous functions? The reason for the difficulty and the obscurity of the subject of denotational semantics is much the same as the crisis in mathematical logic which Russell's Paradox created in 1905: the functions that we intuitively want are functions which have the property that they include themselves in their domain. Like sets that contain themselves as members, or barbers who shave all those in town who don't shave themselves, something has to give. Traditional set theory doesn't allow that. What Russell and Gödel did to Frege's set theory[14], Scott did to Strachey's denotational semantics. An intuitive idea was at hand, but when the formalization was made, the story was much more complicated than anticipated. Scott's solution to the problems of denotational semantics uses infinite objects so that they can have the property that they are isomorphic to a proper subset of themselves.

Here is one more version of a factorial function that I shall write with Pascal syntax, but which is illegal in most versions of Pascal. It is not illegal in various functional languages like TLC Lisp or in Scheme. It should not be illegal in a modern computer language.

Example 15.
```
function factorial4 (n: integer): integer;
begin
      factorial4 := f (f, n)
end;
```

where

```
function f( function g: integer; m: integer): integer;
begin
      if m = 0 then f := 1
             else f := m * g (g, m-1)
end;
```

Here I have made the **continuation** g of f explicit. As you can see, a function might very legitimately in this theory have itself as one of its arguments.

VII. The moral of the story: a defense of functional languages

A. "I have a dream." Several style points in traditional programming languages are really folk wisdom about writing programs that are easy to prove correct. Use them! One of the goals of research in semantics of computer languages in my estimation is to eliminate the need for advice

---

[14] I am thinking of Von Neumann, Bernays, and Gödel's set theory, of various type theories, hereditarily finite models, and so on. One conference participant pointed out that Russell's famous 1905 letter to Frege was actually anticipated by Zermelo in 1902.

about style by creating

a language in which only programs in good style can be written!

Here are some examples of good style points that have as their underlying motive a desire for clean (easy to understand, unambiguous) semantics.

Avoid `goto`
Be clear not clever
Avoid global variables
Isolate input and output in their own separate procedures
Select the highest level programming language that you can which most matches the problem domain
Don't modify loop variables
Avoid implementation-dependent features

—and others that you could add

B. Myths about functional languages

1. A functional language is computationally expensive. Not necessarily! If they are, it is a problem that can be solved by changing the hardware on which the language is run. These languages are traditionally interpreted rather than compiled just because hardware is not optimized for them, so they haven't received widespread use. Texas Instruments even sells a chip which executes the Lisp dialect Scheme in the hardware, a dialect in which the above plan can be carried out. In fact, semantically clean functional language can actually take advantage of parallelism more easily than current imperative third-generation languages.

2. The functional programming style is hard to understand. Every computer scientist knows about the turgid style of functional programming illustrated by APL one-liners. The intention of APL authors who use one-liners is a good one: to avoid updating individual variables. Functional programs need not be hard to understand, however, as evidenced by the popularity and power of spreadsheets, which rely on such functional constructions as a statement `if`, and which simplify the relationship between store and output by displaying all stores.

3. Functional languages are inflexible. I'm a COBOL programmer, you say, and I need to be closer to the machine hardware, especially as it relates to the variety of input and output formats that I must manage. Perhaps you will think of this paper then as an encouragement to think about moving to a language like SQL. SQL can be optimized to compete with COBOL for speed. SQL is a more functional language: it treats tables of data as whole-data structures.

In conclusion, I have argued that if the semantics of a computer program is ultimately to be useful, then it must conform to the requirements of being geometric, canonical, and compositional. And I have argued that progress will be made in Computer Science if the programming languages of the future move in the direction of being more mathematical (hence functional) to allow them to be semantically cleaner.

References

Allison, Lloyd. *A practical introduction to denotational semantics.* Cambridge, 1986.
Here is where to start in the literature. An excellent introductory text, although the motivation will have to be provided by the instructor. I hope that this paper might be some of that motivation.

Backus, John. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." *CACM 21,* 8, Aug 1978.
Argues for learning to program in a semantically clean functional language. Lecture given on the occasion of receiving the prestigious A. M. Turing Award for 1977. By the leader of the team that invented Fortran.

Barendregt, H. P. *The lambda calculus,* revised edition. Elsevier, 1984.
Very sophisticated mathematically. A landmark reference.

"Curriculum '78, recommendations for the undergraduate program in computer science," *CACM 22,* 3, March 1979: 147-165.

Chase, Gene B. An information processing model of mathematics education. Ph.D. dissertation, Cornell University, 1979.
Argues in one chapter that semantic models of computer programs have pedagogical value. Compare Mayer [1979] and Schneiderman and Mayer [1979] for similar ideas.

Chase, Gene B. "An integration of integrations of Christianity and mathematics—a response to Harold Heie." *A third conference on mathematics from a Christian perspective.* Wheaton College, 1981.
See Chase [1987].

Chase, Gene B. "Complementarity as a Christian philosophy of mathematics." *The reality of Christian learning,* ed. Harold Heie and David L. Wolfe. Eerdmans, 1987.
The meaning of a mathematical system can never be found in the system itself. Revision of Chase [1981], which is cited above because this version dropped several lines of text on page 232, creating some confusion about what follows.

Dale, Nell and Susan C. Lilly. *Pascal plus data structures,* second edition. Heath, 1988.
Illustrates the faddish importance of axiomatic semantics to a second-semester freshman text (course CS 2 in Curriculum [1979]) by including loop invariants in Chapter 1, with a technical appendix about the Floyd-Hoare method. Contrast this with the first edition (1985) in which the same material occurred in Chapter 10, and with more recent texts, which again omit semantics altogether in favor of the latest fad of object-oriented programming.

Department of Defense trusted computer system evaluation criteria. CSC-STD-001-83. DoD Computer Security Center, Ft. Meade, MD, 1983.

Donahue, James E. *Complementary definitions of programming language semantics.* Springer-Verlag, Berlin, 1976.
Carries out the program of denotational semantics for all of the Pascal language, showing how operational and axiomatic semantics complement the denotational approach. Does not skip any of the mathematical technicalities.

Gordon, Michael J. C. *The denotational description of programming languages; an introduction.* Springer-Verlag, 1976.
Covers the same ground as Donahue, but intuitively, skipping all of the mathematics. Aimed at computer scientists who want to know that it works. Argues, as I have in Chase [1981], that historically useful parts of mathematics have been used before they were formally admitted as mathematical objects. Infinitesimals and the Dirac delta function were used before consistent models of them were devised; similarly an intuition of denotational semantics by Strachey preceded its formalization by Scott.

Gries, David. *The science of programming.* Springer-Verlag, 1981.
Argues that axiomatic semantics must be taught from the very beginning to prospective computer programmers, and shows how that might be done.

Halmos, Paul. *Cylindrical algebras.* Chelsea Pub.
Shows that the lattice geometry of propositional calculus can be extended to predicate calculus with ∃ as the projection operator. That is, Boolean Algebra : Cylindrical Algebra :: Propositional Calculus : Predicate Calculus.

Harel, David. "On folk theorems." *CACM 23,* 7, July 1980: 379-389.
Traces the historical development of flowchart programs.

Hennessy, Matthew. *Algebraic theory of processes.* MIT Press, 1988.
Typical of how soon a gentle introduction can slip into obscurity for the practitioner.

Jensen, K. *Pascal user manual and report,* second edition. Springer-Verlag, 1978.
Cited for an example of how Pascal is underspecified in its semantics.

Kitchen, Andrew. "Abstract semantics: making the austere beautiful." *IEEE software.* March 1989: 110.
Review of Hennessy [1988].

Kuyk, Willem. *Complementarity in mathematics: A first introduction to the foundations of mathematics and its history.* D. Reidel, Dordrecht, Holland, 1977.
Philosophical basis of unifying several semantical systems under the heading of geometry, contrasting the discrete (syntax) with the continuous (semantics).

Lee, Peter and Uwe Pleban. "The automatic generation of realistic compilers from high-level semantic descriptions; a progress report; or relief for the downtrodden compiler hacker." CRL-TR-13-86, Univ. of Michigan, June 1986.
Shows that all these high-level generalities are practical after all.

Loeckx, J. and K. Seiber. *The foundations of program verification,* second edition. J. Wiley & Sons, NY 1987.
The Goedel β-function solution to finding pre-conditions and post-conditions automatically.

Lucas, P. and K. Walk. On the formal description of PL/I. *Annual review in automatic programming 6,* 105-182. 1969.

Mayer, Richard E. "A psychology of learning BASIC," *CACM,* Nov. 1979.
Builds on his dissertation work. Compare Shneiderman & Mayer [1979] and Chase [1979].

Mel'zak, I. *Mathematical ideas, modeling and applications.* Volume 2 of *Companion to concrete mathematics.* Wiley, 1976.
The appendix on transform-solve-invert as a technique in mathematics is worth the price of the book. In denotational semantics, the interpretation map M from syntax to semantics is the transform. Background only; not cited.

Montague, Richard. "The proper treatment of quantification in ordinary English." In Hintikka, Moravcsik, and Suppes, *Approaches to natural language.* D. Reidel, 1973, 221-242; reprinted in Montague, *Formal philosophy: selected papers of Richard Montague,* Richmond H. Thomason, ed., Yale University Press, 1974.
A start on a denotational semantics for English, cut short by his tragic and untimely death.

Paulson, Lawrence C. *Logic and computation.* Cambridge, 1987.
A long list of applications of denotational semantics is given in the context of a programming language, LCF, which supports it.

Rich, Charles and Linda M. Wills. “Recognizing a program's design: a graph-parsing approach.” *IEEE Software,* Jan 1990: 82-89.
Describes a heuristic, data-flow approach to ascribing meaning to programs based on collecting a list of programming idioms which are recognized by subgraph matching. What I call an “expert system” approach.

Rogers, Hartley Jr. *Theory of recursive functions and effective computability.* McGraw-Hill, 1967.
Still the best specialized book from which to learn about recursive functions for mathematicians; not easy.

Schneider, David I. *Microsoft QuickBasic: An introduction to structured programming.* MacMillan, 1989.
I cite just one example from this reference.

Scott, Dana S. *The lattice of flow diagrams.* Technical monograph, PRG-3, Oxford University Computing Laboratory, Programming Research Group, 1970. Reprinted in E. Engeler's (editor) *Symposium on Semantics of Algorithmic Languages,* Springer-Verlag, 1971: 311-366.
The person who put denotational semantics on a solid mathematical foundation gives flowchart language examples, one of which I use as Example 12 of Part V. A clear paper which you should read secondly after Allison's text.

Shneiderman, Ben and Richard E. Mayer. Syntactic/semantic interactions in programmer behavior: a model and experimental results. *International J. of Comp. & Inf. Science 3,* 1979: 219-238.

Schmidt, David A. *Denotational semantics; a methodology for language development.* Allyn & Bacon, 1986.
A good second book, after Allison, of a tutorial nature.

Sebesta, Robert. *Concepts of programming languages.* Benjamin Cummings, 1993.
Well-written popular undergraduate text cited as an example of formal semantics introduced without motivation.

Slonneger, Kenneth and Barry L. Kurtz. *Formal syntax and semantics of programming languages; a laboratory based approach.* Addison-Wesley, 1995.
Cited for its uniform treatment of five kinds of semantics, only the first three of which are dealt with in the present paper.

van Emden, M. H. and Robert A. Kowalski. “The Semantics of Predicate Logic as a Programming Language.” *J. ACM 23,* 4, Oct 1976: 733-742.
Shows that operational semantics is to proof theory as denotational semantics is to model theory.

Acknowledgements

This paper was written on a sabbatical in 1990. At that time it benefitted from feedback on the occasion of three presentations based on the paper: an Association of Computing Machinery meeting at Lebanon Valley College; and meetings of mathematicians at Dickinson College and at Salisbury State University. I have revised it only slightly since then in the light of the very new, good graduate text by Slonneger and Kurtz [1995], and have added a reference to a current undergraduate

text by Sebesta [1993].

In particular, I have not tried to update the paper in the light of research since 1990. For example, parallelism now has two candidates for a denotational semantics: Petri nets or process semantics, despite my pessimism in footnote 8. Nor have I tried to eliminate evidence of informal style rooted in the fact that this paper is a manuscript from which to speak.

July 17, 1995, Version 1.4a, **s.g.d.**