Semantics of a structure

[\text{carrot}]

[\text{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, essence of imperative programming act, class invariants (Meyer, OO Software Construction, 1997)
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.)
“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 Standards Organization [sic], we are able to distribute electronic versions of the Fortran 95 standard: ISO/IEC 1539-1 : 1997, Information technology–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:

```fortran
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
The syntax has been adopted by Python which makes it not so unfamiliar these days.

>>> A = [1,-2,3,-4,5,-6,7,-8,9]
>>> 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]
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
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.

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
Formal Semantics

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.
This 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 Kssler 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 expressions into false friends in those languages as well as in Greek, where it started out as “a place for naked exercise.”
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
What does the **for** statement mean? \( \text{for } (expr_1; expr_2; expr_3) \{ stmt \} \)

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

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

\[
\begin{align*}
E_1 \rightarrow v_1 & \quad E_2 \rightarrow v_2 \\
\hline
E_1 + E_2 & \rightarrow v_1 + v_2
\end{align*}
\]

\[
\begin{align*}
E_1 \rightarrow v_1 & \quad E_2 \rightarrow v_2 \\
\hline
E_1 \ast E_2 & \rightarrow v_1 \cdot v_2
\end{align*}
\]

\[
\begin{align*}
0 & \rightarrow 0 \\
1 & \rightarrow 1
\end{align*}
\]
Natural Semantics

\text{expr} ::= \text{expr} + \text{expr} \mid \text{expr} \ast \text{expr} \mid "0" \mid "1" \mid \text{var}

\text{var} ::= "A" \mid "B" \mid "C"

\begin{align*}
\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 \ast E_2, C \rangle &\rightarrow v_1 \cdot v_2
\end{align*}

Add a context \( C \).
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.

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 $D \rightarrow D$? OK, but a program is also a thing, so we need $D = D \rightarrow D$.

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 λ-calculus, a model in which $D \cong D \to D$. (See Raymound 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).
Denotational Semantics

A mapping of syntax to mathematical quantities.

Where have we seen this technique already?
Denotational Semantics

A mapping of syntax to mathematical quantities.

Where have we seen this technique already?
Example: regular expressions denoting formal languages.
Denotational Semantics

A mapping of syntax to mathematical quantities.

Where have we seen this technique already? Example: regular expressions denoting formal languages.

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.
Axiomatic Semantics

Axiomatic semantics involves several subcomponents:

- First-order logic
- Assertions: pre and postconditions
- Language or structure of elementary number theory
- Statements in a simple programming language
- Hoare triples, e.g.,
- Deductive systems
C. A. R. Hoare

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

\[ \bot \text{ false} \]
\[ \top \text{ true} \]
\[ A \& B \text{ A and B} \]
\[ A \lor B \text{ A or B} \]
\[ \neg A \text{ not A} \]
\[ A \Rightarrow B \text{ A implies B} \]
\[ \forall x \ P(x) \text{ for all } x, \ P(x) \]
\[ \exists x \ P(x) \text{ there exists } x, \ P(x) \]

*Modus ponens*, one of the classic laws of deduction

\[
\begin{align*}
A \Rightarrow B & \quad A \\
\hline
& \quad B
\end{align*}
\]
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

\[
A \Rightarrow (\neg A \Rightarrow A)) \Rightarrow (((\neg A \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A)) \Rightarrow (\neg A \Rightarrow A)
\]

\[
((\neg A \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A)
\]

\[
A \Rightarrow A
\]
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 postconditions.
Assertions

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.

An assertion used as a postcondition characterizes the state of the program guaranteed to be established by the preceding statement/code/procedure. If the postcondition 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 postcondition is wrong. A postcondition is an assertion that guarantees the output of the preceding statement/code/procedure. It is used to make precise the expected output of a statement/block/procedure.
Language Support

Ada has the pragma `Assert`.

`invariant/exp.adb`

Java 1.4 now has the `assert` statement.

`ser/ser1996/ConvexHull.java`

Tip: use the `assert` to document assumptions in your code.
A Hoare triple is a way of expressing a program is correct with respect to a specification

LOGIC

OR

A Hoare triple is a way of constraining the implementation of a programming language

SEMANTICS
A deductive system has judgments of the form:

\[
\begin{array}{c}
\text{hypothesis} \\
\text{conclusion}
\end{array}
\]
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. We will consider only integer values.
## Storage–Finite Collection of Machine Words

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Machine Words</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>0xFC</td>
<td></td>
</tr>
<tr>
<td>0xFD</td>
<td></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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>0x01</td>
<td>-121</td>
<td></td>
</tr>
<tr>
<td>0x02</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td>23786341</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFC</td>
<td>1574</td>
<td></td>
</tr>
<tr>
<td>0xFD</td>
<td>-3589318</td>
<td></td>
</tr>
<tr>
<td>0xFE</td>
<td>123748</td>
<td></td>
</tr>
<tr>
<td>0xFF</td>
<td>8276</td>
<td></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>123748</td>
</tr>
<tr>
<td>z</td>
<td>8276</td>
</tr>
</tbody>
</table>
Storage–Label, Non-Finite Collection of Cells

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>37</td>
</tr>
<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>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Storage—Snapshot of Memory is a Mapping

\[ a \mapsto 37 \]
\[ b \mapsto -121 \]
\[ c \mapsto 0 \]
\[ d \mapsto 23786341 \]
\[ e \mapsto 5741 \]
\[ f \mapsto -3893158 \]
\[ g \mapsto 837431 \]
\[ h \mapsto 2786 \]

\[ \vdots \]
State — A Snapshot of Memory

Mathematically \( \sigma : A \rightarrow \mathbb{N} \)
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 expression 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.

$P = \{ \sigma \mid \sigma(x = 3) = \top \}$
For example, the formula $y = 8$ characterizes all those states in which the value of memory cell $y$ is eight. 

$$Q = \{ \sigma \mid \sigma(y = 8) = \top \}$$
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 $P\&Q$ are shaded.
\[\sum^* \]

- \(0 < x < y\)
- \(x + x\) while

well-formed Hoare triples

- \(\{y = 0\}x := 1\{y = 0\}\)
- \(\{\top\}x := 1\{x = 1\}\)
- \(\{x > 0\}y := 5\{y < 0\}\)
- \(\{z = 7\}y := y\{x = 0\}\)

truth

- \(xyz\) if
[See the Venn diagrams of all possible connectives in the Post system notes.]
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.
Hoare Triples

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 $\sigma$ 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 $\phi \land \psi$, if $\sigma \models \phi$ and $\sigma \models \psi$. The definition for the other logical connectives is similar. For the universal quantifier we define:

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

The definition of $\sigma \models \exists x.\phi(x)$ is similar.
Hoare Triples

Assertions are formulas of logic that characterize the state of a program. 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. 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 $Q$. The triple $\{P\} S \{Q\}$ can serve as a description of what $S$ does, or it can serve as a definition (a semantics) of how $S$ may be implemented. Hence, these triples serve as a semantics of a language.
The $P$ of $\{ P \} S \{ Q \}$ is called the precondition of the Hoare triple, and $Q$ the postcondition.
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 \textit{partially correct} with respect to the precondition \( P \) and the postcondition \( Q \). We say \textit{partially correct} because we assume that the program terminates. Knowing that a Hoare triple is valid guarantees that the postcondition is established, \textit{if} the program terminates. No assurances are given that the program does indeed terminate. This is not wholly satisfactory and leads to some counterintuitive 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} \ \{ \textquote{Iamthepope.} \}
Partial Versus Total Correctness

\[ A \& B \Rightarrow C \]

\[ A \Rightarrow B \& C \]
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.
Example Hoare Triple

```
z:=0; n:=y; while n>0 do z:=z+x; n:=n-1 end
```
Deductive System

The deductive system we are about to 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.
The Deductive System

Assignment axiom:

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

Conditional rule:

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

While rule:

\[
\begin{align*}
\{ B\&I \} & \ S \{ I \} \\
\{ I \} & \text{while } B \text{ do } S \text{ end } \{ \neg B\&I \}
\end{align*}
\]

Composition rule:

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

\[
\begin{align*}
\{P\} & S_1 \{Q\} & \{Q\} & S_2 \{R\} \\
\{P\} & S_1; & S_2 \{R\}
\end{align*}
\]
Deductive System: Simple Proof Step

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

The conclusion follows if the two Hoare triples, the hypotheses, can be derived.

(Observe that there is arithmetic reasoning going on such as that if \( z < 0 \), then \( 0 \leq y \).)
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.
Several examples:

\[
\begin{align*}
\{2 = 2\} \times &::= 2 \{x = 2\} \\
\{y = 1\} \times &::= 2 \{x = y + 1\} \\
\{y = 17\} \times &::= 2 \{y = 17\} \\
\{2 = 2 + 1\} \times &::= 2 \{x = x + 1\} \\
\{\bot\} \times &::= 2 \{x = 3\}
\end{align*}
\]
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 ensure these properties are never violated. In practice, most programming languages use an assignment statement that does not guarantee the simple assignment axiom. E.g., if \( x \) and \( y \) are aliases as in \( \text{int} & \ x=y \), then the following does not hold:

\[
\{ y = 0 \} x := 1 \{ y = 0 \}
\]
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 postcondition $Q$ and from this choice the precondition is determined, namely, $Q[x := e]$. This “flow” from postcondition 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 postcondition $Q$, written:

$$WP(V := T, Q)$$

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

Conditional rule:

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

Composition rule:

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

Rule of consequence:

\[
\begin{align*}
P' \Rightarrow P & \quad \{ P \} \ S \{ Q \} & \quad Q \Rightarrow Q' \\
\{ P' \} & \quad S \{ Q' \}
\end{align*}
\]
while Rule

\[
\begin{align*}
\{ B \land I \} & S \{ I \} \\
\{ I \} & \text{while } B \text{ do } S \text{ end } \{ \neg B \land I \} \\
\{ x > 0 \land x \geq 0 \} & x := x-1 \{ x \geq 0 \} \\
\{ x \geq 0 \} & \text{while } x > 0 \text{ do } x:=x-1 \text{ end } \{ x = 0 \} \\
\{ y < n \land f = y! \} & S \{ f = y! \} \\
\{ f = y! \} & \text{while } y < n \text{ do } S \text{ end } \{ y \geq n \land f = y! \}
\end{align*}
\]

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

$$\{ I \} S \{ I \}$$
Program Verification

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 postcondition) except for two things:

1. can’t prove all mathematical facts (though theorem proving is quite good)
2. can’t create invariants
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 \cdot y \}

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

Abbreviations:

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

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

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 \}
\begin{align*}
z &:= 0; \\
n &:= y; \\
\text{while } n > 0 \text{ do } z &:= z + x; \\
n &:= n - 1 \text{ end} \\
\{ z = x \cdot y \} \end{align*}

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:

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

\[
\{ 0 = x \ast (y - y) \& y \geq 0 \} \quad z := 0 \quad \{ z = x \ast (y - y) \& y \geq 0 \} \quad (2)
\]

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

\[
\{ 0 = x \ast (y - y) \& y \geq 0 \} \quad z := 0; \quad n := y \quad \{ z = x \ast (y - n) \& n \geq 0 \} \quad (3)
\]
The following fact of arithmetic is needed to derive Hoare triple ?? below.

\[ y \geq 0 \Rightarrow 0 = x \cdot (y - y) \land y \geq 0 \quad (4) \]

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

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

(6)

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

(7)

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

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

(8)

The following formula is a tautology:

\[
z = x \ast (y - n) \& n \geq 0 \& n > 0 \Rightarrow (z + x) = x \ast (y - (n - 1)) \& (n - 1) \geq 0
\]

(9)
The law of consequence applied to Hoare triple ?? and the tautology ?? yields the following Hoare triple:

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

(10)

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

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

(11)

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

\[
z = x \times (y - n) \& n \geq 0 \& \neg (n > 0) \Rightarrow z = x \times y
\]

(12)
\{z = x \ast (y - n) \& n \geq 0\}
while \ n > 0 \ do \ z:=z + x; \ n:=n - 1 \ end \tag{13}
\{z = x \ast y\}

Using Hoare triples \?? and \??, 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 \tag{14}
\{z = x \ast y\}
\[
\{y \geq 0\}
\{0 = x \cdot (y - y) & y \geq 0\}
\]
\[
z := 0; \quad n := y;
\]
\[
\{z = x \cdot (y - n) & n \geq 0\}
\]
while \(n > 0\) do
\[
\{z = x \cdot (y - n) & n \geq 0 & n > 0\}
\]
\[
\{z + x = x \cdot (y - (n - 1)) & (n - 1) \geq 0\}
\]
\[
z := z + x; \quad n := n - 1
\]
end
\[
\{z = x \cdot (y - n) & n \geq 0 & \neg(n > 0)\}
\]
\[
\{z = x \cdot y\}


“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