Overview:

- Syntax and Semantics
- Approaches to Specifying Semantics
- Sets, Semantic Domains, Domain Algebra, and Valuation Functions
- Semantics of Expressions
- Semantics of Assignments
- Other Issues

References:

Three main characteristics of programming languages:

- **Syntax:** What is the appearance and structure of its programs?

- **Semantics:** What is the meaning of programs?
  The static semantics tells us which (syntactically valid) programs are semantically valid (i.e., which are type correct) and the dynamic semantics tells us how to interpret the meaning of valid programs.

- **Pragmatics:** What is the usability of the language?
  How easy is it to implement? What kinds of applications does it suit?
Uses of Semantic Specifications

- Semantic specifications are useful for language designers to communicate to the implementors as well as to programmers. A semantic specification is:
  - A precise standard for a computer implementation:
    - How should the language be implemented on different machines?
  - User documentation:
    - What is the meaning of a program, given a particular combination of language features?
  - A tool for design and analysis:
    - How can the language definition be tuned so that it can be implemented efficiently?
  - An input to a compiler generator:
    - How can a reference implementation be obtained from the specification?
Methods for Specifying Semantics

- **Operational Semantics:**
  - $\llbracket \text{program} \rrbracket = \text{abstract machine program}
  - can be simple to implement
  - hard to reason about

- **Denotational Semantics:**
  - $\llbracket \text{program} \rrbracket = \text{mathematical denotation (typically, a function)}$
  - facilitates reasoning
  - not always easy to find suitable semantic domains

- **Axiomatic Semantics:**
  - $\llbracket \text{program} \rrbracket = \text{set of properties}$
  - good for proving theorems about programs
  - somewhat distant from implementation

- **Structured Operational Semantics:**
  - $\llbracket \text{program} \rrbracket = \text{transition system (defined using inference rules)}$
  - good for concurrency and non-determinism
  - hard to reason about equivalence
Concrete and Abstract Syntax

- How to parse “4 * 2 + 1”?

- Abstract syntax is compact but ambiguous:
  \[
  \begin{align*}
  \text{Expr} & ::= \text{Num} \\
  & \quad | \quad \text{Expr Op Expr} \\
  \text{Op} & ::= \text{`+`} | \text{`-`} | \text{`*`} | \text{`/`} \\
  \end{align*}
  \]

- Concrete syntax is unambiguous, but verbose:
  \[
  \begin{align*}
  \text{Expr} & ::= \text{Expr LowOp Expr} \\
  & \quad | \quad \text{Term} \\
  \text{Term} & ::= \text{Term HighOp Factor} \\
  & \quad | \quad \text{Factor} \\
  \text{Factor} & ::= \text{Num} \\
  & \quad | \quad \text{'( Expr )'} \\
  \text{LowOp} & ::= \text{`+`} | \text{`-' } \\
  \text{HighOp} & ::= \text{`*`} | \text{`/`} \\
  \end{align*}
  \]
Set, Functions, and Domains

- A set is a collection: it can contain numbers, persons, other sets, or (almost) anything one wishes:
  - \{ 1, \{1, 2, 4\}, 4 \}
  - \{ red, yellow, gray \}
  - \{\}

- A function is like “black box” that accepts an object as its input and then transforms it in some way to produce another object as output. We must use an “external approach” to characterize functions. Sets are ideal for formalizing the method. (See “Extensional and Intentional Views”)

- The sets that are used as value spaces in programming language semantics are called **semantic domains**. Semantic domains may have a different structure than a set, and in practice not all of the sets and set building operations are needed for building domains.
Basic Domains

- **Primitive domains:**
  - Natural numbers $\mathbb{N}$
  - Boolean values $\mathbb{B}$
  - Floating point numbers $\mathbb{F}$

- **Compound domains:**
  - Product domains $A \times B$
  - Sum domains $A + B$
  - Function domains $A \rightarrow B$

- **Lifted domains:**
  - Lifted domains add a special value $\bot$ (“bottom”) that denotes non-termination or “no value at all”. Including $\bot$ as a value is an alternative to using a theory of partial functions.
  
  Lifted domains are written $A_{\bot}$, where $A_{\bot} = A \cup \{\bot\}$
Domain Algebra

- The format for representing semantic domains is called *semantic algebra* and defines a grouping of a set with the fundamental operations on the set.

- This format is used because it:
  - Clearly states the structure of a domain and how its elements are used by the functions,
  - Encourages the development of “standard” algebra “modules” or “kits” that can be used in a variety of semantics definitions,
  - Makes it easier to analyze a semantic definition concept by concept,
  - Makes it straightforward to alter a semantic definition by replacing one semantic algebra with another.
Domain $\textbf{Rational}$

Domain $\textbf{Rational} = (\mathbb{Z} \times \mathbb{Z})_\perp$

Operations

$\text{makerational :: } \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \textbf{Rational}$

$\text{makerational} = \lambda p \lambda q. (q = 0) \rightarrow \bot \mapsto (p, q)$

$\text{addrational :: } \textbf{Rational} \rightarrow \textbf{Rational} \rightarrow \textbf{Rational}$

$\text{addrational} = \lambda (p_1, q_1) \lambda (p_2, q_2). (p_1 \cdot q_2 + p_2 \cdot q_1), q_1 \cdot q_2)$

$\text{mulrational :: } \textbf{Rational} \rightarrow \textbf{Rational} \rightarrow \textbf{Rational}$

$\text{mulrational} = \lambda (p_1, q_1) \lambda (p_2, q_2). p_2 \cdot p_1, q_1 \cdot q_2)$
module Rational (Rational, makerational, addrational, mulrational) where

data Rational = Rat Int Int

makerational :: Int -> Int -> Rational
makerational p q
  | q == 0         = error "Rational: division by zero"
  | otherwise      = Rat p q

addrational :: Rational -> Rational -> Rational
addrational = \(Rat p1 q1) -> \(Rat p2 q2) -> Rat ((p1 * q2) + (p2 * q1)) (q1 * q2)

mulrational :: Rational -> Rational -> Rational
mulrational = \(Rat p1 q1) -> \(Rat p2 q2) -> Rat (p1 * p2) (q1 * q2)

instance Show Rational where
  show (Rat p q) = "(" ++ show p ++ ", " ++ show q ++ ")"
Valuation Functions

- A valuation function maps a language’s abstract syntax structures or “syntactic categories” to meanings governed by its semantic domains.

- The domain of a valuation function is the set of derivation trees of a syntactic category (abstract syntax trees).

- The meaning of one syntactic category is determined by determining and combining all meanings of its sub-trees.

- With $P$ identifying the syntactic category, $L$ denoting an element of the category, a set of parameters $n_1, ..., n_m$ carrying a state, and the actual mathematical denotation $F_L^P (n_1, ..., n_m)$, the corresponding valuation function is written $P \parallel L \parallel (n_1, ..., n_m) = F_L^P (n_1, ..., n_m)$.

- Examples of syntactic categories are: *Program*, *Statement*, *Expression*, *Number*, etc.
A denotational definition of a language consists of three parts:

- The abstract syntax definition
  - compact, minimal, but ambiguous language definition

- The semantic algebras
  - set of semantic domains that share some common properties or use

- The valuation functions
  - functions that assign every language element a precise meaning
A Calculator Language

Abstract syntax:

Prog ::= ‘ON’ Stmt
Stmt ::= Expr ‘TOTAL’ Stmt
| ‘OFF’

Expr ::= Expr ( ‘+’ | ‘*’ ) Expr
| ‘IF’ Expr ‘,’ Expr ‘,’ Expr
| ‘LASTANSWER’
| ‘(’ Expr ‘)’
| Num

The program “ON 4 * ( 3 + 2 ) TOTAL OFF” should print out 20 and stop.
Calculator Semantics – Semantic Algebras

- Truth values:
  Domain **Booleans**
  Operations
    - \texttt{true, false :: Booleans}

- Natural numbers:
  Domain **Numbers**
  Operations
    - \texttt{zero, one, two, ... :: Numbers}
    - \texttt{plus, times :: Numbers \times Numbers \rightarrow Numbers}
    - \texttt{equals :: Numbers \times Numbers \rightarrow Booleans}
Calculator Semantics – Valuation Functions

$$\text{Prog} :: \text{Program} \rightarrow \text{Numbers}$$
$$\text{Prog} \ [(\text{ON} \ S)] = \text{Stmt} \ [(S)] (0)$$

$$\text{Stmt} :: \text{Statements} \rightarrow \text{Numbers} \rightarrow \text{Numbers}$$
$$\text{Stmt} \ [(E \ \text{TOTAL} \ S)] (n) = \text{let} n = \text{Expr} \ [(E)] (n) \text{ in } n : \text{Stmt} \ [(S)] (n)$$
$$\text{Stmt} \ [(OFF')] (n) = []$$

$$\text{Expr} :: \text{Expression} \rightarrow \text{Numbers} \rightarrow \text{Numbers}$$
$$\text{Expr} \ [(E_1 \ + \ E_2)] (n) = \text{plus} (\text{Expr} \ [(E_1)] (n), \text{Expr} \ [(E_2)] (n))$$
$$\text{Expr} \ [(E_1 \ \text{\textbullet} \ E_2)] (n) = \text{times} (\text{Expr} \ [(E_1)] (n), \text{Expr} \ [(E_2)] (n))$$
$$\text{Expr} \ [(\text{IF} \ E_1 \ \text{\textbullet} \ E_2 \ \text{\textbullet} \ E_3)] (n) =$$
$$\quad \text{equals} (\text{Expr} \ [(E_1)] (n), \text{zero}) \rightarrow \text{Expr} \ [(E_2)] (n) \ | \ \text{Expr} \ [(E_3)] (n)$$
$$\text{Expr} \ [(\text{LASTANS} \ \text{ER}')] (n) = n$$
$$\text{Expr} \ [(\text{'}(E\text{')}\text{')}] (n) = \text{Expr} \ [(E)] (n)$$
$$\text{Expr} \ [(N)] (n) = \text{Numeral} \ [(N)]$$
We can represent programs in our calculator language as syntax trees:

- `data Program = On Statements`
- `data Statements = Total Expression Statements
  | Off`
- `data Expression = Plus Expression Expression
  | Times Expression Expression
  | If Expression Expression Expression
  | LastAnswer
  | Braced Expression
  | Numeral Int`
The program “ON 4 * ( 3 + 2 ) TOTAL OFF” can be parsed as:

And represented as:

```
test = On (Total (Times (N 4) (Braced (Plus (Numeral 3) (Numeral 2)))))
```
Haskell Implementation

prog :: Program -> [Int]
prog (On s) = stmt s 0

stmt :: Statements -> Int -> [Int]
stmt (Total e s) n = let n' = (expr e n) in n' : (stmt s n')
stmt Off n = []

expr :: Expression -> Int -> Int
expr (Plus e1 e2) n = (expr e1 n) + (expr e2 n)
expr (Times e1 e2) n = (expr e1 n) * (expr e2 n)
expr (If e1 e2 e3) n
    | (expr e1 n) == 0 = (expr e2 n)
    | otherwise        = (expr e3 n)
expr LastAnswer n = n
expr (Braced e) n = expr e n
expr (Numeral n) _ = n

val = On (Total (Times (Numeral 4) (Braced (Plus (Numeral 3) (Numeral 2)))))) Off)
A Language With Assignment

Abstract Syntax:

Program ::= Command ‘.’

Command ::= Command ‘;’ Command
| ‘IF’ Boolean ‘THEN’ Command ‘ELSE’ Command
| Identifier ‘:=’ Expression

Expression ::= Expression ‘+’ Expression
| Identifier
| Numeral

Boolean ::= Expression ‘=’ Expression
| ‘NOT’ Boolean

Example: “z := 1 ; IF a = 0 THEN z := 3 ELSE z := z + a .”

Programs take a single number as input, which initializes the variable ‘a’. The output of a program is the final value of the variable ‘z’.

Com S 541
Abstract Syntax Trees

Data Structures:

- `data Program = Dot Command`
- `data Command = Sequence Command Command | Assign Identifier Expression | If Boolean Command Command`
- `data Expression = Plus Expression Expression | Ident Identifier | Numeral Int`
- `data Boolean = Equal Expression Expression | Not Boolean`
- `type Identifier = Char`

Example:

```
Dot (Sequence (Assign 'z' (Numeral 1))
    (If (Equal (Ident 'a') (Numeral 0))
        (Assign 'z' (Numeral 3)) (Assign 'z' (Plus (Ident 'z') (Ident 'a'))) ) )
```
Store Algebra

Domain $\textit{Store} = \textit{Identifier} \to \textit{Numbers}$

Operations

$new\ \textit{store} :: \textit{Store}$
$new\ \textit{store} = \lambda \text{zero}$

$\text{access} :: \textit{Identifier} \to \textit{Store} \to \textit{Numbers}$
$\text{access} = \lambda i.\lambda s.s(i)$

$update :: \textit{Identifier} \to \textit{Numbers} \to \textit{Store} \to \textit{Store}$
$update = \lambda i.\lambda n.\lambda s.s[i.a\ n]s$
For a function $f :: A \rightarrow B$, we let $[a_0 \; a \; b_0]f$ be the function that acts just like $f$ except that it maps the specific value $a_0 \in A$ to $b_0 \in B$:

$$( [a_0 \; a \; b_0]f)(a_0) = b_0$$

$$( [a_0 \; a \; b_0]f)(a) = f(a), \text{ for all other } a \in A \text{ such that } a \neq a_0$$
A store is a mapping from identifiers to values:

```haskell
type Store = Identifier -> Int

newstore :: Store
newstore id = 0

access :: Identifier -> Store -> Int
access id store = store id

update :: Identifier -> Int -> Store -> Store
update id val store = store'
    where store' id'
        | id' == id = val
        | otherwise = store id'
```

Com S 541
Store Trace

- How does the store work?

? access 'a' (update 'a' 3 (update 'z' 2 newstore))
3 :: Int

Trace:
- update 'z' 2 newstore --> \id' -> if id' == 'z'
  then 2
  else newstore id'
- update 'a' 3 (`1Xupdate`) --> \id' -> if id' == 'a'
  then 3
  else (`1Xupdate`) id'
- access 'a' (`2Xupdate`) --> (`2Xupdate`) 'a'
  ---> if 'a' == 'a' then 3 else (`1Xupdate`) 'a'
  ---> 3
Semantics of Assignments

\[ \text{prog} :: \text{Program} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ \text{prog} \ D \ o t c \ n = \text{access} \ 'z' \ (\text{com m} \ c \ \text{update} \ b \ 'n \ \text{new store}) \]

\[ \text{com m} :: \text{Command} \rightarrow \text{Store} \rightarrow \text{Store} \]
\[ \text{com m} \ (\text{Sequence} \ c1 \ c2) \ s = \text{com m} \ c2 \ (\text{com m} \ c1 \ s) \]
\[ \text{com m} \ (\text{Assign} \ id \ e) \ s = \text{update} \ id \ (\text{expr} \ e) \ s \]
\[ \text{com m} \ (\text{If} \ b \ c1 \ c2) \ s = \text{ifelse} \ (\text{boolean} \ b) \ s \ (\text{com m} \ c1 \ s) \ (\text{com m} \ c2 \ s) \]

\[ \text{expr} :: \text{Expression} \rightarrow \text{Store} \rightarrow \text{Int} \]
\[ \text{expr} \ (\text{Plus} \ e1 \ e2) \ s = (\text{expr} \ e1 \ s) + (\text{expr} \ e2 \ s) \]
\[ \text{expr} \ (\text{Ident} \ id) \ s = \text{access} \ id \ s \]
\[ \text{expr} \ (\text{Numeral} \ n) \ s = n \]

\[ \text{boolean} :: \text{Boolean} \rightarrow \text{Store} \rightarrow \text{Bool} \]
\[ \text{boolean} \ (\text{Equal} \ e1 \ e2) \ s = (\text{expr} \ e1 \ s) == (\text{expr} \ e2 \ s) \]
\[ \text{boolean} \ (\text{Not} \ b) \ s = \text{not} \ (\text{boolean} \ b) \ s \]

\[ \text{ifelse} :: \text{Bool} \rightarrow \text{a} \rightarrow \text{a} \rightarrow \text{a} \]
\[ \text{ifelse} \ \text{True} \ x \ y = x \]
\[ \text{ifelse} \ \text{False} \ x \ y = y \]

Com S 541
Practical and Theoretical Issues

- **Modeling**
  - Errors and non-termination:
    - Need a special “error” in semantic domains
  - Branching:
    - Semantic domains in which “continuations” model “the rest of the program” make it easy to transfer control
  - Interactive input
  - Dynamic typing

- **What is the semantics of recursive functions?**
  - Recursive domain specifications
  - Need least fixed point theory

- **How to model concurrency and non-determinism?**
  - Powerdomains – set of all possible results for different evaluations
  - “true concurrency” requires other models (e.g. CCS, \(\pi\)-calculus, Petri-Nets)