Semantics of a structure

[ ] = carrot

[ ] = bowling pin
Tom Murphy
CS PhD student at CMU
Illustrated Notes from Computer Science Classes
Fall 2002
Overview of Semantics

• Value of formal semantics
• Major approaches: operational, denotational, axiomatic
• Axiomatic
  • Assertions, preconditions, postconditions, loop invariants
  • Hoare triples, weakest precondition calculus
  • Wider impact: assertions in languages (SPARK), essence of imperative programming act, class invariants (Meyer, OO Software Construction, 1997), non-standard logic (formal methods and model checking)
Lewis Carroll was a professional logician. Mathematical logicians study the logic—the language of mathematical proofs. So, mathematical logic is closely tied with the study of programming languages.
Lewis Carroll is a favorite among semanticists. (The illustration is by John Tenniel.)
Alice in Wonderland

“I don’t know what you mean by ‘glory,’” Alice said.
Humpty Dumpty smiled contemptuously. “Of course you don’t—
till I tell you. I meant ‘there’s a nice knock-down argument for you!’”
“But ‘glory’ doesn’t mean ‘a nice knock-down argument,’” Alice objected.
“When I use a word,” Humpty Dumpty said, in rather a scornful
tone, “it means just what I choose it to mean—neither more nor less.”
“The question is,” said Alice, “whether you can make words mean
so many different things.”
“The question is,” said Humpty Dumpty, “which is to be master—
that’s all.” . . .
“When I make a word do a lot of work like that,” said Humpty Dumpty, “I always pay it extra.”
How do you know what constructs in a programming language mean?
Consider for a moment FORTRAN. A standard exists for FORTRAN 95.

*In accordance with an official agreement with the International Stan-
dards Organization [sic], we are able to distribute electronic versions of
Programming languages–Fortran. Cost: USD 175.*

www.fortran.com
You expect that the standard communicates the purpose and meaning of the FORTRAN 95 constructs. Here is an example, adapted from Chapman, of a (possibly) unfamiliar construct:

```
INTEGER :: i=3, j=7, k=2
REAL, DIMENSION(9): A=(/1.,-2.,3.,-4.,5.,-6.,7.,-8.,9./)
```

1. `A(:)` means the whole array
2. `A(i:j)` means the subset starting at 3 and ending at 7
3. `A(i:)` means the subset starting at 3 and ending at the end of the array
4. `A(:j)` means the subset starting at 1 and ending at 7
5. `A(:k)` means the subset of all the odd elements
6. `A(i::k)` means every other element starting at 3
Python

The syntax has been adopted by Python.

```python
» A = [1,-2,3,-4,5,-6,7,-8,9]
» i,j,k = 3,7,2
» A[:]
[1, -2, 3, -4, 5, -6, 7, -8, 9]
» A[i:j]
[-4, 5, -6, 7]
» A[i:]
[-4, 5, -6, 7, -8, 9]
» A[:j]
[1, -2, 3, -4, 5, -6, 7]
» A[:k]
[1, 3, 5, 7, 9]
» A[i:k]
[-4, -6, -8]
» A[i:-3]
[1, -2, 3, -4, 5, -6]
```
Haskell List

[3..7]
[3,5..7]
Another example about guessing.

```python
while condition:
    statements
else:
    statements
```

What does this do? The else is skipped only if the loop is interrupted by a break statement.
Not very intuitive!
Guido van Rossum: “I would not have the feature at all if I had to do it over.”
How does one usually learn a programming language construct or a new programming language?

• By analogy (guessing from previous experience)
• By example
• Trial and error

But still how does one what is correct?
A precise description of the semantics of a programming language may be quite challenging. So, rarely is a reference consulted by users.

Consider the Ada Reference Manual, For example, the assignment statement is quite complex.

Consider the Java Language Specification, For example, the assignment statement.

Kotlin Language Specification Assignments

But would not a careful, i.e., formal, semantics for programming languages be a good idea?
Formal Semantics

- **Standardization of programming languages.** Much effort is spent on standardizing languages.
- **Reference for users.** “Try it and see if it works” — takes a lot of effort, and is often inconclusive.
- **Proof of program correctness.** Mathematical reasoning about what programs do.
- **Reference for implementers.** Prevent ill-defined and incompatible dialects.
- **Automatic implementation.** Tools that automate creating language translators that go beyond parsing.
- **Better understanding of language design.** What’s hard to define is hard to understand.
Remember the point is to define something you don’t understand with something you do understand. The semantics of programming languages is very complex. Some computer science students find the syntax suggestive or even familiar and the explanation of the meaning confusing.
# Cherokee Script

<table>
<thead>
<tr>
<th>glyph</th>
<th>Unicode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>𝚅</td>
<td>U+13A0</td>
<td>CHEROKEE LETTER A</td>
</tr>
<tr>
<td>𝚆</td>
<td>U+13A1</td>
<td>CHEROKEE LETTER E</td>
</tr>
<tr>
<td>𝚇</td>
<td>U+13A2</td>
<td>CHEROKEE LETTER I</td>
</tr>
<tr>
<td>𝚈</td>
<td>U+13A3</td>
<td>CHEROKEE LETTER O</td>
</tr>
<tr>
<td>𝚉</td>
<td>U+13A4</td>
<td>CHEROKEE LETTER U</td>
</tr>
<tr>
<td>𝚊</td>
<td>U+13A5</td>
<td>CHEROKEE LETTER V</td>
</tr>
<tr>
<td>𝚋</td>
<td>U+13A7</td>
<td>CHEROKEE LETTER KA</td>
</tr>
<tr>
<td>𝚌</td>
<td>U+13CE</td>
<td>CHEROKEE LETTER SE</td>
</tr>
<tr>
<td>𝚍</td>
<td>U+13D2</td>
<td>CHEROKEE LETTER SV</td>
</tr>
</tbody>
</table>
This is not to deny that experience and experimentation are important when clear explanations are not available.

Do not guess what constructs mean. Do not let intuition or *false friends* fool you.
The French phrase *faux amis (du traducteur)* was introduced in a book by linguists Maxime Kœssler and Jules Derocquigny in 1928. False friends are words in two languages that look or sound similar, but differ significantly in meaning. An example is the English “embarrassed” and the Spanish “embarazada” (which means pregnant), or the word “sensible,” which means reasonable in English, but sensitive in French and Spanish. Also, gymnasium meant both “a place of education” and “a place for exercise” in Latin, but its meaning was restricted to the former in German and to the latter in English, making the expression into false friends in those languages as well as in Greek, where it started out as “a place for naked exercise” – from $\gamma\nu\mu\nu\omicron\varsigma$ (gumnos, “naked”).
Types of Semantics

- **operational** — the operation of an abstract machine
- **denotational** — the denotation of programs as mathematical entities
- **axiomatic** — a system of rules for proving properties about the program
Operational semantics. This method defines a language by describing its actions in terms of the operations of an actual or hypothetical machine. Of course, this requires that the operations of the machine used in the description also be precisely define, and for this reason a very simple hypothetical machine is often used that bears little resemblance to an actual computer.

What does the `for` statement mean? `for (expr_1; expr_2; expr_3) {stmt}`

Well, define it in terms of machine code .... `expr_1`; loop: if `expr_2` goto end; `stmt` `expr_3`; goto loop; end:

Sebesta, 3.5.1 Operational Semantics, page 130.
During the ’60s and ’70s, operational semantics was generally regarded as inferior to the other two styles—useful for quick and dirty definitions of language features, but inelegant and mathematically weak. But in the 80s, the more abstract methods began to encounter increasingly thorny technical problems (the bête noire of denotational semantics turned out to be the treatment of nondeterminism and concurrency; for axiomatic semantics, it was procedures), and the simplicity of flexibility of operational methods came to be seem more and more attractive by comparison—especially in the light of new developments in the area by a number of researchers, beginning with Plotkin’s Structural Operational Semantics (1981), Kahn’s Natural Semantics (1987), and Milner’s work on CCS.
Modern Operational Semantics

When the state of an abstract machine can be described simply in terms of the language (rather than some low-level instruction set), operational semantics has two important subcategories.

- **structural operational** — a deductive system that defines a transition function that gives the next state of the machine (so-called *small-step* style of operational semantics)
- **natural semantics** — a deductive system that defines the final state of the machine (so-called *big-step* style of operational semantics)
Suppose terms (or expressions) of the language can be used to describe the “state” of the computation. We can use term rewriting, structural or natural semantics, to define the language.

What does $2 + 3 \times 4$ mean?
It means: $2 + 3 \times 4 \Rightarrow 2 + 12 \Rightarrow 14$. 
**Natural Semantics**

\[
\text{expr ::= expr } + \text{ expr } | \text{ expr } \times \text{ expr } | "0" | "1"
\]

\[
\begin{align*}
E_1 \rightarrow v_1 & \quad E_2 \rightarrow v_2 \\
\hline
E_1 + E_2 \rightarrow v_1 + v_2 \\
E_1 \rightarrow v_1 & \quad E_2 \rightarrow v_2 \\
\hline
E_1 \times E_2 \rightarrow v_1 \cdot v_2 \\
0 \rightarrow 0 \\
\hline
1 \rightarrow 1
\end{align*}
\]
Natural Semantics

expr ::= expr "+" expr | expr "*" expr | "0" | "1" | var
var ::= "A" | "B" | "C"

\[ \frac{\text{??????}}{x \rightarrow \text{?????}} \]

Add a context \( C \).

\[ \langle x, C \rangle \rightarrow l(x, C) \]

\[ \langle E_1, C \rangle \rightarrow v_1 \quad \langle E_2, C \rangle \rightarrow v_2 \]

\[ \langle E_1*E_2, C \rangle \rightarrow v_1 \cdot v_2 \]
In computer science, the phrase denotational semantics refers to a specific style of mathematical semantics for imperative programs. This approach was developed in the late 1960s and early 1970s, following the pioneering work of Christopher Strachey and Dana Scott at Oxford University. The term denotational semantics suggests that a meaning or denotation is associated with each program or program phrase (expression, statement, declaration, etc.). The denotation of a program is a mathematical object, typically a function, as opposed to an algorithm or a sequence of instructions to execute.

Denotational Semantics

A mapping of syntax to mathematical quantities.

\[ \lambda \]

\( \omega \)

\[ \langle \text{program} \rangle ::= \ldots \implies \]

Where have we seen this technique already?
Denotational Semantics

A mapping of syntax to mathematical quantities.

\[ \langle \text{program} \rangle ::= \ldots \Rightarrow \lambda \]

Where have we seen this technique already?
Example: regular expressions denoting formal languages. \[ \mathcal{D} : RE \to \wp(\Sigma^*) \]
Denotational Semantics

A mapping of syntax to mathematical quantities.

\[ \langle \text{program} \rangle ::= \ldots \implies \lambda \]

Where have we seen this technique already?
Example: regular expressions denoting formal languages. \( D : RE \rightarrow \varphi(\Sigma^*) \)

An important tenet of denotational semantics is that semantics should be compositional: the denotation of a program phrase should be built out of the denotations of its sub-phrases.
What is the natural choice for the denotation of a program?

\[ \mathcal{P} : \text{programs} \rightarrow (\text{inputs} \rightarrow \text{outputs}) \]

Doomed by Cantor’s theorem!
MICHAEL RABIN & DANA SCOTT

Introduced nondeterministic machines by publishing "Finite Automata and Their Decision Problem."

A.M. TURING AWARD
1976
Dana S. Scott (1932–)

Scott was born on October 11, 1932, in Berkeley, California. He studied under Alfred Tarski at the University of California, Berkeley. He took his doctoral degree at Princeton University in 1958 with Alonzo Church as his thesis advisor. He was a professor of philosophy at Princeton University from 1969 until 1972, when he became a professor of mathematical logic at Oxford University. His work on automata theory earned him the ACM Turing Award in 1976 (with Michael O. Rabin). In 1981 he moved from Oxford to Carnegie Mellon University, where he is the Hillman University Professor of Computer Science, Philosophy, and Mathematical Logic (Emertius).
Scott found a replacement for the concept of a mathematical function suitable for the denotation of a program.

What is the denotation of a program? Well, a program computes on “things,” say a domain $D$ of things. So why not the mathematical function $D \to D$? OK, but a program is also a thing, so we need $D = D \to D$. But Cantor’s theorem says there are no such things!
In his “Outline of a mathematical theory of computation” (1970):

*To date no mathematical theory of functions has ever been able to supply conveniently a free-wheeling notion of function except at the cost of being inconsistent. The main mathematical novelty of the present study is the creation of a proper mathematical theory of functions which accomplishes these aims (consistently!) and which can be used as the basis for the metamathematical project of providing the ‘correct’ approach to semantics.*

The first mathematical model of the type-free $\lambda$-calculus, a model in which $D \cong D \to D$. (See Raymond Turner, in *Handbook of Logic and Language* edited by Johan van Benthem and Alice G. B. ter Meulen.) We don’t need an uncountable number of functions, there are only a countable number of computable ones (i.e., the ones that have programs).
Axiomatic semantics involves several subcomponents:

- First-order logic
- Assertions: pre and post-conditions
- Language or structure of elementary number theory
- Statements in a simple programming language
- Hoare triples, e.g.,
- Deductive systems
expresses a program is correct with respect to a specification

LOGIC

AND

constrains the implementation of a programming language

SEMANTICS
ANTONY HOARE

Made enduring contributions to programming language design and definition

A.M. TURING AWARD 1980
Emeritus Professor of Computing at the University of Oxford and is now a senior researcher at Microsoft Research in Cambridge, England. He received the 1980 ACM Turing Award for “his fundamental contributions to the definition and design of programming languages.” Knighted by the Queen of England in 2000.
Recap of First-Order Logic

⊥ false
⊤ true
$A \& B$ $A$ and $B$
$A \lor B$ $A$ or $B$
$\neg A$ not $A$
$A \Rightarrow B$ $A$ implies $B$
$\forall x \ P(x)$ for all $x$, $P(x)$
$\exists x \ P(x)$ there exists $x$, $P(x)$

Modus ponens and modus tollens, classic laws of deduction

\[
\begin{array}{ccc}
A \Rightarrow B & A & A \Rightarrow B \\
\hline
B & \neg B & \neg A
\end{array}
\]
Non-standard logics

Non-standard aka non-classical aka alternative logics.

- Many-valued logic allow for truth values other than true and false. The most popular forms are three-valued logic, as initially developed by Jan Łukasiewicz, and infinitely-valued logics such as fuzzy logic, which permit any real number between 0 and 1 as a truth value.
- Intuitionistic logic rejects the law of the excluded middle, double negation elimination, and part of De Morgan’s laws;
- Linear logic rejects idempotency of entailment as well;
- Modal logic extends classical logic with non-truth-functional ("modal") operators. that express statements about necessity and possibility. The most common basic modal operators are "□" for "necessarily" and "◊" for "Possibly".
A Proof

1. \((\neg A \Rightarrow A) \Rightarrow A\)  
   axiom 1
2. \(A \Rightarrow (\neg A \Rightarrow A)\)  
   axiom 2
3. \((A \Rightarrow (\neg A \Rightarrow A)) \Rightarrow (((\neg A \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A))\)
4. \(((\neg A \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A)\)  
   \(MP\ 2, 3\)
5. \(A \Rightarrow A\)  
   \(MP\ 1, 4\)
A Proof is a Tree

\[
\begin{align*}
\text{axiom3} & \quad (A \Rightarrow (\neg A \Rightarrow A)) \Rightarrow (((\neg A \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A)) \\
\text{axiom2} & \quad A \Rightarrow (\neg A \Rightarrow A) \\
\text{axiom1} & \quad (\neg A \Rightarrow A) \Rightarrow A
\end{align*}
\]

\[
\begin{align*}
((\neg A \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A) & \Rightarrow A \\
A \Rightarrow A
\end{align*}
\]
 Assertions

An assertion is a logical expression (a formula of FOL) characterizing the state of a program by the relationship of the variables. It is a two-valued, either true or false. An assertion, if false, indicates an error. An assertion is not part of the normal execution of the program, but can be used in debugging by catching “can’t happen” errors.

Assertions are used in two important ways: as preconditions and post-conditions.
An assertion used as precondition characterizes the state of the program that is required in order for the following statement/code/procedure to work correctly. If the precondition is false, then the error was in preparing to call the statement/code/procedure and in establishing the logical relations required for the statement/code/procedure to work correctly.

A precondition is an assertion that is used to require the condition be true before the execution of a statement/block/procedure. It is used to make precise the assumptions or requirements made by the statement/block/procedure.
Post-condition

An assertion used as a post-condition characterizes the state of the program guaranteed to be established by the preceding statement/code/procedure. If the post-condition is false, then the statement/code/procedure failed to establish the guaranteed outcome. So either the statement/code/procedure has a bug, the preconditions were not met, or the post-condition is wrong.

A post-condition is an assertion that guarantees the output of the preceding statement/code/procedure. It is used to make precise the intended purpose or actions of a statement/block/procedure.
Ada has the pragma Assert.

`invariant/exp.adb`

Java has the assert statement.

`recursion/Hamming.java`

Tip: use the assert to document assumptions in your code.
A variety of program properties can be specified via *contracts*

```ada
Char : Character := ... ;

procedure Replace( S : in out String;
                   Index : in Integer )
  with
    Global => (Input => Char),
    Depends => (S => (S, Char, Index)),
    Pre => S'Length > 0 and then
           Index in S'Range,
    Post => S(Index) = Char
  is
begin
  S(Index) := Char;
end Replace;
```

- **Global**: the only access to non-local data is to read the value of Char
- **Depends**: the value of S on return depends on its value on entry and the values of Char and Index, and S depends only on these values
- **Pre**: Boolean condition that subprogram assumes on entry
- **Post**: Boolean condition that is guaranteed at subprogram return

*data dependencies ✔*
*flow dependencies ✔*
*functionality ✔*
*absence of run-time errors ✔*
package Array_Handling is
  type Row is array (1..100) of Float;
  function Max (R : Row) return Float
  with Global => null,
    Depends => (Max.Result => R),
    Post => (for all X of R => Max.Result >= X) and
           (for some X of R => Max.Result = X);
end Array_Handling;

package body Array_Handling is
  function Max (R : Row) return Float is
    Current_Max : Float := R(1);
  begin
    for I in 2..100 loop
      pragma Loop.Invariant( for all J in 1 .. I-1 =>
                              Current_Max >= R(J) );
      pragma Loop.Invariant( for some J in 1 .. I-1 =>
                              Current_Max = R(J) );
      if Current_Max < R(I) then
        Current_Max := R(I);
      end if;
      end loop;
    return Current_Max;
  end Max;
end Array_Handling;

SPARK can prove the Global and Depends contracts, but needs additional guidance to prove the postcondition
• Loop invariant

SPARK checks pragma Loop.Invariant
• True at the 1st iteration
• If true on the (K-1)st iteration then also true on the Kth iteration

SPARK can prove absence of run-time errors and, given the pragmas, can prove the postcondition
• In a postcondition for function F, F.Result is the value being returned

Note: in the body of Max the indexing uses uses specific values (1, 2, 100) for simplicity. In practice these would be expressed as R'.First, R'.First+1, R'.Last for robustness.
Hoare Triples: State

The axiomatic approach defines each language construct in terms of a statement about what the construct accomplishes when executed.

Accomplishment will be gauged by describing the state of the computation before and after the execution of the construct.

We will view the memory of the computer as a collection of cells, each uniquely labeled.

The contents of the labeled cells is the state.

So, we view the state as a function from names to values.
## Storage–Finite Collection of Machine Words

<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Binary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0100 1101</td>
</tr>
<tr>
<td>0x01</td>
<td>1100 0001</td>
</tr>
<tr>
<td>0x02</td>
<td>0111 1111</td>
</tr>
<tr>
<td>0x03</td>
<td>1101 1011</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0xFC</td>
<td>1101 1011</td>
</tr>
<tr>
<td>0xFD</td>
<td>1101 1010</td>
</tr>
<tr>
<td>0xFE</td>
<td>0000 1100</td>
</tr>
<tr>
<td>0xFF</td>
<td>0100 1010</td>
</tr>
</tbody>
</table>
## Storage–Finite Collection of Cells With Arbitrary Values

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>37</td>
</tr>
<tr>
<td>0x01</td>
<td>121</td>
</tr>
<tr>
<td>0x02</td>
<td>0</td>
</tr>
<tr>
<td>0x03</td>
<td>23786341</td>
</tr>
<tr>
<td>0xFC</td>
<td>1574</td>
</tr>
<tr>
<td>0xFD</td>
<td>3589318</td>
</tr>
<tr>
<td>0xFE</td>
<td>123743</td>
</tr>
<tr>
<td>0xFF</td>
<td>8276</td>
</tr>
</tbody>
</table>
Storage–Labeled, Finite Collection of Cells

<table>
<thead>
<tr>
<th>a</th>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>121</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>23786341</td>
</tr>
<tr>
<td>w</td>
<td>1574</td>
</tr>
<tr>
<td>x</td>
<td>3589318</td>
</tr>
<tr>
<td>y</td>
<td>123743</td>
</tr>
<tr>
<td>z</td>
<td>8276</td>
</tr>
</tbody>
</table>
## Storage–Label, Non-Finite Collection of Cells

<table>
<thead>
<tr>
<th>a</th>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>-121</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>23786341</td>
</tr>
<tr>
<td>e</td>
<td>5741</td>
</tr>
<tr>
<td>f</td>
<td>-3893158</td>
</tr>
<tr>
<td>g</td>
<td>837431</td>
</tr>
<tr>
<td>h</td>
<td>2786</td>
</tr>
</tbody>
</table>
Storage– Snapshot of Memory is a Mapping

\[
\begin{align*}
    a & \mapsto 37 \\
    b & \mapsto 121 \\
    c & \mapsto 0 \\
    d & \mapsto 23786341 \\
    e & \mapsto 5741 \\
    f & \mapsto 3893158 \\
    g & \mapsto 837431 \\
    h & \mapsto 2786 \\
\end{align*}
\]

\ldots
Mathematically $\sigma : A \rightarrow \mathbb{N}$

Same as a model of first-order predicate logic.

(A model is everything you need to know in order to figure out if a formula is true or not.)

This “pun” or fortuitous alignment is the bridge between the computer and logic.
To describe sets of program states we need a specification language. An ingenious way of specifying states takes advantage of the fact that an assignment in logic is just like a snapshot of memory: they are both functions from labels/names to values.

logic: can express relations about names called variables, e.g., $x \leq 7$.
programs: can express actions on named cells, also called variables, e.g., $y := 3$. 
Formulas of FOL using Arithmetic

- **Terms.**
  1. If \( x \) is a variable, then \( x \) is a term.
  2. If \( n \) is an integer constant, then \( n \) is a term.
  3. If \( t_1 \) and \( t_2 \) are terms, then \( t_1 + t_2 \) and \( t_1 \times t_2 \) are terms.

- **Formulas.**
  1. \( \top \) and \( \bot \) are formulas.
  2. If \( t_1 \) and \( t_2 \) are terms, then \( t_1 = t_2 \) and \( t_1 < t_2 \) are formulas.
  3. If \( \phi \) and \( \psi \) are formulas, then \( \phi \& \psi, \phi \mid \psi, \neg \phi \), and \( \phi \Rightarrow \psi \) are formulas.
  4. If \( \phi(x) \) is a formula possibly containing the variable \( x \) free, then \( \forall x.\phi(x) \) and \( \exists x.\phi(x) \) are formulas.
Each dot is a state, $\sigma : A \rightarrow \mathbb{N}$ where $A$ is a set of memory/storage addresses or labels.
For example, the formula $x = 3$ characterizes all those states in which the value of memory cell $x$ is three. $P = \{ \sigma \mid \sigma(x = 3) = T \}$
For example, the formula \( y = 8 \) characterizes all those states in which the value of memory cell \( y \) is eight. \( Q = \{ \sigma \mid \sigma(y = 8) = T \} \)
Using propositional connectives, ever more complicated sets of states can be described.

\[ x = 3 \& y = 8 \]
\[ x = 3 \Rightarrow y = 8 \]
The states characterized by $P \lor Q$ are shaded.
The states characterized by $P \& Q$ are shaded.
\[
P \land Q \Rightarrow R
\]
$P \Rightarrow Q \land R$
Hoare Triples

Bounded variables in a formula of FOL act like “variables,” they stand for a value. While unbound variables act as an address in the computer’s memory.
Hoare Triples

In using formulas to characterize states there is a vast difference between free and bound variables. Formulas with free variables, like the formula $x = 3$, have an intuitive reading, like

"the states in which the cell $x$ has the contents 3."

But formulas with bound variables, like the formula $\exists x. x = 3$, can be misleading. The names of bound variables are not relevant. The formula $\exists x. x = 3$ is the same as $\exists y. y = 3$. Thus the name of a bound variable is not significant and does not have any relation to the "labels" for the cells in memory. The formula $\exists z. z = 3$ does not mean that some cell has contents 3. Rather it asserts that some value is equal to 3. Quantification ranges over the set of possible values, not labels. Hence $\exists z. z = 3$ is equivalent to $3 = 3$, or any other true formula. As such, it characterizes all states. More generally, formulas without free variables are either true or false, and hence characterize all the states or none of the states.
More formally, we define the set of states characterized by the formula $\phi$ to be the set $\{ \sigma \in \Sigma \mid \sigma \models \phi \}$, where “$\models$” is the relation that the state/assignment $\sigma$ satisfies the formula $\phi$. This relation is the basic semantic definition of first-order predicate logic. The definition of the relation goes as follows. We define the value of a term in state $\sigma$, denoted $t^\sigma$, to be $\sigma(x)$, if $t$ is the variable $x$, and $n^\sigma = n$ for integer constants. The state $\sigma$ is said to satisfy the formula $t_1 = t_2$, if $t_1^\sigma = t_2^\sigma$. The state $\sigma$ is said to satisfy the formula $\phi \& \psi$, if $\sigma \models \phi$ and $\sigma \models \psi$. The definition for the other logical connectives is similar. For the universal quantifier we define:

$$\sigma \models \forall x. \phi(x) \text{ iff } \sigma[x := n] \models \phi(n) \text{ for every integer } n$$

The definition of $\sigma \models \exists x. \phi(x)$ is similar.
\[ \Sigma^* \text{ strings of symbols} \]

- \[ 0 < x < y \]
- \[ x + x \text{ while} \]

well-formed formulas

- \[ x \leq 5 \lor 3 \leq x \]
- \[ x \leq 5 \Rightarrow x \leq 9 \]
- \[ x \leq 5 \Rightarrow y \leq 5 \]
- \[ xyz \{ \forall \} \]
- \[ x = 5 \lor x = 8 \]
Assertions

Assertions are formulas of logic that characterize the state of a program during its executions.

The execution of a construct $S$ in a programming language can be described by the state obtained by executing a program segment. This suggests that we consider triples $\{ P \} S \{ Q \}$, where $P$ and $Q$ are formulas of first order logic and $S$ is a piece of code. Constructs of this form are called Hoare triples.

Definition

We say the triple $\{ P \} S \{ Q \}$ is valid, if execution of the program segment $S$ is begun in any state satisfying $P$, and if $S$ terminates, then it terminates in a state satisfying $Q$. 
Hoare Triples

The $P$ of $\{ P \} S \{ Q \}$ is called the precondition of the Hoare triple, and $Q$ the post-condition. $S$ is a code fragment or statement.
Partial Correctness

Hoare triples are especially useful in proving programs correct because proof systems exist for deriving valid Hoare triples. (We give one in the next section.) A “correct” program is one that meets its specification. Sometimes, instead of saying the Hoare triple \{ P \} S \{ Q \} is valid, we say that the program segment \( S \) is partially correct with respect to the precondition \( P \) and the post-condition \( Q \). We say partially correct because we assume that the program terminates. Knowing that a Hoare triple is valid guarantees that the post-condition is established, if the program terminates. No assurances are given that the program does indeed terminate. This is not wholly satisfactory and leads to some counter-intuitive behavior. A total-correctness semantics, where termination is assured instead of assumed, is possible.
Partial Correctness

\{ \top \} \textbf{while } \top \textbf{ do } x := x; \textbf{ end } \{ \text{“I am the pope.”} \}
Figure 5.3
Partial and total correctness.

Partial correctness:
- Any legal input
- Algorithm A
- If this point is reached
- Then this is the desired output

Total correctness:
- Any legal input
- Algorithm A
- Indeed this point is reached
- And this is the desired output
Partial Versus Total Correctness

Partial Correctness:

$$A \& B \Rightarrow C$$

Total Correctness:

$$A \Rightarrow B \& C$$
Suppose that we know that the following Hoare triple is valid:

\[
\{ 0 \leq a \& 0 \leq b \} \ S \ { z = a \ast b } 
\]

If the program segment \( S \) is \( z := 0; \ a := 0 \), we can prove the formal correctness of \( S \) with respect to the assertion \( z = a \ast b \), but \( S \) does not perform any multiplication!
A Hoare triple concerns a piece of code. To be formal we need to define what code we are talking about. In fact, the Hoare triples serve to define the meaning of the code.

To make things easier we use a simple, but Turing complete programming language, we call the *while* language.
Simple While Language

To make things easier we use a simple, but Turing complete programming language, we call the while language.

\[ W ::= V := T \]
\[ W ::= \text{if } B \text{ then } W \text{ else } W \]
\[ W ::= \text{while } B \text{ do } W \text{ end} \]
\[ W ::= W ; W \]

It is an idealized, but nonetheless quite powerful, programming language. The boolean conditions \( B \) and terms \( T \) share with the language of first-order predicate logic a domain of integers.

We carefully observe that syntactic distinction between formulas (and their constituent terms) and statements in the \textit{while} language. Most especially between the notation for assignment \( := \) and that of the relation of equality \( = \).
Example Program

Here is an example program fragment in the simple while language:

\[ z := 0; \ n := y; \ \textbf{while} \ n > 0 \ \textbf{do} \ z := z + x; \ n := n - 1 \ \textbf{end} \]
The deductive system we are about study is remarkably small and powerful. It closely follows the simple programming language (the while language). It is obvious how to construct derivations.

- **Sound**: Only truths are derivable.
- **Completeness**: All truths are derivable.
$\Sigma^*$ strings of symbols

- $0 < x < y$
- $x + x$ while

well-formed Hoare triples

- $\{y = 0\}x := 1\{y = 0\}$
- $\{\top\}x := 1\{x = 1\}$
- $\{x > 0\}y := 5\{y < 0\}$
- $xyz if$
- $\{z = 7\}y := y\{x = 0\}$
A deductive system has judgments of the form:

\[
\frac{\text{hypothesis}}{\text{conclusion}}
\]
The Deductive System

Assignment axiom:
\[ \{ Q[V := T] \} \ V := T \{ Q \} \]

Conditional rule:
\[ \{ B \& P \} \ S_1 \{ Q \} \quad \{ \neg B \& P \} \ S_2 \{ Q \} \]
\[ \{ P \} \text{ if } B \text{ then } S_1 \text{ else } S_2 \{ Q \} \]

While rule:
\[ \{ B \& I \} \ S \{ I \} \]
\[ \{ I \} \text{ while } B \text{ do } S \text{ end } \{ \neg B \& I \} \]

Composition rule:
\[ \{ P \} \ S_1 \{ Q \} \quad \{ Q \} \ S_2 \{ R \} \]
\[ \{ P \} \ S_1 ; \ S_2 \{ R \} \]
Deductive System: Simplest Rules First

Conditional rule:

\[
\begin{align*}
\{ B & \& P \} & S_1 \{ Q \} & \{ \neg B & \& P \} & S_2 \{ Q \} \\
\{ P \} & \text{if} \ B & \text{then} \ S_1 & \text{else} \ S_2 \{ Q \}
\end{align*}
\]

Composition rule:

\[
\begin{align*}
\{ P \} & S_1 \{ Q \} & \{ Q \} & S_2 \{ R \} \\
\{ P \} & S_1; \ S_2 \{ R \}
\end{align*}
\]

So for all formulas \( P, Q \), etc, and all programs \( S_1 \) and \( S_2 \).
\[ \{ x > 0 \} \ y := x \ { 0 \leq y } \quad \{ \neg(x > 0) \} \ y := -x \ { 0 \leq y } \]

\[ \{ \top \} \text{ if } x > 0 \text{ then } y := x \text{ else } y := -x \ { 0 \leq y } \]

This is an instance of the conditional rule.

The conclusion follows if the two Hoare triples, the hypotheses, can be derived. The premises are, in fact, provable, using the axiom for the assignment statement which follows shortly.
Assignment Axiom

Here is the rule for discovering valid Hoare triples about assignment statements. No hypotheses appear above the horizontal line, so the Hoare triple below the line is true in all circumstances.

\[
\{ Q[V := T] \} \quad V := T \quad \{ Q \}
\]

The notation \( Q[V := T] \) stands for the assertion obtained by substituting the term \( T \) for the variable \( V \) in formula \( Q \). All Hoare triples of this form (for all terms \( T \), variables \( V \), and formulas \( Q \)) are valid Hoare triples. So, it is really an infinite family of axioms, not just one axiom.
Assignment Axiom

Here is the rule for discovering valid Hoare triples about assignment statements. No hypotheses appear above the horizontal line, so the Hoare triple below the line is true in all circumstances.

\[
\{ Q[V := T] \} \ V := T \{ Q \}
\]

The notation \( Q[V := T] \) stands for the assertion obtained by substituting the term \( T \) for the variable \( V \) in formula \( Q \).

All Hoare triples of this form (for all terms \( T \), variables \( V \), and formulas \( Q \)) are valid Hoare triples. So, it is really an infinite family of axioms, not just one axioms.
Several examples:

\[ \{ 2 = 2 \} \ x := 2 \ \{ x = 2 \} \]
Several examples:

\[
\{ 2 = 2 \} \ x \ := \ 2 \ \{ x = 2 \}
\]

\[
\{ y = 1 \} \ x \ := \ 2 \ \{ x = y + 1 \}
\]
Several examples:

\[
\{2 = 2\} \ x := 2 \ \{x = 2\}
\]

\[
\{y = 1\} \ x := 2 \ \{x = y + 1\}
\]

\[
\{y = 17\} \ x := 2 \ \{y = 17\}
\]
Several examples:

\[
\{ 2 = 2 \} \ x := 2 \ \{ x = 2 \}
\]

\[
\{ y = 1 \} \ x := 2 \ \{ x = y + 1 \}
\]

\[
\{ y = 17 \} \ x := 2 \ \{ y = 17 \}
\]

\[
\{ 2 = 2 + 1 \} \ x := 2 \ \{ x = x + 1 \}
\]
Several examples:

\[
\{ 2 = 2 \} \ x \ := \ 2 \ \{ x = 2 \}
\]

\[
\{ y = 1 \} \ x \ := \ 2 \ \{ x = y + 1 \}
\]

\[
\{ y = 17 \} \ x \ := \ 2 \ \{ y = 17 \}
\]

\[
\{ 2 = 2 + 1 \} \ x \ := \ 2 \ \{ x = x + 1 \}
\]

\[
\{ \bot \} \ x \ := \ 2 \ \{ x = 3 \}
\]
Earlier we wanted some valid Hoare triples to be used as premises. The axiom of assignment is required.

\[
\{0 \leq x\} \ y := x \ {0 \leq y}\]

\[
\{0 \leq -x\} \ y := -x \ {0 \leq y}\]

There is a troubling syntactic mismatch in the if example which disappears considering some arithmetic reasoning:

\[x > 0 \iff 0 \leq x\]

\[x \leq 0 \iff 0 \leq -x\]

\[\neg(x > 0) \iff 0 \leq -x\]
Again, how is this an approach to semantics? How is an axiom or rule a definition of meaning?

The assignment axiom captures the properties which must be true of the execution of the assignment statement. E.g., an implementation must insure these properties are never violated.
The rules are simple, obvious, and sound. Yet, most programming languages violate the rules. This illustrates that understating most programs, is much harder than people imagine.

For example, aliasing invalidates the axiom of assignment. E.g., if $x$ and $y$ are aliases as in `int& x=y`, then the following does *not* hold:

\[
\{ y = 0 \} x := 1 \{ y = 0 \}
\]

So, only very disciplined subsets of languages can be verified. And, only very careful programmers can write correct programs.
Weakest Precondition Calculus

At first glance the rule for assignment appears to be backward. There is no correct direction in a Hoare triple: it is valid or it is not; there is no direction involved. The mechanics of the rule imply that one picks the post-condition \( Q \) and from this choice the precondition is determined, namely, \( Q[x := e] \). This “flow” from post-condition to precondition has been formalized by Dijkstra and Gries into a weakest precondition calculus. The weakest or most useful precondition of the assignment \( V := T \) and the arbitrary post-condition \( Q \), written:

\[
WP(V := T, Q)
\]

is the condition \( Q[V := T] \). It is possible to describe the most useful or strongest post-condition in terms of the precondition for the assignment statement, but this is harder.
Hoare Logic

Conditional rule:
\[
\{ B \& P \} S_1 \{ Q \} \quad \{ \neg B \& P \} S_2 \{ Q \}
\]
\[
\{ P \} \text{ if } B \text{ then } S_1 \text{ else } S_2 \{ Q \}
\]

Composition rule:
\[
\{ P \} S_1 \{ Q \} \quad \{ Q \} S_2 \{ R \}
\]
\[
\{ P \} S_1 ; S_2 \{ R \}
\]

Rule of consequence:
\[
P' \Rightarrow P \quad \{ P \} S \{ Q \} \quad Q \Rightarrow Q'
\]
\[
\{ P' \} S \{ Q' \}
\]
Rule of Consequence

$P \Rightarrow Q$ means $P$ is “stronger” can $Q$.

Rules of consequence:

$P' \Rightarrow P \quad \{ P \} S \{ Q \}$

$\{ P' \} S \{ Q \}$

$P'$ is stronger so we can replace the precondition with something stronger.

$\{ P \} S \{ Q \} \quad Q \Rightarrow Q'$

$\{ P \} S \{ Q' \}$

$Q'$ is weaker so we can replace the post-condition with something weaker.

Sometimes we have more information than we need and that makes it difficult to comprehend.
Rule of Consequence

Since \( y = 5 \Rightarrow y + 1 > 3 \). From

\[
\{ y + 1 > 3 \} x := y + 1 \{ x > 3 \}
\]

(which is valid since it is an instance of the axiom of assignment) we can get, by the rule of consequence:

\[
\{ y = 5 \} x := y + 1 \{ x > 3 \}
\]

And, since \( x > 3 \Rightarrow x > 0 \), we can get:

\[
\{ y + 1 > 3 \} x := y + 1 \{ x > 0 \}
\]

Or, using both implications:

\[
\{ y = 5 \} x := y + 1 \{ x > 0 \}
\]
while Rule

\[
\begin{align*}
\{ B & \land l \} & S \{ l \} \\
\{ l \} & \text{while } B \text{ do } S \text{ end } \{ \neg B & \land l \}
\end{align*}
\]
\( \{ x \geq 0 \} \quad \text{while} \quad x > 0 \quad \text{do} \quad x := x - 1 \quad \text{end} \quad \{ x = 0 \} \)
while Rule: Example 1

\[
\{ y < n \& f = y! \} \ S \ \{ f = y! \} \\
\{ f = y! \} \ \text{while} \ y < n \ \text{do} \ \ S \ \text{end} \ \{ y \geq n \& f = y! \}
\]

where \( S \) is \( y := y + 1; \ f := f \times y \)
Loop Invariant

A loop invariant is an assertion which if true before the execution of the body of a loop, is true after.
So, if $S$ is the body of a loop, then $I$ is a loop invariant if:

\[
\{ I \} \ S \ \{ I \}
\]
It is possible for a computer program to verify that a proof of a Hoare triple is correct or not. It is even possible for a computer program to build a correct program from a specification (a post-condition) except for two things:

1. can’t prove all mathematical facts (though theorem proving is quite good)
2. can’t create invariants
If the loop invariant is too strong, it could be t be able to prove it holds either initially or after loop-body.
If the loop invariant is too weak, it may Leave the post-condition too weak to prove what you And/or be impossible to re-establish after the loop body.
This is the essence of why there is no complete automatic procedure for conjuring a loop.
Requires thinking or guessing.
If proof does not work, invariant or code or both may need work.
Fortunately, programming is creative and inventive.
Example Proof

To prove:

\[ \{ y \geq 0 \} \ z := 0 ; \ n := y ; \textbf{while} \ n > 0 \ \textbf{do} \ z := z + x ; \ n := n - 1 \ \textbf{end} \ \{ z = x \times y \} \]

(Multiply \( x \) and \( y \) by repeated addition to get \( z \).)

Abbreviations:

\[
\begin{align*}
I &= n \geq 0 \& z = x \times (y - n) \\
P &= n - 1 \geq 0 \& z + x = x \times (y - (n - 1))
\end{align*}
\]
The goal is to prove that the program

\[ z := 0; \ n := y; \ \text{while} \ n > 0 \ \text{do} \ z := z + x; \ n := n - 1 \ \text{end} \]

computes the product of \( x \) and \( y \) by repeated addition. This program works only if \( y \) is not negative, so we take \( y \geq 0 \) as the precondition.
We want to prove that the following Hoare triple is valid.

\[ \{ y \geq 0 \} \]
\[ z := 0; \quad n := y; \quad \text{while } n > 0 \text{ do } z := z + x; \quad n := n - 1 \text{ end} \]
\[ \{ z = x \times y \} \]

The proof requires four applications of the assignment axiom, three applications of the composition rule, one application of the rule for while loops, three tautologies of arithmetic, and three applications of the rule of consequence.
We begin by using the axiom of assignment to prove the following two Hoare triples:

\[
\begin{align*}
\{ & z = x \ast (y - y) \land y \geq 0 \} \quad n := y \quad \{ & z = x \ast (y - n) \land n \geq 0 \} \\
\{ & 0 = x \ast (y - y) \land y \geq 0 \} \quad z := 0 \quad \{ & z = x \ast (y - y) \land y \geq 0 \}
\end{align*}
\]  

(1)  

(2)  

By the composition rule using valid Hoare triples 1 and 2 above, we obtain the following Hoare triple:

\[
\begin{align*}
\{ & 0 = x \ast (y - y) \land y \geq 0 \} \quad z := 0; \quad n := y \quad \{ & z = x \ast (y - n) \land n \geq 0 \}
\end{align*}
\]  

(3)
The following fact of arithmetic is needed to derive Hoare triple 5 below.

\[ y \geq 0 \Rightarrow 0 = x \times (y - y) \& y \geq 0 \quad (4) \]

\[ \{ y \geq 0 \} \ z:=0; \ n:=y \ \{ z = x \times (y - n) \& n \geq 0 \} \quad (5) \]
The assignment axiom yields the following two Hoare triples:

\[
\{(z + x) = x \ast (y - (n - 1)) \& (n - 1) \geq 0\}
\]
\[
z := z + x
\]
\[
\{z = x \ast (y - (n - 1)) \& (n - 1) \geq 0\}
\]

\[
\{z = x \ast (y - (n - 1)) \& (n - 1) \geq 0\}
\]
\[
n := n - 1
\]
\[
\{z = x \ast (y - n) \& n \geq 0\}
\]

Applying the rule of composition to Hoare triples 6 and 7 yields:

\[
\{(z + x) = x \ast (y - (n - 1)) \& (n - 1) \geq 0\}
\]
\[
z := z + x; n := n - 1
\]
\[
\{z = x \ast (y - n) \& n \geq 0\}
\]

The following formula is a tautology:

\[
z = x \ast (y - n) \& n \geq 0 \& n > 0 \Rightarrow (z + x) = x \ast (y - (n - 1)) \& (n - 1) \geq 0
\]
The law of consequence applied to Hoare triple 8 and the tautology 9 yields the following Hoare triple:

\[
\begin{align*}
\{ z = x \ast (y - n) & \land n \geq 0 \land n > 0 \} \\
z := z + x; \ n := n - 1 \\
\{ z = x \ast (y - n) & \land n \geq 0 \}
\end{align*}
\]  \hfill (10)

Applying the rule for while statements with Hoare triple 10 yields the following Hoare triple:

\[
\begin{align*}
\{ z = x \ast (y - n) & \land n \geq 0 \} \\
\text{while } \ n > 0 \ \text{do} \ z := z + x; \ n := n - 1 \ \text{end} \\
\{ z = x \ast (y - n) & \land n \geq 0 & \neg (n > 0) \}
\end{align*}
\]  \hfill (11)

The loop invariant \( I \) is \( z = x \ast (y - n) \land n \geq 0 \).

\[
z = x \ast (y - n) \land n \geq 0 \land \neg (n > 0) \Rightarrow z = x \ast y
\]  \hfill (12)
\{z = x \ast (y - n) \& n \geq 0\}

while \ n \ > \ 0 \ do \ z := z + x; \ n := n - 1 \ end

\{z = x \ast y\} \tag{13}

Using Hoare triples 5 and 13, and by applying the rule of composition we obtain the Hoare triple that we were seeking:

\{y \geq 0\}

\ z := 0; \ n := y; \ while \ n \ > \ 0 \ do \ z := z + x; \ n := n - 1 \end

\{z = x \ast y\} \tag{14}
{y \geq 0}
{0 = x \ast (y - y) \& y \geq 0}
z := 0; n := y;
{z = x \ast (y - n) \& n \geq 0}
while n > 0 do
    {z = x \ast (y - n) \& n \geq 0 \& n > 0}
    {z + x = x \ast (y - (n - 1)) \& (n - 1) \geq 0}
    z := z + x; n := n - 1
end
{z = x \ast (y - n) \& n \geq 0 \& \neg (n > 0)}
{z = x \ast y}
\{ 0 \leq y \} x := 2y + 1 \{ 0 \leq y \}

This follows immediately from axiom assignment

\[ 0 \leq y[x := 2y + 1] = 0 \leq y \]
\{ x = 2 \& x = 4 \} \ x := x + 1 \ { x = 3 \}

The axiom assignment yields

\((x = 2 \& x = 4)[x := x + 1] = (x + 1 = 2 \& x + 1 = 4)\)

But of course, \((x = 2 \& x = 4)\) iff \(\bot\) from the beginning. So the Hoare triple is true.
\{x = 2 \& x = 4\} \ x := x + 1 \ \{x = 3\}

The axiom assignment yields

\[(x = 2 \& x = 4)[x := x + 1] = (x + 1 = 2 \& x + 1 = 4)\]

But of course, \((x = 2 \& x = 4)\) iff \(\perp\) from the beginning. So the Hoare triple is true.
\{ x = 3 \} x := x + 1 \{ 0 \leq 0 & x \leq 5 \}

The axiom assignment yields

\[(x \leq 0 & x \leq 5)[x := x + 1] = (x + 1 \leq 0 & x + 1 \leq 5)\]

But \(x = 3\) implies \(x \leq -1 & x \leq 4\). So the Hoare triple is true.
\[
\{ \top \} \text{ if } x = 0 \text{ then } y := 2 \text{ else } y := x + 1 \text{ end } \{ x \leq y \}
\]

We have

\[
\{ x = 0 \} y := 2 \{ x \leq y \} \quad \{ x \neq 0 \} y := x + 1 \text{ end } \{ x \leq y \}
\]

Because \( x = 0 \) implies \( x \leq 2 \) and also Because \( x \neq 0 \) implies \( x \leq x + 1 \).
\{ s = 2^i \} \quad i := i + 1; \quad s := 2s \quad \{ s = 2^1 \}
// { x > 0 }  (PRE)
// { 1 = 0! }  (WEAK)
y := 1;
// { y = 0! }  (ASG)
z := 0;
// { y = z! }  (ASG)
while (z != x) {
    // { y = z! and z != x}  (WHILE)
    // { y * (z+1) = (z+1)!}  (WEAK)
z := z + 1;
    // { y * z = z! }  (ASG)
y := y * z;
    // { y = z! }  (ASG)
}
// { y = z! and ~(z != z)}  (WHILE)
// { y = x! }

PLC: Semantics
Axiomatic Semantics

© 2022
\[
\begin{array}{c}
\{ P \} C \{ Q \} \\
\{ R \ast R \} C \{ Q \ast R \} \\
\end{array}
\text{if } \text{mod}(C) \cap \text{fv}(R) = \emptyset
\]

The frame rule: that executes safely in all state (satisfying $P$), and also executes in any bigger state (satisfying $P \ast R$). The side condition enforces that none of the variables modified by $C$ occur free in $R$. 
Reading


“A mathematician is a machine for turning coffee into theorems.”


A programmer is a machine for turning coffee into invariants (which are not captured by most programming languages.)

Howard-Curry correspondence.

A programmer is a machine for turning coffee into proofs (which are the same as programs).
The end of semantics