

Comp 205: Comparative Programming Languages

Semantics of Imperative Programming Languages

- denotational semantics
- operational semantics
- logical semantics

Lecture notes, exercises, etc., can be found at:
www.csc.liv.ac.uk/~grant/Teaching/COMP205/

Syntax ...

Syntax concerns notation and the rules that describe when an expression is legitimate (i.e., when an expression is - potentially - meaningful).

For example, the following Java code

```
public doNothing(int i, j)
{ return }
```

contains a number of syntactic errors
(mistyped keywords, missing semicolons, etc.)

... and Semantics

Semantics is concerned with meaning.

In Computer Science, we are particularly interested in the meanings of programs and their constituent parts

(expressions, function or type definitions, etc.)

Why?

Semantics is important for:

- **Language design and implementation**
E.g., the Java Virtual Machine makes Java portable because it has a simple but formal semantics
- **Verification of programs**
In order to prove something about a program you have to know what the program means
- **Fun, Depth of understanding, ...**

Flavours of Semantics

- **Denotational Semantics**
describes the meaning of a program by saying what the program denotes (usually, some kind of mathematical object)
- **Operational Semantics**
describes how programs are evaluated
- **Logical Semantics** (or **Axiomatic Semantics**)
describes the meaning of programs indirectly, by specifying what abstract properties they have

Denotational Semantics

The notion of **state** is central to the imperative paradigm and to its denotational semantics.

A state records the values that are associated with the variables used in a computation.

Programs change these values and therefore change the state.

The denotation of a program is a function from states to states.

States

In order to understand how programs behave when executed starting from a certain state, all we need know is the values of the variables in that state.

A reasonable abstraction of the notion of state is:

A **state** is a function of type $Id @ Nat$, where Id is the set of variable names.

(For simplicity, we assume that all variables take values in the natural numbers.)

Values and Expressions

If a state is a function of type $Id @ Nat$, we can use a state to obtain a value for expressions that contain variables.

For example if $S : Id @ Nat$ is a state, then a variable i will have a certain value, $S(i)$, in that state; similarly, the value of an expression, say

$$i * 2 + j,$$

will have a value

$$S(i) * 2 + S(j)$$

Some Notation

Given a state $S : Id @ Nat$, we write $S[[i]]$ instead of $S(i)$; similarly, given an expression, e , we write $S[[e]]$ for the value of e in the state S .

For example,

$$S[[i * 2 + j]] = S(i) * 2 + S(j).$$

Assignment

Consider the program

```
i = i + 1;
```

This program changes a state by changing only the value associated with the variable i .

If we run this program in a state S , we obtain a new state, S' , such that

- $S'[[i]] = S[[i]] + 1$
- $S'[[x]] = S[[x]]$, for all $x \in Id$ such that $x \neq i$.

Semantics of Assignment

The denotation of an assignment

```
y = e;
```

is a function that takes a state S and returns the state S' such that

- $S'[[y]] = S[[e]]$
- $S'[[x]] = S[[x]]$, for all $x \in Id$ such that $x \neq y$.

We write $U[y = e](S)$ for S' .

Operational Semantics

While denotational semantics says **what** a program means, **operational semantics** describes **how** programs are evaluated.

Typically this is done by presenting either:

- an abstract machine that evaluates programs (e.g., a Turing Machine, the JVM), or
- an "evaluation" relation that describes the step-by-step evaluation of programs

An Evaluation Relation

As an example, we define a relation \Rightarrow on pairs of states and programs.

The pair (S, P) represents a state S and a program P which is to be run in the state S .

An Evaluation Relation

We write $(S, P) \mathcal{D} (S', P')$ to indicate that when the program P is run in state S ,

- a single step of the evaluation of P results in the new state S' , and
- P' is the remaining part of the program to be evaluated (i.e., the rest of P after the first step of the evaluation).

Done Evaluation

It is convenient to write *done* to denote a fully -evaluated program. This allows us to denote a complete evaluation of a program P in a state S by a chain of the form:

$$(S, P) \mathcal{D} (S_1, P_1) \mathcal{D} (S_2, P_2) \mathcal{D} \dots \mathcal{D} (S_n, \text{done})$$

which means that when run in state S , the program P terminates, producing a new state S_n .

Assignment Operationally

As with denotational semantics, assignment forms the basis of operational semantics.

The first part of our definition of the evaluation relation is:

$$(S, y = e) \mathcal{D} (U[y = e](S), \text{done})$$

which is, more or less, exactly what denotational semantics says about assignment.

Composition Operationally

Operational semantics is also hierarchical (or "structured"), in that the semantics of other constructs are built from the semantics of assignment.

If $(S, P_1) \mathcal{D} (S', P_1')$
then $(S, P_1 ; P_2) \mathcal{D} (S', P_1' ; P_2)$

and if $(S, P_1) \mathcal{D} (S', \text{done})$
then $(S, P_1 ; P_2) \mathcal{D} (S', P_2)$

If-then-else Operationally

If $S[[e]]$ is true
then $(S, \text{if}(e)\{P_1\}\text{else}\{P_2\}) \mathcal{D} (S, P_1)$

and if $S[[e]]$ is false
then $(S, \text{if}(e)\{P_1\}\text{else}\{P_2\}) \mathcal{D} (S, P_2)$

Loops Operationally

If $S[[e]]$ is false
then $(S, \text{while}(e)\{P\}) \Downarrow (S, \text{done})$
and if $S[[e]]$ is true
then $(S, \text{while}(e)\{P\}) \Downarrow$
 $(S, P; \text{while}(e)\{P\})$

Loops Operationally

Note that we don't have the difficulty we faced with denotational semantics:
we don't need to explicitly say what a non-terminating computation **denotes**;
we can simply allow an infinite chain of evaluation steps:

$$(S, P) \Downarrow (S_1, P_1) \Downarrow (S_2, P_2) \Downarrow \dots$$

Summary

- The semantics of imperative languages is based on the semantics of assignment
- The semantics of other constructs is built up from that of assignment

Next: Semantics of Functional Languages