Goals of this tutorial
You should be able to...
- correctly apply stepping rules.
- understand and use structures.

Review: Stepping Rules
Always evaluate the topmost, leftmost unsimplified expression first.

Application of built-in functions: \((f \, v_1 \ldots \, v_n) \Rightarrow v\)
where \(f\) is a built-in function and \(v\) is the value of \(f(v_1, \ldots, v_n)\)

Substitution of Constants: \(id \Rightarrow val\), where \((\text{define id val})\) occurs previously.
Review: Stepping Rules

Application of user-defined functions: The general substitution rule is:
\[(f\ v_1 \ldots v_n) \Rightarrow \exp'\]
where \((\text{define}\ (f\ x_1 \ldots x_n)\ \exp)\) occurs previously, and \(\exp'\) is obtained by substituting all occurrences of the formal parameter \(x_i\) replaced by the value \(v_i\) (for \(i\) from 1 to \(n\)) into the expression.

Clicker Question
What are the next two steps for this code? (Do not skip any steps.)
\[\text{(define } x\ 3)\]
\[\text{(define } (\text{foo } a\ b)\ (-\ +\ a\ b\ x\ (\text{min } a\ (\text{sqr } b))))\]
\[\text{(foo } 1\ x)\]

A \(\Rightarrow\ (\ +\ 1\ 3\ 3\ (\text{min } 1\ (\text{sqr } 3))) \Rightarrow (\ +\ 1\ 3\ 3\ (\text{min } 1\ 9))\)
B \(\Rightarrow\ (\ +\ 1\ 3\ x\ (\text{min } 1\ (\text{sqr } 3))) \Rightarrow (\ +\ 1\ 3\ 3\ (\text{min } 1\ (\text{sqr } 3)))\)
C \(\Rightarrow\ (\text{foo } 1\ 3) \Rightarrow (\ +\ 1\ 3\ 3\ (\text{min } 1\ (\text{sqr } 3)))\)
D \(\Rightarrow\ (\text{foo } 1\ 3) \Rightarrow (\ +\ 1\ 3\ x\ (\text{min } 1\ (\text{sqr } 3)))\)

Clicker Question
The following definitions have been processed:
\[\text{(define } x\ 10)\]
\[\text{(define } y\ (\ +\ x\ x))\]
what are the next two steps for this code?
\[\ (\ +\ y\ y)\]

A \(\Rightarrow\ (\ +\ (\ +\ x\ x)\ y) \Rightarrow (\ +\ (\ +\ 10\ x)\ y)\)
B \(\Rightarrow\ (\ +\ (\ +\ 10\ x)\ y) \Rightarrow (\ +\ (\ +\ 10\ 10)\ y)\)
C \(\Rightarrow\ (\ +\ 20\ 20) \Rightarrow 40\)
D \(\Rightarrow\ (\ +\ 20\ y) \Rightarrow (\ +\ 20\ 20)\)
Review: Stepping Rules
Substitution in cond expressions

There are three rules: when the first expression is false, when it is true, and when it is else.

\[(\text{cond} \ [\text{false} \ \text{exp}] \ldots) \Rightarrow (\text{cond} \ldots)\]

\[(\text{cond} \ [\text{true} \ \text{exp}] \ldots) \Rightarrow \text{exp}\]

\[(\text{cond} \ [\text{else} \ \text{exp}]) \Rightarrow \text{exp}\]

These suffice to simplify any \text{cond} expression, note the error case too:

\[(\text{cond} \ [\text{false} \ \text{exp}]) \Rightarrow (\text{cond}) \Rightarrow \text{ERROR}\]

Group Problem - Stepping \text{cond}

The following have been processed in the Beginning Student language:

\(\text{(define x 1)}\)
\(\text{(define y 1)}\)

Step through the following:

\[(\text{cond} \ [(\mathord{=} \ x \ 0) \ \text{one}]\]
\n\n\n\[\text{[else} \ (< \ (/ \ y \ x) \ c)]\]

Review: Stepping Rules
Simplification Rules for \text{and} and \text{or}

The simplification rules we use for Boolean expressions involving \text{and} and \text{or} differ from the ones the Stepper in DrRacket uses.

\[(\text{and} \ \text{false} \ldots) \Rightarrow \text{false}\]

\[(\text{and} \ \text{true} \ldots) \Rightarrow (\text{and} \ldots)\]

\[(\text{and}) \Rightarrow \text{true}\]

\[(\text{or} \ \text{true} \ldots) \Rightarrow \text{true}\]

\[(\text{or} \ \text{false} \ldots) \Rightarrow (\text{or} \ldots)\]

\[(\text{or}) \Rightarrow \text{false}\]
Group Problem - Stepping and
The following have been processed in the Beginning Student language:

\[
\begin{align*}
\text{(define } & \text{x } 0) \\
\text{(define } & \text{y } ((+ \text{x } 1))
\end{align*}
\]
Step through the following:
\[
\text{(and } \neg \text{ (=} \text{x } 0) \langle=} \text{ (//= (/ \text{y } \text{x}) \text{c})}
\]

Review: Posn structures
- **constructor** function `make-posn`, with contract
  ;; make-posn: Num Num → Posn
- **selector** functions `posn-x` and `posn-y`, with contracts
  ;; posn-x: Posn → Num
  ;; posn-y: Posn → Num

Example:
\[
\begin{align*}
\text{(define mypoint } & \text{(make-posn } 8 \text{ } 1)) \\
\text{(posn-x mypoint) } & \Rightarrow \text{8} \\
\text{(posn-y mypoint) } & \Rightarrow \text{1}
\end{align*}
\]

Review: Posn structures
- **posn?**, with contract
  ;; posn?: Any → Bool

Example:
\[
\begin{align*}
\text{(posn? } & \text{(make-posn } 5 \text{ } 4)) \Rightarrow \text{true} \\
\text{(posn? } & \text{(make-posn 'red "snake") } \Rightarrow \text{true}
\end{align*}
\]
Review: Posn structures
Possible uses:
- coordinates of a point on a two-dimensional plane
- positions on a screen or in a window
- a geographical position

Note:
- An expression such as `(make-posn 8 1)` is considered a value, which will not be simplified further by the Stepper or our semantic rules.
- The expression `(make-posn (+ 4 4) (− 3 2))` would be simplified further to (eventually) `(make-posn 8 1)`.

Review - User-defined Structures
Consider the following structures used to represent a card and a hand of 3 cards:

```scheme
(define-struct card (suit value))
;; A Card is a (make-card Str Nat)
;; requires: suit is one of ("hearts", "spades", "clubs", "diamonds")
;; value is between 1 and 13, inclusive

(define-struct hand (card1 card2 card3))
;; A Hand is a (make-hand Card Card Card)
```

Review - User-defined Structures
These structures behave just like posn, which we looked at earlier:

```scheme
(define first-card (make-card "hearts" 3))
(define second-card (make-card "spades" 1))
(define bad-hand (make-hand second-card second-card second-card))

(card-suit first-card)
⇒ "hearts"

(card-value (hand-card1 bad-hand))
⇒ 1
```
Group Problem - Structures

Write a template function for Card called my-card-fn.

Group Problem - Structures

A card has a score given by the following rules:

- Hearts score 5 points
- Diamonds score 4 points
- Spades score 0 points
- Clubs score -5 points
- A card scores points equal to sum of its value and its suit score

Write a function hand-score that consumes a Hand and produces the total score of positive score cards in the hand. Include purpose, contract, examples for the main function.

Group Problem - Structures

Given the following constants below. Write an expression (without using numbers and only use arithmetic operators) that produces 3, and then write an expression that produces -7, under the same restrictions.

```
(define h1 (make-hand (make-card "hearts" 6)
                        (make-card "diamonds" 4)
                        (make-card "diamonds" 2)))

(define h2 (make-hand (make-card "spades" 12)
                        (make-card "clubs" 4)
                        (make-card "hearts" 9)))
```