Operational vs. denotational

- **Operational semantics**
  - meaning of program defined by syntactic transitions
  - structural operational semantics: how to write a recursive-descent interpreter
  - meaning of language terms defined by other language terms
    \[(\lambda x . x) = (\lambda x . x)\]

- **Denotational semantics**
  - defines meaning of program in terms of underlying semantic domain (intrinsic meaning)
  - semantic function maps expressions to meanings
  - how to write a compiler

Semantic Function

- Denotational semantics operates on expressions to produce objects that are the meaning of the expression (usually mathematical function)

\[
\llbracket (\lambda x . x) \rrbracket = \lambda x \in D. x
\]

\[
\llbracket (a, a) | a \in D \rrbracket
\]

Parsing λ’s

- Notation for describing a mathematical function of several variables:
  \[\lambda y z. e = \lambda y . \lambda z. e\]

- Lambda expression extends as far to the right as possible (like ∀, ∃)
  \[\lambda x y z. \lambda w. w = \lambda x . (\lambda y . (\lambda z . x) (\lambda w . w)))\]

- Application left-associates:
  \[xy = (x y) z = x(y z)\]

  \[f = \lambda y z. x = \lambda x y z. e\]

  \[fabe = f(a, b, c)\]

Typed functions

- For mathematical functions, we will usually write the types of the arguments
  \[PLUS = \lambda x . Z \cdot \lambda y . Z. x + y = \lambda x, y . Z. x + y\]
  \[PLUS \in Z \rightarrow (Z \rightarrow Z)\]

- Type \((T_1 \rightarrow T_2)\) is domain of functions that maps elements from domain \(T_1\) to domain \(T_2\)

- Application associates to the left ⇒ function constructor \((\rightarrow)\) associates to right
  \[Z \rightarrow (Z \rightarrow Z) = Z \rightarrow Z \rightarrow Z\]

Back to IMP

- Recall IMP has three kinds of expressions:
  \[a ::= n | X | a_0 + a_1 | a_0 - a_1 | a_0 \times a_1\]
  \[b ::= a_0 \leq a_1 | b_0 = b_1 | b_0 \land b_1 | b_0 \lor b_1\]
  \[c ::= X ::= a_0 | \text{skip} | \text{if } b \text{ then } c_0 \text{ else } c_1 | \text{while } b_0 \text{ do } c_0\]

  \[a : \text{Aexp, } b : \text{Bexp, } c : \text{Com}\]

  What is the meaning of these three syntactic categories?

  Does an element from \(a\) mean an integer?
Natural semantics as functions

- Expression $a$ denotes a unique integer given a particular store $\langle a, \sigma \rangle \downarrow n$
- Expression $b$ denotes a unique truth value given a particular store $\langle b, \sigma \rangle \downarrow t$
- Command $c$ maps one store into another $\langle c, \sigma \rangle \downarrow \sigma'$
- Deterministic evaluation $\Rightarrow$ exists functions $f, g$ such that:
  \[
  f \circ (a, \sigma) = n \quad \text{iff} \quad \langle a, \sigma \rangle \downarrow n
  
  g \circ (b, \sigma) = t \quad \text{iff} \quad \langle b, \sigma \rangle \downarrow t
  
  \gamma \circ (c, \sigma) = \sigma' \quad \text{iff} \quad \langle c, \sigma \rangle \downarrow \sigma'
  \]

Semantic functions for IMP

- Meaning functions $\gamma, \delta : \cdot \Rightarrow\cdot$ translate syntactic expressions into meaning: mathematical functions

  $\gamma (a) = n \quad \delta (a) \in Aexp \to (\Sigma \to N)$
  $\gamma (b) = t \quad \delta (b) \in Bexp \to (\Sigma \to T)$
  $\gamma (c) = \sigma' \quad \delta (c) \in Com \to (\Sigma \to \Sigma)$

- $\gamma (a)$ is a unique truth value $\in \Sigma$
- $\gamma (b)$ is a unique integer $\in \Sigma$

Semantic function as a set

$\gamma ([n]) = \lambda \sigma \in \Sigma . \ n = \{(\sigma, n) \mid \sigma \in \Sigma\}

\gamma ([a_0 + a_1]) = \lambda \sigma \in \Sigma . \ \gamma ([a_0]) \sigma + \gamma ([a_1]) \sigma

\gamma ([a_0 - a_1]) = \lambda \sigma \in \Sigma . \ \gamma ([a_0]) \sigma - \gamma ([a_1]) \sigma

\gamma ([a_0 \times a_1]) = \lambda \sigma \in \Sigma . \ \gamma ([a_0]) \sigma \cdot \gamma ([a_1]) \sigma

\gamma ([a_0]) = f_0

\gamma ([a_1]) = f_1

\gamma ([a_0 + a_1]) = \lambda \sigma \in \Sigma . \ f_0 \sigma + f_1 \sigma

Arithmetic denotations

- The function $\gamma : Aexp \to \Sigma \to Z$ is defined using induction on structure of exprs:

  $\gamma ([n]) = \lambda \sigma \in \Sigma . \ n$
  $\gamma ([X]) = \lambda \sigma \in \Sigma . \ \sigma X$
  $\gamma ([a_0 + a_1]) = \lambda \sigma \in \Sigma . \ \gamma ([a_0]) \sigma + \gamma ([a_1]) \sigma$
  $\gamma ([a_0 - a_1]) = \lambda \sigma \in \Sigma . \ \gamma ([a_0]) \sigma - \gamma ([a_1]) \sigma$
  $\gamma ([a_0 \times a_1]) = \lambda \sigma \in \Sigma . \ \gamma ([a_0]) \sigma \cdot \gamma ([a_1]) \sigma$

Boolean denotations

$\delta : Bexp \to \Sigma \to T$

$\delta ([a_0] = a_1] \sigma = \text{if} \ \gamma ([a_0]) \sigma = \gamma ([a_1]) \sigma \quad \text{then true else false}$

$\delta ([a_0] \leq a_1] \sigma = \text{if} \ \gamma ([a_0]) \sigma \leq \gamma ([a_1]) \sigma \quad \text{then true else false}$

$\delta ([b_0 \land b_1] \sigma = \text{if} \ \gamma ([b_0]) \sigma \land \gamma ([b_1]) \sigma \quad \text{then true else false}$

$\delta ([b_0 \lor b_1] \sigma = \text{if} \ \gamma ([b_0]) \sigma \lor \gamma ([b_1]) \sigma \quad \text{then true else false}$

Boolean denotations

Command denotations

- Some commands do not terminate ($\sim \exists \sigma' . \ (c, \sigma) \downarrow \sigma'$)
- Commands are partial functions from states to states ($\Sigma \to \Sigma$)
- Idea: make denotations total by adding special state to represent non-termination: $\bot$
- Domain $\Sigma_1$ has elements of $\Sigma \cup \{ \bot \}$ (shift of $\Sigma$)

  $\sigma \in \text{Com} \to \Sigma_1 \to \Sigma_1$

  Advantage over large-step: can specify non-terminating behavior
Command denotations

\[ \begin{align*}
\lbrack \text{skip} \rbrack & \sigma = \sigma \\
\lbrack X := a \rbrack & \sigma = \sigma[X \leftarrow \lbrack a \rbrack \sigma] \\
\lbrack \text{if } b \text{ then } c_0 \text{ else } c_1 \rbrack & \sigma = \\
& \begin{cases} 
\llbracket c_0 \rrbracket \sigma & \text{if } \llbracket b \rrbracket \sigma \\
\llbracket c_1 \rrbracket \sigma & \text{else}
\end{cases} \\
\llbracket c_0 ; c_1 \rrbracket & \sigma = \llbracket c_1 \rrbracket (\llbracket c_0 \rrbracket \sigma)
\end{align*} \]

Note: \( \sigma \) could be \( \bot \); need to wrap \( \text{if } \sigma = \bot \text{ then } \bot \text{ else } \ldots \) around :=, if, while definitions

while command

\[ \llbracket \text{while } b \text{ do } c \rrbracket \sigma = \\
\begin{cases} 
\sigma & \text{if } \llbracket b \rrbracket \sigma \\
\llbracket \text{while } b \text{ do } c \rrbracket (\llbracket c \rrbracket \sigma) & \text{else}
\end{cases} \]

This is an equation, not a definition (induction is not well-founded)

\[ \llbracket \text{while } b \text{ do } c \rrbracket = \\
\{ (\sigma, \sigma') \mid \llbracket b \rrbracket \sigma \land (\sigma, \sigma') \in \llbracket \text{while } b \text{ do } c \rrbracket \circ (\llbracket c \rrbracket \sigma) \}
\]

Denotation as a fixed point

\[ \llbracket \text{while } b \text{ do } c \rrbracket = \\
\{ (\sigma, \sigma') \mid \llbracket b \rrbracket \sigma \land (\sigma, \sigma') \in (\llbracket \text{while } b \text{ do } c \rrbracket \circ (\llbracket c \rrbracket \sigma)) \}
\]

Define \( \Gamma(f) \) where \( f \) is a command denotation

\[ \Gamma = \lambda f : \Sigma_\bot \rightarrow \Sigma_\bot. \text{if } \llbracket b \rrbracket \sigma \text{ then } \sigma \text{ else } f (\llbracket c \rrbracket \sigma) \]

\[ \llbracket \text{while } b \text{ do } c \rrbracket = \Gamma(\llbracket \text{while } b \text{ do } c \rrbracket) \]

Denotation of while is fixed point of \( \Gamma \)

Question: how do we define least fixed point operator \( \text{fix} \) for domain \( \Sigma_\bot \rightarrow \Sigma_\bot \)?

Answer: next lecture