3 Semantics

How goes the obedience training?

Great! Watch this.

Heel.

Next week we start on semantics.
3 Semantics

en.wikipedia.org/wiki/Blissymbols

Semantics

Syntax
3 Semantics

Why semantics?

What is semantics?

Semantics of simple expression languages

Elements of semantic definitions

Examples: Shape & Move languages

Advanced Semantic Domains

Translating Haskell into denotational semantics

Haskell as a metalanguage
Why Semantics?

**Modus Ponens**

\[
\begin{array}{c}
S \rightarrow G \\
S \\
\hline
G
\end{array}
\]

If this statement is true, then God exists

If this statement is true, then God does not exist

If this statement is true, then Donald Trump is humble


See also: John Allen Paulos: Irreligion, Hill and Wang 2008

Curry’s Paradox

Haskell B. Curry, 1900-1982
Why Semantics?

Recursion without a base case

\[ S = \text{If } S \text{ is true, then God exists} \]

\[ S = \text{If } S \text{ is true, then God does not exist} \]

\[ S = \text{If } S \text{ is true, then Donald Trump is humble} \]
Why Semantics?

```c
{ int x=2;
  int f(int y) {return y+x;};
  { int x=4;
    printf("%d", f(3));
  }
}
```

Output? 5

Access to non-local variables
Why Semantics?

Swap the values of two variables

```
{ 
  int x=1;
  int y=8;
  y = x;
  x = y;
}
```

Effect?  

| y: 1  |
| x: 1  |

✓ 

What?!

```
{ 
  int x=1;
  int y=8;
  int z;
  z = y;
  y = x;
  x = z;
}
```

Effect?  

| y: 1  |
| x: 8  |

```
{ 
  int x=1;
  int y=8;
  y = x + 0*(x = y);
}
```

Effect?  

| y: 1  |
| x: 8  |
Why Semantics?

- Understand what program constructs do
- Judge the correctness of a program
  (compare expected with observed behavior)
- Prove properties about languages
- Compare languages
- Design languages
- Specification for implementations

Syntax: Form of programs
Semantics: Meaning of programs
The Meaning of Programs

What is the meaning of a program?
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>Logo</td>
<td>Picture</td>
</tr>
</tbody>
</table>

Denotational Semantics of a language:
Transformation of representation
(abstract syntax \(\rightarrow\) semantic domain)
Simple Examples

BoolSyn.hs
BoolSem.hs

ExprSyn.hs
ExprSem.hs
Exercises

(1) Extend the boolean expression language by an and operation (abstract syntax and semantics)

(2) Extend the arithmetic expressions by multiplication and division (abstract syntax and semantics)

(3) Define a Haskell function to apply DeMorgan’s laws to boolean expression, i.e., a function to transform any expression \( \text{not} (x \text{ and } y) \) into \( \text{not} x \) or \( \text{not} y \) (and accordingly for \( \text{not} (x \text{ or } y) \))
Defining Semantics in 3 Steps

(1) Define the **abstract syntax** $S$,  
i.e. set of syntax trees

(2) Define the **semantic domain** $D$,  
i.e. the representation of semantic values

(3) Define the **semantic function / valuation** $\llbracket \cdot \rrbracket : S \rightarrow D$  
that maps trees to semantic values

**Example Language**

"Arithmetic expressions"

$S$: $Expr$

$D$: $Int$

$\llbracket \cdot \rrbracket$: $sem :: Expr \rightarrow Int$
Language Definitions

Definition Profile

Language

Aspect

Metalanguage
Example Expression Language

**Syntax**

```
data Expr = N Int
          | Plus Expr Expr
          | Neg Expr
```

**Haskell**

```
Expr = N Int
     | Plus Expr Expr
     | Neg Expr
```

**Semantic Domain**

```
sem :: Expr → Int
sem (N i)    = i
sem (Plus e e') = sem e + sem e'
sem (Neg e)  = -(sem e)
```

**Semantic Function**

```
[·] : Expr → Int
[n]    = n
[e+e'] = [e] + [e']
[-e]   = -[e]
```

**Syntactic Symbol**

```
Expr ::= Num | Expr+Expr | -Expr
```

**Semantic Operation**

```
```
Related: Expressions vs. Values

\[
\text{tail } [1 \ldots 4] \quad \text{semantics} \quad [2,3,4] \quad \text{or} \quad 2:3:4:[] \\
\]

\[
3 + 5 \quad \text{semantics} \quad 8 \\
\]

\[
3 \times 5 \text{ and } 5 + 5 + 5 \text{ have different meaning but the same referent}
\]

Gottlob Frege: Meaning vs. Reference
Example: Shape Language

A language for constructing bitmap images: an image is either a pixel or a vertical or horizontal composition of images.

- Operation `TD s1 s2` puts `s1` on top of `s2`

- Operation `LR s1 s2` puts `s1` left next to `s2`
Abstract Syntax

**data**

\[
\text{Shape} = X \\
\text{TD} \text{ Shape} \text{ Shape} \\
\text{LR} \text{ Shape} \text{ Shape}
\]

**Example:**

- LR (TD X X) X
- TD X (LR X X)
- TD (LR X X) X

LR aligns at bottom
TD aligns at left

...part of semantics
How to represent a bitmap image?

Data: Shape = X
  | TD Shape Shape
  | LR Shape Shape

Type: Image = Array (Int,Int) Bool

Pixel = (Int,Int)

Image = [Pixel]

[(1,1),(1,2),(2,1)]

Semantics

LR (TD X X) X
Semantic Function (I)

Approach: Translate individual shapes separately into *pixel lists* and then compose pixel lists

```
data Shape = X
|   TD Shape Shape
|   LR Shape Shape

type Pixel = (Int,Int)
type Image = [Pixel]
```

Base case: Individual pixel

```
sem :: Shape -> Image
sem X = [(1,1)]
```
Semantic Function (2)

How can we compose (horizontally and vertically) two pixel lists images without overlapping?

sem (TD s1 s2) = adjustY ht p1 ++ p2

where p1 = sem s1
p2 = sem s2
ht = maxY p2

Take bounding boxes and adjust y-coordinates of top shape by height of bottom shape
Semantic Function (3)

\[
\text{sem } (\text{TD } s1 \; s2) = \text{adjustY } \; \text{ht } \; p1 \; ++ \; p2
\]

\textbf{where}
\[
p1 = \text{sem } s1
\]
\[
p2 = \text{sem } s2
\]
\[
\text{ht} = \text{maxY } \; p2
\]

\[
\text{maxY } :: [(\text{Int},\text{Int})] \rightarrow \text{Int}
\]
\[
\text{maxY } \; p = \text{maximum } (\text{map } \text{snd} \; p)
\]

\[
\text{adjustY } :: \text{Int} \rightarrow [(\text{Int},\text{Int})] \rightarrow [(\text{Int},\text{Int})]
\]
\[
\text{adjustY } \; \text{ht} \; p = [(x, y+\text{ht}) \mid (x, y) \leftarrow p]
\]
Exercise

(1) Define the functions:

\[
\text{sem (LR } s_1 \ s_2) \quad \text{maxX} \quad \text{adjustX}
\]

\[
\text{sem (LR } s_1 \ s_2) = p_1 ++ \text{adjustX } wd \ p_2
\]

\[
\text{where } p_1 = \text{sem } s_1
\]
\[
p_2 = \text{sem } s_2
\]
\[
wd = \text{maxX } p_1
\]

\[
\text{maxX :: ([Int,Int]) } \rightarrow \text{ Int}
\]
\[
\text{maxX } p = \text{maximum (map fst } p)
\]

\[
\text{adjustX :: Int } \rightarrow \text{ ([Int,Int]) } \rightarrow \text{ ([Int,Int])}
\]
\[
\text{adjustX } ht \ p = [(x, y+ht) | (x, y) \leftarrow p]
\]
Example

Shape.hs
ShapePP.hs
Example: Move Language

A language describing vector-based movements in the 2D plane. A *step* is an *n*-unit horizontal or vertical move, a *move* is a sequence of steps.

Go Up 3;
Go Right 4;
Go Down 1
Abstract Syntax

\begin{verbatim}
data Dir = Lft | Rgt | Up | Dwn
data Step = Go Dir Int
type Move = [Step]
\end{verbatim}

Example:

\[\text{[Go Up 3, Go Rgt 4, Go Dwn 1]}\]
Exercises

(1) Give a type definition for the data type \texttt{Step}

\begin{verbatim}
data Step = Go Dir Int
\end{verbatim}

(2) Define the data type \texttt{Move} without using built-in lists

\begin{verbatim}
type Step = (Dir,Int)
data Move = One Step | Seq Step Move
\end{verbatim}

(3) Write the move \texttt{[Go Up 3, Go Rgt 4, Go Dwn 1]} using the representation from (1) and (2)

\begin{verbatim}
(Up,3) `Seq` (Rgt,4) `Seq` One (Dwn,1)
\end{verbatim}
What is the meaning of a move?

The distance traveled, the final position, or both.

data Dir = Lft | Rgt | Up | Dwn

data Step = Go Dir Int

type Move = [Step]

type Pos = (Int, Int)

[Go Up 3, Go Rgt 4, Go Dwn 1]  \rightarrow  (4, 2)

**Semantic Domain**
**Semantic Function**

```haskell
sem :: Move -> Pos
sem [] = (0,0)
sem (Go d i:ss) = (dx*i+x,dy*i+y)
  where (dx,dy) = vector d
        (x,y) = sem ss

vector :: Dir -> (Int,Int)
vector Lft = (-1,0)
vector Rgt = (1,0)
vector Up = (0,1)
vector Dwn = (0,-1)
```

*pattern matching in definitions*
Example

Move.hs
Exercises

sem :: Move -> Pos
sem [] = (0,0)
sem (Go d i:ss) = (dx*i+x,dy*i+y)
    where (dx,dy) = vector d
        (x,y) = sem ss

(1) Define the semantic function for the move language for the semantic domain

\[
\text{type } \text{Dist} = \text{Int}
\]

(2) Define the semantic function for the move language for the semantic domain

\[
\text{type } \text{Trip} = (\text{Dist}, \text{Pos})
\]
Advanced Semantic Domains

The story so far: Semantic domains were mostly simple types (such as \texttt{Int} or \texttt{[(Int,Int)]})

How can we deal with language features, such as \textit{errors}, \textit{union types}, or \textit{state}?

(1) \textit{Errors}: Use the \texttt{Maybe} data type

(2) \textit{Union types}: Use corresponding \textit{data types}

(3) \textit{State}: Use \textit{function types}
If $T$ is the type representing “regular” values, define the semantic domain as $\texttt{Maybe } T$

```
data Maybe a = Just a | Nothing
```
Example

ExprErr.hs
If $T_1 \ldots T_k$ are types representing different semantic values for different nonterminals, define the semantic domain as a data type with $k$ constructors.

```haskell
data T = C1 T1 | \ldots | Ck Tk
```
Example

Expr2.hs
Exercises

(1) Extend the semantic domain for the two-type expression language to include errors

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

(2) Extend the semantic function for the two-type expression language to handle errors
Function Domains

If a language operates on a state that can be represented by a type $T$, define the semantic domain as a function type $T \rightarrow T$

```
type D = T -> T
```

```
sem :: S -> D
=
sem :: S -> (T -> T)
=
sem :: S -> T -> T
```

Semantic function takes state as an additional argument
Example

RegMachine.hs
Exercises

(1) Extend the machine language to work on two registers A and B

(2) Define a new semantic domain for the extended language

(3) Define the semantics functions for the extended language

RegMachine2.hs

data Op = LD Int | INC | DUP

type RegCont = Int

type D = RegCont -> RegCont
Translating Haskell into Mathematical Denotational Semantics

(1) Replace *type definitions* by *sets (should actually be CPOs)*

(2) Replace *patterns* by *grammar productions*

(3) Replace *function names* by *semantic brackets* that enclose only syntactic objects

**Semantics**

\[
\begin{align*}
\text{sem} & : \text{Expr} \rightarrow \text{Int} \\
\text{sem} (N \ i) & = i \\
\text{sem} (\text{Plus} \ e \ e') & = \text{sem} \ e + \text{sem} \ e' \\
\text{sem} (\text{Neg} \ e) & = -(\text{sem} \ e)
\end{align*}
\]

**Haskell**

\[
\begin{align*}
\text{Expr} & ::= \text{Num} \mid \text{Expr} + \text{Expr} \mid -\text{Expr}
\end{align*}
\]

**Math**

\[
\begin{align*}
\mathbf{[\cdot]} : \text{Expr} & \rightarrow \text{Int} \\
\mathbf{[n]} & = n \\
\mathbf{[e+e']} & = \mathbf{[e]} + \mathbf{[e']} \\
\mathbf{[-e]} & = -\mathbf{[e]}
\end{align*}
\]
Haskell as a Mathematical Metalanguage

Math World

Grammars (Languages) → Functions (Semantics) → Sets (Semantic domains)

Haskell World

Data Types → Functions → Data Types

= Executable Math World