Semantics
Outline

What is semantics?

Denotational semantics

Semantics of naming
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?

<table>
<thead>
<tr>
<th>Formal specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>denotational semantics</strong>: relates terms directly to values</td>
</tr>
<tr>
<td>• <strong>operational semantics</strong>: describes how to evaluate a term</td>
</tr>
<tr>
<td>• <strong>axiomatic semantics</strong>: describes the effects of evaluating a term</td>
</tr>
<tr>
<td>• ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Informal/non-specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>reference implementation</strong>: execute/compile program in some implementation</td>
</tr>
<tr>
<td>• <strong>community/designer intuition</strong>: how people “think” a program should behave</td>
</tr>
</tbody>
</table>
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

• 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?

Denotational semantics

Semantics of naming
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**

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

*what is a meaning? — it depends on the language!*

<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>MiniLogo program</td>
<td>Drawing</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*

   ```haskell
data Term = ...
```

2. Identify or define the **semantic domain**, \( V \)  
   *the representation of semantic values*

   ```haskell
type Value = ...
```

3. Define the **valuation function**, \( \llbracket \cdot \rrbracket : T \to V \)  
   *the mapping from ASTs to semantic values*

   ```haskell
sem :: Term -> Value
```
Example: simple arithmetic expressions

1. Define abstract syntax

```haskell
data Exp = Add Exp Exp
    | Mul Exp Exp
    | Neg Exp
    | Lit Int
```

2. Identify semantic domain

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

3. Define the valuation function

```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 semantic domain is denoted by some program
- ensures that semantic domain and language align
- if not, language has expressiveness gaps, or semantic domain is too general

**Soundness**: if two programs are “equivalent” then they have the same denotation
- equivalence: e.g. by some syntactic rule or law
- ensures the equivalence relation and denotational semantics are correct
**Compositionality**: a program’s denotation is built from the denotations of its parts

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: simple arithmetic expressions (again)

1. Define abstract syntax
   `data Exp = Add Exp Exp |
   Mul Exp Exp |
   Neg Exp |
   Lit Int`

2. Identify semantic domain
   Use the set of all integers, `Int`

3. Define the valuation function
   `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`
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**

```haskell
data Dir = N | S | E | W
data Step = Go Dir Int
type Move = [Step]
```

```
[Go N 3, Go E 4, Go S 1]
```
Semantics of move language

1. Abstract syntax

```haskell
data Dir = N | S | E | W
data Step = Go Dir Int
type Move = [Step]
```

2. Identify semantic domain

```haskell
type Pos = (Int,Int)
Domain: Pos -> Pos
```

3. Valuation function (Step)

```haskell
step :: Step -> Pos -> Pos
step (Go N k) = \(x,y) -> (x,y+k)
step (Go S k) = \(x,y) -> (x,y-k)
step (Go E k) = \(x,y) -> (x+k,y)
step (Go W k) = \(x,y) -> (x-k,y)
```

3. Valuation function (Move)

```haskell
move :: Move -> Pos -> Pos
move [] = \p -> p
move (s:m) = move m . step 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

type Dist = Int

dstep :: Step -> Int

dstep (Go _ k) = k

dmove :: Move -> Int

dmove [] = 0

dmove (s:m) = dstep s + dmove m

Combined trip information

trip :: Move -> Pos -> (Dist, Pos)

trip m = \p -> (dmove m, move m p)
Picking the right semantic domain (1/2)

Simple semantic domains can be combined in two ways:

- **sum**: contains a value from one domain or the other
  - e.g. IntBool language can evaluate to **Int** or **Bool**
  - use Haskell **Either a b** or define a new data type

- **product**: contains a value from both domains
  - e.g. combined trip information for move language
  - use Haskell **(a,b)** or define a new data type
Picking the right semantic domain (2/2)

Can errors occur?
- use Haskell `Maybe` or define a new data type

Does the language manipulate state or use names?
- use a function type

**Example stateful domains**

- Read-only state: \[ \text{State} \rightarrow \text{Value} \]
- Modify as only effect: \[ \text{State} \rightarrow \text{State} \]
- Modify as side effect: \[ \text{State} \rightarrow (\text{State},\text{Value}) \]
Outline

What is semantics?

Denotational semantics

Semantics of naming
What is naming?

Most languages provide a way to name and reuse stuff

### Naming concepts

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Introduce a new name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding</td>
<td>Associate a name with a thing</td>
</tr>
<tr>
<td>Reference</td>
<td>Use the name to stand for the bound thing</td>
</tr>
</tbody>
</table>

### C/Java variables

```java
int x; int y;
x = slow(42);
y = x + x + x;
```

### In Haskell:

#### Local variables

```haskell
let x = slow 42
in x + x + x
```

#### Type names

```haskell
type Radius = Float
data Shape = Circle Radius
```

#### Function parameters

```haskell
area r = pi * r * r
```
Semantics of naming

Environment: a mapping from names to things

type Env = Name -> Thing

Naming concepts

- **declaration**: add a new name to the environment
- **binding**: set the thing associated with a name
- **reference**: get the thing associated with a name

Example semantic domains for expressions with ...

<table>
<thead>
<tr>
<th>Type</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>immutable vars (Haskell)</td>
<td>Env -&gt; Val</td>
</tr>
<tr>
<td>mutable vars (C/Java/Python)</td>
<td>Env -&gt; (Env, Val)</td>
</tr>
</tbody>
</table>

We’ll come back to mutable variables in unit on **scope**