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:

```
exp := var | int
| '( '+' exp exp ')' |
| '( 'let' var exp exp ')' |
| '( ':'=' var exp ')' |
| '( 'if' exp exp exp ')' |
| '( 'while' exp exp ')' |
| etc.
```
State machine transition judgments

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

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

current expression

result value

initial store

final store

maps locations $l$ to values $v$

maps locations $l$ to values $v$

current environment

maps vars $x$ to locations $l$
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 essentially the 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)**

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

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

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

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

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

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

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
  
  ```
  Value eval(Exp e, Env env) { .... }
  ```

  returns a value $v$ and has side effects on a global store $store$ such that

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

- The implementation of `eval` dispatches on the syntactic form of $e$, choosing the appropriate rule,

- and makes recursive calls on `eval` corresponding to the premises of that rule.

- Question: How deep can the derivation tree get?