$$s(n) = n \qquad s(b_1 \;\&\&\; b_2) = s(b_1) \text{ and } s(b_2) \qquad \frac{}{<Skip, s> \Downarrow s}\,,$$
$$s(a_1 + a_2) = s(a_1) \text{ plus } s(a_2) \qquad s(a_1 < a_2) = s(a_1) \text{ is less than } s(a_2)$$
$$s(a_1 \times a_2) = s(a_1) \text{ times } s(a_2)$$

$$\frac{s(a) = n}{(x \leftarrow a, s) \to (\textbf{skip}, s[x \mapsto n])} \qquad \frac{(c_1, s) \to (c_1', s')}{(c_1; c_2, s) \to (c_1'; c_2, s')} \qquad \frac{}{(\textbf{skip}; c_2, s) \to (c_2, s)}$$

$$\frac{s(b) = \mathit{true}}{(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, s) \to (c_1, s)} \qquad \frac{s(b) = \mathit{false}}{(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, s) \to (c_2, s)}$$

$$\frac{s(b) = \mathit{true}}{(\textbf{while } b \textbf{ do } c, s) \to (c; \textbf{while } b \textbf{ do } c, s)} \qquad \frac{s(b) = \mathit{false}}{(\textbf{while } b \textbf{ do } c, s) \to (\textbf{skip}, s)}$$

The pair $(\textbf{skip}, s)$ does not step: this is the halting configuration, representing a program that is done.

The idea is that cond {b ⇒ c, _⇒ c'} executes c if b is true in the initial store, otherwise it executes c′.
Likewise, cond {b1 ⇒ c1, b2 ⇒ c2, _⇒ c3} executes c1 if b1 is true, otherwise it executes c2 if b2 is true, otherwise it executes c3.

$$\frac{s(b) = \mathit{true}}{(\textbf{cond } \{b \Rightarrow c, \_ \Rightarrow c'\}, s) \to (c, s)} \qquad \frac{s(b) = \mathit{false}}{(\textbf{cond } \{b \Rightarrow c, \_ \Rightarrow c'\}, s) \to (c', s)}$$

$$\frac{s(b_1) = \mathit{true}}{(\textbf{cond } \{b_1 \Rightarrow c_1, b_2 \Rightarrow c_2, \_ \Rightarrow c_3\}, s) \to (c_1, s)}$$

$$\frac{s(b_1) = \mathit{false}}{(\textbf{cond } \{b_1 \Rightarrow c_1, b_2 \Rightarrow c_2, \_ \Rightarrow c_3\}, s) \to (\textbf{cond } \{b_2 \Rightarrow c_2, \_ \Rightarrow c_3\}, s)}$$

This command should always execute the body c at least once. Then, it should exit if b is true, and continue looping if b is false.

Solution to While-language: do-until

$$\frac{}{(\textbf{do } c \textbf{ until } b, s) \to (c; \textbf{if } b \textbf{ then skip else do } c \textbf{ until } b, s)}$$

(a) $(x \leftarrow x + 1, s_0) \to (\textbf{skip}, [1, 0])$ because $s_0(x + 1) = 1$

(b)

$$\begin{aligned}(x \leftarrow 3; y \leftarrow y + x, s_0) &\to (\textbf{skip}; y \leftarrow y + x, [3, 0]) && \text{because } (x \leftarrow 3, s_0) \to (\textbf{skip}, [3, 0])\\ &\to (y \leftarrow y + x, [3, 0])\\ &\to (\textbf{skip}, [3, 3]) && \text{because } [3, 0](y + x) = 3\end{aligned}$$

(c)

$$\begin{aligned}(\textbf{if } x < y \textbf{ then } x \leftarrow 2 \textbf{ else } x \leftarrow -2, s_0) &\to (x \leftarrow -2, s_0) && \text{because } s_0(x < y) = \mathit{false}\\ &\to (\textbf{skip}, [-2, 0]) && \text{because } s_0(-2) = -2\end{aligned}$$

(d)

$$\begin{aligned}(x \leftarrow -1; &\textbf{if } x < y \textbf{ then } x \leftarrow 2 \textbf{ else } x \leftarrow -2, s_0)\\ &\to (\textbf{skip}; \textbf{if } x < y \textbf{ then } x \leftarrow 2 \textbf{ else } x \leftarrow -2, [-1, 0]) && \text{because } (x \leftarrow -1, s_0) \to (\textbf{skip}, [-1, 0])\\ &\to (\textbf{if } x < y \textbf{ then } x \leftarrow 2 \textbf{ else } x \leftarrow -2, [-1, 0])\\ &\to (x \leftarrow 2, s_0) && \text{because } [-1, 0](x < y) = \mathit{true}\\ &\to (\textbf{skip}, [2, 0])\end{aligned}$$

(e)

$$\begin{aligned}(x \leftarrow 2; &\textbf{while } y < x \textbf{ do } y \leftarrow y + 1, s_0)\\ &\to (\textbf{skip}; \textbf{while } y < x \textbf{ do } y \leftarrow y + 1, [2, 0]) && \text{because } (x \leftarrow 2, s_0) \to (\textbf{skip}, [2, 0])\\ &\to (\textbf{while } y < x \textbf{ do } y \leftarrow y + 1, [2, 0])\\ &\to (y \leftarrow y + 1; \textbf{while } y < x \textbf{ do } y \leftarrow y + 1, [2, 0]) && \text{because } [2, 0](y < x) = \mathit{true}\\ &\to (\textbf{while } y < x \textbf{ do } y \leftarrow y + 1, [2, 1]) && \text{because } (y \leftarrow y + 1, [2, 0]) \to (\textbf{skip}, [2, 1])\\ &\to (y \leftarrow y + 1; \textbf{while } y < x \textbf{ do } y \leftarrow y + 1, [2, 1]) && \text{because } [2, 1](y < x) = \mathit{true}\\ &\to (\textbf{while } y < x \textbf{ do } y \leftarrow y + 1, [2, 2]) && \text{because } (y \leftarrow y + 1, [2, 1]) \to (\textbf{skip}, [2, 2])\\ &\to (\textbf{skip}, [2, 2]) && \text{because } [2, 2](y < x) = \mathit{false}\end{aligned}$$

CPS style:

```
fun div(x, y) {
  x / y
}
fun f(x, y) {
  3 * y * div(2, y)
}
f(div(3,4), 5))
```

```
fun div(x, y, cc) {
  cc(x / y)
}
fun f(x, y, cc) {
  div(2, y, fun (res) { cc(3 * y * res) })
}
div(3, 4, fun (res) { f(res, 5, top_cc) })
```

Any other answer with a single β-reduction is fine. For example,

```
fun div(x, y, cc) {
  cc(x / y)
}
fun f(x, y, cc) {
  div(2, y, fun (res) { cc(3 * y * res) })
}
// div(3, 4, fun (res) { f(res, 5, top_cc) }) =
(fun (res) { f(res, 5, top_cc) })(3/4)
```

```
fun div(x, y) {
  if (y != 0) {
    x / y
  } else {
    throw "Divide by zero";
  }
}
fun f(x, y) {
  try {
    3 * y * div(2, y)
  } catch (e) {
    0
  }
}
f(div(3,4), 5)
```

```
fun div(x, y, cc_ok, cc_fail) {
  if (y != 0) {
    cc_ok(x / y)
  } else {
    cc_fail("Divide by zero")
  }
}
fun f(x, y, cc) {
  div(2, y, fun (res) { cc(3 * y * res) }, fun (err) { cc(0) });
}
div(3, 4, fun(res) { f(res, 5, top_cc_ok);}, top_cc_fail);
```

Haskell:

```
gt = \x y -> x > y -- \xy.(x > y) -- this is a lambda abstraction, equivalent to gt x y = x > y
squaredList = map (\x -> x * x) [1, 2, 3, 4]
isOdd = \x -> x `mod` 2 == 1
```