SYNTAX and SEMANTICS

The Languages of Computer Science: some conceptual issues
Outline

1. The Languages of Computer Science
2. Grammar and its Limitations
3. Normative Semantics
4. Metacircular Interpreters
5. Machines
6. Informal Semantics
7. Operational Semantics
8. Towards a Theory of Operations
9. Denotational Semantics
10. Mathematics, Elegance and Design
1. THE LANGUAGES OF COMPUTER SCIENCE
Varieties of Languages

- Programming Languages (Haskell)
- Query Languages (SQL)
- Specification Languages (VDMSL)
- Architectural Description Languages (ACME)
- Mark-Up Languages (HTML)
- Web Ontology Languages (OWL)
- Knowledge Representation Languages (KRSS)
- Modelling Languages (UML)
- etc.
Semantic Issues

- These languages are at the heart of computer science: it is concerned with their purpose, their design and their syntax and semantics. Consequently, a proper conceptual analysis of the nature of these languages will form a significant part of the philosophy of computer science.
- In particular, semantics raises immediate and recognisable philosophical concerns. It touches upon issues in:

1. The philosophy of language
2. Ontology
3. The philosophy of mathematics
4. The philosophies of science and engineering
But adds a computational twist.
Varieties of Semantics

- SQL is given its interpretation in the relational algebra/calculus.
- The semantics of specification and AD languages, when given, is usually given in set theory with a natural language commentary.
- The semantics of OWL uses some form of first-order model theory. Much the same is so for description logics.
- The semantics of HTML uses very little; UML uses everything.
  
  Some of the central philosophical issues are common to them all and flow from their artificial nature. But we shall illustrate many of the conceptual issues with programming languages.
I am not only temperamentally a Platonist and prone to talking about abstracts if I think they throw light on a discussion, but I also regard syntactical problems as essentially irrelevant to programming languages at their present state of development. In a rough and ready sort of way, it seems to be fair to think of the semantics as being what we want to say and the syntax as how to say it. In these terms the urgent task in programming languages is to explore the field of semantic possibilities.

When we have discovered the main outlines and the principal peaks we can go about describing a suitable neat and satisfactory notation for them. But first we must try to get a better understanding of the processes of computing and their description in programming languages. In computing we have what I believe to be a new field of mathematics which is at least as important as that opened up by the discovery (or should it be invention) of calculus.
Two Themes

1. Semantics must come first.
2. The notions underlying programming language semantics are mathematical ones. We shall examine these two interrelated issues.
2. GRAMMAR and its LIMITATIONS
Grammar

• Syntax is given via a grammar of some sort – e.g., context free, BNF, inference rules, syntax diagrams, graphical means.

• But a grammar only pins down what the legal strings of the language are. It does not determine what they mean.

• We shall illustrate some issues with the following toy programming language.
The While Language

\[ P ::= \ x:=E \ | \ \text{skip} \ | \ P; P \ | \ \text{if} \ B \ \text{then} \ P \ \text{else} \ P \ | \ \text{while} \ B \ \text{do} \ P \ | \]

\[ E ::= \ x \ | \ 0 \ | \ 1 \ | \ E+E \ | \ E*E \ | \]

\[ B ::= \ x \ | \ true \ | \ false \ | \ E<E \ | \ \neg B \ | \ B\land B \ | \]
Semantics

According to the grammar, the following is legitimate

\[
\begin{align*}
  x &:= 0; \\
  y &:= 1; \\
  \textbf{while } x < n \textbf{ do } (x := x + 1; y := x \times y)
\end{align*}
\]

- But in order too construct or understand this program one needs to know more than the syntax of its host language; one must possess some \textit{semantic} information about the language.
3. NORMATIVE SEMANTICS
Normative Requirements

- The fact that the expression means something implies, that there is a whole set of normative truths about my behavior with that expression: namely, that my use of it is **correct** in application to certain objects and not in application to others. ....

- The normativity of meaning turns out to be, in other words, simply a new name for the familiar fact that, regardless of whether one thinks of meaning in truth-theoretic or assertion-theoretic terms, meaningful expressions possess conditions of correct use.

- Kripke's insight was to realize that this observation may be converted into a condition of adequacy on theories of the determination of meaning: any proposed candidate for the property in virtue of which an expression has meaning, must be such as to ground the 'normativity' of meaning-it ought to be possible to read off from any alleged meaning constituting property of a word, what is the correct use of that word.

*Boghssian on rule following*
Correctness and Obligation

Contemporary philosophers of language understand this requirement on a theory of meaning to have two components.

1. A criterion of correctness
2. An obligation to do what is correct

We shall only be concerned with 1. This is in contrast to Microsoft who seem to be concerned with 2 but not 1.
The Designer’s Intentions

- A semantics must reflect the intentions of the designer about the constructs of the language.
- However it is expressed or conveyed, it carries normative force: at some level, it must guide all categories of users.
Guide the Compiler Writer

- A semantics must guide a compiler writer in implementing the language.
- It must enable a distinction to be drawn between the correct and incorrect implementation of a construct.
- It must facilitate a specification of compiler correctness.
Program Correctness

- Semantics must enable a distinction to be drawn between *correct* and *incorrect* use of programming constructs - not just syntactically, but in the sense of meeting their intended specifications (formal or otherwise).
- Assume the specification is the factorial function. Then a semantic account must determine whether or not the following meets it.

  ```latex
  x:=0; \quad y:=1;
  \textbf{while} \; x<n \; \textbf{do} \; (x:=x+1; \; y:=x*y)
  ```

- Syntax alone cannot do this.
- It must distinguish between software whose specification is a web browser and that used to aid in asset management of power generating. Software engineers who produce one instead of the other will get fired - well not quite all. All will be ok if you return an error message.
Approaches to Semantics

1. Metacircular interpreters
2. Machines
3. Natural language manuals
4. Operational specifications
5. Denotational accounts
4. METACIRCULAR INTERPRETERS
Metacircular Interpreters

- Defined via an interpretation into another programming language (or a subset of the source one)
  \[ L_1 \Rightarrow L_2 \]
- Here the second language is intended to provide the semantics of the first.
- May even involve several layers of translation
  \[ L_1 \Rightarrow L_2 \Rightarrow L_3 \Rightarrow L_4 \ldots \]
Not Grounded

- On the face of it, such an approach does not satisfy the our normative requirements.
- It is not grounded: it just passes the burden of normativity to another language.
- How, for example, does this guide the implementer?
- Translations are themselves implementations; they are not independent guides to one.
5. MACHINES
Semantics Grounded in Machines

- Programming languages get their semantic interpretation in terms of a machine.

- Maybe achieved layer by layer, one language getting its interpretation in the next, until a machine ($M$) provides the final and actual mechanism of semantic interpretation.

$$L_1 \Rightarrow L_2 \Rightarrow L_3 \Rightarrow L_4 \ldots \ldots \Rightarrow M$$
Machine Structure

For our toy language we require a machine with an underlying state whose role is to store numerical values in locations

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>67</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>u</td>
<td>w</td>
<td>z</td>
</tr>
</tbody>
</table>

Locations
What kind of Machine?

- One view is that *the* semantic account is given by an implementation on a *physical* machine i.e., the intended meaning is to be given by the actual effect on the state of a physical machine.

- We might express the positive demand as:
  
  *No (normative) semantics without physical implementation*

- But ..
Actual machines can malfunction: through melting wires or slipping gears they may give the wrong answer. How is it determined when a malfunction occurs? By reference to the program of the machine, as intended by its designer, not simply by reference to the machine itself. Depending on the intent of the designer, any particular phenomenon may or may not count as a machine malfunction. A programmer with suitable intentions might even have intended to make use of the fact that wires melt or gears slip, so that a machine that is malfunctioning for me is behaving perfectly for him. Whether a machine ever malfunctions and, if so, when, is not a property of the machine itself as a physical object but is well defined only in terms of its program, stipulated by its designer. Given the program, once again, the physical object is superfluous for the purpose of determining what function is meant.
Consider the assignment instruction

\[ x := E \]

How is its semantics to be given on a physical machine?

Presumably, the machine does what it does when the program is run - and what it does determines the meaning of assignment.

There is no notion of *malfunction* since there is no independent specification. And so there is no notion of *correctness*.
Operation Specification

- Semantic accounts are not empirical claims about the behaviour of a physical machine; they are normative statements.
- The semantic account of assignment must determine what it means for the physical machine to behave correctly.
  
  *When the state is updated by placing \( v \) in location \( x \), and then the contents of \( x \) is retrieved, \( v \) will be returned. For any other location, the contents remains unchanged.*

- Accordingly, if the command \( x:=10 \) places 28 in location \( y \), this is not correct.
Abstract Machines

- But this operation description makes the physical machine (semantically) superfluous.
- It determines an operation on an abstract machine.
- And this supplies the specification of the physical one.
- In other words, normative semantics requires an abstract machine at its base.
6. INFORMAL SEMANTICS
The semantics of our toy programming language is to be given in terms of its impact upon such an abstract machine.

The most common form of semantics is given in natural language.

Such descriptions most often take the form of a reference manual for the language.

They can be big: the one for Java Language is almost 600 pages.
The while Statement

The while statement executes an Expression and a Statement repeatedly until the value of the Expression is false.

While Statement: while ( Expression ) Statement
WhileStatementNoShortIf: while ( Expression ) StatementNoShortIf

The Expression must have type Boolean or Boolean, or a compile-time error occurs. A while statement is executed by first evaluating the Expression. If the result is of type Boolean, it is subject to unboxing conversion (§5.1.8). If evaluation of the Expression or the subsequent unboxing conversion (if any) completes abruptly for some reason, the while statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the resulting value:

If the value is true, then the contained Statement is executed. Then there is a choice:
- If execution of the Statement completes normally, then the entire while statement is executed again, beginning by re-evaluating the Expression.
- If execution of the Statement completes abruptly, see §14.12.1 below.

If the (possibly unboxed) value of the Expression is false, no further action is taken and the while statement completes normally.

If the (possibly unboxed) value of the Expression is false the first time it is evaluated, then the Statement is not executed.
An Informal Semantics

1. If the evaluation of $E$ in the state $s$ returns the value $v$, then the evaluation of $x:=E$ in a state $s$, returns the state that is the same as $s$ except that the value $v$ replaces the current value in location $x$.

2. The evaluation of $\text{skip}$ in a state $s$, returns $s$.

3. If the evaluation of $B$ in $s$ returns true and the evaluation of $P$ in $s$ returns $s'$, then the evaluation of if $B$ then $P$ else $Q$ in $s$, evaluates to $s'$. If on the other hand, the evaluation of $B$ in $s$ returns false and the evaluation of $Q$ in $s$ returns $s'$, then the evaluation of if $B$ then $P$ else $Q$ in $s$, evaluates to $s'$.

4. If the evaluation of $P$ in $s$ yields the state $s'$ and the evaluation of $Q$ in $s'$ returns the state $s''$, then the evaluation of $P;Q$ in $s$, returns the state $s''$.

5. If the evaluation of $B$ in $s$ returns true, the evaluation of $P$ in $s$ returns $s'$, and the evaluation of while $B$ do $P$ in $s'$ yields $s''$, then the evaluation of while $B$ do $P$ in $s$, returns $s''$. If the evaluation $B$ in $s$ returns false, return $s$. 

Transparency and Informality

- It is *evaluation* that is being recursively specified (cf. semantic conception of truth)
- Works with simple languages but not so well with real ones involving complex mixtures of notions.
- Difficult to express what are essentially technical notions: natural languages does not always facilitate being clear about what we are talking about.
- It is hard to be simultaneously transparent and unambiguous: hard to be normative when hampered by language.
- The consequences of design decisions articulated in natural language may not be as sharp as they could be. Ambiguity cuts deeper than scope distinctions.
Java has integrated multithreading to a far greater extent than most programming languages. It is also one of the only languages that specifies and requires safety guarantees for improperly synchronized programs. It turns out that understanding these issues is far more subtle and difficult than was previously thought. The existing specification makes guarantees that prohibit standard and proposed compiler optimizations; it also omits guarantees that are necessary for safe execution of much existing code. Jeremy Manson and William Pugh
Semantic Coherence

• To make sense of our semantic description, to ensure that it is coherent, the designer will need to ensure that expression evaluation does not change the state.

• She must argue, by induction on the structure of expressions: the evaluation of variables does not change the state, and on the (inductive) assumption that the evaluation of both $E$ and $E'$ do not, it is clear that the evaluation of $E+E'$ does not.
A compiler writer will need to argue (at some level of clarity) that the compiler is correct.

This will (almost certainly) involve an inductive argument—not after the construction, but during it.

Such arguments are not optional; at some level, and with some degree of precision, one cannot construct a compiler without reasoning.

But a little notation helps.
7. OPERATIONAL SEMANTICS
A Little Notation

We shall write

\[ <P,s> \downarrow s' \]

to indicate that evaluating \( P \) in state \( s \) terminates in \( s' \).
Sequencing and Conditionals

\[
\begin{align*}
&P, s \downarrow s' & Q, s' \downarrow s'' \\
\frac{}{P;Q, s \downarrow s''}
\end{align*}
\]

\[
\begin{align*}
&B, s \downarrow \text{true} & P, s \downarrow s' \\
\frac{}{\text{If } B \text{ do } P \text{ else } Q, s \downarrow s'}
\end{align*}
\]

\[
\begin{align*}
&B, s \downarrow \text{false} & Q, s \downarrow s'' \\
\frac{}{\text{If } B \text{ do } P \text{ else } Q, s \downarrow s''}
\end{align*}
\]
While

\(<B, s> \downarrow \text{true} \quad <P, s> \downarrow s' \quad <\text{while } B \text{ do } P, s'> \downarrow s''\)

\(\text{-----------------------}\)

\(<\text{while } B \text{ do } P, s> \downarrow s''\)

\(<B, s> \downarrow \text{false}\)

\(\text{-----------------------}\)

\(<\text{while } B \text{ do } P, s> \downarrow s\)
From Informal to Formal

- We have not moved that far from the informal account. We have:
  1. Introduced a little notation
  2.Expressed the conditional nature of the informal rules as premise and conclusion
- But this is little more than exercising some care in expressing the informal concepts.
- But this is already helpful....
**Semantic Coherence**

**Theorem**  For all expressions $E$ and states $s$
if $<E,s>\downarrow<v,s'>$ then $s=s'$

**Proof**  By induction on expressions.

Without this the evaluation of programs needs to be adjusted in order to take account of state change during expression evaluation—coherence
8. THEORIES OF OPERATIONS
Evaluation and Termination

• The relation $\Downarrow$ is taken to be sui-generis in the proposed theory of operations. It is axiomatised by the rules.

• Define

$$<P,s> \Downarrow \equiv \exists s'. <P,s> \Downarrow s'$$

This provides a notion of *terminating* operation.
Equivalence

\[ P \cong Q \triangleq \forall s_1 \cdot \forall s_2 \cdot <P,s_1> \downarrow s_2 \leftrightarrow <Q,s_1> \downarrow s_2 \]

i.e., we cannot tell them apart in terms of their extensional behaviour. This is an equivalence relation. Moreover,

**Theorem**

(1) If **true** the \( P \) else \( Q \cong P \)
(2) If **false** the \( P \) else \( Q \cong Q \)
(3) while \( B \) do \( P \cong \) If \( B \) then \((P; \) while \( B \) do \( P \))
else skip
Towards a *Theory* of operations.

- We have the beginnings of a *theory* of operations.
- Not a deep and exciting theory, but still a theory.
- *Strachey*: *In computing we have what I believe to be a new field of mathematics which is at least as important as that opened up by the discovery (or should it be invention) of calculus.*
- A programming language (i.e., the bundle that is its syntax and semantics) is a mathematical object.
- *BUT*........
Formal versus Mathematical

- Does the argument that semantic accounts are mathematical depend upon the semantics and underlying theory being formally expressed?
- There is a distinction to be drawn between being formal and being mathematical.
- Formal axiomatisation is not a necessity.
- Even with the informal account, there is still an underlying theory of operations.
- Informal theories often get rigorously axiomatised later (Hilbert’s Geometry). But Euclid’s geometry is still mathematics.
- But...
9. Denotational Semantics
Mathematics is about Abstract Objects?

- One may question whether such theories (formal or not) are mathematical ones.
- Surely mathematics is about *abstract* objects and a mathematical theory must point beyond the axioms and rules to these objects.
- Such a criticism of the present position applies to both the informal and formal operational accounts: operational accounts are just translations into other languages; they do not reach this *abstract world of mathematics*. 
It cannot be syntax all the way down

We can apparently get quite a long way expounding the properties of a language with purely syntactic rules and transformations.......One such language is the Lambda Calculus and, as we shall see, it can be presented solely as a formal system with syntactic conversion rules........But we must remember that when working like this all we are doing is manipulating symbols - we have no idea at all of what we are talking about. To solve any real problem, we must give some semantic interpretation. We must say, for example, "these symbols represent the integers".

Stoy in his book on Denotational Semantics

- Peter Landin ISWIM/Dana Scott OWHY
Semantic Function

\( C: \text{Program} \Rightarrow (\text{State} \Rightarrow \text{State}) \)

We write

\( C \mid P \mid s \)

for the result of applying \( C \) to the program \( P \) in state \( s \).

1. \( C\mid x:=n \mid s = \text{Update}(x,n,s) \)
2. \( C\mid \text{skip} \mid s = s \)
3. \( C\mid P_1;P_2 \mid s = C\mid P_2 \mid (C\mid P_1 \mid s) \)
4. \( C\mid \text{if } B \text{ then } P_1 \text{ else } P_2 \mid s = \text{if } C\mid B \mid s = \text{true} \text{ then } C\mid P_1 \mid s \text{ else } C\mid P_2 \mid s \)
5. \( C\mid \text{while } B \text{ do } P \mid s = \text{if } C\mid B \mid s = \text{true} \text{ then } C\mid \text{while } B \text{ do } P \mid (C\mid P_1 \mid s) \text{ else } s \)
Mathematical Semantics

- Meaning via denotation into sets and functions
- Often generalised to category theory: objects and morphisms are the building blocks with enough structure to support the semantics e.g. Cartesian closed
- If such a semantics were required for a semantics to be normative, programming languages would be, via their semantics, mathematical objects. But it does not seem to be required.
- Rules can be normative without constituting a mathematical theory (the law). Our operational account suffices.
- But if the latter is not taken to be mathematical, then we cannot conclude that programming languages, as theories of operations, are mathematical objects.
What is Different About Set Theory?

• But why are set theoretic (category theoretic) accounts philosophically different to our operational ones? All are written in languages with rules/axioms: all end in axiomatic theories.

• If there is any substance to the above metaphysical claims about set-theory, it must go beyond the syntax of the language of set-theory. Indeed, it must go beyond its axioms. Otherwise, we will have not reached the abstract universe of sets.
The Metaphysics of Set Theory

- Gödel gives us some insight into what might be involved

*Despite their remoteness from sense experience, we do have something like a perception also of the objects of set-theory, as is seen from the fact that the axioms force themselves upon us as being true. I don't see any reason why we should have less confidence in this kind of perception i.e., in mathematical intuition than in sense perception, which induces us to build up physical theories and to expect that future sense perceptions will agree with them, and moreover, to believe that a question not decidable now has meaning and may be decided in the future*. Gödel 1964

- Gödel draws an analogy with the perception of physical objects; sets are perceived in an analogous way but what is perceived is neither the axioms and rules, nor the expressions that generate sets, but the sets themselves. It would seem that such knowledge must be taken as *knowledge of sets* rather than *knowledge that some proposition about sets is true*. 
Why are Operations and Sets Different?

- But even under Gödel’s perspective, it seems hard to see how the difference between operations and sets could be made out. It is certainly not clear that Gödel would have supported such a distinction. In his Gibbs Lecture he writes:

  *The greatest improvement was made possible through the precise definition of the concept of finite procedure, which plays a decisive role in these results. There are several different ways of arriving at such a definition, which, however, all lead to the same concept. The most satisfactory way, in my opinion, is that of reducing the concept of finite procedure to that of a machine with a finite number of parts, as has been done by the British mathematician Turing.*

- During this period, Gödel thought that Turing's analyses of finite procedure was definitive. In Wang's words, Gödel saw the problem of defining computability as:

  *an excellent example of a concept which did not appear sharp to us but has become so as a result of a careful reflection*

- One assumes that he would have assigned the notion of finite procedure a similar metaphysical status to sets.
Not just any old Theory

• Set theories and theories of operations have the same metaphysical status as mathematical theories. So on this view, programming languages as theories of operations are mathematical objects.

• However, while it is hard to pinpoint exactly why some are blessed and some are not, there is still some uneasiness in dubbing any old theory mathematical.

• Perhaps, small, elegant, aesthetically pleasing theories of operations deserve that status (e.g. Martin-Lof's type theory, the theory of constructions, Feferman's theories of operations and types, The second order Lambda Calculus, Milnor's calculus of processes).
I0. Mathematics, Elegance and Design
When we have discovered the main outlines and the principal peaks we can go about describing a suitable neat and satisfactory notation for them. But first we must try to get a better understanding of the processes of computing and their description in programming languages. In computing we have what I believe to be a new field of mathematics which is at least as important as that opened up by the discovery (or should it be invention) of calculus.
Mathematics First, Last or Not At All

• Most theories that result from real programming languages are just too ugly and mathematically intractable.
• Are we not starting at the wrong end i.e. language design followed by semantics?
• Should the mathematics play a more central role in design? I think this is what Strachey intended.
Conclusion?

1. This talk has been more about clarification, mapping out the logical space of possibilities, than reaching fixed conclusions.

2. The status of programming languages, as mathematical theories, raises issues that impinge upon some of the central and contemporary questions in the philosophies of language and mathematics.

3. In particular, in examining Strachey’s claim about programming languages being mathematical objects, we are as much engaged in clarifying the nature of mathematical theories as we are in examining the nature of programming languages.

But ...........
11. Programming Languages as Scientific Artefacts

An Alternative Perspective
Engineering Design: a caricature

- Engineers deal with constructed artifacts that are designed and built to meet some design specification. These are not intended to be explanatory but normative¹.
- Once designed, the engineer turns to the construction and testing of the actual device.
- Tests may show that it does not measure up to its abstract specification. If so, it needs to be rebuilt.
- This is the normative picture that we have implicitly applied to programming language semantics (specification) and construction (e.g. compiler).
- However, there is an alternative perspective.

¹. At some point Freeze the design Koen. Discussion of The Method. Conducting The Engineer's Approach to Problem Solving.
Theory Construction: a caricature

- Given a naturally occurring artifact, a scientist attempts to understand it. To do so, she makes some initial observations, prods and pocks it, literally and figuratively, and formulates some hypothesis about its function: what is it and what does it do?
- Eventually, she may formulate a theory about it. Such theories are often mathematically expressed but have empirically testable implications.
- If these turn out to be false, the theory may have to be revised. Subsequently, theory construction cycles through alternate stages of theory articulation and empirical verification.
Programming Languages as Scientific Artefacts

- Modern software is so complex that once built, even to specification, the consequences of running it on a physical machine are unpredictable.
- In order to program a user needs to know what will actually happen on a given physical machine; not what the semantics says. The semantic description maybe irrelevant in practice.
- Consequently, a programming language must be treated as an artifact that is subject to scientific investigation.
- We need to construct theories (that might even look like our theories of operations) about its behavior. But they are scientific not mathematical theories. They are subject to falsification.
Theory Formulation

1. We build an initial scientific theory of the language.
2. This is then used to inform the user. It must act as (a reverse engineered) semantic specification of the language. No programming without semantic content.
3. Hence, from the point where the theory is accepted, it must take on a normative role.
Problems

1. Is such theory construction feasible?
2. In the future, the whole system may malfunction in new ways not predicted by the theory.
3. In which case, the user requirement that initiated the scientific perspective (i.e., the user needs to know what actually happens) will lead to the development of a new theory. And so on..
4. Indeed, it would seem that this user requirement is unobtainable: continual revision is required to feed this desire to know exactly what happens.
5. Moreover, there must have been an initial normative account that underpinned the original compiler.