The **dynamic semantics** of a language defines what happens when you run a program.

There are many approaches to defining the dynamic semantics of a language:

- Small-step operational semantics
- Big-step operational semantics
- Abstract machines
- Translation to another language
- Denotational semantics (translation to a mathematical system)
- Axiomatic semantics

In this class we will primarily be concerned with the small-step operational semantics, which is the most widely used approach.
Recall the language of Arithmetic Expressions

\[
e ::= \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \\
   \quad \mid 0 \mid \text{succ } e \mid \text{pred } e \mid \text{iszero } e
\]

\[
v ::= \text{true} \mid \text{false} \mid \text{nv}
\]

\[
nv ::= 0 \mid \text{succ } \text{nv}
\]
Small-step operational semantics

In a small-step semantics, we take textual rewriting view of running a program. That is, we evaluate a program by changing the program text a little bit at a time.

Example

\[
\text{if iszero } (\text{pred } (\text{succ } 0)) \text{ then } 0 \text{ else succ } 0
\]

\[
\rightarrow
\]

\[
\text{if iszero } 0 \text{ then } 0 \text{ else succ } 0
\]

\[
\rightarrow
\]

\[
\text{if } \text{true } \text{ then } 0 \text{ else succ } 0
\]

\[
\rightarrow
\]

\[
0
\]
We define a relation between two expressions that says when one expression evaluates to the other expression. We’ll call this the ReducesTo relation and define the relation inductively with the following rules. The first bunch of rules do some real computation.

(1) \((\text{pred } 0, 0) \in \text{ReducesTo}\)

(2) \((\text{pred succ } \text{nv, nv}) \in \text{ReducesTo}\)

(3) \((\text{iszero succ } \text{nv, false}) \in \text{ReducesTo}\)

(4) \((\text{iszero } 0, \text{true}) \in \text{ReducesTo}\)

(5) \((\text{if } \text{true then } e_2 \text{ else } e_3, e_2) \in \text{ReducesTo}\)

(6) \((\text{if } \text{false then } e_2 \text{ else } e_3, e_3) \in \text{ReducesTo}\)
Before defining the rest of the rules for ReducesTo, we introduce the following notation:

\[ e \mapsto e' \equiv (e, e') \in \text{ReducesTo} \]
The second bunch of rules, called congruence rules, define the evaluation order by saying where you can reach inside a large expression to evaluate a subexpression.

\[ (7) \quad e \mapsto e' \quad \text{succ} \quad e \mapsto \text{succ} \quad e' \]

\[ (8) \quad e \mapsto e' \quad \text{pred} \quad e \mapsto \text{pred} \quad e' \]

\[ (9) \quad e \mapsto e' \quad \text{iszero} \quad e \mapsto \text{iszero} \quad e' \]

\[ (10) \quad e_1 \mapsto e'_1 \quad (\text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3) \mapsto (\text{if} \ e'_1 \ \text{then} \ e_2 \ \text{else} \ e_3) \]
Inversion Lemmas

Suppose you know that for some $e$ and $e'$, $e \rightarrow e'$. What further information can you glean from this based on the rules for ReducesTo? We can do case analysis on the rules. For example, suppose

\[
\text{if true then } e_2 \text{ else } e_3 \rightarrow e_4
\]

Then you know that $e_2 = e_4$ because the only rule that could have appeared as the final step in the derivation is rule (5). Why not rule (10)? Because then there would need to be a rule of the form $true \rightarrow \_\_\_$, but there is no such rule.
Theorem

If $e \rightarrow e'$ and $e \rightarrow e''$ then $e' = e''$.

Proof by rule induction on $e \rightarrow e'$.

Case (1) $\text{pred } 0 \rightarrow 0$:

So $e = \text{pred } 0$, $e' = 0$ and by assumption we have $\text{pred } 0 \rightarrow e''$. There are two rules that could have applied (1) and (8), but (8) can’t apply because there are no rules of the form $0 \rightarrow \_$. Thus, $e'' = 0$ and we conclude that $e' = e''$. 
A Proof by Rule Induction on ReducesTo, continued

Case (7) \[
\begin{array}{c}
e_1 \rightarrow e'_1 \\
\text{succ } e_1 \rightarrow \text{succ } e'_1
\end{array}
\]

So \( e = \text{succ } e_1 \) and \( e' = \text{succ } e'_1 \). From the induction hypothesis we have

(a) \( \forall x. e_1 \rightarrow x \) implies \( x = e'_1 \).

Also, by assumption we have \( \text{succ } e_1 \rightarrow e'' \). The only rule that could have applied is (7), so we know there was some \( e''_1 \) such that \( e_1 \rightarrow e''_1 \). Then using (a) we have \( e''_1 = e'_1 \). Thus \( e = \text{succ } e'_1 = \text{succ } e''_1 = e'' \).

The other cases are left as an exercise.
Definition
An expression \( e \) is in **normal form** if \( \forall e'. \ e \not\rightarrow e' \).

Definition
Multi-step reduction (written \( \rightarrow^* \)) is the relation inductively defined by the following rules:

\[
\begin{align*}
(1) & \quad e \rightarrow^* e \\
(2) & \quad e_1 \rightarrow e_2 \quad e_2 \rightarrow^* e_3 \quad e_1 \rightarrow^* e_3
\end{align*}
\]

(This is an alternate but equivalent definition than the one in the textbook.)
The Untyped Lambda Calculus

Syntax:

\[
\begin{align*}
  x & \quad \text{Variables} \\
  e & ::= \quad x \mid (\lambda x. e) \mid (e \ e) \quad \text{Expressions} \\
  v & ::= \quad \lambda x. e \quad \text{Values}
\end{align*}
\]

- An expression of the form \((\lambda x. e)\) create an anonymous function. The \(x\) is the one parameter of the function and \(e\) is the function body.

- An expression of the form \((e_1 \ e_2)\) is a function call. The expression \(e_1\) should evaluate to a function, which is then called using the result of \(e_2\) as the argument.
Call-by-value Small-step Semantics

(1) \[ ((\lambda x. e_1) \, v_2) \leadsto [x \mapsto v_2]e_1 \]

(2) \[ e_1 \leadsto e_1' \]
(2) \[ e_2 \leadsto e_2' \]

(2) \[ (e_1 \, e_2) \leadsto (e_1' \, e_2) \]
(2) \[ (v_1 \, e_2) \leadsto (v_1 \, e_2') \]

(Note where we use \( v \) instead of \( e \) to restrict the order of evaluation!)
The variable name associated with a λ doesn’t really matter. The meaning of the program remains unchanged if you change the variable name and consistently replace the occurrences of that variable in the λ.

\[ \lambda y. e = \lambda z. [y \mapsto z] e \quad \text{provided } z \text{ is not in } e \]
Substitution and Free Variables

We write \([x \mapsto e']e\) to say that \(e'\) is substituted for the free occurrences of \(x\) in \(e\).

For example, in the following we substitute the expression \((y \ z)\) for the variable \(x\) in the expression \(((\lambda x. \ x) \ x)\).

\[
[x \mapsto (y \ z)]((\lambda x. \ x) \ x) = ((\lambda x. \ x) (y \ z))
\]

A free occurrence of a variable \(x\) within an expression \(e\) is an occurrence of \(x\) in \(e\) that does not have a surrounding \(\lambda\) in \(e\) that binds \(x\).

The following examples, free occurrences are enclosed in a box.

- \((z \ x)\)
- \((\lambda y. \ y)\)
- \((\lambda y. \ (y \ x))\)
- \(((\lambda x. \ x) \ x)\)
- \(y\)
The function $FV$ computes the set of free variables in a given expression.

\[
\begin{align*}
FV(x) &= \{x\} \\
FV((e_1 e_2)) &= FV(e_1) \cup FV(e_2) \\
FV((\lambda x. e)) &= FV(e) - \{x\}
\end{align*}
\]
We need to be careful when defining substitution \([x \mapsto e']e\) to make sure that free variables in \(e'\) don’t get captured by \(\lambda\)s in \(e\).

The following is an example of what happens if you don’t define substitution correctly:

\[
(\lambda y. (\lambda x. \lambda y. x)\ y) \rightarrowto (\lambda y. (\lambda y. y))
\]

Here’s a correct definition of substitution:

\[
\begin{align*}
[x \mapsto e']y &= \text{if } x = y \text{ then } e' \text{ else } y \\
[x \mapsto e'](\lambda y. e) &= (\lambda z. [x \mapsto e'][y \mapsto z]e) \quad \text{(where } z \text{ is fresh)} \\
[x \mapsto e'](e_1\ e_2) &= ([x \mapsto e']e_1\ [x \mapsto e']e_2)
\end{align*}
\]
Congruence rules are a verbose way to specify evaluation order. A more succinct way is to use evaluation contexts.

Recall the small-step textual rewriting view of semantics:

```
if iszero (pred (succ 0)) then 0 else succ 0
```

↦−→

```
if iszero 0 then 0 else succ 0
```

An **evaluation context** is everything *not* inside an evaluation box. We can use a grammar to define where the boxes are allowed and what the surrounding evaluation context looks like.

```latex
E ::= []  \hspace{1cm} \text{The evaluation box forms a hole in the context.}
(E e)
(v E)
```
Filling an Evaluation Context

\[
\begin{align*}
\text{fill}([], e) &= e \\
\text{fill}((E \ e'), e) &= (\text{fill}(E, e) \ e') \\
\text{fill}((\nu \ E), e) &= (\nu \ \text{fill}(E, e))
\end{align*}
\]

Notation: \( E[e] \equiv \text{fill}(E, e) \)

Example

\((\lambda y. \ y)[[]][\lambda x. \ x] = (\lambda y. \ y)(\lambda x. \ x))\)
We separate out the computational reduction rules into the relation $\rightarrow$. For the lambda calculus there is just one rule:

$$\beta\quad (\lambda x.e_1 \, v_2) \rightarrow [x \mapsto v_2]e_1$$

We then define top-level reduction using evaluation contexts:

$$e \rightarrow e' \quad \frac{E[e] \rightarrow E[e']}{E[e] \rightarrow E[e']}$$