Exercise 2.1: Write Mini-ML programs for multiplication, exponentiation, subtraction, and a function that returns a pair of (integer) quotient and remainder of two natural numbers.

Solution:

\[
\begin{align*}
\text{add} &= \text{fix } f. \text{lam } x. \text{lam } y. \text{case } x \text{ of } z \Rightarrow y | s \ x' \Rightarrow s \ (f \ x') \\
\text{sub} &= \text{fix } f. \text{lam } x. \text{lam } y. \\
&\quad \text{case } x \text{ of } z \Rightarrow z \\
&\quad | s \ x' \Rightarrow \text{case } y \text{ of } z \Rightarrow x | s \ y' \Rightarrow f \ x' \ y' \\
\text{mult} &= \text{fix } f. \text{lam } x. \text{lam } y. \\
&\quad \text{case } x \text{ of } z \Rightarrow z | s \ x' \Rightarrow \text{add } (f \ x') \ y \\
\text{expo} &= \text{fix } f. \text{lam } x. \text{lam } n. \\
&\quad \text{case } n \text{ of } z \Rightarrow (s \ z) | s \ n' \Rightarrow \text{mult } x \ (f \ x \ n') \\
\text{quot} &= \text{fix } f. \text{lam } x. \text{lam } y. \\
&\quad \text{case } \text{sub } x \ y \ of \\
&\quad \quad z \Rightarrow \text{case } \text{sub } y \ x \ of \ z \Rightarrow (s \ z, z) | s \ x' \Rightarrow (z, x) \\
&\quad | s \ w \Rightarrow \text{let } v = f \ (s \ w) \ y \ \text{in } (s \ (\text{fst } v), \text{snd } v)
\end{align*}
\]

Exercise 2.13: Specify a call-by-name operational semantics for our language where the constructors are lazy that is they should not evaluate their arguments.

Solution: We start by defining lazy values. If we discover an expression \( s \ e \) then we reached a value as we will only evaluate \( e \) when needed. Similarly, a pair \( \langle e_1, e_2 \rangle \) is a value.

\[
\begin{array}{c}
\text{z Lazy\_Val} \\
\text{lvalz}
\end{array}
\quad
\begin{array}{c}
\text{s e Lazy\_Val} \\
\text{lvals}
\end{array}
\quad
\begin{array}{c}
\text{lval\_lam} \\
\text{lam } x. e \ \text{Lazy\_Val}
\end{array}
\quad
\begin{array}{c}
\text{lval\_pair} \\
\text{\langle e_1, e_2 \rangle Lazy\_Val}
\end{array}
\]
We proceed by revising the operational semantics of Mini-ML.

We have:

\[
\begin{align*}
\text{letn} & : \quad \frac{}{x \vdash x} \\
\text{letv} & : \quad \frac{\text{let val } x = e_1 \text{ in } e_2 \vdash v}{v_1 \vdash v}
\end{align*}
\]

The \text{evl\_letn} rule does not change as it already is lazy, i.e. it does not evaluate the argument \(x\). In order to force the evaluation of an expression, we choose to include the \text{evl\_letv} rule.

\[
\begin{align*}
\text{letv} & : \quad \frac{\text{let name } u = e_1 \text{ in } e_2 \vdash v}{v_1 \vdash v}
\end{align*}
\]

The \text{evl\_fix} rule stays the same.

\[
\begin{align*}
\text{fix} & : \quad \frac{[\text{fix } e/x] \vdash v}{v \vdash v}
\end{align*}
\]

**Theorem 1** (Value Soundness). \(\text{If } D : e \vdash v \text{ then } E : v \text{ Lazy Val.}\)

**Proof.** The proof follows by induction over the structure of the deduction \(D : e \vdash v\). We will only show a few typical cases.

Case: \(D = \frac{}{x \vdash x}\). Then \(x\) Lazy Val by the rule \text{lval\_z}.

Case: \(D = \frac{\text{let val } x = e_1 \text{ in } e_2 \vdash v}{v_1 \vdash v}\). Then \(e\) Lazy Val by the rule \text{lval\_s}.

Case: \(D = \frac{\text{let name } u = e_1 \text{ in } e_2 \vdash v}{v_1 \vdash v}\). Then \(e\) Lazy Val by the rule \text{lval\_s}.
Then \( \text{lam} \ x.e \text{ Lazy Val} \) by the rule \( \text{vallam} \).

\[
\begin{array}{c}
\text{Case: } \mathcal{D} = \frac{D_1 \quad D_2}{e_1 \overset{l}{\mapsto} \text{lam } x.e' \quad [e_2/x]e' \overset{l}{\mapsto} v} \text{ evapp} \quad\quad e_1 e_2 \overset{l}{\mapsto} v
\end{array}
\]

The induction hypothesis on \( \mathcal{D}_2 \) yields a deduction \( \mathcal{E} :: v \text{ Lazy Val} \).

\[
\begin{array}{c}
\text{Case: } \mathcal{D} = \frac{D_1 \quad D_2}{e \overset{l}{\mapsto} \langle e_1, e_2 \rangle \quad e_1 \overset{l}{\mapsto} v} \text{ evfst} \quad \text{fst } e \overset{l}{\mapsto} v
\end{array}
\]

The induction hypothesis on \( \mathcal{D}_2 \) yields a deduction \( \mathcal{E} :: v \text{ Lazy Val} \).

Exercise 2.14 - Part 1: Prove that \( v \text{ Value} \) is derivable if and only if \( v \mapsto v \) is derivable. That is, values are exactly those expressions that evaluate to themselves.

Solution: Theorem 2. If \( D :: v \text{ Value} \) then \( E :: v \mapsto v \).

Proof. By induction over the structure of the deduction \( D :: v \text{ Value} \).

Case: \( D = \text{val } z \). Then \( z \mapsto z \) by the rule \( \text{evz} \).

Case: \( D = v \text{ Value} \)

The induction hypothesis on \( \mathcal{D}_1 \) yields a deduction \( \mathcal{E}_1 :: v \mapsto v \). Using the inference rule \( \text{evs} \) we conclude that \( s v \mapsto s v \).

Case: \( D = \text{lam } x.e \text{ Value} \).

Then \( \text{lam } x.e \mapsto \text{lam } x.e \) by the rule \( \text{evlam} \).

Case: \( D = v_1 \text{ Value} \quad v_2 \text{ Value} \)

\( \langle v_1, v_2 \rangle \mapsto \langle v_1, v_2 \rangle \) by induction hypothesis on \( \mathcal{D}_1 \)

\( v_1 \mapsto v_1 \) by induction hypothesis on \( \mathcal{D}_2 \)

\( v_2 \mapsto v_2 \) by rule \( \text{evpair} \)

Theorem 3. If \( \mathcal{E} :: v \mapsto v \) then \( D :: v \text{ Value} \).

Proof. Follows immediately from the value-soundness theorem Theorem 2.1 p 19 of the lecture notes. □
Exercise 2.14 - Part 2: Write a Mini-ML function \( \text{observe} : \text{nat} \rightarrow \text{nat} \) that, given a lazy value of type \( \text{nat} \), returns the corresponding eager value if it exists.

Solution:
There are two possible ways to observe the value of a lazy expression. The first solution uses the \texttt{let val} construct to force the evaluation of a lazy expression.

\[
\text{observe} = \text{fix } f. \lambda x. \text{case } x \Rightarrow z \mid s \ x' \Rightarrow \text{let } v = f \ x' \text{ in } s \ v.
\]

The second solution is based on continuations. The basic idea is the following: any function \( f : t \rightarrow s \) can be rewritten into a function \( f' \) of type \( t \rightarrow (s \rightarrow b) \rightarrow b \). In contrast to \( f \), the function \( f' \) takes an extra function as an argument, called a \textit{continuation}, which accumulates the results. To use the function \( f' \) to compute the original function \( f \), we give it the \textit{initial continuation} which is often the identity function as an argument. Applying this idea to define \( \text{observe} \) we first define a function \( \text{observe}' \) which takes \( x \) and a continuation \( k \) as an argument. In the base case, we just call the continuation \( k \) applied to \( z \). In the recursive case, we apply the successor function to the result of the continuation. Note that the successor function will be only applied to values once it is executed.

\[
\text{observe}' = \text{fix } f. \lambda x. \lambda k. \text{case } x \Rightarrow k \ z \mid s \ x' \Rightarrow f \ x' (\lambda v. k (s v)).
\]

\[
\text{observe} = \lambda x. \text{observe}' x (\lambda v. v).
\]

Let us consider the following evaluation: \( \text{observe}' \ s \ ((\lambda x. x) z) \ k. \)

\[
\begin{align*}
\text{first rec. call: } & \text{observe}' \ (s ((\lambda x. x) z)) \ (\lambda v_1. k (s v_1)) \\
\text{sec. rec. call: } & \text{observe}' ((\lambda x. x) z) \ (\lambda v_2. (\lambda v_1. k (s v_1)) (s v_2))
\end{align*}
\]

Now \( \text{observe}' \) will evaluate \( ((\lambda x. x) z) \) to \( z \) and reach the base case where we need to compute \( (\lambda v_2. (\lambda v_1. k (s v_1))(s v_2)) \ z. \)