Lecture 2
Introduction to 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 none of these is a satisfactory solution.

---

**A8.7 Void**

The (nonexistent) value of a `void` object may not be used in any way, and neither explicit nor implicit conversion to any non-void type may be applied. Because a `void` expression denotes a nonexistent value, such an expression may be used only where the value is not required, for example as an expression statement (§A9.2) or as the left operand of a comma operator (§A7.18).

An expression may be converted to type `void` by a cast. For example, a `void` cast documents the discarding of the value of a function call used as an expression statement.

`void` did not appear in the first edition of this book, but has become common since.
Formal Semantics

Three Approaches

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

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

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

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

\[ \llbracket e \rrbracket \]

\[ \vdash \{ \phi \} e \{ \psi \} \]
Arithmetic Expressions
Syntax

A language of integer arithmetic expressions with assignment.
Syntax

A language of integer arithmetic expressions with assignment.

Metavariables:

\[
\begin{align*}
x, y, z & \in \text{Var} \\
n, m & \in \text{Int} \\
e & \in \text{Exp}
\end{align*}
\]
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:

\[ e ::= x \]
\[ n \]
\[ e_1 + e_2 \]
\[ e_1 \times e_2 \]
\[ x := e_1 ; e_2 \]
What expression does the string “1 + 2 * 3” describe?
What expression does the string “1 + 2 * 3” describe?

There are two possible parse trees:
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:

\[ e ::= x \]
\[ | n \]
\[ | e_1 + e_2 \]
\[ | e_1 * e_2 \]
\[ | x := e_1 ; e_2 \]
Representing Expressions

BNF Grammar:

\[
e ::= x \\

| n \\
| e_1 + e_2 \\
| e_1 * e_2 \\
| 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:

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

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 error?
Quiz

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

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 $b \in B$. 
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** $R; S$: $\{(a, c) \mid \exists b. (a, b) \in R \land (b, c) \in S\}$
Functions

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 $f : A \rightarrow B$ instead of $f \subseteq A \times B$.

The *image* of $f$ is the set of elements $b \in B$ that are mapped to by at least one $a \in A$. 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:

\[
(g \circ f)(x) = g(f(x)) \quad \text{Note order!}
\]

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 \( A' \).

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 \( f : A \rightarrow B \) 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 \( A' \).

A function \( f : A \rightarrow B \) 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 \( f : A \rightarrow B \) is said to be surjective (or onto) if and only if the image of \( f \) is \( B \).
Operational Semantics
Overview

An operational semantics describes how a program executes on some abstract (imaginary) machine.
An **operational semantics** describes how a program executes on some abstract (imaginary) machine.

A **small-step** semantics describes how such an execution proceeds from configuration to configuration: $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$
Overview

An **operational semantics** describes how a program executes on some abstract (imaginary) machine.

A **small-step semantics** describes how such an execution proceeds from configuration to configuration: \( \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \)

For our language, a **configuration** \( \langle \sigma, e \rangle \) is a pair of:

- 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 abstract (imaginary) machine.

A small-step semantics describes how such an execution proceeds from configuration to configuration: \( \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \)

For our language, a configuration \( \langle \sigma, e \rangle \) is a pair of:

- a store \( \sigma \) that records the values of variables,
- and the expression \( e \) being evaluated.

More formally:

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

(A store is a partial function from variables to integers.)
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: $\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$

which means $(\langle \sigma, e \rangle, \langle \sigma', e' \rangle) \in \rightarrow$. 
Operational Semantics

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

Notation: \( \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \)
which means \((\langle \sigma, e \rangle, \langle \sigma', e' \rangle) \in "\rightarrow".\)

Question: How should we define this relation?
The small-step operational semantics itself is a relation on configurations—i.e., a subset of \textbf{Config} $\times$ \textbf{Config}.

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

which means $(\langle \sigma, e \rangle, \langle \sigma', e' \rangle) \in \text{“} \rightarrow \text{”}.$

**Question:** How should we define this relation? Remember that there are an infinite number of configurations and possible steps!
Answer: Define it inductively, using inference rules:

\[
\begin{array}{cccc}
\text{premise}_1 & \text{premise}_2 & \cdots & \text{Name} \\
\hline
\text{conclusion} & & & \\
\end{array}
\]
Inference Rules

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

\[
\begin{array}{c}
\text{premise}_1 & \text{premise}_2 & \cdots \\
\hline \\
\text{conclusion} & \text{NAME} \\
\end{array}
\]

An inference rule defines an implication: if all the *premises* hold, then the *conclusion* also holds.

Formally, “→” is the smallest relation that is closed under all the inference rules.
Variables

\[ n = \sigma(x) \]

\[ \langle \sigma, x \rangle \rightarrow \langle \sigma, n \rangle \quad \text{VAR} \]
Addition

\[ p = m + n \]

\[ \langle \sigma, n + m \rangle \rightarrow \langle \sigma, p \rangle \]
Addition

\[
p = m + n
\]

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

\[
\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e'_1 \rangle
\]

\[
\langle \sigma, e_1 + e_2 \rangle \rightarrow \langle \sigma', e'_1 + e_2 \rangle \quad \text{LADD}
\]
Addition

\[ p = m + n \]

\[ \langle \sigma, n + m \rangle \rightarrow \langle \sigma, p \rangle \]  \text{ADD}

\[ \langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e_1' \rangle \]  \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}

\[ \langle \sigma, n + e_2 \rangle \rightarrow \langle \sigma', n + e_2' \rangle \]
Multiplication

\[ p = m \times n \]

\[ \langle \sigma, m \times n \rangle \rightarrow \langle \sigma, p \rangle \]

\[ \text{MUL} \]
Multiplication

\[ p = m \times n \]

\[ \langle \sigma, m \times n \rangle \rightarrow \langle \sigma, p \rangle \quad \text{MUL} \]

\[ \langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e'_1 \rangle \]

\[ \langle \sigma, e_1 \times e_2 \rangle \rightarrow \langle \sigma', e'_1 \times e_2 \rangle \quad \text{LMUL} \]

\[ \langle \sigma, e_2 \rangle \rightarrow \langle \sigma', e'_2 \rangle \]

\[ \langle \sigma, n \times e_2 \rangle \rightarrow \langle \sigma', n \times e'_2 \rangle \quad \text{RMUL} \]
Assignment

\[
\sigma' = \sigma[x \mapsto n] \\
\langle \sigma, x := n ; e_2 \rangle \rightarrow \langle \sigma', e_2 \rangle \quad \text{ASSGN}
\]

Notation: \( \sigma[x \mapsto n] \) is a new function that mostly behaves like \( \sigma \), except that it maps \( x \) to \( n \).
\[
\sigma' = \sigma[x \mapsto n] \\
\langle \sigma, x := n ; e_2 \rangle \rightarrow \langle \sigma', e_2 \rangle
\]

\text{ASSGN}

\textbf{Notation:} \( \sigma[x \mapsto n] \) is a \textit{new} function that mostly behaves like \( \sigma \), except that it maps \( x \) to \( n \).

\[
\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e'_1 \rangle \\
\langle \sigma, x := e_1 ; e_2 \rangle \rightarrow \langle \sigma', x := e'_1 ; e_2 \rangle
\]

\text{ASSGN1}
Operational Semantics

\[
\begin{align*}
  n &= \sigma(x) \\
  \langle \sigma, x \rangle &\to \langle \sigma, n \rangle & \text{VAR} \\
  \langle \sigma, e_2 \rangle &\to \langle \sigma', e'_2 \rangle & \text{RADD} \\
  \langle \sigma, n + e_2 \rangle &\to \langle \sigma', n + e'_2 \rangle & \text{LADD} \\
  \langle \sigma, e_1 \rangle &\to \langle \sigma', e'_1 \rangle & \text{LMUL} \\
  \langle \sigma, e_1 * e_2 \rangle &\to \langle \sigma', e'_1 * e_2 \rangle & \text{RMUL} \\
  p &= m \times n \\
  \langle \sigma, m * n \rangle &\to \langle \sigma, p \rangle & \text{MUL} \\
  \langle \sigma, x := e_1 ; e_2 \rangle &\to \langle \sigma', x := e'_1 ; e_2 \rangle & \text{ASSGN1} \\
  \sigma' &= \sigma[x \mapsto n] \\
  \langle \sigma, x := n ; e_2 \rangle &\to \langle \sigma', e_2 \rangle & \text{ASSGN}
\end{align*}
\]
Multi-Step Evaluation

We can define the multi-step evaluation relation, written $\rightarrow^*$, as the reflexive and transitive closure of the small-step evaluation relation.

\[
\begin{align*}
\langle \sigma, e \rangle &\rightarrow^* \langle \sigma, e \rangle & \text{REFL} \\
\langle \sigma, e \rangle &\rightarrow \langle \sigma, e' \rangle \langle \sigma', e' \rangle & \langle \sigma', e' \rangle &\rightarrow^* \langle \sigma'', e'' \rangle & \text{STEP} \\
\langle \sigma, e \rangle &\rightarrow^* \langle \sigma'', e'' \rangle & \text{TRANS}
\end{align*}
\]