Big-Step Semantics

( + introduction to Semantics)
"Semantics" = meaning/significance (ancient Greek) of programs

vs. "Syntax" (rules for writing valid programs)
i.e., given a program — a collection of valid syntactic objects... semantic analysis describes the program's behavior mathematically
"Mathematical"?

- rigorous/precise
- abstract/symbolic representation
- amenable to proof methods (e.g., proof by induction)
Approaches to modeling semantics

- **Operational semantics** — describe the execution of a program and its effects directly, as on an abstract machine.

- **Denotational semantics** — model language constructs as "denotations", i.e., values in a mathematical domain, and reason about those.

- **Axiomatic semantics** — describe a program in terms of logical predicates that are satisfied before and after its execution.
Type of Operational Semantics

- **Big-step** – evaluate an entire expression in one big step
  \[ e \Downarrow v : \text{expression } e \text{ evaluates to value } v \]

- **Small-step** – evaluation is modeled as a series of small steps
  \[ e \rightarrow e' \rightarrow e'' \rightarrow e''' \rightarrow \ldots \rightarrow v \]
  \[ e \rightarrow^* v : e \text{ evaluate to } v \text{ after a series of steps} \]
How can we assert/prove that evaluations and computations proceed a certain way?

Need to be able to:

1. Make assertions about individual syntactic objects
2. Connect these assertions
3. Use these connections to create larger narratives (proofs!)
Inference Rules

form of a rule: \[ \text{[NAME]} \rightarrow \text{premise}_1, \ldots, \text{premise}_k \text{[side conditions]} \]

where each premise \( \vdash \) the conclusion is an assertion about a syntactic object

aka "judgement"

e.g., assertions: \( n \) is odd
\( e \rightarrow e' \)
\( e \in V \)
\( \vdash e : t \leftarrow \text{type of } e \text{ is } t \)
Inference Rules

form of a rule: \[ \text{[NAME]} \quad \text{premise}_1, \ldots, \text{premise}_k \quad \Rightarrow \text{conclusion} \quad \text{[side conditions]} \]

meaning of a rule: if all premises (and side condition(s), if present) hold, then the conclusion holds.

a rule with no premises is an axiom;
otherwise, it is a proper rule.
e.g., inference rules for even/odd integers over multiplication

\[
\begin{align*}
\text{MOD } 0 & \quad x \mod 2 = 0 \\
& \quad x \text{ is even} \\
\text{MOD } 1 & \quad x \text{ is odd} \\
\end{align*}
\]

\[
\begin{align*}
\text{EVEN \times EVEN} & \quad x \text{ is even} \quad y \text{ is even} \\
& \quad x \times y \text{ is even} \\
\text{ODD \times EVEN} & \quad x \text{ is odd} \quad y \text{ is even} \\
& \quad x \times y \text{ is even} \\
\text{EVEN \times ODD} & \quad x \text{ is even} \quad y \text{ is odd} \\
& \quad x \times y \text{ is even} \\
\text{ODD \times ODD} & \quad x \text{ is odd} \quad y \text{ is odd} \\
& \quad x \times y \text{ is odd}
\end{align*}
\]
Proof Trees

- We can derive a proof tree for a given assertion, where:
  - the root of the proof tree (its final conclusion) is the assertion we wish to prove
  - each subtree is a proof tree of the conclusion’s premises

\[
\begin{align*}
\text{leaves} &= \text{axioms} \\
\frac{P_1}{C_1} &\quad \frac{P_2}{P_3} &\quad \frac{P_4, P_5}{C_4} \\
\frac{C_1, P_2}{C_3} &\quad \frac{C_3}{C} \leftarrow \text{"root"}
\end{align*}
\]
Derivation Strategies

- forward chaining / bottom-up construction
  - start w/ axioms and work towards conclusion
  - end goal may not be reachable — when to stop?

- backward chaining / top-down construction
  - start w/ conclusion and work "backwards" towards axioms
  - goal is known — can we derive it?
e.g., prove "21347 \times 12345 is odd"

\[
\begin{align*}
\text{MOD} & \quad 21347 \mod 2 = 1 \\
\text{ODD} \times \text{ODD} & \quad 21347 \times 12345 \text{ is odd}
\end{align*}
\]
Big-step Semantics

- a big-step semantic rule makes an assertion about what a syntactic object would evaluate to on an abstract machine.

- state of the abstract machine (aka "configuration") includes:
  - the syntactic object to evaluate (e.g., expressions, statements, etc.)
  - the environment (e.g., variable → value mappings)
The Evaluation Relation: "\(\downarrow\)"

\(\downarrow\) is a relation over:
- syntactic objects
- environments
- possible results

i.e., \(\downarrow \in S \times \Sigma \times R\)

we write (in "mixfix" form): \(\langle s, \sigma \rangle \downarrow r\)
"Syntactic Objects"

i.e., as found in a program written in some language.

- what syntactic objects are possible/valid?
- need a formal way of specifying a language
  - Grammar
a grammar $G = (V, T, S, P)$ consists of:

- $V$, the **vocabulary**: a non-empty set of symbols.
- $T \subseteq V$: the set of **terminal symbols** (cannot be rewritten as other symbols).
- $S \in V$: the **start symbol**
- $P$: a set of **productions**

Each production is a "rewrite rule" of form $LHS \rightarrow RHS$; $LHS$ and $RHS$ consist of symbols, and $LHS$ contains at least one non-terminal.

$V - T = N$: the set of non-terminal symbols.
e.g., use the following grammar to generate some sentences:

\[ V = \{ \epsilon \}, E \}\]

\[ T = \{ \epsilon \}, (, ) \}\]

\[ S = E \]

\[ P = \{ E \rightarrow EE, E \rightarrow (E), E \rightarrow \epsilon \} \]

1. \[ E \rightarrow \epsilon = \"\" \]
2. \[ E \rightarrow (E) \rightarrow ((E)) \rightarrow ((\epsilon)) \rightarrow () \]
3. \[ E \rightarrow EE \rightarrow (E)E \rightarrow (\epsilon)E \rightarrow ()(E) \rightarrow ()(\epsilon) = ()() \]
Noam Chomsky created a classification system for formal grammars in 1956.

Chomsky hierarchy:

- **type 0**: unrestricted
- **type 1**: context sensitive
- **type 2**: context free
- **type 3**: regular
**Type 0 (unrestricted):** no restrictions on grammar

**Type 1 (context-sensitive):** productions have form \( \alpha A \beta \rightarrow \alpha X \beta \), where

A is a non-terminal and \( \alpha, \beta, X \) are strings of terminals/non-terminals (\( \neq \) non-empty).

**Type 2 (context-free):** productions have form \( A \rightarrow \alpha \), where

A is a non-terminal and \( \alpha \) is a string of terminals/non-terminals.

**Type 3 (regular):**

- **[Right regular]:** productions have form \( A \rightarrow aB \) or \( A \rightarrow a \),

- **[Left regular]:** productions have form \( A \rightarrow Ba \) or \( A \rightarrow a \), where A, B are non-terminals, and a is a terminal symbol (or \( \varepsilon \))
Regular grammars, the simplest type, are often used to specify the syntax of tokens for parsers.

- Regular expressions (regexes) are a way to specify these grammars.

Virtually all programming languages are specified using context-free grammars (CFGs).

- Backus-Naur Form (BNF) is often used to specify a CFG.
Regular Expressions

e.g., for a floating point number

\[ [+][-] ? ([0-9]+ \cdot \)? [0-9]+ \]

- char group
- optional preceding elem
- 0 or more of preceding elem

+42.15
-99.0
123
+8

- note lack of "memory"
- cannot describe structures w/ matching/nested elements

(real world regex libraries are more complex)
Backus-Naur Form

- list all productions in form

\[
\langle \text{non-terminal} \rangle ::= \langle \text{term/non-term} \rangle + \left( \langle \text{term/non-term} \rangle \right) *
\]

e.g., English phrases

\[
\text{SENTENCE} ::= \text{NP} \ \text{VP}
\]

\[
\text{NP} ::= \text{ARTICLE} \ \text{ADJ} \ \text{NOUN} \ | \ \text{ARTICLE} \ \text{NOUN}
\]

\[
\text{VP} ::= \text{VERB} \ \text{ADV} \ | \ \text{VERB}
\]

\[
\text{ARTICLE} ::= \text{the} \ | \ \text{a}
\]

\[
\text{ADJ} ::= \text{red} \ | \ \text{big}
\]

\[
\text{NOUN} ::= \text{car} \ | \ \text{computer}
\]

\[
\text{VERB} ::= \text{runs} \ | \ \text{jumps}
\]

\[
\text{ADV} ::= \text{hungrily} \ | \ \text{happily}
\]
"Simple Imperative Programming Language" (IMP)

arithmetic exps: \( E ::= \text{Integer} \mid \text{Var} \mid E + E \)

boolean exps: \( B ::= \text{true} \mid \text{false} \mid E \neq E \)

statements: \( S ::= \text{skip} \)
  \| \( \text{Var} ::= E \)
  \| \( S_1; S_2 \)
  \| \text{if } B \text{ then } S_1 \text{ else } S_2 \)
  \| \text{while } B \text{ do } S \)
Evaluation relations:

• \[ \text{arith exps: } \langle E, \sigma \rangle \downarrow_e v \]
  - \text{resulting value (integer)}
  - \text{environment (vars)}

• \[ \text{boolean exps: } \langle B, \sigma \rangle \downarrow_b b \]
  - \text{boolean value}
  - \text{new environment}

• \[ \text{statements: } \langle S, \sigma \rangle \downarrow_s \sigma' \]

* We will overload \( \downarrow \) when it is clear by context which relation to use.
Arithmetic Expression Rules

**LITERAL**
\[
\langle i, \sigma \rangle \Downarrow i \quad \text{if } i \in \mathbb{Z}
\]

**VARIABLE**
\[
\sigma(x) = v
\]

**ARITH**
\[
\langle e_1, \sigma \rangle \Downarrow v_1, \langle e_2, \sigma \rangle \Downarrow v_2 \\
\langle e_1 \oplus e_2, \sigma \rangle \Downarrow v_1 \oplus v_2
\]

x is a var, and \( x := v \in \sigma \)
Boolean Expression Rules

\[ \text{LITERAL} \quad \langle b, \sigma \rangle \Downarrow b \]

\[ \text{REL} \quad \langle e_1, \sigma \rangle \Downarrow e_1 V_1 \langle e_2, \sigma \rangle \Downarrow e_2 V_2 \]

\[ \Downarrow_b v_1 \sim v_2 \]
skip Rule

\[ \langle \text{skip}, \sigma \rangle \uparrow \sigma \]
Assignment Rule

\[
\text{ASSIGN} \quad \langle e, \sigma \rangle \Downarrow e \Downarrow v \\
\langle x := e, \sigma \rangle \Downarrow \sigma (x := v)
\]

- create new/update
- var mapping in env
Sequencing \((S_1; S_2)\) Rule

\[
\text{SEQ} \quad \frac{\langle s_1, \sigma \rangle \Downarrow \sigma' \quad \langle s_2, \sigma' \rangle \Downarrow \sigma''}{\langle s_1; s_2, \sigma \rangle \Downarrow \sigma''}
\]
**if-statement Rules**

**IF-T**

\[
\langle b, \sigma \rangle \Downarrow \text{b true} \quad \langle s_1, \sigma \rangle \Downarrow \sigma' \\
\frac{\langle \text{if } b \text{ then } s_1 \text{ else } s_2, \sigma \rangle \Downarrow \sigma'}
\]

**IF-F**

\[
\langle b, \sigma \rangle \Downarrow \text{b false} \quad \langle s_2, \sigma \rangle \Downarrow \sigma' \\
\frac{\langle \text{if } b \text{ then } s_1 \text{ else } s_2, \sigma \rangle \Downarrow \sigma'}
\]
**while-statement Rules**

\[
\text{WHILE-F} \quad \frac{\langle b, \sigma \rangle \Downarrow \text{false}}{\langle \text{while } b \text{ do } S, \sigma \rangle \Downarrow \sigma} \\
\text{WHILE-T} \quad \frac{\langle b, \sigma \rangle \Downarrow \text{true} \quad \langle S, \sigma' \rangle \Downarrow \sigma'' \quad \langle \text{while } b \text{ do } S, \sigma' \rangle \Downarrow \sigma''}{\langle \text{while } b \text{ do } S, \sigma \rangle \Downarrow \sigma''}
\]
e.g., proof trees for IMP

- prove that \(3a + 4b = 39\) in \(\sigma = \{a := 5, b := 6\}\)

- prove that \(\text{while } b \text{ do } s\)
  is equivalent to \(\text{if } b \text{ then } (s; \text{while } b \text{ do } s) \text{ else } \text{skip}\)
Prove that $3 \times a + 4 \times b = 39$ in $\sigma = \{ a := 5, b := 6 \}$.
- Prove that \( \text{while } b \text{ do } s \) is equivalent to \( \text{if } b \text{ then } (s; \text{while } b \text{ do } s) \text{ else skip} \)

- Two statements \( S_1 \) and \( S_2 \) are equivalent \((S_1 \sim S_2)\) if, for any two environments \( \sigma, \sigma' \),

\[
\langle S_1, \sigma \rangle \downarrow \sigma' \iff \langle S_2, \sigma \rangle \downarrow \sigma'
\]

- Let \( W \) be "while \( b \) do \( S \)"; we want to show:

\[
\langle W, \sigma \rangle \downarrow \sigma' \iff \langle \text{if } b \text{ then } (s; W) \text{ else } \text{skip}, \sigma \rangle \downarrow \sigma'
\]
- prove that \( \text{while } b \text{ do } s \) is equivalent to \( \text{if } b \text{ then } (s; \text{while } b \text{ do } s) \text{ else skip} \)

- consider \text{WHILE-F} and \text{WHILE-T} rules for \( W \)

\[
\text{WHILE-F} \quad \frac{<b, \sigma> \Downarrow \text{false}}{<W, \sigma> \Downarrow \sigma} \quad \Rightarrow \quad \text{IF-F} \quad \frac{<b, \sigma> \Downarrow \text{false}}{<\text{if } b \text{ then } (s; W) \text{ else skip}, \sigma> \Downarrow \sigma}
\]

\[
\frac{<\text{skip}, \sigma> \Downarrow \sigma}{<\text{skip}, \sigma> \Downarrow \sigma}
\]
- prove that \( \text{while } b \text{ do } s \) is equivalent to \( \text{if } b \text{ then } ( s ; \text{while } b \text{ do } s ) \text{ else skip} \)

- consider \text{WHILE-F} and \text{WHILE-T} rules for \( W \)

\[
\text{WHILE-T} \quad \begin{cases} 
\langle b, \sigma \rangle \Downarrow \text{true} & \Rightarrow \langle s, \sigma \rangle \Downarrow \sigma' \\
\langle w, \sigma \rangle \Downarrow \sigma' & \Rightarrow \langle w, \sigma' \rangle \Downarrow \sigma' 
\end{cases}
\]
- prove that \( \text{while } b \text{ do } s \) is equivalent to \( \text{if } b \text{ then } (s ; \text{while } b \text{ do } s) \text{ else skip} \)

- consider WHILE-F and WHILE-T rules for \( W \)

\[
\begin{align*}
\langle b, \sigma \rangle &\Downarrow \text{true} \\
\langle s, \sigma \rangle &\Downarrow \delta' \\
\langle w, \sigma \rangle &\Downarrow \delta' \quad \text{IF-T}
\end{align*}
\]

\[
\frac{\langle s; w, \sigma \rangle \Downarrow \delta'}{\langle \text{if } b \text{ then } (s; w) \text{ else skip}, \sigma \rangle \Downarrow \delta'}
\]
Back to Interpreters

\[ \Downarrow \equiv \text{eval} \]

\[ \sigma \equiv \text{env} \]

i.e., Big step semantics also gives us a rigorous, concise way to specify the behavior of an interpreter/language!

e.g., dozens of pages of formal semantics vs. hundreds of English for specification of Java lang.