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 \( b c \). 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 \( a \to b \), 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:

\[
(\text{let } x = v \text{ in } a) \Rightarrow (\lambda x.a) v \to a[x \leftarrow v]
\]

by \( \beta_v \)-reduction. For **if/then/else**:

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

\[
\begin{align*}
\text{fst}(v_1, v_2) & \approx (\text{match Pair}(v_1, v_2) \text{ with } \text{Pair}(x_1, x_2) \to x_1) \to x_1[x_1 \leftarrow v_1, x_2 \leftarrow v_2] = v_1 \\
\text{snd}(v_1, v_2) & \approx (\text{match Pair}(v_1, v_2) \text{ with } \text{Pair}(x_1, x_2) \to x_2) \to x_2[x_1 \leftarrow v_1, x_2 \leftarrow v_2] = v_2
\end{align*}
\]

again by \texttt{match} reductions.

**Exercise I.3** Assume \( 1 \ 2 \Rightarrow v \) 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:

\[
\begin{align*}
\lambda x. x & \Rightarrow \lambda x. x \\
\lambda x. x & \Rightarrow \lambda x. x \\
(x \ x)[x \leftarrow \lambda x. x] & = a' \Rightarrow v
\end{align*}
\]

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 \( a = (\lambda x. c) \ v' \to c[x \leftarrow v'] = b \). We can build the following derivation of \( a \Rightarrow v \) from that of \( b \Rightarrow v \):

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

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 \( a' \to b' \). The evaluation derivation for \( b \Rightarrow v \) is of the following form:

\[
\begin{align*}
 b' & \Rightarrow \lambda x. d \\
 c & \Rightarrow v' \\
 d[x \leftarrow v'] & \Rightarrow v
\end{align*}
\]

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

\[
\begin{align*}
a' \Rightarrow \lambda x. d \\
 c & \Rightarrow v' \\
 d[x \leftarrow v'] & \Rightarrow v
\end{align*}
\]

\( a' \ c \Rightarrow v \)

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:

\[
\begin{align*}
v' \Rightarrow \lambda x.c & \quad b' \Rightarrow v'' \quad c[x \leftarrow v''] \Rightarrow v \\
v' b' \Rightarrow v
\end{align*}
\]

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:

\[
\begin{align*}
v' \Rightarrow \lambda x.c & \quad a' \Rightarrow v'' \quad c[x \leftarrow v''] \Rightarrow v \\
v' a' \Rightarrow v
\end{align*}
\]

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 $E[a]$.

```ml
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 $a_2$.

```ml
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:

\[
\begin{align*}
  a \xrightarrow{s} v & \text{ with the left-to-right evaluation strategy} \\
  \text{if and only if} & \quad a \Rightarrow v \\
  \text{if and only if} & \quad a \xrightarrow{a} v \text{ with the non-deterministic evaluation strategy.}
\end{align*}
\]

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.