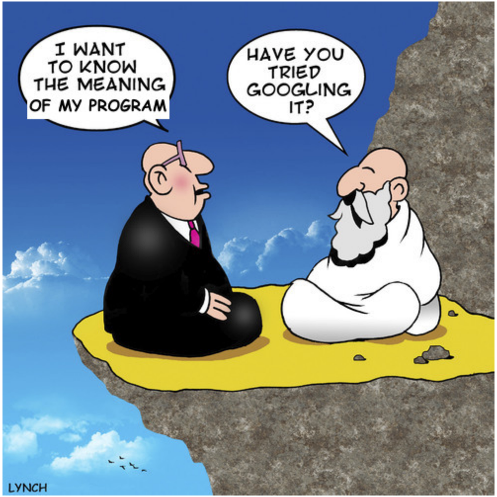
Semantics

Outline

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?

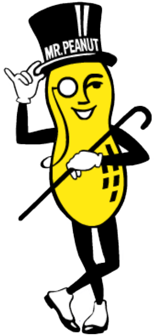
Formal specifications

- **denotational semantics**: relates terms directly to values
- **operational semantics**: describes how to evaluate a term
- **axiomatic semantics**: defines 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
- promotes better language design
  - better understand impact of design decisions
  - apply semantic insights to improve the language design (e.g. *compositionality*)

Outline

Denotational semantics

A denotational semantics relates each **term** to a **denotation**

an abstract syntax tree

a value in some **semantic domain**

Semantic function

$$[\![\cdot]\!] \quad : \quad \text{abstract syntax} \quad \to \quad \text{semantic domain}$$

Semantic function in Haskell

```haskell
eval :: Term -> Value
```

Semantic domains

Semantic domain: captures the set of possible meanings of a program/term

what is a meaning? – it depends on the language!

Example semantic domains

Language	Meaning
Boolean expressions	Boolean value
Arithmetic expressions	Integer
Imperative language	State transformation
SQL query	Set of relations
MiniLogo program	Drawing
MIDI	Music performance

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 **semantic function**, $[\![\cdot]\!] : T \to V$
   the mapping from ASTs to semantic values

   ```
   eval :: Term -> Value
   ```

Example: simple arithmetic expressions

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 semantic function

```
eval :: Exp -> Int
eval (Add l r) = eval l + eval r
eval (Mul l r) = eval l * eval r
eval (Neg e)   = negate (eval e)
eval (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

- if not, language has expressiveness gaps, or semantic domain is too general
- ensures that semantic domain and language align

Soundness: two programs are "equivalent" iff they have the same denotation

- equivalence: same w.r.t. to some other definition
- 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 **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 **op**

Implications:

- The semantic function is probably **recursive**
- Often need different semantic 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`**

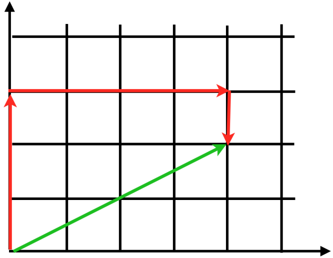
3. Define the semantic function

```
eval :: Exp -> Int
eval (Add l r) = eval l + eval r
eval (Mul l r) = eval l * eval r
eval (Neg e)   = negate (eval e)
eval (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

```
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

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

2. Semantic domain

```
type Pos = (Int,Int)
```

Domain: **Pos -> Pos**

3. Semantic function (**Step**)

```
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. Semantic function (**Move**)

```
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:	**`State -> Value`**
Modify as only effect:	**`State -> State`**
Modify as side effect:	**`State -> (State,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

declaration	introduce a new name
binding	associate a name with a thing
reference	use the name to stand for the bound thing

C/Java variables

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

In Haskell:

Local variables

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

Type names

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

Function parameters

```
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 …

immutable vars (Haskell)	`Env -> Val`
mutable vars (C/Java/Python)	`Env -> (Env,Val)`

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