Exercise 1:
Consider the following PCF expression $u$
\begin{verbatim}
letrec f (x) = 3 in
    letrec g (x) = g (x) in
    f (g 0)
\end{verbatim}

1. This is not a valid expression because the type annotations are missing. Add them.
2. Calculate the denotational semantics of $u$.

Solution:
1. \begin{verbatim}
letrec f _ \rightarrow \text{int} (x_{\text{int}}) = \hat{3} \text{ in }
    letrec g _ \rightarrow \text{int} (x_{\text{int}}) = g _ \rightarrow \text{int} (x_{\text{int}}) \text{ in }
    f _ \rightarrow \text{int} (g _ \rightarrow \text{int} \ 0)
\end{verbatim}
2. Using the rule
\begin{align*}
\llbracket \text{letrec } f_{\sigma \rightarrow \tau} (x_\sigma) = u \text{ in } v \rrbracket_\rho &= \llbracket v \rrbracket (\rho[\sigma \rightarrow \tau \mapsto \text{lfp}(F^\rho_{\sigma \rightarrow \tau, x_\sigma, u})]) \\
\text{where } F^\rho_{\sigma \rightarrow \tau, x_\sigma, u} (\varphi) &= (V \in \llbracket \sigma \rrbracket \mapsto \llbracket u \rrbracket (\rho[\sigma \rightarrow \tau \mapsto \varphi, x_\sigma \mapsto V])).
\end{align*}
We obtain that $\llbracket u \rrbracket_\rho = 3$ for all environments $\rho$.

Exercise 2:
For each OCaml expression below, give the type of the expression, if it exists. Justify.
1. \begin{verbatim}
let f x = x in (f 3, f "trois")
\end{verbatim}
2. \begin{verbatim}
(fun f -> (f 3, f "trois")) (fun x -> x)
\end{verbatim}
3. \begin{verbatim}
let f x = x in let g = ref f in (!g 3, !g "trois")
\end{verbatim}

Solution:
1. The type is \text{int} * \text{string} .
2. This does not type, because the generalization only applies to \text{let} thus the function \text{fun} x -> x is not generalized.
3. We trigger the "value restriction". It is important because otherwise we can do things like
\begin{verbatim}
let r = (fun x -> ref x) [];;
\end{verbatim}
\begin{verbatim}
r := [ 1 ];;
\end{verbatim}
\begin{verbatim}
let cond = (!r = [ "foo" ]);;
\end{verbatim}
Exercise 3:
We consider the following language

\[ M := x \mid \lambda x : \tau. M \mid MN \mid \text{let } x : \tau = M \text{ in } N \mid \text{ff} \mid \text{tt} \mid \text{if } M \text{ then } N \text{ else } P \]

1. Propose an adapted typing system.
2. Give a derivation of \( \Gamma \vdash (\lambda x : \text{bool}. \text{if } x \text{ then } \text{ff} \text{ else } x) \text{tt} : \text{bool} \)
3. Which element of the programming language syntax is crucial to guarantee typing determinism? Explain with an example.
4. Show that the \text{let} is encoded using the other constructs in a well-typed way.
5. Propose small-step semantics for this language.
6. Show that there is a theorem of subject reduction, that is, small-step semantics preserves typing.
7. We add to the syntax the following two constructions

\[ \text{try } M \text{ with } N \mid \text{abort} \]

Propose an extension of the typing system.
8. Propose an extension of the small step semantics.

Solution:

1. \[ \begin{array}{c}
\Gamma \vdash \text{tt} : \text{bool} \\
\Gamma \vdash \text{ff} : \text{bool} \\
\Gamma, x : \tau \vdash x : \tau \\
\hline \\
\Gamma \vdash M : \sigma \rightarrow \tau \\
\Gamma \vdash N : \sigma \\
\Gamma \vdash MN : \tau \\
\Gamma, x : \sigma \vdash M : \tau \\
\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau \\
\hline \\
\Gamma \vdash P : \text{bool} \\
\Gamma \vdash M : \tau \\
\Gamma \vdash N : \tau \\
\hline \\
\Gamma \vdash \text{if } P \text{ then } M \text{ else } N : \tau \\
\Gamma \vdash \text{let } x : \sigma = M \text{ in } N : \tau \\
\end{array} \]

2. It can be shown using the rules above.

3. The fact that the types are in the syntax. That is, type inference is not deterministic, the type erasure loses information. For example, \( \lambda x. x \).

4. We write \( \text{let } x = M \text{ in } N \triangleq (\lambda x. N)M \).

5. \[ (\lambda x : \tau. M)N \rightarrow M[N/x] \]
\[ \text{let } x : \tau = M \text{ in } N \rightarrow N[M/x] \]
\[ \text{if tt then } M \text{ else } N \rightarrow M \]
\[ \text{if ff then } M \text{ else } N \rightarrow N \]

and \( M \rightarrow N \) implies \( C[M] \rightarrow C[N] \) for all contexts \( C \).

6. This is done by induction on the typing derivation.

7. We give to \text{Abort} the type \( \text{exn} \) and

\[ \begin{array}{c}
\Gamma \vdash M : \tau \\
\Gamma \vdash N : \tau \\
\hline \\
\Gamma \vdash \text{try } M \text{ with } N : \tau \\
\end{array} \]

8. We add the rules

\[ \begin{array}{c}
\text{try abort with } M \rightarrow M \\
\text{try } V \text{ with } M \rightarrow V \\
\end{array} \]

and the following context

\[ \text{try } C \text{ with } M \]
Exercise 4:
We add exceptional constructors that we denote as \( C_1, \ldots, C_n \). These are for example exceptions like `KeyboardInterrupt`. For each \( C_i \), we consider a type \( \tau_i \) of fixed argument and we add the rules of deductions

\[
C_i : \tau_i \rightarrow \text{exn}
\]

1. Adapt the syntax. What are the values? What are the contexts?
2. Adapt the small-step semantics.
3. Use it to reduce the next term assuming that \( M \rightarrow^* V \).
   \[
   \text{try } (\lambda x.\lambda y.y)(\text{abort } M) \text{ with } C_i(x) \mapsto x
   \]
4. OCaml language prohibits building exceptions possessing a polymorphic type. Explain.

Solution:
1. Values are closed terms of the form \( \lambda x.f \), \( \text{tt} \) ou \( \text{ff} \). Exceptions are not considered as values since they will be executed in a context. The contexts are:
   \[
   C := C | \text{try } M \text{ with } C | \text{abort } C | VC | CM | \text{if } C \text{ then } M \text{ else } M
   \]
2. Small-step semantics adapts as follows
   \[
   \text{try } (\text{abort } (C_i V)) \text{ with } C_i(x) \mapsto N \rightarrow N[x/M]
   \]
   \[
   F[\text{abort } V] \rightarrow \text{abort } V
   \]
   \[
   \text{try } V \text{ with } M \rightarrow V
   \]
3. Trivially reduces to \( V \).
4. It is sufficient to imagine the type \( \tau_i \triangleq \forall \alpha.\alpha \).
   So we lose subject reduction as shown by the following term:
   \[
   \text{try abort } (C_i(\text{tt}));1 \text{ with } C_i(x) \mapsto x + 1
   \]