Outline

What is semantics?

Defining a denotational semantics
What is the meaning of a program?

Recall: aspects of a language
- **syntax**: the structure of its programs
- **semantics**: the meaning of its programs
How to define the meaning of a program?

Formal specifications
- **denotational semantics**: relates terms directly to values
- **operational semantics**: describes how to evaluate a term
- **axiomatic semantics**: describes the effects of evaluating a term
- ...

Informal/non-specifications
- **reference implementation**: execute/compile program in some implementation
- **community/designer intuition**: how people “think” a program should behave
Advantages of a formal semantics

A formal semantics …

• is **simpler** than an implementation, **more precise** than intuition
  • can answer: is this implementation correct?

• supports the definition of analyses and transformations
  • prove properties about the language
  • prove properties about programs written in the language

• promotes better language design
  • better understand impact of design decisions
  • apply semantic insights to improve the language design (e.g. *compositionality*)
Outline

What is semantics?

Defining a denotational semantics
Denotational semantics

A denotational semantics relates each term to a denotation:

- an abstract syntax tree
- a value in some semantic domain

Valuation function

\[ [\cdot] : \text{abstract syntax} \rightarrow \text{semantic domain} \]

Valuation function in Haskell

```
eval :: Term -> Value
```
Semantic domain: captures the set of possible meanings of a program/term

What is a meaning? — it depends on the language!

Example semantic domains

<table>
<thead>
<tr>
<th>Language</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean expressions</td>
<td>Boolean value</td>
</tr>
<tr>
<td>Arithmetic expressions</td>
<td>Integer</td>
</tr>
<tr>
<td>Imperative language</td>
<td>State transformation</td>
</tr>
<tr>
<td>SQL query</td>
<td>Set of relations</td>
</tr>
<tr>
<td>ActionScript</td>
<td>Animation</td>
</tr>
<tr>
<td>MIDI</td>
<td>Sound waves</td>
</tr>
</tbody>
</table>
Defining a language with denotational semantics

Example encoding in Haskell:

1. Define the **abstract syntax**, $T$
   the set of abstract syntax trees
data Term = ...

2. Identify or define the **semantic domain**, $V$
   the representation of semantic values
type Value = ...

3. Define the **valuation function**, $\mathbb{L} : T \to V$
   the mapping from ASTs to semantic values
   a.k.a. the “semantic function”
sem :: Term -> Value
Example: simple arithmetic expressions

1. Define abstract syntax

\[\begin{align*}
n \in \text{Nat} &::= 0 \mid 1 \mid 2 \mid \ldots \\
e \in \text{Exp} &::= \text{add } e \ e \\
 &\mid \text{mul } e \ e \\
 &\mid \text{neg } e \\
 &\mid n
\end{align*}\]

2. Define semantic domain

Use the set of all integers, \(\text{Int}\)

Comes with some operations:
\(+, \times, -, \text{toInt} : \text{Nat} \to \text{Int}, \ldots\)

3. Define the valuation function

\[\begin{align*}
[\text{Exp}] : \text{Int} \\
[\text{add } e_1 \ e_2] &= [e_1] + [e_2] \\
[\text{mul } e_1 \ e_2] &= [e_1] \times [e_2] \\
[\text{neg } e] &= -[e] \\
[n] &= \text{toInt}(n)
\end{align*}\]
1. **abstract syntax**: define a new **data type**, as usual
2. **semantic domain**: identify and/or define a new **type**, as needed
3. **valuation function**: define a **function** from ASTs to semantic domain

Valuation function in Haskell

```haskell
sem :: Exp -> Int
sem (Add l r) = sem l + sem r
sem (Mul l r) = sem l * sem r
sem (Neg e)   = negate (sem e)
sem (Lit n)   = n
```
Desirable properties of a denotational semantics

**Compositionality:** a program’s denotation is built from the denotations of its parts
- supports modular reasoning, extensibility
- supports proof by structural induction

**Completeness:** every value in the semantics domain is denoted by some program
- ensures that semantics domain and language align
- if not, language has expressiveness gaps, or semantics domain is too general

**Soundness:** two programs are “equivalent” iff they have the same denotation
- equivalence: have the same observable effects
- ensures that the denotational semantics is correct
More on compositionality

**Compositionality**: a program’s denotation is built from the denotations of its parts

- an AST
- sub-ASTs

Example: What is the meaning of $\text{op } e_1 \ e_2 \ e_3$?

1. Determine the meaning of $e_1, e_2, e_3$
2. Combine these submeanings in some way specific to $\text{op}$

Implications:

- The valuation function is probably **recursive**
- We need different valuation functions for each **syntactic category** (type of AST)
Example: move language

A language describing movements on a 2D plane
- a **step** is an $n$-unit horizontal or vertical movement
- a **move** is described by a sequence of steps

**Abstract syntax**

\[
\begin{align*}
n &\in \text{Nat} &::= &\emptyset | 1 | 2 | \ldots \\
d &\in \text{Dir} &::= &N | S | E | W \\
s &\in \text{Step} &::= &\text{go} \ d \ n \\
m &\in \text{Move} &::= &\epsilon \ | \ s \ ; \ m
\end{align*}
\]

```
go N 3; go E 4; go S 1;
```
Semantics of move language

1. Abstract syntax

\[ n \in \text{Nat} ::= 0 | 1 | 2 | \ldots \]
\[ d \in \text{Dir} ::= N | S | E | W \]
\[ s \in \text{Step} ::= \text{go } d \ n \]
\[ m \in \text{Move} ::= \epsilon | s ; m \]

2. Semantic domain

\[ \text{Pos} = \text{Int} \times \text{Int} \]
Domain: \( \text{Pos} \rightarrow \text{Pos} \)

3. Valuation function \((\text{Step})\)

\[ S[\text{Step}] : \text{Pos} \rightarrow \text{Pos} \]
\[ S[\text{go N } k] = \lambda (x, y). (x, y + k) \]
\[ S[\text{go S } k] = \lambda (x, y). (x, y - k) \]
\[ S[\text{go E } k] = \lambda (x, y). (x + k, y) \]
\[ S[\text{go W } k] = \lambda (x, y). (x - k, y) \]

3. Valuation function \((\text{Move})\)

\[ M[\text{Move}] : \text{Pos} \rightarrow \text{Pos} \]
\[ M[\epsilon] = \lambda p. p \]
\[ M[s ; m] = M[m] \circ S[s] \]
Alternative semantics

Often multiple interpretations (semantics) of the same language

Example: Database schema
One declarative spec, used to:
- initialize the database
- generate APIs
- validate queries
- normalize layout
- ...

Distance traveled
\[
S_D[\text{Step}] : \text{Int} \\
S_D[\text{go } d \ k] = k \\
M_D[\text{Move}] : \text{Int} \\
M_D[\epsilon] = 0 \\
M_D[s ; m] = S_D[s] + M_D[m]
\]

Combined trip information
\[
M_C[\text{Move}] : \text{Int} \times (\text{Pos} \to \text{Pos}) \\
M_C[m] = (M_D[m], M[m])
\]