Formal Operational Semantics

So far, we’ve presented operational semantics using interpreters: precise and executable, but verbose and too concrete.

One alternative: describe semantics using state transition judgments.

Example language

```latex
exp := \text{var} | \text{int} \\
| \text{'}\text{' }+\text{' }\text{exp exp '}\text{' }\text{'}
| \text{'}\text{' }\text{let' }\text{var exp exp '}\text{' }\text{'}
| \text{'}\text{' }\text{:=' }\text{var exp '}\text{' }\text{'}
| \text{'}\text{' }\text{if' }\text{exp exp exp '}\text{' }\text{'}
| \text{'}\text{' }\text{while' }\text{exp exp '}\text{' }\text{'}
| \text{etc.}
```
To evaluate this language, we choose a machine state consisting of:

- the current environment $E$, which maps each in-scope variable $x$ to a location $l$.
- the current store $S$, which maps each location $l$ to an integer value $v$.
- the current expression $e$, to be evaluated.

We give the state transitions in the form of judgments:

$$\langle e, E, S \rangle \downarrow \langle v, S' \rangle$$

Intuitively, this says that evaluating expression $e$ in environment $E$ and store $S$ yields the value $v$ and the (possibly) changed store $S'$. 

Evaluating expression $e$ in environment $E$ and store $S$ yields value $v$ and (possibly) changed store $S'$. 
Evaluate by inference

To describe the machine's operation, we give rules of inference that say when a judgment can be derived from judgments about sub-expressions.

\[
\frac{\text{premises}}{\text{conclusion}} \quad \text{(Name of rule)}
\]

We can view evaluation of the program as the process of building an inference tree.

Notation has similarities to axiomatic semantics: idea of derivation is that same, but contents of judgments is different.
Environments and Stores, Formally

- We write $E(x)$ means the result of looking up $x$ in environment $E$. (This notation is because an environment is like a function taking a name as argument and returning a meaning as result.)
- We write $E + \{x \mapsto v\}$ for the environment obtained from existing environment $E$ by extending it with a new binding from $x$ to $v$. If $E$ already has a binding for $x$, this new binding replaces it.

The domain of an environment, $\text{dom}(E)$, is the set of names bound in $E$.

Analogously with environments, we’ll write
- $S(l)$ to mean the value at location $l$ of store $S$
- $S + \{l \mapsto v\}$ to mean the store obtained from store $S$ by extending (or updating) it so that location $l$ maps to value $v$.
- $\text{dom}(S)$ for the set of locations bound in store $S$.

Also, we’ll write
- $S - \{l\}$ to mean the store obtained from store $S$ by removing the binding for location $l$. 
**Evaluation Rules (1)**

\[
\begin{align*}
    & l = E(x) \quad v = S(l) \quad \frac{}{\langle x, E, S \rangle \downarrow \langle v, S \rangle} \quad \text{(Var)} \\
    & \langle i, E, S \rangle \downarrow \langle i, S \rangle \quad \text{(Int)} \\
    & \langle e_1, E, S \rangle \downarrow \langle v_1, S' \rangle \quad \langle e_2, E, S' \rangle \downarrow \langle v_2, S'' \rangle \quad \frac{}{\langle (+ e_1 e_2), E, S \rangle \downarrow \langle v_1 + v_2, S'' \rangle} \quad \text{(Add)} \\
    & \langle e_1, E, S \rangle \downarrow \langle v_1, S' \rangle \quad l \notin \text{dom}(S') \quad \langle e_2, E + \{x \mapsto l\}, S' + \{l \mapsto v_1\} \rangle \downarrow \langle v_2, S'' \rangle \quad \frac{}{\langle (\text{let } x e_1 e_2), E, S \rangle \downarrow \langle v_2, S'' - \{l\} \rangle} \quad \text{(Let)} \\
    & \langle e, E, S \rangle \downarrow \langle v, S' \rangle \quad l = E(x) \quad \frac{}{\langle (:= x e), E, S \rangle \downarrow \langle v, S' + \{l \mapsto v\} \rangle} \quad \text{(Assgn)}
\end{align*}
\]
**EVALUATION RULES (2)**

\[
\begin{align*}
\langle e_1, E, S \rangle \downarrow \langle v_1, S' \rangle & \quad v_1 \neq 0 & \langle e_2, E, S' \rangle \downarrow \langle v_2, S'' \rangle & \quad \text{(If-nzero)} \\
\langle (\text{if } e_1 \ e_2 \ e_3), E, S \rangle \downarrow \langle v_2, S'' \rangle
\end{align*}
\]

\[
\begin{align*}
\langle e_1, E, S \rangle \downarrow \langle 0, S'' \rangle & \quad \langle e_3, E, S' \rangle \downarrow \langle v_3, S''' \rangle & \quad \text{(If-zero)} \\
\langle (\text{if } e_1 \ e_2 \ e_3), E, S \rangle \downarrow \langle v_3, S''' \rangle
\end{align*}
\]

\[
\begin{align*}
\langle e_1, E, S \rangle \downarrow \langle v_1, S' \rangle & \quad v_1 \neq 0 & \langle e_2, E, S' \rangle \downarrow \langle v_2, S'' \rangle & \quad \text{(While-nzero)} \\
\langle (\text{while } e_1 \ e_2), E, S'' \rangle \downarrow \langle v_3, S''' \rangle
\end{align*}
\]

\[
\begin{align*}
\langle e_1, E, S \rangle \downarrow \langle 0, S' \rangle & \quad \text{(While-zero)} \\
\langle (\text{while } e_1 \ e_2), E, S \rangle \downarrow \langle 0, S' \rangle
\end{align*}
\]
Example Derivation Trees

\[
\begin{align*}
\langle 10, \emptyset, \emptyset \rangle \Downarrow \langle 10, \emptyset \rangle & \quad \text{(Int)} \\
\langle 21, E_1, S_1 \rangle \Downarrow \langle 21, S_1 \rangle & \quad \text{(Int)} \\
\langle (+ (:= x \ 21) \ x), E_1, S_1 \rangle \Downarrow \langle 42, S_2 \rangle & \quad \text{(Let)} \\
\langle x, E_1, S_2 \rangle \Downarrow \langle 21, S_2 \rangle & \quad \text{(Add)} \\
\langle 10, \emptyset, \emptyset \rangle \Downarrow \langle 10, \emptyset \rangle & \quad \text{(Var)} \\
\langle (+ (:= x \ 21) \ x), E_1, S_1 \rangle \Downarrow \langle 42, S_2 \rangle & \quad \text{(Let)} \\
\langle 10, \emptyset, \emptyset \rangle \Downarrow \langle 10, \emptyset \rangle & \quad \text{(Int)} \\
\langle (+ (:= x \ 21) \ x), E_1, S_1 \rangle \Downarrow \langle 42, S_2 \rangle & \quad \text{(Let)} \\
\langle x, E_1, S_2 \rangle \Downarrow \langle 21, S_2 \rangle & \quad \text{(Add)} \\
\end{align*}
\]

where \( E_1 = \{ x \mapsto L_1 \}, \ S_1 = \{ L_1 \mapsto 10 \}, \ S_2 = \{ L_1 \mapsto 21 \} \).

\[
\begin{align*}
\langle x, E_1, S_1 \rangle \Downarrow \langle -1, S_1 \rangle & \quad \text{(Var)} \\
\langle (+ (:= x \ 21) \ x), E_1, S_1 \rangle \Downarrow \langle 0, S_2 \rangle & \quad \text{(Add)} \\
\langle x, E_1, S_2 \rangle \Downarrow \langle 0, S_2 \rangle & \quad \text{(Var)} \\
\langle (\text{while } x \ (\text{:= } x \ (\text{+ } 1))), E_1, S_1 \rangle \Downarrow \langle 0, S_2 \rangle & \quad \text{(While-zero)} \\
\langle (\text{while } x \ (\text{:= } x \ (\text{+ } 1))), E_1, S_1 \rangle \Downarrow \langle 0, S_2 \rangle & \quad \text{(While-nzero)} \\
\end{align*}
\]

where \( E_1 = \{ x \mapsto L_1 \}, \ S_1 = \{ L_1 \mapsto -1 \}, \ S_2 = \{ L_1 \mapsto 0 \} \).
About the rules

- As usual in inference systems, a rule only applies if all the premises above the line can be shown.

- Structure of rules guarantees that at most one rule is applicable at any point.

- Store relationships constrain the order of evaluation of premises.
  
  (For simplicity here, we use just a single global store)

- If no rules apply, the evaluation gets stuck; this corresponds to an (unchecked) runtime error.
Rules vs. Interpreter

- We can view a recursive interpreter as implementing a bottom-up exploration of the inference tree.

- A function like

\[
\text{Value eval}(\text{Exp } e, \text{Env } env) \{ \ldots \}
\]

returns a value \( v \) and has side effects on a global store \text{store} such that

\[
\langle e, env, store \rangle_{\text{before}} \Downarrow \langle v, store \rangle_{\text{after}}
\]

- The implementation of \text{eval} dispatches on the syntactic form of \( e \), choosing the appropriate rule,

- and makes recursive calls on \text{eval} corresponding to the premises of that rule.

- Question: How deep can the derivation tree get?