Part I: Operational semantics

**Exercise I.1** Note that terms that can reduce are necessarily applications $a = a_1 a_2$. This is true for head reductions (the $\beta_v$ rule) and extends to reductions under contexts because non-trivial contexts are also applications. Since values are not applications, it follows that values do not reduce. Now, assume $a = E_1[a_1] = E_2[a_2]$ where $a_1$ and $a_2$ reduce by head reduction and $E_1, E_2$ are evaluation contexts. We show $E_1 = E_2$ and $a_1 = a_2$ by induction over the structure of $a$. By the previous remark, $a$ must be an application $bc$. We argue by case on whether $b$ or $c$ are applications.

- **Case 1:** $b$ is an application. $b$ is not a $\lambda$-abstraction, so $a$ cannot head-reduce by $\beta_v$, and therefore we cannot have $E_i = []$ for $i = 1, 2$. Similarly, $b$ is not a value, therefore we cannot have $E_i = b E'_i$. The only case that remains possible is $E_i = E'_i c$ for $i = 1, 2$. We therefore have two decompositions $b = E'_1[a_1] = E'_2[a_2]$. Applying the induction hypothesis to $b$, which is a strict subterm of $a$, it follows that $a_1 = a_2$ and $E'_1 = E'_2$, and therefore $E_1 = E_2$ as well.

- **Case 2:** $b$ is not an application but $c$ is. $b$ cannot reduce, so the case $E_i = E'_i c$ is impossible. $c$ is not a value, so the case $E_i = []$ is also impossible. The only possibility is therefore that $b$ is a value and $E_i = b E'_i$. The result follows from the induction hypothesis applied to $c$ and its two decompositions $c = E'_1[a_1] = E'_2[a_2]$.

- **Case 3:** neither $b$ nor $c$ are applications. The only possibility is $E_1 = E_2 = []$ and $a_1 = a_2 = a$.

**Exercise I.2** For each proposed rule $a \rightarrow b$, we expand the derived forms in $a$ (written $\approx$ below), perform reductions with the rules for the core constructs, then reintroduce derived forms in the result when necessary. For the let rule, this gives:

$$(\text{let } x = v \text{ in } a) \approx (\lambda x.a) v \rightarrow a[x \leftarrow v]$$

by $\beta_v$-reduction. For if/then/else:

$$\begin{align*}
\text{if true then } a \text{ else } b & \approx \text{match True()} \text{ with True()} \rightarrow a \mid \text{False()} \rightarrow b \\
& \rightarrow a \\
\text{if false then } a \text{ else } b & \approx \text{match False()} \text{ with True()} \rightarrow a \mid \text{False()} \rightarrow b \\
& \rightarrow \text{match False()} \text{ with False()} \rightarrow b \\
& \rightarrow b
\end{align*}$$
by match-reduction. Note that the second rule actually corresponds to two reductions in the base language. Finally, for pairs and projections:

\[
\text{fst}(v_1,v_2) \approx \text{(match Pair}(v_1,v_2)\text{ with Pair}(x_1,x_2) \rightarrow x_1) \rightarrow x_1[x_1 \leftarrow v_1,x_2 \leftarrow v_2] = v_1
\]

\[
\text{snd}(v_1,v_2) \approx \text{(match Pair}(v_1,v_2)\text{ with Pair}(x_1,x_2) \rightarrow x_2) \rightarrow x_2[x_1 \leftarrow v_1,x_2 \leftarrow v_2] = v_2
\]

again by match reductions.

Exercise I.3  Assume $1 \Rightarrow v$ for some $v$. There is only one evaluation rule that can conclude this:

\[
1 \Rightarrow \lambda x.c \quad 2 \Rightarrow v' \quad c[x \leftarrow v'] \Rightarrow v
\]

but of course $1$ evaluates only to $1$ and not to any $\lambda$-abstraction.

Now, assume that we have a derivation $a' \Rightarrow v$. By examination of the rules that can conclude this derivation, it can only be of the following form:

\[
\vdots
\]

\[
\lambda x.x \Rightarrow \lambda x.x \quad \lambda x.x \Rightarrow \lambda x.x \quad (x\ x)[x \leftarrow \lambda x.x] = a' \Rightarrow v
\]

Therefore, any derivation $D$ of $a' \Rightarrow v$ contains a sub-derivation $D'$ of $a' \Rightarrow v$ that is strictly smaller than $D$. Since derivations for the $\Rightarrow$ predicate are finite, this is impossible.

The difference between these two examples is visible on their reduction sequences: $a$ is an erroneous evaluation (a term that does not reduce but is not a value), while $a'$ reduces infinitely. The evaluation relation does not hold in either of these two cases.

Exercise I.4  The base case for the induction is $a = (\lambda x.c)\ v' \rightarrow c[x \leftarrow v'] = b$. We can build the following derivation of $a \Rightarrow v$ from that of $b \Rightarrow v$:

\[
\lambda x.c \Rightarrow \lambda x.c \quad v' \Rightarrow v' \quad c[x \leftarrow v'] = b \Rightarrow v
\]

\[
a = (\lambda x.c)\ v' \Rightarrow v
\]

using the fact that $v' \Rightarrow v'$ for all values $v'$ (check it by case over $v'$).

The first inductive case is $a = a'\ c \rightarrow b'\ c = b$ where $a' \rightarrow b'$. The evaluation derivation for $b \Rightarrow v$ is of the following form:

\[
b' \Rightarrow \lambda x.d \quad c \Rightarrow v' \quad d[x \leftarrow v'] \Rightarrow v
\]

\[
b'\ c \Rightarrow v
\]

Applying the induction hypothesis to the reduction $a' \rightarrow b'$ and the evaluation $b' \Rightarrow \lambda x.d$, it follows that $a' \Rightarrow \lambda x.d$. We can therefore build the following derivation:

\[
a' \Rightarrow \lambda x.d \quad c \Rightarrow v' \quad d[x \leftarrow v'] \Rightarrow v
\]

\[
a'\ c \Rightarrow v
\]

2
which concludes $a \Rightarrow v$ as claimed.

The second inductive case is $a = v' a' \rightarrow v' b' = b$ where $a' \rightarrow b'$. The evaluation derivation for $b \Rightarrow v$ is of the following form:

$$v' \Rightarrow \lambda x.c \quad b' \Rightarrow v'' \quad c[x \leftarrow v''] \Rightarrow v$$

Applying the induction hypothesis to the reduction $a' \rightarrow b'$ and the evaluation $b' \Rightarrow v''$, it follows that $a' \Rightarrow v''$. We can therefore build the following derivation:

$$v' \Rightarrow \lambda x.c \quad a' \Rightarrow v'' \quad c[x \leftarrow v''] \Rightarrow v$$

which concludes $a \Rightarrow v$ as claimed.

**Exercise I.5** A convenient representation for contexts $E$ is as Caml functions taking a term $a$ and returning the term $E[a]$.

```
type context = term -> term

let top : context = fun x -> x

let appleft (c: context) (b: term) : context = fun x -> App(c x, b)

let appright (a: term) (c: context) : context = fun x -> App(a, c x)
```

The decomposition of a term $a$ into a context and a subterm that potentially reduces follows the same reasoning as in exercise I.1. The base cases are 1- $a$ is not an application, and 2- $a$ is an application of a value to a value. In these cases, the context must be the “top” context. Otherwise, we have a application $a = a_1 a_2$ and we hunt for a potential redex in $a_1$, unless $a_1$ is already a value in which case we should look into $a_2$.

```
let rec decomp = function
  | App(a, b) ->
    if isvalue a then
      if isvalue b then
        (top, App(a, b))
      else
        let (c, b') = decomp b in (appright a c, b')
    else
      let (c, a') = decomp a in (appleft c b, a')
  | a -> (top, a)
```

Reductions at head and under contexts:
let head_reduce = function
  | App(Lam(x, a), v) when isvalue v -> Some(subst x v a)
  | _ -> None

let reduce a =
  let (c, a') = decomp a in
  match head_reduce a' with
  | Some a'' -> Some (c a'')
  | None -> None

Iterated reductions:

let rec evaluate a =
  match reduce a with None -> a | Some a' -> evaluate a'

Concerning efficiency, this interpreter has the same (bad) complexity as the SOS-based interpreter from the lecture. It is slightly less efficient in practice because the context must be explicitly constructed by decomp, then applied in reduce. Instead, the SOS-based interpreter combines the three phases (decompose, head-reduce, reconstruct by applying context) in one single traversal.

Exercise I.6 For question 1, define \( I = \lambda x. x \) and take \( a = (I I) (I I) \). We can reduce on the left of the top-level application to \( a_1 = I (I I) \). But we can also reduce on the right, obtaining \( a_2 = (I I) I \).

For question 2, the reduction sequences built during the proof of theorem 3 happen to use only left-to-right reductions, but remain valid with non-deterministic reductions. Concerning theorem 4, the proof of the second inductive case (see exercise I.4) never uses the hypothesis that the left part of the application is a value, therefore it remains valid if the reduction rule (app-r) is replaced by (app-r'). We therefore have the following equivalences:

\[
\frac{a \rightarrow^* v}{a \Rightarrow v} \quad \text{if and only if} \quad \frac{a \Rightarrow v}{a \rightarrow^* v} \quad \text{with the non-deterministic evaluation strategy.}
\]

Question 3: in light of question 2, we must look for a term that does not evaluate to a value, but instead diverges or causes an error. An example is \( a = (1 2) \omega \), where \( \omega \) is a term that diverges. With left-to-right reductions, \( a \) cannot reduce and is not a value, therefore its evaluation terminates immediately on an error. With non-deterministic reductions, we can choose to reduce infinitely often in \( \omega \), the argument part of the top-level evaluation, therefore observing divergence.
Part II: Abstract machines

Exercise II.1

\[
\begin{align*}
N(n) &= \text{ACCESS}(n) ; \text{APPLY} \\
N(\lambda.a) &= \text{GRAB} ; N(a) \\
N(a b) &= \text{CLOSURE}(N(b)) ; N(a)
\end{align*}
\]

We represent function arguments and values of variables by zero-argument closures, i.e. thunks. The \text{ACCESS} instruction of Krivine’s machine is simulated in the ZAM by an \text{ACCESS} (which fetches the thunk associated with the variable) followed by an \text{APPLY} (which jumps to this thunk, forcing its evaluation). The \text{GRAB} ZAM instruction behaves like the \text{GRAB} of Krivine’s machine if we never push a mark on the stack, which is the case in the compilation scheme above. Finally, the \text{PUSH} instruction of Krivine’s machine and the \text{CLOSURE} instruction of the ZAM behave identically.

Exercise II.2

Quite simply:

\[
\begin{align*}
C(n, k) &= \text{ACCESS}(n) ; k \\
C(\lambda.a, k) &= \text{CLOSURE}(T(a)) ; k \\
C(\text{let } a \text{ in } b, k) &= C(a, \text{GRAB}; C(b, \text{ENDLET}; k)) \\
C(a_1 \ldots a_n, k) &= \text{PUSHRETADDR}(k); C(a_n, \ldots ; C(a_1, C(a, \text{APPLY})))
\end{align*}
\]

The \(T\) schema is adjusted accordingly:

\[
\begin{align*}
T(\lambda.a) &= \text{GRAB}; T(a) \\
T(\text{let } a \text{ in } b) &= C(a, \text{GRAB}; T(b)) \\
T(a_1 \ldots a_n) &= C(a_n, \ldots ; C(a_1, T(a))) \\
T(a) &= C(a, \text{RETURN}) \quad \text{(otherwise)}
\end{align*}
\]

Exercise II.3

At the level of the instruction set, we can add a \text{COND}(c_1, c_2) instruction that tests the boolean value at the top of the stack and continues execution with one of two possible instruction sequences, \(c_1\) if the boolean is \text{true}, \(c_2\) otherwise. The transitions for this new instruction can be:

<table>
<thead>
<tr>
<th>Machine state before Code</th>
<th>Code</th>
<th>Env</th>
<th>Stack</th>
<th>Machine state after Code</th>
<th>Env</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{COND}(c_1, c_2); c</td>
<td>e</td>
<td>\text{true.s}</td>
<td>c_1</td>
<td>e</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>\text{COND}(c_1, c_2); c</td>
<td>e</td>
<td>\text{false.s}</td>
<td>c_2</td>
<td>e</td>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>

In the compilation scheme, the translation of \text{if/then/else} in tail-call position is straightforward:

\[
T(\text{if } a \text{ then } a_1 \text{ else } a_2) = C(a, \text{COND}(T(a_1), T(a_2)))
\]

An \text{if/then/else} in non-tail-call position is more delicate. The naive approach just duplicates the continuation code \(k\) in both arms of the conditional:

\[
C(\text{if } a \text{ then } a_1 \text{ else } a_2, k) = C(a, \text{COND}(C(a_1, k), C(a_2, k)))
\]
However, this can cause code size explosion if many conditionals are nested. Another approach uses `PUSHRETTADDR` and `RETURN` to share the continuation code \( k \) between both branches:

\[
C(\text{if } a \text{ then } a_1 \text{ else } a_2, k) = \text{PUSHRETTADDR}(k); C(a, \text{COND}(C(a_1, \text{RETURN}), C(a_2, \text{RETURN})))
\]

Yet another solution modifies the dynamic semantics (the transition rule) for `COND`, so that the code \( c \) that follows the `COND` is not discarded, but magically appended to whatever arm is taken:

<table>
<thead>
<tr>
<th>Machine state before</th>
<th>Machine state after</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Env</td>
</tr>
<tr>
<td><code>COND(c_1, c_2); c</code></td>
<td><code>true.s</code></td>
</tr>
<tr>
<td><code>COND(c_1, c_2); c</code></td>
<td><code>false.s</code></td>
</tr>
</tbody>
</table>

In this case, compilation without code duplication is straightforward:

\[
C(\text{if } a \text{ then } a_1 \text{ else } a_2, k) = C(a, \text{COND}(C(a_1, \varepsilon), C(a_2, \varepsilon)); k)
\]

However, it looks like the machine is generating new code sequences on the fly during execution, which is not very realistic. To address this issue, “real” abstract machines (like Caml’s or Java’s) introduce conditional and unconditional branch instructions that skip over a given number of instructions.

**Exercise II.4** Since the machine state decompiles to \( a \), the machine state is of the form

\[
\begin{align*}
\text{code} & = C(a') \\
\text{env} & = C(e') \\
\text{stack} & = C(a_1[e_1] \ldots a_n[e_n])
\end{align*}
\]

and \( a = a'[e'] a_1[e_1] \ldots a_n[e_n] \).

Since the machine is stopped (cannot make a transition), we are in one of the two following cases:

1. A `GRAB` instruction on an empty stack, meaning that \( n = 0 \) and \( C(a') = \text{GRAB}; c \) for some code \( c \). By examination of the compilation scheme, it follows that \( a' = \lambda a'' \). Therefore, \( a = (\lambda a'')[e'] \) is a value.

2. An `ACCESS(m)` instruction where \( C(e')(m) \) is undefined. By examination of the compilation scheme, it follows that \( a' = m \). Therefore, \( a = m[e'] \ a_1[e_1] \ldots a_n[e_n] \) is not a value and cannot reduce, since \( m[e'] \) cannot reduce (\( e'(m) \) is undefined).

**Exercise II.5** We write \( D(c, S) = a \) to mean that the symbolic machine, started in code \( c \) and symbolic stack \( S \), stops on the configuration \((\varepsilon, a, \varepsilon)\). By definition of the transitions of the symbolic machine, this partial function \( D \) satisfies the following equations:

\[
\begin{align*}
D(\varepsilon, a, \varepsilon) & = a \\
D(\text{CONST}(N).c, S) & = D(c, N.S) \\
D(\text{ADD}.c, b.a.S) & = D(c, (a + b).S) \\
D(\text{SUB}.c, b.a.S) & = D(c, (a - b).S)
\end{align*}
\]
By definition of decompilation, the concrete machine state \((c, s)\) decompiles to \(a\) iff \(D(c, s) = a\).

We start by the following technical lemma that shows the compatibility between symbolic execution and reduction of one expression contained in the symbolic stack.

**Lemma 1 (Compatibility)** Let \(s\) be a stack of integer values, \(a\) an expression and \(S\) a stack of expressions. Assume that \(D(c, S.a.s) = r\) and that \(a \rightarrow a'\). Then, there exists \(r'\) such that \(D(c, S.a'.s) = r'\) and \(r \rightarrow r'\).

**Proof:** By induction on \(D\). Note that \(c\) is \(D\). By definition of decompilation, the concrete machine state \((c, s)\) decompiles to \(a\) iff \(D(c, s) = a\). The result follows by induction hypothesis.

The compatibility lemma therefore shows the existence of \(a\) such that \(D(c, a.s) = a\) and \(D(c, s) = a\).

**Lemma 2 (Simulation)** If the HP calculator performs a transition from \((c, s)\) to \((c', s')\), and \(D(c, s) = a\), there exists \(a'\) such that \(a \rightarrow a'\) and \(D(c', s') = a'\).

**Proof:** By case analysis on the transition.

**Case** \(\text{CONST}\) transition: \((\text{CONST}(N); c, s) \rightarrow (c, N.s)\). We have \(D(\text{CONST}(N); c, s) = D(c, N.s)\) since the symbolic machine can perform the same transition. Therefore by definition of decompilation, the two states decompile to the same term. The result follows by taking \(a' = a\).

**Case** \(\text{ADD}\) transition: \((\text{ADD}; c, n_2.n_1.s) \rightarrow (c, n.s)\) where the integer \(n\) is the sum of \(n_1\) and \(n_2\). We have \(a = D(\text{ADD}; c, n_2.n_1.s) = D(c, b.s)\) where \(b\) is the expression \(n_1 + n_2\). Since \(b \rightarrow n\), the compatibility lemma therefore shows the existence of \(a'\) such that \(a \rightarrow a'\) and \(D(c, n.s) = a'\). This is the desired result.

**Case** \(\text{SUB}\) transition: similar to the previous case.

**Lemma 3 (Initial state)** The state \((C(a), \varepsilon)\) decompiles to \(a\).

**Proof:** We show by induction on \(a\) that the symbolic machine can perform transitions from \((C(a), k, S)\) to \((k, a.S)\) for all codes \(k\) and symbolic stack \(S\). (The proof is similar to that of theorem 10 in lecture II.) The result follows by taking \(k = \varepsilon\) and \(S = \varepsilon\).

**Lemma 4 (Final state)** If the machine stops on a state \((c, s)\) that decompiles to the expression \(a\), then \((c, s)\) is a final state \((\varepsilon, n.\varepsilon)\) and \(a = n\).

**Proof:** By case analysis on the code \(c\). If \(c\) is empty, by definition of decompilation we must have \(s = n.\varepsilon\) and \(a = n\) for some integer \(n\). If \(c\) starts with a \(\text{CONST}(N)\) instruction, the machine can perform a \(\text{CONST}\) transition and therefore is not stopped. If \(c\) starts with an \(\text{ADD}\) or \(\text{SUB}\) instruction, the stack \(s\) must contain at least two elements, otherwise the symbolic machine would get stuck and the decompilation of \((c, s)\) would be undefined. Therefore, the concrete machine can perform an \(\text{ADD}\) or \(\text{SUB}\) transition and is not stopped.
Exercise II.6  We show that for all $n$ and $a$, if $a \Rightarrow \infty$, there exists a reduction sequence of length $\geq n$ starting from $a$. The proof is by induction over $n$ and sub-induction over $a$. By hypothesis $a \Rightarrow \infty$, there are three cases to consider:

**Case** $a = b \ c$ and $b \Rightarrow \infty$. By induction hypothesis applied to $n$ and $b$, we have a reduction sequence $b \rightarrow b'$ of length $\geq n$. Therefore, $a = b \ c \Rightarrow b' \ c$ is a reduction sequence of length $\geq n$.

**Case** $a = b \ c$ and $b \Rightarrow v$ and $c \Rightarrow \infty$. By theorem 3 of lecture I, $b \rightarrow v$. By induction hypothesis applied to $n$ and $c$, we have a reduction sequence $c \rightarrow c'$ of length $\geq n$. Therefore, $a = b \ c \rightarrow v \ c'$ is a reduction sequence of length $\geq n$.

**Case** $a = b \ c$ and $b \Rightarrow \lambda x.d$ and $c \Rightarrow v$ and $d[x \leftarrow v] \Rightarrow \infty$. By theorem 3 of lecture I, $a \Rightarrow \lambda x.d$ and $b \rightarrow v$. By induction hypothesis applied to $n - 1$ and $d[x \leftarrow v]$, we have a reduction sequence $d[x \leftarrow v] \rightarrow e$ of length $\geq n - 1$. Therefore,

$$a = b \ c \Rightarrow (\lambda x.d) c \Rightarrow (\lambda x.d) v \rightarrow d[x \leftarrow v] \Rightarrow e$$

is a reduction sequence of length $\geq 1 + (n - 1) = n$.

Exercise II.7 For question (1), we show that $\forall a, \mathcal{E}_n(a) \leq \mathcal{E}_{n+1}(a)$ by induction over $n$. The base case $n = 0$ is obvious since $\mathcal{E}_0(a) = \perp$. For the inductive case, we assume the result for $n$ and consider $\mathcal{E}_{n+1}(a)$ by case over $a$. The non-trivial case is $a = b \ c$. We know (induction hypothesis) that $\mathcal{E}_n(b) \leq \mathcal{E}_{n+1}(b)$ and $\mathcal{E}_n(c) \leq \mathcal{E}_{n+1}(c)$.

If $\mathcal{E}_n(b) = \perp$ or $\mathcal{E}_n(c) = \perp$, then $\mathcal{E}_{n+1}(a) = \perp$ and the result is obvious.

Otherwise, $\mathcal{E}_{n+1}(b) = \mathcal{E}_n(b)$ and $\mathcal{E}_{n+1}(c) = \mathcal{E}_n(c)$, from which it follows that either $\mathcal{E}_{n+2}(a) = \text{err} = \mathcal{E}_{n+1}(a)$, or $\mathcal{E}_{n+2}(a) = \mathcal{E}_{n+1}(d[x \leftarrow v'])$ and $\mathcal{E}_{n+1}(a) = \mathcal{E}_n(d[x \leftarrow v'])$ for the same $d$ and $v'$, and the result follows by induction hypothesis.

We then conclude that $\mathcal{E}_n(a) \leq \mathcal{E}_m(a)$ if $n \leq m$ by induction on the difference $m - n$ and transitivity of $\leq$.

Consider now the sequence $(\mathcal{E}_n(a))_{n \in \mathbb{N}}$ for a fixed $a$. Either $\forall n, \mathcal{E}_n(a) = \perp$, or $\exists n, \mathcal{E}_n(a) \neq \perp$. In the first case, the sequence is constant and equal to $\perp$, hence $\lim_{n \rightarrow \infty} \mathcal{E}_n(a) = \perp$. In the second case, for all $m \geq n$, $\mathcal{E}_m(a) \geq \mathcal{E}_n(a) \neq \perp$, that is, $\mathcal{E}_n(a) = \mathcal{E}_m(a)$. The sequence is therefore constant starting from rank $n$, hence $\lim_{n \rightarrow \infty} \mathcal{E}_m(a)$ is defined and equal to $\mathcal{E}_n(a)$.

This limit corresponds to the behavior of the eval Caml function, in the following sense: if the limit is a value $v$, eval $a$ terminates and returns $v$; if the limit is err, eval $a$ terminates on an uncaught exception Error; and if the limit is $\perp$, eval $a$ loops.

For question (2), we show that $a \Rightarrow v$ implies $\exists n, \mathcal{E}_n(a) = v$ by induction on the derivation of $a \Rightarrow v$. The cases $a = N$ and $a = \lambda x.b$ are trivial: take $n = 1$. For the case $a = b \ c$, the induction hypothesis gives us integers $p, q, r$ such that

$$\mathcal{E}_p(b) = \lambda x.d \quad \mathcal{E}_q(c) = v' \quad \mathcal{E}_r(d[x \leftarrow v']) = v$$

Taking $n = 1 + \max(p, q, r)$ and using the monotonicity of $\mathcal{E}$, we have that $\mathcal{E}_n(b \ c) = v$.

Conversely, we show that $\mathcal{E}_n(a) = v$ implies $a \Rightarrow v$ by induction over $n$ and case analysis over $a$. Again, the cases $a = N$ and $a = \lambda x.b$ are trivial: we must have $v = a$. For the case $a = b \ c$, the fact that $\mathcal{E}_{n+1}(a) = v$ (and not err neither $\perp$) implies that

$$\mathcal{E}_n(b) = \lambda x.d \quad \mathcal{E}_n(c) = v' \quad \mathcal{E}_n(d[x \leftarrow v']) = v$$
The result follows by induction hypothesis applied to these three computations, and an application of the (app) rule.

For question (3), we show $\forall a, \ a \Rightarrow \infty$ implies $E_n(a) = \bot$ by induction over $n$. The case $n = 0$ is trivial. Assuming this property for $n$, we consider the evaluation rule that concludes $a \Rightarrow \infty$. For instance, if $a = b \ c$ and $b \Rightarrow \infty$, by induction hypothesis, $E_n(b) = \bot$, from which it follows that $E_{n+1}(a) = \bot$. The proof is similar for the other two rules.

For the converse implication ($\forall n, \ E_n(a) = \bot$ implies $a \Rightarrow \infty$), see the paper *Coinductive big-step semantics*. 
Part III: Program transformations

Exercise III.1 The translation rule for $\lambda$-abstraction needs to be changed:

$$\left[ \lambda x.a \right] = \text{tuple}(\lambda c, x. \text{let } x_1 = \text{field}_1(c) \text{ in } \ldots \text{let } x_n = \text{field}_n(c) \text{ in } \left[ a \right], x_1, \ldots, x_n)$$

so that the variables $x_1, \ldots, x_n$ are not just the free variables of $\lambda x.a$, but all variables currently in scope. To do this, the translation scheme should take the list of such variables as an additional argument $V$:

$$\left[ x \right]_V = x$$

$$\left[ \lambda x.a \right]_V = \text{tuple}(\lambda c, x. \text{let } x_1 = \text{field}_1(c) \text{ in } \ldots \text{let } x_n = \text{field}_n(c) \text{ in } \left[ a \right]_x V, x_1, \ldots, x_n)$$

where $V = x_1 \ldots x_n$

$$\left[ a \ b \right]_V = \text{let } c = \left[ a \right]_V \text{ in field}_0(c)(c, \left[ b \right]_V)$$

$$\left[ \text{let } x = a \text{ in } b \right]_V = \text{let } x = \left[ a \right]_V \text{ in } \left[ b \right]_x V$$

Exercise III.2 For a two-argument function $\lambda x.\lambda x'.a$, the two-argument method $\text{apply2}$ will be defined as $\text{return } \left[ a \right]$. The one-argument method $\text{apply}$ will build an intermediate closure (corresponding to $\lambda x'.a$) which, when applied, will call back to $\text{apply2}$.

Symmetrically, for a one-argument function $\lambda x.a$, we define $\text{apply}$ as $\text{return } \left[ a \right]$ and $\text{apply2}$ as calling $\text{apply}$ on the first argument, then applying again the result to the second argument.

We encapsulate this construction in the following generic classes, from which we will inherit later:

```
abstract class Closure {
    abstract Object apply(Object arg);
    Object apply2(Object arg1, Object arg2) {
        return ((Closure)(apply(arg1))).apply(arg2);
    }
}
abstract class Closure2 extends Closure {
    Object apply(Object arg) {
        return new PartialApplication(this, arg);
    }
    abstract Object apply2(Object arg1, Object arg2);
}
class PartialApplication extends Closure {
    Closure2 fn; Object arg1;
```
PartialApplication(Closure2 fn, Object arg1) {
    this.fn = fn; this.arg1 = arg1;
}
Object apply(Object arg2) {
    return fn.apply2(arg1, arg2);
}

Now, the class generated for a two-argument function $\lambda x.\lambda y.a$ of free variables $x_1, \ldots, x_n$ is

class $C_{\lambda x.\lambda y.a}$ extends Closure2 {
    Object $x_1$; ...; Object $x_n$;
    $C_{\lambda x.\lambda y.a}$ (Object $x_1$, ..., Object $x_n$) {
        this.$x_1$ = $x_1$; ...; this.$x_n$ = $x_n$;
    }
    Object apply2(Object $x$, Object $y$) { return $a$; }
}

The class generated for a one-argument function $\lambda x.a$ of free variables $x_1, \ldots, x_n$ is

class $C_{\lambda x.a}$ extends Closure {
    Object $x_1$; ...; Object $x_n$;
    $C_{\lambda x.a}$ (Object $x_1$, ..., Object $x_n$) {
        this.$x_1$ = $x_1$; ...; this.$x_n$ = $x_n$;
    }
    Object apply(Object $x$) { return $a$; }
}

Finally, the translation of expressions receives one additional case for curried applications to two arguments:

$[a \ b \ c] = [a].apply2([b], [c])$

**Exercise III.3** Quite simply,

$[\text{match} \ a \ \text{with} \ x \rightarrow a \ | \ \text{exception} \ y \rightarrow c] = \text{match} \ [a] \ \text{with} \ V(x) \rightarrow [b] \ | \ E(y) \rightarrow [c]$}

Note that $\text{try} \ a \ \text{with} \ x \rightarrow b$ can then be viewed as syntactic sugar for

$\text{match} \ a \ \text{with} \ y \rightarrow y \ | \ \text{exception} \ x \rightarrow b$