Part I: Interpreters and operational semantics

Exercise I.1 (**): Prove theorem 2 (the unique decomposition theorem). Hint: use the fact that a value cannot reduce.

Exercise I.2 (*): Check that the reduction rules for derived forms (let, if/then/else, fst, snd) are valid.

Exercise I.3 (*): Consider \( a = 1 \ 2 \). Does there exist a value \( v \) such that \( a \Rightarrow v \)? Same question for \( a' = (\lambda x. x \ x) \ (\lambda x. x \ x) \). Do you see a difference between these two examples?

Exercise I.4 (**): As claimed in the proof of theorem 4, show that if \( a \rightarrow b \) and \( b \Rightarrow v \), then \( a \Rightarrow v \). Hint: proceed by induction on the derivation of \( a \rightarrow b \) in SOS and by case analysis on the last rule used to derive \( b \Rightarrow v \).

Programming exercise I.5 (**): Implement a naive interpreter that follows exactly the Wright-Felleisen presentation of CBV semantics. The core function to implement is decomp, taking a term \( a \) as argument and returning a pair of a reduction context \( E \) and a subterm \( a' \) such that \( a = E[a'] \) and \( a \) reduces iff \( a' \) reduces at head. How will you represent contexts? Compare the efficiency of this interpreter with that of the naive interpreter based on the SOS presentation shown in the course.

Exercise I.6 (*): We relax the (app-r) reduction rule as follows:

\[
\frac{b \rightarrow b'}{a \ b \rightarrow a \ b'} \quad \text{(app-r')}
\]

That is, we allow reductions to take place on the right of an application, even if the left part of the application is not yet reduced to a value.

1. Show by way of an example that the reduction relation is no longer deterministic: give three terms \( a, a_1, a_2 \) such that \( a \rightarrow a_1 \) and \( a \rightarrow a_2 \) and \( a_1 \neq a_2 \).
2. Check that theorems 3 and 4 ($a \Rightarrow v$ if and only if $a \rightarrow^* v$) still hold for this relaxed, non-deterministic reduction relation. Conclude that the left-to-right evaluation order makes no difference for terminating terms.

3. Give an example of a term that evaluates differently under the two reduction strategies (left-to-right and non-deterministic).
Part II: Abstract machines

Exercise II.1 (**): A Krivine machine is hidden in the ZAM. Can you find it? More precisely, define a compilation scheme $N$ from $\lambda$-terms to ZAM instructions that implements call-by-name evaluation of the source term.

Exercise II.2 (*): The $C$ compilation schema for the ZAM is not mathematically precise: in the application case, it is not 100% clear what the $k$ in $\text{PUSHRETD}\text{ADDR}(k)$ is, exactly. Reformulate the compilation schema more precisely as a 2-argument function $C(a, k)$, which should prepend to the code $k$ the instructions that evaluate the expression $a$ and deposit its value on the top of the stack.

Exercise II.3 (**/***): How would you extend the ZAM and its compilation scheme to handle if/then/else conditional expressions?

Exercise II.4 (**): Prove lemma 8 (the Final states lemma for Krivine’s machine).

Exercise II.5 (**/*): State and prove the analogues of the Simulation, Initial state and Final state lemmas (lemmas 6, 7, 8 in the case of Krivine’s machine) for the HP calculator. Use the notion of decompilation defined in the lecture notes and the following reduction semantics for arithmetic expressions:

\[
\begin{align*}
N_1 + N_2 &\rightarrow N \quad \text{(if } N = N_1 + N_2) \\
N_1 - N_2 &\rightarrow N \quad \text{(if } N = N_1 - N_2) \\
a &\rightarrow a' \\
a \cdot b &\rightarrow a' \cdot b \\
N \cdot a &\rightarrow N \cdot b' \\
\end{align*}
\]

Exercise II.6 (**): Complete the proof of theorem 14 (if $a \Rightarrow \infty$ then $a$ reduces infinitely).

Exercise II.7 (**): Consider the following function $E_n(a)$, defined by recursion over $n$, that evaluates a term $a$ up to recursion depth $n$:

\[
\begin{align*}
E_0(a) &= \bot \\
E_{n+1}(x) &= \text{err} \\
E_{n+1}(N) &= N \\
E_{n+1}(\lambda x.a) &= \lambda x.a \\
E_{n+1}(a b) &= \text{case } E_n(a) \text{ of } \bot \rightarrow \bot \mid \text{err} \rightarrow \text{err} \mid v \rightarrow \text{case } E_n(b) \text{ of } \bot \rightarrow \bot \mid \text{err} \rightarrow \text{err} \mid v' \rightarrow \text{case } v \text{ of } N \rightarrow \text{err} \mid \lambda x.c \rightarrow E_n(c[x \leftarrow v'])
\end{align*}
\]

The result of $E_n(a)$ is either a value $v$ (denoting termination), the constant $\text{err}$ (denoting an erroneous evaluation), or the constant $\bot$ (denoting an evaluation that cannot terminate at recursion depth $n$).

We define a partial order $\leq$ on evaluation results by $r \leq r$ and $\bot \leq r$ for all results $r$. Note that distinct values, as well as $\text{err}$ and a value, are not comparable by $\leq$. 

3
1. Show that $E_n(a)$ is increasing in $n$, that is $E_n(a) \leq E_{n+1}(a)$. Conclude that $\lim_{n \to \infty} E_n(a)$ exists for all terms $a$. What is the relationship between this limit and the behavior of the Caml function $\text{eval}$ defined in part I?

2. Show that $a \Rightarrow v$ if and only if $\exists n, E_n(a) = v$.

3. Show that $a \Rightarrow \infty$ if and only if $\forall n, E_n(a) = \bot$. (The “if” part requires a proof by coinduction; do not attempt it if you are not familiar with the coinduction principle.)
Part III: Program transformations

Exercise III.1 (*) How would you modify closure conversion so that it builds full closures rather than minimal closures?

Exercise III.2 (***） Consider again closure conversion targetting a class-based object-oriented language such as Java. (Slide 14.) How would you extend this transformation to efficiently handle curried applications to 2 arguments? Hint: each closure becomes an object with 2 methods, `apply` and `apply2`, performing applications to 1 and 2 arguments respectively.

Exercise III.3 (*) OCaml version 4.02 introduces support for the following construct

\[ \text{match } a \text{ with } x \to a \mid \text{exception } y \to c \]

that behaves not at all like \( \text{try (let } x = a \text{ in } b \text{ with } y \to c, \) but as follows:

\[
\begin{align*}
\text{(match } v \text{ with } x \to a \mid \text{exception } y \to c) & \xrightarrow{\xi} b[x \leftarrow v] \\
\text{(match (raise } v) \text{ with } x \to a \mid \text{exception } y \to c) & \xrightarrow{\xi} c[y \leftarrow v]
\end{align*}
\]

In other terms, the exception handler `with y -> c` catches exceptions arising during the evaluation of \( a \), but not those arising during evaluation of \( b \). Extend the exception-returning conversion to deal with this \text{match-and-try} construct.

Exercise III.4 (**) Give a natural semantics for references. Hint: the evaluation predicate has the form \( a/s \Rightarrow v/s' \) where \( s \) is the initial store at the beginning of evaluation and \( s' \) is the final store at the end of evaluation.

Exercise III.5 (**) What function is computed by the following expression?

\[
\begin{align*}
\text{let fact = ref } (\lambda n. 0) \text{ in} \\
\text{fact := } (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \ast (!!\text{fact})(n-1)) \\
\text{!!fact}
\end{align*}
\]

Define a translation scheme from a functional language with recursive functions \( \mu f. \lambda x. a \) to a functional language with only plain functions \( \lambda x. a \) and references. Hint: a fixpoint combinator would do the job, but please use references instead.

Programming exercise III.6 (***） Write an OCaml implementation of the polymorphic store operations from slide 48 of lecture III. The implementation must be statically type-safe (no Obj.magic).

Exercise III.7 (*) Define the CPS conversion of arithmetic operations \( a \ op \ b \), constructor applications \( C(a_1, \ldots, a_n) \) and pattern-matchings `match a with p_1 | \ldots | p_n`.

Exercise III.8 (***） Define a translation from exceptions to references + continuations. Hint: use an imperative stack containing continuations corresponding to the `with` part of active `try...with` constructs.
Exercise III.9 (*) Prove the first case of Theorem 5: if \( a \Rightarrow v \) and \( k \) is a value, then \([a] \xrightarrow{\downarrow} k \ [v]_v\).

Exercise III.10 (**) — the “double-barreled” CPS transformation  Define a CPS translation for a source language that includes mini-ML and exceptions (\texttt{raise}, \texttt{try...with}). Each source term \( a \) becomes a term \([a] = \lambda k_1 \lambda k_2 \ldots\) that expects not one, but two continuations. The continuation \( k_1 \) is to be called on the value of \( a \) if \( a \) terminates normally. The continuation \( k_2 \) is to be called on an exception value if \( a \) terminates early by raising this exception.

Programming exercise III.11 (*) Using Danvy’s et al.’s technique, derive a tail-recursive function equivalent to the familiar \texttt{map} function over lists:

```ocaml
let rec map f l =  
match l with 
[ | -> [] | hd :: tl -> f hd :: map f tl
```

Exercise III.12 (**) Consider the CPS transformation for call-by-name given in the lecture. What is its effect on types? In other words, if \( a \) is a closed term of type \( \tau \), what is the type of its call-by-name CPS translation \([a]\)?