CS 225: Midterm Practice—Short

1. Consider the `Value` Haskell datatype:

```
data Value = IntV Integer
           | BoolV Bool
```

which corresponds to the mathematical set value defined as follows:

$$
\begin{array}{rcll}
i \in & \mathbb{Z} & & \textit{(integers)} \\
b \in & \mathbb{B} & & \textit{(booleans)} \\
v \in & \text{value} & ::= i \mid b &
\end{array}
$$

Write a Haskell function `convert` that converts a list of `Value` values into a list of `Integer` values, where integers are mapped to themselves and booleans are mapped to either 1 for true, or 0 for false. Define this function by recursion and pattern matching on the structure of Haskell lists, and by pattern matching on the two possible cases of `Value`. The mathematical specification for this function is:

$$
\begin{array}{l}
\text{convert}(v_1, \ldots, v_n) \triangleq i_1, \ldots, i_n \\
\quad \textit{where} \quad \begin{array}{lll}
i_i = v_i & \textit{if} & v_i \in \mathbb{Z} \\
i_i = 1 & \textit{if} & v_i = \texttt{true} \\
i_i = 0 & \textit{if} & v_i = \texttt{false}
\end{array}
\end{array}
$$

```
convert :: [Value] -> [Integer]
convert vs =
```

2. Consider a core programming language with integers, addition, variables, and let-expressions:

$$
\begin{array}{rcll}
i \in & \mathbb{Z} & & \textit{(integers)} \\
x \in & \text{variable} & & \textit{(program variables)} \\
e \in & \text{expr} & ::= i \mid e + e \mid x \mid \texttt{LET}\ x = e\ \texttt{IN}\ e &
\end{array}
$$

For each of the following expressions, (1) draw a square around every *binder*, (2) draw an arrow from every *bound* variable to its *binder*, and (3) draw a circle around every *free variable*.

(a)

```
LET x = 1 IN (y + x)
```

(b)

```
(LET x = 1 IN y) + x
```

(c)

```
LET x = x IN (LET x = x IN x)
```

3. Consider the same programming language as the previous question, and a *substitution* based semantics for the interpretation of let. Write out each substitution step for interpreting each of the following terms. An worked out example is given:

```
LET x = (1 + 2) IN             LET y = (1 + 2) IN
LET y = x IN           =       y + y                  =   (1 + 2) + (1 + 2)   =   6
y + y
```

(a)

```
LET x = 1 IN
LET y = 2 IN
x
```

(b)

```
LET x = 1 IN
LET x = 2 IN
x
```

(c)

```
LET x = (LET x = 1 IN x) IN
x + x
```

(d)

```
LET x = 1 IN
LET y = (LET x = 2 IN x) + x IN
y + x
```

4. Consider a core programming language with integers, addition, and first-class functions of *only a single argument*:

$$
\begin{array}{rrl}
i \in & \mathbb{Z} & \textit{(integers)} \\
x \in & \text{variable} & \textit{(program variables)} \\
e \in & \text{expr} ::= & i \mid e + e \mid \texttt{FUN}(x) \Rightarrow e \mid e(e) \\
v \in & \text{value} ::= & i \mid \langle \texttt{FUN}(x) \Rightarrow e, \gamma \rangle \\
\gamma \in & \text{env} \triangleq & \text{var} \rightharpoonup \text{value} \\
a \in & \text{answer} ::= & v \mid \texttt{BAD}
\end{array}
$$

The Haskell definitions for these sets are as follows:

```
data Expr = IntE Integer
          | PlusE Expr Expr
          | FunE String Expr
          | AppE Expr Expr
data Value = IntV Integer
           | CloV String Expr Env
type Env = Map String Value
data Answer = ValA Value
            | BadA
```

Complete the definition of an interpreter for this language. The mathematical specification for the interpreter is:

$$
\begin{array}{l}
\text{interp} \in \text{env} \times \text{expr} \to \text{answer} \\
\text{interp}(\gamma, i) \triangleq i \\
\text{interp}(\gamma, e_1 + e_2) \triangleq i_1 + i_2 \\
\quad \textit{where} \quad i_1 = \textit{interp}(\gamma, e_1) \\
\qquad\qquad\quad i_2 = \textit{interp}(\gamma, e_2)
\end{array}
$$

$$
\begin{array}{l}
\text{interp}(\gamma, \texttt{FUN}(x) \Rightarrow e) \triangleq \langle \texttt{FUN}(x) \Rightarrow e, \gamma \rangle \\
\text{interp}(\gamma, e_1(e_2)) \triangleq \text{interp}(\gamma'[x \mapsto v], e') \\
\quad \textit{where} \quad \langle \texttt{FUN}(x) \Rightarrow e', \gamma' \rangle = \text{interp}(\gamma, e_1) \\
\qquad\qquad\qquad\qquad\qquad\quad v = \text{interp}(\gamma, e_2)
\end{array}
$$

```
interp :: Env -> Expr -> Answer
interp env (IntE i) =




interp env (PlusE e1 e2) =




interp env (FunE x e) =




interp env (AppE e1 e2) =
```