Semantics of Programming Languages

COMP 317

Lecture 1

Semantics:
- What?
- Why?
Where?

Me:
grant@liv.ac.uk   Room 2.19, Ashton Bdg.

Office Hours:
Wednesdays 2–3 pm   or use email to arrange a time

Module Web Page:
http://www.csc.liv.ac.uk/~grant/Teaching/COMP317/
notes, exercises, practical assignments, past papers, etc.

Recommended Text:
Joseph A. Goguen and Grant Malcolm.
Algebraic Semantics of Imperative Programs.
Where?

Me:
grant@liv.ac.uk  Room 2.19, Ashton Bdg.

Office Hours:
Wednesdays 2–3 pm  or use email to arrange a time

Module Web Page:
http://www.csc.liv.ac.uk/~grant/Teaching/COMP317/
notes, exercises, practical assignments, past papers, etc.

Recommended Text:
Joseph A. Goguen and Grant Malcolm. 
Algebraic Semantics of Imperative Programs. 
Where?

Me:
grant@liv.ac.uk  Room 2.19, Ashton Bdg.

Office Hours:
Wednesdays 2–3 pm or use email to arrange a time

Module Web Page:
http://www.csc.liv.ac.uk/~grant/Teaching/COMP317/
notes, exercises, practical assignments, past papers, etc.

Recommended Text:
Joseph A. Goguen and Grant Malcolm.
Algebraic Semantics of Imperative Programs.
Tutorials:
  start next week
  Mondays 3pm
  Thursdays 3pm
  (optional for Masters students)

You’ll be given problem sheets each week; you should attempt the problems before each tutorial
What is a computer?

(and as a digression: Who built the first computer?)

What is a program?

What does a program do?

(and: How do we know it does?

— we make mathematical models of programs!)
What is a computer?
(And as a digression: Who built the first computer?)

What is a program?

What does a program do?
(And: How do we know it does?
— We make mathematical models of programs!)
What is a computer?

(and as a digression: Who built the first computer?)

What is a program?

What does a program do?

(and: How do we know it does?

— we make mathematical models of programs!)
What is a computer?

(and as a digression: Who built the first computer?)

What is a program?

What does a program do?

(and: How do we know it does?

— we make mathematical models of programs!)
Semantics as Modelling

What is a mathematical model?

E.g.,

\[ F = ma \]

Turing was the first to give a mathematical model of computers:
Semantics as Modelling

What is a mathematical model?

E.g.,

\[ F = ma \]

Turing was the first to give a mathematical model of computers:
What is a mathematical model?

E.g.,

\[ F = ma \]

Turing was the first to give a mathematical model of computers:
What is the $n^{\text{th}}$ number in this series?

Turing wanted to write a program to compute this . . .
Turing and Sums

... and he came up with this:

```
count = 0;
sum = 0;
while count < n
    count = count + 1;
    sum = sum + count;
```

Compute $\sum_{i=0}^{n} i$

Was he right?
Of course he was!

But he wanted a rigorous way of demonstrating† this.

So he invented the notion of invariant.

This allows a proof that his program is correct, but depends upon having a mathematical model of what that program does.

† As in Quod Erat Demonstrandum.
Turing and Sums

Of course he was!

But he wanted a rigorous way of demonstrating† this.

So he invented the notion of invariant.

This allows a proof that his program is correct, but depends upon having a mathematical model of what that program does.

† As in Quod Erat Demonstrandum.
Formal semantics builds mathematical models of programs.

These models allow us to reason about programs (e.g., do they do what they’re meant to do) in much the same way that we can reason about numbers (or other mathematical objects).

All formal methods, insofar as they are formal, depend upon such mathematical models.
What and Why

**Formal semantics** builds mathematical models of programs.

These models allow us to reason about *programs* (e.g., do they do what they’re meant to do) in much the same way that we can reason about *numbers* (or other mathematical objects).

*All* formal methods, insofar as they are formal, depend upon such mathematical models.
Formal semantics builds mathematical models of programs.

These models allow us to reason about programs (e.g., do they do what they’re meant to do) in much the same way that we can reason about numbers (or other mathematical objects).

All formal methods, insofar as they are formal, depend upon such mathematical models.
Kinds of Semantics

Semantics comes in three flavours:

1. **Operational** — how programs are evaluated,
2. **Denotational** — what a program is (typically a function from ‘input’ to ‘output’)
3. **Axiomatic** — what logical inferences we can make about programs.
Kinds of Semantics

Semantics comes in three flavours:

1. Operational — how programs are evaluated,
2. Denotational — what a program is (typically a function from ‘input’ to ‘output’)
3. Axiomatic — what logical inferences we can make about programs.
Kinds of Semantics

Semantics comes in three flavours:

1. **Operational** — *how* programs are evaluated,
2. **Denotational** — *what* a program is (typically a function from ‘input’ to ‘output’)
3. **Axiomatic** — what logical inferences we can make about programs.
Axiomatic Semantics

mainly due to Hoare (Hoare Logic)

Most axiomatic approaches say how programs transform predicates about the values of variables. E.g.,

$$\{ y + 1 \leq 0 \} \ x = y + 1 \ \{ x \leq 0 \}$$

says that if $$y + 1 \leq 0$$ (i.e., $$y < 0$$), then after executing the assignment $$x = y + 1$$, it will be the case that $$x \leq 0$$.

and Dijkstra (weakest-precondition semantics)
Axiomatic Semantics

mainly due to Hoare (Hoare Logic)

Most axiomatic approaches say how programs transform predicates about the values of variables. E.g.,

\{ y + 1 \leq 0 \} x = y + 1 \{ x \leq 0 \}

says that if \( y + 1 \leq 0 \) (i.e., \( y < 0 \)), then after executing the assignment \( x = y + 1 \), it will be the case that \( x \leq 0 \).

and Dijkstra (weakest-precondition semantics)
Denotational Semantics

mainly due to Christopher Strachey

Denotational approaches say what programs (and expressions, etc.) \textit{denote}.

Typically, expressions denote numbers (sort of), and programs denote functions from states to states, where states are tables giving the values of variables.

E.g., $x = y + 1$ takes a state where $y$ has the value 23 to a state where $x$ has the value 24.

and Dana Scott
Denotational Semantics

mainly due to Christopher Strachey

Denotational approaches say what programs (and expressions, etc.) denote.

Typically, expressions denote numbers (sort of), and programs denote functions from states to states, where states are tables giving the values of variables.

E.g., \( x = y + 1 \) takes a state where \( y \) has the value 23 to a state where \( x \) has the value 24.

and Dana Scott
Operational Semantics

perhaps originating with Alan Turing

Before the first (electronic) computer was built, Turing used the idea of a **Turing Machine** to investigate what is computationally possible.

A Turing Machine strips computation down to bare essentials: it follows sequences of simple instructions, reading and writing 0s and 1s on a tape as it moves along.
Operational Semantics

perhaps originating with Alan Turing

Mathematics thrives on simplicity. Using this stripped-down model of computation, Turing was able to show exactly what computers could and could not do.

For example, there is no computer program that can decide whether any given program will or will not terminate/do what it’s meant to do.
Operational Semantics

Operational approaches often use mathematical **automata**, abstract machines that use a small set of instructions.

A notable example of operational semantics is the JVM.

Java is a rather complicated language; source code is compiled to byte code, instructions for a simple (abstract/mathematical/virtual) machine.

Mathematical abstraction is here a prerequisite to Java’s goal of platform-independent re-usability.
Operational approaches often use mathematical automata, abstract machines that use a small set of instructions.

A notable example of operational semantics is the JVM.

Java is a rather complicated language; source code is compiled to byte code, instructions for a simple (abstract/mathematical/virtual) machine.

Mathematical abstraction is here a prerequisite to Java’s goal of platform-independent re-usability.
Operational Semantics

Operational approaches often use mathematical automata, abstract machines that use a small set of instructions.

A notable example of operational semantics is the JVM.

Java is a rather complicated language; source code is compiled to byte code, instructions for a simple (abstract/mathematical/virtual) machine.

Mathematical abstraction is here a prerequisite to Java’s goal of platform-independent re-usability.
Our Approach

combines axiomatic, denotational and operational elements.

We use Maude to define the syntax and semantics of a SIMPLE† programming language

Maude is:

- a logical language (the logic of equations)
- a functional language
- executable

†Simple IMPerative LanguageE
Our Approach

combines axiomatic, denotational and operational elements.

We use Maude to define the syntax and semantics of a SIMPLE† programming language

Maude is:

- a logical language (the logic of equations)
- a functional language
- executable

†Simple IMPerative LanguagE
Our Approach

combines axiomatic, denotational and operational elements.

We use Maude to define the syntax and semantics of a SIMPLE† programming language

Maude is:

- a logical language (the logic of equations)
- a functional language
- executable

†Simple IMPerative LanguageE
Our Approach

combines axiomatic, denotational and operational elements.

We use Maude to define the syntax and semantics of a SIMPLE† programming language

Maude is:

- a logical language (the logic of equations)
- a functional language
- executable

†Simple IMPerative LanguageE
Our Approach

combines axiomatic, denotational and operational elements.

We use Maude to define the syntax and semantics of a SIMPLE† programming language

Maude is:

- a logical language (the logic of equations)
- a functional language
- executable

†Simple IMPerative LanguageE
Outline of the Module

- **Denotational Semantics**
  - binary numerals
  - semantics of SIMPLE
- **Algebraic Semantics**
  - denotational semantics re-done in Maude
- **Semantics of Maude**
  - denotational semantics (abstract data types)
  - operational semantics (term-rewriting)
    which gives an interpreter for SIMPLE
- **Program Verification**
  - invariants
  - Maude’s semantics lets us write and execute proofs of program correctness
- **Data Structures**
  - extending SIMPLE with arrays, stacks, etc.
Outline of the Module

- **Denotational Semantics**
  - binary numerals
  - semantics of SIMPLE

- **Algebraic Semantics**
  - denotational semantics re-done in Maude

- **Semantics of Maude**
  - denotational semantics (abstract data types)
  - operational semantics (term-rewriting)
    which gives an interpreter for SIMPLE

- **Program Verification**
  - invariants
  - Maude’s semantics lets us write and execute proofs of program correctness

- **Data Structures**
  - extending SIMPLE with arrays, stacks, etc.
Outline of the Module

- Denotational Semantics
  - binary numerals
  - semantics of SIMPLE

- Algebraic Semantics
  - denotational semantics re-done in Maude

- Semantics of Maude
  - denotational semantics (abstract data types)
  - operational semantics (term-rewriting)
    which gives an interpreter for SIMPLE

- Program Verification
  - invariants
  - Maude’s semantics lets us write and execute proofs of
    program correctness

- Data Structures
  - extending SIMPLE with arrays, stacks, etc.
Outline of the Module

- Denotational Semantics
  - binary numerals
  - semantics of SIMPLE

- Algebraic Semantics
  - denotational semantics re-done in Maude

- Semantics of Maude
  - denotational semantics (abstract data types)
  - operational semantics (term-rewriting)
    which gives an interpreter for SIMPLE

- Program Verification
  - invariants
  - Maude’s semantics lets us write and execute proofs of program correctness

- Data Structures
  - extending SIMPLE with arrays, stacks, etc.
Outline of the Module

- **Denotational Semantics**
  - binary numerals
  - semantics of SIMPLE

- **Algebraic Semantics**
  - denotational semantics re-done in Maude

- **Semantics of Maude**
  - denotational semantics (abstract data types)
  - operational semantics (term-rewriting)
    - which gives an interpreter for SIMPLE

- **Program Verification**
  - invariants
  - Maude’s semantics lets us write and execute proofs of program correctness

- **Data Structures**
  - extending SIMPLE with arrays, stacks, etc.
Why Semantics?

Intellectual Satisfaction

- Understanding syntax and semantics
- Basic mathematical computer science
Why Semantics?

Language Design

- there are many, many languages (programming, logical, design, formatting, . . .)
- most are ‘designed’ in an ad hoc way, with new features added as the need arises
- understanding the semantics of the language allows features to be added in a more disciplined way.

Case in point: Java generics.
Formal Methods

Most software life-cycles have little time for requirements and design, even though failings there tend to be most expensive. Very few software projects devote any time to verification or other formal methods.

There are exceptions, mainly in life-critical settings:

- the US Department of Defense requires all delivered software to be formally verified
- the avionics industry verifies much of its software
Why Semantics?

Formal Methods

- monitors and semaphores
- the alternating-bit protocol
- AVL trees
- the Knuth-Bendix pattern-matching algorithm
- scanning for metamorphic viruses
- the Java applet security protocol

All but the last were developed formally, hand-in-hand with a proof of correctness

(in the case of the last, the proof of correctness came later, and uncovered a bug, which was subsequently fixed).
Semantics builds mathematical models of programs
these models can be used to reason about programs
the main goal is verification, but semantics also helps with developing correct programs

Next
Syntax and semantics of binary numerals