Lecture 2
Introduction to Semantics
24 August 2012
Announcements

OCaml Demo
- 7-9pm tonight in Upson B7

Teaching Assistants
- Brittany Nkounkou
- Raghu Rajkumar

Homework #1
- Out: Monday, August 27th
- Due: Monday, September 3rd
- Distributed via CMS
Semantics

**Question:** What is the meaning of a program?
Semantics

Question: What is the meaning of a program?

Answer: We could execute the program using an interpreter or a compiler, or we could consult a manual...

...but neither of these is a satisfactory solution.
Formal Semantics

Three Approaches

• Operational
  ▶ Model program by execution on abstract machine
  ▶ Useful for implementing compilers and interpreters

\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle

• Axiomatic
  ▶ Model program by the logical formulas it obeys
  ▶ Useful for proving program correctness

\Gamma \vdash \{ \phi \} e \{ \psi \}

• Denotational:
  ▶ Model program as mathematical objects
  ▶ Useful for theoretical foundations

[ e ]
Arithmetic Expressions
Syntax

A language of integer arithmetic expressions with assignment.

BNF Grammar:

\[ e ::= x | n | e_1 + e_2 | e_1 \times e_2 | x := e_1 ; e_2 \]
Syntax

A language of integer arithmetic expressions with assignment.

Metavariables:

\[ x, y, z \in \text{Var} \]
\[ n, m \in \text{Int} \]
\[ e \in \text{Exp} \]
Syntax

A language of integer arithmetic expressions with assignment.

Metavariables:

\[ x, y, z \in \text{Var} \]
\[ n, m \in \text{Int} \]
\[ e \in \text{Exp} \]

BNF Grammar:

\[ e ::= x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid x := e_1 ; e_2 \]
Ambiguity

What expression does the string "1 + 2 * 3" describe?
Ambiguity

What expression does the string “1 + 2 * 3” describe?

There are two possible parse trees:

```
+  
  /\  
 1  *  
  /   
 2   3

*  
  /\  
 1  +  
  /   
 2   3
```
Ambiguity

What expression does the string “1 + 2 * 3” describe?

There are two possible parse trees:

In this course, we will distinguish abstract syntax from concrete syntax, and focus primarily on abstract syntax (using conventions or parentheses at the concrete level to disambiguate as needed).
Representing Expressions

BNF Grammar:

\[ e ::= x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid x := e_1 ; e_2 \]
Representing Expressions

BNF Grammar:

\[ e ::= x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid x := e_1 ; e_2 \]

OCaml:

```ocaml
type exp = Var of string |
| Int of int |
| Add of exp * exp |
| Mul of exp * exp |
| Assgn of string * exp * exp
```

Example: Mul(Int 2, Add(Var “foo”, Int 1))
Representing Expressions

BNF Grammar:

\[ e ::= x \mid n \mid e_1 + e_2 \mid e_1 * e_2 \mid x := e_1 ; e_2 \]

Java:

```java
abstract class Expr {
}
class Var extends Expr { String name; .. }
class Int extends Expr { int val; ... }
class Add extends Expr { Expr exp1, exp2; ... }
class Mul extends Expr { Expr exp1, exp2; ... }
class Assgn extends Expr { String var, Expr exp1, exp2; .. }
```

Example: new Mul(new Int(2), new Add(new Var("foo"), new Int(1)))
Quiz

- $7 + (4 \times 2)$ evaluates to ...?
Quiz

- $7 + (4 \times 2)$ evaluates to 15
Quiz

- $7 + (4 \times 2)$ evaluates to 15
- $i := 6 + 1$; $2 \times 3 \times i$ evaluates to ...?
Quiz

- $7 + (4 \times 2)$ evaluates to 15
- $i := 6 + 1; \ 2 \times 3 \times i$ evaluates to 42
Quiz

- $7 + (4 \times 2)$ evaluates to 15
- $i := 6 + 1; \ 2 \times 3 \times i$ evaluates to 42
- $x + 1$ evaluates to ...?
Quiz

- $7 + (4 \times 2)$ evaluates to 15
- $i := 6 + 1$; $2 \times 3 \times i$ evaluates to 42
- $x + 1$ evaluates to nothing?
Quiz

- $7 + (4 \times 2)$ evaluates to 15
- $i := 6 + 1; \ 2 \times 3 \times i$ evaluates to 42
- $x + 1$ evaluates to nothing?

The rest of this lecture will make these intuitions precise...
Mathematical Preliminaries
Binary Relations

The *product* of two sets $A$ and $B$, written $A \times B$, contains all ordered pairs $(a, b)$ with $a \in A$ and $b \in B$. 
Binary Relations

The *product* of two sets $A$ and $B$, written $A \times B$, contains all ordered pairs $(a, b)$ with $a \in A$ and $b \in B$.

A *binary relation* on $A$ and $B$ is just a subset $R \subseteq A \times B$. 
Binary Relations

The *product* of two sets $A$ and $B$, written $A \times B$, contains all ordered pairs $(a, b)$ with $a \in A$ and $b \in B$.

A *binary relation* on $A$ and $B$ is just a subset $R \subseteq A \times B$.

Given a binary relation $R \subseteq A \times B$, the set $A$ is called the *domain* of $R$ and $B$ is called the *range* (or *codomain*) of $R$. 
Binary Relations

The *product* of two sets $A$ and $B$, written $A \times B$, contains all ordered pairs $(a, b)$ with $a \in A$ and $b \in B$.

A *binary relation* on $A$ and $B$ is just a subset $R \subseteq A \times B$.

Given a binary relation $R \subseteq A \times B$, the set $A$ is called the *domain* of $R$ and $B$ is called the *range* (or *codomain*) of $R$.

Some Important Relations

- empty – $\emptyset$
- total – $A \times B$
- identity on $A$ – $\{(a, a) \mid a \in A\}$.
- composition $R; S$ – $\{(a, c) \mid \exists b. (a, b) \in R \land (b, c) \in S\}$
A (total) function \( f \) is a binary relation \( f \subseteq A \times B \) with the property that every \( a \in A \) is related to exactly one \( b \in B \).
A (total) function $f$ is a binary relation $f \subseteq A \times B$ with the property that every $a \in A$ is related to exactly one $b \in B$.

When $f$ is a function, we usually write $f : A \rightarrow B$ instead of $f \subseteq A \times B$. 
A (total) function $f$ is a binary relation $f \subseteq A \times B$ with the property that every $a \in A$ is related to exactly one $b \in B$.

When $f$ is a function, we usually write $f : A \to B$ instead of $f \subseteq A \times B$.

The domain and range of $f$ are defined the same way as for relations.
A (total) function $f$ is a binary relation $f \subseteq A \times B$ with the property that every $a \in A$ is related to exactly one $b \in B$.

When $f$ is a function, we usually write $f : A \rightarrow B$ instead of $f \subseteq A \times B$.

The domain and range of $f$ are defined the same way as for relations.

The image of $f$ is the set of elements $b \in B$ that are mapped to by at least one $a \in A$. More formally: $\text{image}(f) \triangleq \{f(a) \mid a \in A\}$.
Some Important Functions

Given two functions $f : A \rightarrow B$ and $g : B \rightarrow C$, the composition of $f$ and $g$ is defined by: $(g \circ f)(x) = g(f(x))$  

Note order!
Some Important Functions

Given two functions \( f : A \rightarrow B \) and \( g : B \rightarrow C \), the composition of \( f \) and \( g \) is defined by: \((g \circ f)(x) = g(f(x))\)  

Note order!

A partial function \( f : A \rightarrow B \) is a total function \( f : A' \rightarrow B \) on a set \( A' \subseteq A \). The notation \( \text{dom}(f) \) refers to \( A' \).
Some Important Functions

Given two functions \( f : A \to B \) and \( g : B \to C \), the composition of \( f \) and \( g \) is defined by: \((g \circ f)(x) = g(f(x))\)  
Note order!

A partial function \( f : A \to B \) is a total function \( f : A' \to B \) on a set \( A' \subseteq A \). The notation \( \text{dom}(f) \) refers to \( A' \).

A function \( f : A \to B \) is said to be injective (or one-to-one) if and only if \( a_1 \neq a_2 \) implies \( f(a_1) \neq f(a_2) \).
Some Important Functions

Given two functions $f : A \to B$ and $g : B \to C$, the composition of $f$ and $g$ is defined by: $(g \circ f)(x) = g(f(x))$ Note order!

A partial function $f : A \to B$ is a total function $f : A' \to B$ on a set $A' \subseteq A$. The notation $\text{dom}(f)$ refers to $A'$.

A function $f : A \to B$ is said to be injective (or one-to-one) if and only if $a_1 \neq a_2$ implies $f(a_1) \neq f(a_2)$.

A function $f : A \to B$ is said to be surjective (or onto) if and only if the image of $f$ is $B$. 
Operational Semantics
An operational semantics describes how a program executes on some (typically idealized) abstract machine.
An **operational semantics** describes how a program executes on some (typically idealized) abstract machine.

A **small-step** semantics describes how such an execution proceeds in terms of successive reductions: \( \langle \sigma, e \rangle \longrightarrow \langle \sigma', e' \rangle \)
Overview

An operational semantics describes how a program executes on some (typically idealized) abstract machine.

A small-step semantics describes how such an execution proceeds in terms of successive reductions: $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$

For our language, a configuration $\langle \sigma, e \rangle$ has two components:

- a store $\sigma$ that records the values of variables
- and the expression $e$ being evaluated
Overview

An operational semantics describes how a program executes on some (typically idealized) abstract machine.

A small-step semantics describes how such an execution proceeds in terms of successive reductions: \( \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \)

For our language, a configuration \( \langle \sigma, e \rangle \) has two components:
- a store \( \sigma \) that records the values of variables
- and the expression \( e \) being evaluated

More formally,

\[
\text{Store} \triangleq \text{Var} \rightarrow \text{Int} \\
\text{Config} \triangleq \text{Store} \times \text{Exp}
\]

Note that a store is a partial function from variables to integers.
Operational Semantics

The small-step operational semantics itself is a relation on configurations—i.e., a subset of $\text{Config} \times \text{Config}$. 
Operational Semantics

The small-step operational semantics itself is a relation on configurations—i.e., a subset of $\text{Config} \times \text{Config}$.

Notation: $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$
Operational Semantics

The small-step operational semantics itself is a relation on configurations—i.e., a subset of $\text{Config} \times \text{Config}$.

Notation: $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$

Question: How should we define this relation?
Operational Semantics

The small-step operational semantics itself is a relation on configurations—i.e., a subset of $\text{Config} \times \text{Config}$.

Notation: $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$

Question: How should we define this relation? Note that there are an infinite number of configurations and possible steps!
Operational Semantics

The small-step operational semantics itself is a relation on configurations—i.e., a subset of $\text{Config} \times \text{Config}$.

Notation: $\langle \sigma, e \rangle \mathrel{\rightarrow} \langle \sigma', e' \rangle$

Question: How should we define this relation? Note that there are an infinite number of configurations and possible steps!

Answer: define it inductively, using inference rules:

\[
p = m + n
\]

\[
\langle \sigma, n + m \rangle \mathrel{\rightarrow} \langle \sigma, p \rangle \quad \text{Add}
\]
Operational Semantics

The small-step operational semantics itself is a relation on configurations—i.e., a subset of $\text{Config} \times \text{Config}$.

**Notation:** $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$

**Question:** How should we define this relation? Note that there are an infinite number of configurations and possible steps!

**Answer:** define it inductively, using inference rules:

\[
\frac{p = m + n}{\langle \sigma, n + m \rangle \rightarrow \langle \sigma, p \rangle} \quad \text{Add}
\]

Intuitively, if facts above the line hold, then facts below the line hold. More formally, “$\rightarrow$” is the smallest relation “closed” under the inference rules.
Variables

\[ n = \sigma(x) \]

\[ \langle \sigma, x \rangle \longrightarrow \langle \sigma, n \rangle \]

Var
Addition

\[
\begin{align*}
\langle \sigma, e_1 \rangle & \rightarrow \langle \sigma', e'_1 \rangle & \text{LAdd} \\
\langle \sigma, e_1 + e_2 \rangle & \rightarrow \langle \sigma', e'_1 + e_2 \rangle
\end{align*}
\]
Addition

\[
\begin{align*}
\langle \sigma, e_1 \rangle & \rightarrow \langle \sigma', e'_1 \rangle & \text{LAdd} \\
\langle \sigma, e_1 + e_2 \rangle & \rightarrow \langle \sigma', e'_1 + e_2 \rangle \\
\langle \sigma, e_2 \rangle & \rightarrow \langle \sigma', e'_2 \rangle & \text{RAdd} \\
\langle \sigma, n + e_2 \rangle & \rightarrow \langle \sigma', n + e'_2 \rangle
\end{align*}
\]
Addition

\[
\begin{align*}
\langle \sigma, e_1 \rangle & \longrightarrow \langle \sigma', e'_1 \rangle \quad \text{LAdd} \\
\langle \sigma, e_1 + e_2 \rangle & \longrightarrow \langle \sigma', e'_1 + e_2 \rangle \\
\langle \sigma, e_2 \rangle & \longrightarrow \langle \sigma', e'_2 \rangle \quad \text{RAdd} \\
\langle \sigma, n + e_2 \rangle & \longrightarrow \langle \sigma', n + e'_2 \rangle \\
\langle \sigma, n + m \rangle & \longrightarrow \langle \sigma, p \rangle \quad \text{Add}
\end{align*}
\]
Multiplication

\[
\begin{align*}
\langle \sigma, e_1 \rangle & \quad \longrightarrow \quad \langle \sigma', e'_1 \rangle \\
\langle \sigma, e_1 \ast e_2 \rangle & \quad \longrightarrow \quad \langle \sigma', e'_1 \ast e_2 \rangle \\
\text{LMul} & \\
\langle \sigma, e_2 \rangle & \quad \longrightarrow \quad \langle \sigma', e'_2 \rangle \\
\langle \sigma, n \ast e_2 \rangle & \quad \longrightarrow \quad \langle \sigma', n \ast e'_2 \rangle \\
\text{RMul} & \\
p & = m \\
\langle \sigma, m \ast n \rangle & \quad \longrightarrow \quad \langle \sigma', p \rangle \\
\text{Mul} & 
\end{align*}
\]
Multiplication

\[
\begin{align*}
\langle \sigma, e_1 \rangle & \rightarrow \langle \sigma', e'_1 \rangle & \text{LMul} \\
\langle \sigma, e_1 \ast e_2 \rangle & \rightarrow \langle \sigma', e'_1 \ast e_2 \rangle \\
\langle \sigma, e_2 \rangle & \rightarrow \langle \sigma', e'_2 \rangle & \text{RMul} \\
\langle \sigma, n \ast e_2 \rangle & \rightarrow \langle \sigma', n \ast e'_2 \rangle
\end{align*}
\]
Multiplication

\[
\begin{align*}
\langle \sigma, e_1 \rangle & \rightarrow \langle \sigma', e'_1 \rangle & \text{LMul} \\
\langle \sigma, e_1 \star e_2 \rangle & \rightarrow \langle \sigma', e'_1 \star e_2 \rangle \\
\langle \sigma, e_2 \rangle & \rightarrow \langle \sigma', e'_2 \rangle & \text{RMul} \\
\langle \sigma, n \star e_2 \rangle & \rightarrow \langle \sigma', n \star e'_2 \rangle \\
p = m \times n & \\
\langle \sigma, m \star n \rangle & \rightarrow \langle \sigma, p \rangle & \text{Mul}
\end{align*}
\]
Assignment

\[
\begin{align*}
\langle \sigma, e_1 \rangle & \rightarrow \langle \sigma', e'_1 \rangle \\
\langle \sigma, x := e_1 ; e_2 \rangle & \rightarrow \langle \sigma', x := e'_1 ; e_2 \rangle
\end{align*}
\]

Assgn1

Notation: \([x := n]\) maps \(x\) to \(n\) and otherwise behaves like 20
Assignment

\[
\begin{align*}
\langle \sigma, e_1 \rangle & \rightarrow \langle \sigma', e'_1 \rangle & \text{Assgn1} \\
\langle \sigma, x := e_1 ; e_2 \rangle & \rightarrow \langle \sigma', x := e'_1 ; e_2 \rangle \\
\sigma' &= \sigma[x \mapsto n] & \text{Assgn} \\
\langle \sigma, x := n ; e_2 \rangle & \rightarrow \langle \sigma', e_2 \rangle
\end{align*}
\]

Notation: \( \sigma[x \mapsto n] \) maps \( x \) to \( n \) and otherwise behaves like \( \sigma \)
Operational Semantics

\[
\begin{align*}
\text{LMul} & : \langle \sigma', e_1 \rangle \rightarrow \langle \sigma', e'_1 \rangle \\
\text{RAdd} & : \langle \sigma, n + e_2 \rangle \rightarrow \langle \sigma, n + e'_2 \rangle \\
\text{Add} & : \langle \sigma, n \rangle \rightarrow \langle \sigma, p \rangle \\
\text{Var} & : \langle \sigma, x \rangle \rightarrow \langle \sigma, x \rangle \\
\text{Assign1} & : \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \\
\text{LMul} & : \langle \sigma', e_1 \rangle \rightarrow \langle \sigma', e'_1 \rangle \\
\text{RAdd} & : \langle \sigma, m \times n \rangle \rightarrow \langle \sigma, p \rangle \\
\text{Mul} & : \langle \sigma, e_1 \rangle \rightarrow \langle \sigma, e'_1 \rangle \\
\text{Assign2} & : \langle \sigma, x = e \rangle \rightarrow \langle \sigma, x = e' \rangle \\
\text{Mul} & : \langle \sigma, m \times n \rangle \rightarrow \langle \sigma, p \rangle \\
\text{Add} & : \langle \sigma, n \rangle \rightarrow \langle \sigma, p \rangle \\
\end{align*}
\]