CS 381: Programming Language Fundamentals

Summer 2015

Semantics
July 6, 2015

Outline

What is semantics?

Denotational semantics

Semantics of naming

Why do we need semantics?

Understand what program constructs do

Judge the correctness of a program (compare the expected with observed behaviors)

Prove properties about languages

Compare languages

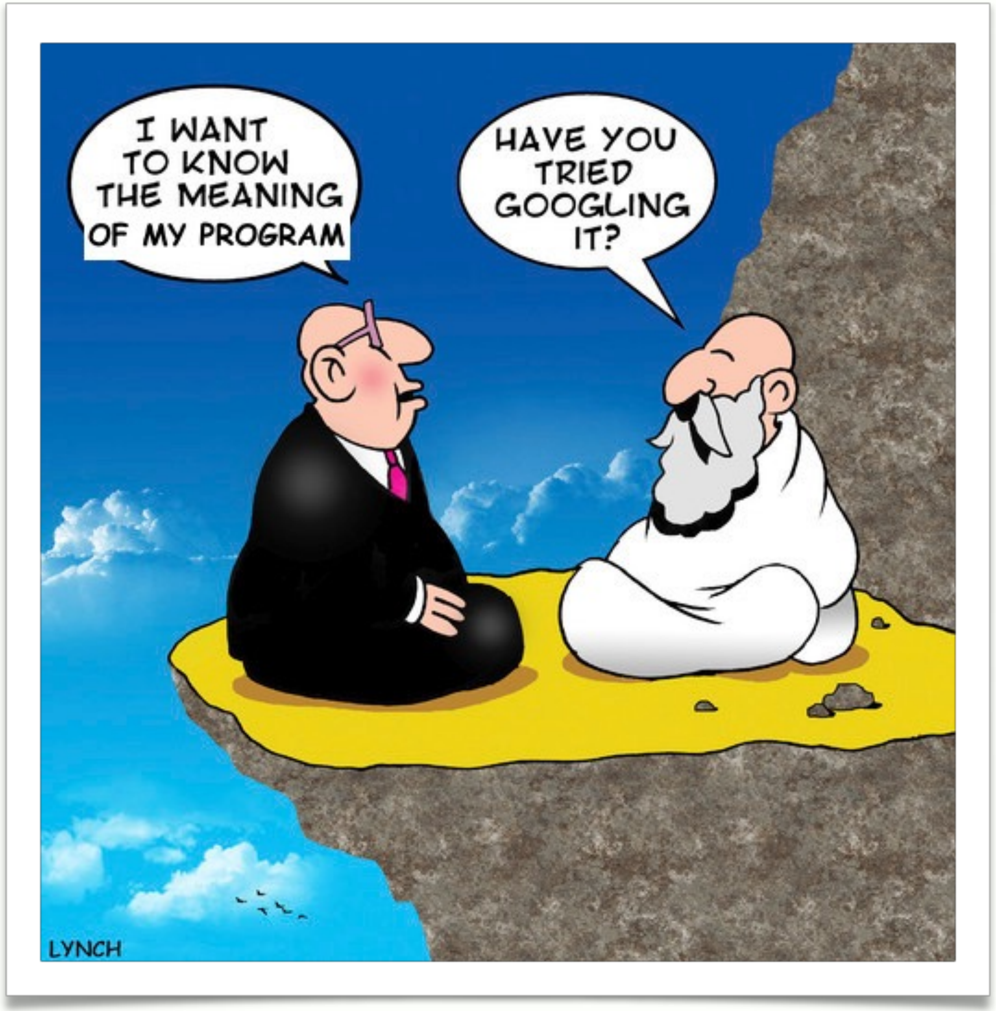
Design languages

Specification for implementation

What is the meaning of a program?

Recall aspects of a language

- **syntax**: the structure of its program
- **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

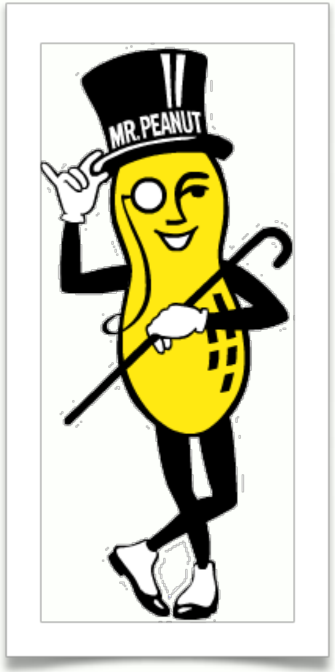
Informan/non-specifications

- **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 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 elegance (simplicity + power)



Oregon State UNIVERSITY

Outline

What is semantics?

Denotational semantics

Semantics of naming

Denotational semantics

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

an abstract syntax tree

a **value** in a **semantic domain**

Semantic function

$[\![\cdot]\!] \quad : \quad \texttt{abstract syntax} \quad \rightarrow \quad \texttt{semantics domain}$

Semantic function in Haskell

```
sem :: Term -> Value
```

Semantics domains

Semantics 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 expression	*Boolean value*
Arithmetic expression	*Integer*
Imperative	*State Transformation*
SQL query	*Set of relations*
MiniLogo program	*Drawing*

Defining a language with denotational semantics

1. Define the **abstract syntax**, T

   the set of abstract syntax trees

2. Identify or define the **semantics domain**, V

   the representation of semantic values

3. Define the **semantic function**, $[\![\cdot]\!] : T \to V$

   the mapping from ASTs to semantic values

Haskell encoding

```
data Term = …
type Value = …
sem :: Term -> Value
```

Example: simple arithmetic expression language

ExprSem.hs

1. Define abstract syntax

```
data Expr = Add Expr Expr
          | Mul Expr Expr
          | Neg Expr
          | Lit Int
```

2. Identify semantic domain

Let’s just use `Int`.

3. Define semantic function

```
sem :: Expr -> 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
```


Oregon State University

Exercise: simple arithmetic expression language

Extend the expression language with **`sub`** and **`div`** operations (abstract syntax and semantics)

Abstract syntax

```
data Expr = Add Expr Expr
          | Mul Expr Expr
          | Neg Expr
          | Lit Int
```

Semantic function

```
sem :: Expr -> 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
```

Exercise: simple arithmetic expression language

ExprSem.hs

Abstract syntax

```
data Expr = Add Expr Expr
          | Mul Expr Expr
          | Neg Expr
          | Lit Int
          | Sub Expr Expr
          | Div Expr Expr
```

Semantic function

```
sem :: Expr -> 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
sem (Sub l r) = sem l - sem r
sem (Div l r) = sem l `div` sem r
```

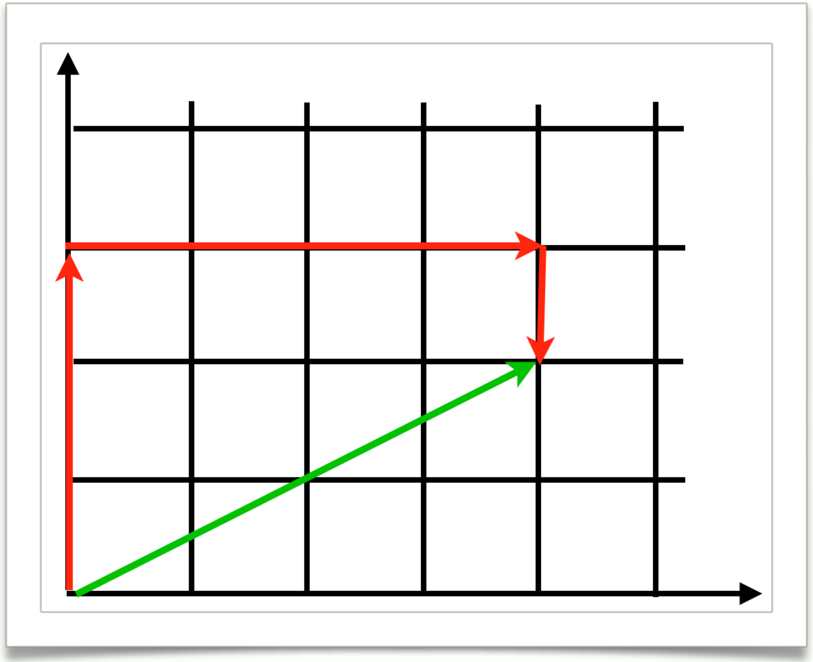
Example: move language

A language describing movements on a 2D plane

- a **step** is an *n*-unit horizontal or vertical shift
- a **move** is a sequence of **steps**

Abstract syntax

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



```
MoveLang> [Go N 3, Go E 4, Go S 1]
```

Semantics of move language

Move.hs

1. Define abstract syntax

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

2. Identify semantic domain

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

3. Define semantic function

```
sem :: Move -> Pos
sem = foldr step (0,0)
```

Define semantics of Step (helper)

```
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)
```

Alternative semantics

Often multiple **interpretations** (semantics) of the same language

Distance traveled

```
type Dist = Int

dist :: Move -> Dist
dist = sum . move distS
  where distS (Go _ k) = k
```

Combined trip information

```
trip :: Move -> (Dist,Pos)
trip m = (dist m,sem m)
```

Example: Recipe language

- Library — estimate time and difficulty
- Market — extract ingredients to buy
- Kitchen — execute to make the dish

Oregon State UNIVERSITY

Picking the right semantic domain

IntBool.hs

Simple semantics domains can be combined in two ways:

- **products**: contains a value from both domains
  - e.g. combined trip information for move language
  - use Haskell (a,b) or define a new data type
- **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

Can errors occur?

- use Haskell `Maybe a` or define a new data type

Does the language manipulate the state or use naming?

- use a **function type**

Exercise: expression language with two types

Extend the IntBool language by a **cond** operation (abstract syntax and semantics)

Abstract syntax

```
data Expr = Lit Int
          | Add Expr Expr
          | Equ Expr Expr
          | Not Expr

data Val = I Int
         | B Bool
         | TypeError
```

Semantic function

```
sem :: Expr -> Val
sem (Lit n)   = I n
sem (Add l r) = case (sem l, sem r) of
                  (I i, I j) -> I (i + j)
                  _          -> TypeError
sem (Equ l r) = case (sem l, sem r) of
                  (I i, I j) -> B (i == j)
                  (B a, B b) -> B (a == b)
                  _          -> TypeError
sem (Not e)   = case sem e of
                   B b -> B (not b)
                   _   -> TypeError
```

Exercise: expression language with two types

Abstract syntax

```
data Expr = Lit Int
          | Add Expr Expr
          | Equ Expr Expr
          | Not Expr
          | Cond Expr Expr Expr

data Val = I Int
         | B Bool
         | TypeError
```

Semantic function

```
sem :: Expr -> Val
sem (Lit n)      = I n
sem (Add l r)    = case (sem l, sem r) of
                     (I i, I j) -> I (i + j)
                     _          -> TypeError
sem (Equ l r)    = case (sem l, sem r) of
                     (I i, I j) -> B (i == j)
                     (B a, B b) -> B (a == b)
                     _          -> TypeError
sem (Not e)      = case sem e of
                     B b -> B (not b)
                     _   -> TypeError
sem (Cond f t e) = case sem f of
                     B True  -> sem t
                     B False -> sem e
```

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 in 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

RegCalc.hs LetSem.hs

Environment: a mapping associating names with 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 semantics domains for expressions with…

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