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:

```plaintext
exp := var | int
    | '(' '+' exp exp ')' 
    | '(' 'let' var exp exp ')' 
    | '(' '==' var exp ')' 
    | '(' 'if' exp exp exp ')' 
    | '(' 'while' exp exp ')' 
    | etc.
```
To evaluate this language, we choose a machine state consisting of:

- the current environment $E$, which maps each in-scope variable 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.

\[
\begin{array}{c}
\text{premises} \\
\hline
\text{conclusion}
\end{array}
\] (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)**

\[
\begin{align*}
  l &= E(x) \quad v = S(l) & \text{(Var)} \\
  \langle x, E, S \rangle &\downarrow \langle v, S \rangle \\
  \langle i, E, S \rangle &\downarrow \langle i, S \rangle & \text{(Int)} \\
  \langle e_1, E, S \rangle &\downarrow \langle v_1, S' \rangle \quad \langle e_2, E, S' \rangle &\downarrow \langle v_2, S'' \rangle & \text{(Add)} \\
  \langle (+ e_1 e_2), E, S \rangle &\downarrow \langle v_1 + v_2, S'' \rangle \\
  \langle e_1, E, S \rangle &\downarrow \langle v_1, S' \rangle \quad l \notin \text{dom}(S') \\
  \langle e_2, E + \{x \mapsto l\}, S' + \{l \mapsto v_1\} \rangle &\downarrow \langle v_2, S'' \rangle & \text{(Let)} \\
  \langle (\text{let } x e_1 e_2), E, S \rangle &\downarrow \langle v_2, S'' - \{l\} \rangle \\
  \langle e, E, S \rangle &\downarrow \langle v, S' \rangle \quad l = E(x) & \text{(Assgn)} \\
  \langle (\text{:= } x e), E, S \rangle &\downarrow \langle v, S' + \{l \mapsto v\} \rangle 
\end{align*}
\]
\[
\langle e_1, E, S \rangle \downarrow \langle v_1, S' \rangle \quad v_1 \neq 0 \quad \langle e_2, E, S' \rangle \downarrow \langle v_2, S'' \rangle \\
\frac{\langle (\text{if } e_1 \ e_2 \ e_3), E, S \rangle \downarrow \langle v_2, S'' \rangle}{\langle (\text{if } e_1 \ e_2 \ e_3), E, S \rangle \downarrow \langle v_2, S'' \rangle} \quad \text{(If-nzero)}
\]

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

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

\[
\langle e_1, E, S \rangle \downarrow \langle 0, S' \rangle \quad \langle e_1, E, S \rangle \downarrow \langle 0, S' \rangle \\
\frac{\langle (\text{while } e_1 \ e_2), E, S \rangle \downarrow \langle 0, S' \rangle}{\langle (\text{while } e_1 \ e_2), E, S \rangle \downarrow \langle 0, S' \rangle} \quad \text{(While-zero)}
\]
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 \( \text{store} \) such that

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

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?