CS 4110

Programming Languages & Logics

Lecture 2
Introduction to Semantics

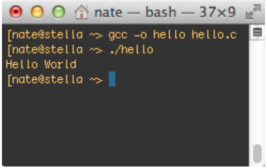
Semantics

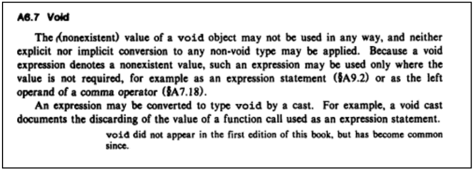
Question: What is the meaning of a program?

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...




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.

...but none of these is a satisfactory solution.

Formal Semantics

Three Approaches

- Operational $\langle \sigma, e \rangle \longrightarrow \langle \sigma', e' \rangle$
  - Model program by execution on abstract machine
  - Useful for implementing compilers and interpreters
- Denotational: $[\![e]\!]$
  - Model program as mathematical objects
  - Useful for theoretical foundations
- Axiomatic $\vdash \{\phi\}\ e\ \{\psi\}$
  - Model program by the logical formulas it obeys
  - Useful for proving program correctness

Arithmetic Expressions

Syntax

A language of integer arithmetic expressions with assignment.

Syntax

A language of integer arithmetic expressions with assignment.

Metavariables:

$$\begin{array}{rcl} x, y, z & \in & \mathbf{Var} \\ n, m & \in & \mathbf{Int} \\ e & \in & \mathbf{Exp} \end{array}$$

Syntax

A language of integer arithmetic expressions with assignment.

Metavariables:

$$\begin{array}{rcl} x, y, z & \in & \mathbf{Var} \\ n, m & \in & \mathbf{Int} \\ e & \in & \mathbf{Exp} \end{array}$$

BNF Grammar:

$$\begin{array}{rl} e ::= & x \\ | & n \\ | & e_1 + e_2 \\ | & e_1 * e_2 \\ | & x := e_1 \,;\; e_2 \end{array}$$

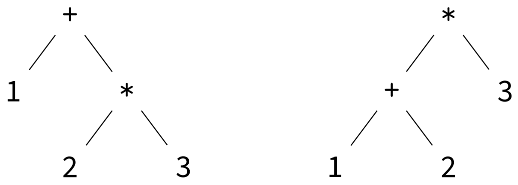
Ambiguity

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

Ambiguity

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

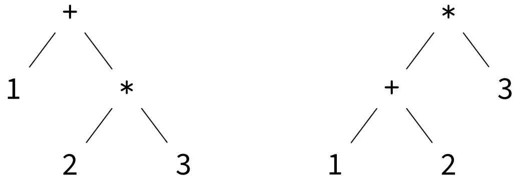
There are two possible parse trees:



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:

$$
\begin{aligned}
e ::= {} & x \\
\mid {} & n \\
\mid {} & e_1 + e_2 \\
\mid {} & e_1 * e_2 \\
\mid {} & x := e_1 \,;\; e_2
\end{aligned}
$$

Representing Expressions

BNF Grammar:

$$
\begin{aligned}
e ::= \; & x \\
\mid \; & n \\
\mid \; & e_1 + e_2 \\
\mid \; & e_1 * e_2 \\
\mid \; & x := e_1 \,;\; e_2
\end{aligned}
$$

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:

$$
\begin{aligned}
e ::= {} & x \\
\mid {} & n \\
\mid {} & e_1 + e_2 \\
\mid {} & e_1 * e_2 \\
\mid {} & x := e_1 \,;\; e_2
\end{aligned}
$$

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

- $7 + (4 * 2)$ evaluates to ...?

Quiz

- $7 + (4 * 2)$ evaluates to 15

Quiz

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

Quiz

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

Quiz

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

Quiz

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

Quiz

- $7 + (4 * 2)$ evaluates to 15
- $i := 6 + 1\,;\; 2 * 3 * i$ 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 $b \in B$.

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 $R; S$: $\{(a, c) \mid \exists b. \, (a, b) \in R \wedge (b, c) \in S\}$

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 $b \in B$.

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 $b \in B$.

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

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 $b \in B$.

When f is a function, we usually write $f : A \to B$ 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:

$$\mathrm{image}(f) \triangleq \{f(a) \mid a \in A\}$$

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!

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

Overview

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

Overview

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

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

Overview

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

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

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

- a store σ 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 operational semantics describes how such an execution proceeds from configuration to configuration:
$\langle \sigma, e \rangle \to \langle \sigma', e' \rangle$

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

- a store σ that records the values of variables,
- and the expression e being evaluated.

More formally:

$$\begin{array}{rcl} \textbf{Store} & \triangleq & \textbf{Var} \rightharpoonup \textbf{Int} \\ \textbf{Config} & \triangleq & \textbf{Store} \times \textbf{Exp} \end{array}$$

(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 **Config** $\times$ **Config**.

Operational Semantics

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

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

Operational Semantics

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

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

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 **Config** $\times$ **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? Remember that there are an infinite number of configurations and possible steps!

Inference Rules

Answer: Define it inductively, using inference rules:

$$\frac{\text{premise}_1 \qquad \text{premise}_2 \qquad \cdots}{\text{conclusion}}\ \textsc{Name}$$

Inference Rules

Answer: Define it inductively, using inference rules:

$$\frac{\text{premise}_1 \qquad \text{premise}_2 \qquad \cdots}{\text{conclusion}}\ \textsc{Name}$$

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

Formally, "$\rightarrow$" is the smallest relation that is closed under all the inference rules.

Variables

$$\frac{n = \sigma(x)}{\langle \sigma, x \rangle \to \langle \sigma, n \rangle} \textsc{Var}$$

Addition

$$\frac{p = m \mathbin{\color{blue}{+}} n}{\langle \sigma, n + m \rangle \to \langle \sigma, p \rangle}\ \textsc{Add}$$

Addition

$$\frac{p = m \mathbin{\color{blue}{+}} n}{\langle \sigma, n + m \rangle \to \langle \sigma, p \rangle}\ \textsc{Add}$$

$$\frac{\langle \sigma, e_1 \rangle \to \langle \sigma', e_1' \rangle}{\langle \sigma, e_1 + e_2 \rangle \to \langle \sigma', e_1' + e_2 \rangle}\ \textsc{LAdd}$$

Addition

$$\frac{p = m \mathbin{\color{blue}+} n}{\langle \sigma, n + m \rangle \rightarrow \langle \sigma, p \rangle} \text{ ADD}$$

$$\frac{\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e_1' \rangle}{\langle \sigma, e_1 + e_2 \rangle \rightarrow \langle \sigma', e_1' + e_2 \rangle} \text{ LADD}$$

$$\frac{\langle \sigma, e_2 \rangle \rightarrow \langle \sigma', e_2' \rangle}{\langle \sigma, n + e_2 \rangle \rightarrow \langle \sigma', n + e_2' \rangle} \text{ RADD}$$

Multiplication

$$\frac{p = m \times n}{\langle \sigma, m * n \rangle \rightarrow \langle \sigma, p \rangle}\ \textsc{Mul}$$

Multiplication

$$\frac{p = m \times n}{\langle \sigma, m * n \rangle \rightarrow \langle \sigma, p \rangle}\ \textsc{Mul}$$

$$\frac{\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e_1' \rangle}{\langle \sigma, e_1 * e_2 \rangle \rightarrow \langle \sigma', e_1' * e_2 \rangle}\ \textsc{LMul}$$

$$\frac{\langle \sigma, e_2 \rangle \rightarrow \langle \sigma', e_2' \rangle}{\langle \sigma, n * e_2 \rangle \rightarrow \langle \sigma', n * e_2' \rangle}\ \textsc{RMul}$$

Assignment

$$\frac{\sigma' = \sigma[x \mapsto n]}{\langle \sigma, x := n\,;\; e_2 \rangle \to \langle \sigma', e_2 \rangle}\ \textsc{Assgn}$$

Notation: $\sigma[x \mapsto n]$ is a *new* (partial) function that mostly behaves like σ, except that it maps x to n.

Assignment

$$\frac{\sigma' = \sigma[x \mapsto n]}{\langle \sigma, x := n\,;\ e_2 \rangle \to \langle \sigma', e_2 \rangle}\ \textsc{Assgn}$$

Notation: $\sigma[x \mapsto n]$ is a *new* (partial) function that mostly behaves like σ, except that it maps x to n.

$$\frac{\langle \sigma, e_1 \rangle \to \langle \sigma', e_1' \rangle}{\langle \sigma, x := e_1\,;\ e_2 \rangle \to \langle \sigma', x := e_1'\,;\ e_2 \rangle}\ \textsc{Assgn1}$$

Operational Semantics

$$\frac{n = \sigma(x)}{\langle \sigma, x \rangle \rightarrow \langle \sigma, n \rangle} \textsc{Var}$$

$$\frac{\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e_1' \rangle}{\langle \sigma, e_1 + e_2 \rangle \rightarrow \langle \sigma', e_1' + e_2 \rangle} \textsc{LAdd}$$

$$\frac{\langle \sigma, e_2 \rangle \rightarrow \langle \sigma', e_2' \rangle}{\langle \sigma, n + e_2 \rangle \rightarrow \langle \sigma', n + e_2' \rangle} \textsc{RAdd}$$

$$\frac{p = m + n}{\langle \sigma, n + m \rangle \rightarrow \langle \sigma, p \rangle} \textsc{Add}$$

$$\frac{\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e_1' \rangle}{\langle \sigma, e_1 * e_2 \rangle \rightarrow \langle \sigma', e_1' * e_2 \rangle} \textsc{LMul}$$

$$\frac{\langle \sigma, e_2 \rangle \rightarrow \langle \sigma', e_2' \rangle}{\langle \sigma, n * e_2 \rangle \rightarrow \langle \sigma', n * e_2' \rangle} \textsc{RMul}$$

$$\frac{p = m \times n}{\langle \sigma, m * n \rangle \rightarrow \langle \sigma, p \rangle} \textsc{Mul}$$

$$\frac{\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e_1' \rangle}{\langle \sigma, x := e_1 \,;\; e_2 \rangle \rightarrow \langle \sigma', x := e_1' \,;\; e_2 \rangle} \textsc{Assgn1}$$

$$\frac{\sigma' = \sigma[x \mapsto n]}{\langle \sigma, x := n \,;\; e_2 \rangle \rightarrow \langle \sigma', e_2 \rangle} \textsc{Assgn}$$

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.

$$\frac{}{\langle \sigma, e \rangle \rightarrow^* \langle \sigma, e \rangle}\ \textsc{Refl}$$

$$\frac{\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \qquad \langle \sigma', e' \rangle \rightarrow^* \langle \sigma'', e'' \rangle}{\langle \sigma, e \rangle \rightarrow^* \langle \sigma'', e'' \rangle}\ \textsc{Trans}$$