Instructions: Answer all questions in the space provided. Extra blank pages are provided in the back. Partial credit will be given where appropriate. For reference, the static and dynamic semantics of the simply-typed lambda calculus (and relevant extensions) are given as attachments.
**Question 1.** (15 points)

Explain the flaw (if one exists) in the following proof by induction of the presumed fact “All flowers smell the same.” Be precise in indicating which statement(s) in the proof are false. (It is not acceptable to present a counterexample.)

**Proof.** Let $F$ be the set of all flowers and let $s(f)$ be the smell of flower $f \in F$. Assume that $F$ is countable. Let the property $P(n)$ define all subsets of $F$ of size at most $n$ containing flowers that smell the same.

$$P(n) \overset{df}{=} \forall X \in \mathcal{P}(F). |X| \leq n \Rightarrow (\forall f, f' \in X. s(f) = s(f'))$$

(the notation $|X|$ denotes the number of elements of $X$ and $\mathcal{P}(X)$ denotes the powerset (i.e., the set of all subsets) of a set $X$.)

We will prove by induction that $\forall n \geq 1. P(n)$ holds.

**Base:** $n = 1$. Obviously, all singleton set of flowers contain flowers that smell the same.

**Induction Step.** Let $n$ be arbitrary and assume that all subsets of $F$ of size at most $n$ contain flowers that smell the same. We prove that this property holds for all subsets of size at most $n + 1$. Pick an arbitrary set $X$ such that $|X| = n + 1$. Pick two distinct flowers $f, f' \in X$. We show that $s(f) = s(f')$. Let $Y = X - \{f\}$ and $Y' = X - \{f'\}$. $Y$ and $Y'$ are sets of size at most $n$ so the induction hypothesis holds for both. Pick an arbitrary $x \in Y \cap Y'$. We know $x \neq f$ and $x \neq f'$. By the induction hypothesis on $Y$ and $Y'$, we have $s(f) = s(x)$ and $s(f') = s(x)$. Hence, $s(f) = s(f')$, and the theorem is proved.

The flaw lies in the assumption there exists an $x \in Y \cap Y'$ such that $x \neq f$ and $x \neq f'$. The induction hypothesis does not guarantee that the set need be non-empty. In particular, consider the case when $n = 2$. 
Question 2. (15 points)

The IMP language is given by the following grammar:

\[
e \in AExp ::= n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \ast e_2
\]

\[
b \in BExp ::= \text{true} \mid \text{false} \mid e_1 = e_2 \mid e_1 \leq e_2 \mid \neg b \mid b_1 \land b_2 \mid b_1 \lor b_2
\]

\[
c \in Com ::= \text{skip} \mid x := e \mid c_1 ; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c
\]

Prove the following statement by structural induction: For any boolean command \( b \) and any initial state \( \sigma \), such that \( \sigma(x) \) is even, if

\[
\text{while } b \text{ do } x := x + 2, \sigma \Downarrow \sigma'
\]

then \( \sigma'(x) \) is even.

**Reference:** The natural semantics for while loops is given by the following two commands:

\[
(b, \sigma) \Downarrow \text{false}
\]

\[
(\text{while } b \text{ do } c, \sigma) \Downarrow \sigma
\]

\[
(b, \sigma) \Downarrow \text{true}, (c; \text{while } b \text{ do } c), \sigma \Downarrow \sigma'
\]

\[
(\text{while } b \text{ do } c, \sigma) \Downarrow \sigma'
\]

Consider a derivation \( D \) of this expression. There are two possible rules used at the top of \( D \):

1. while false \( \Rightarrow \) \( \sigma = \sigma' \Rightarrow \) \( \sigma' \) is even.

2. while true. In evaluating the antecedents of the true case, we have (\( x := x + 2, \sigma \Downarrow \sigma'' \) and \( D_1 = \langle \text{while } b \text{ do } \ldots, \sigma'' \Downarrow \sigma' \rangle \). We know that \( \sigma'' = \sigma[x \mapsto \sigma(x) + 2] \) and thus, \( \sigma''(x) \) is even. Apply the induction hypothesis to the derivation rooted at \( D_1 \).
**Question 3.** (15 points)

A **combinator** is a $\lambda$ expression with no free variables. The two most well-known combinators are:

- $S = \lambda x.\lambda y.\lambda z. x \, (y \, z)$
- $K = \lambda x.\lambda y. x$

1. What is the value of $SKK$ when reduced to normal form? Can you think of a good name for this term?

   $SKK$ reduces to $I$, the identity combinator, $\lambda x. x$.

2. What is the value of $S(KS)K$ when reduced to normal form? Can you think of a good name for this term?

   This combinator reduces to $\lambda x.\lambda y.\lambda z. x \, (y \, z)$. It’s equivalent to $B$ (defined below); and could be regarded as a function composition operation.

3. Consider the following two combinators:

   - $B = \lambda x.\lambda y.\lambda z. x \, (y \, z)$
   - $I = \lambda x. x$

   Reduce $BI$ to normal form and show that the resulting term is equivalent to $I$.

   $BI$ reduces to $\lambda y.\lambda z. y \, z$. We can apply $\eta$ reduction on the inner abstraction to get $\lambda y. y$ which is $\alpha$-equivalent to $I$.

4. For each of the following lambda terms, decide whether it is typable in the simply-typed lambda calculus. If a term is typable, give a type for it.

   (a) $\lambda x. \lambda y. x(y \, x)$: $(\sigma_1 \to \sigma_2) \to \sigma_1 \to \sigma_2$

   (b) $\lambda x. \lambda y. y \, x \, y$: *Not typable*

   (c) $\lambda x. \lambda y. y \, x \, (y \, y)$: $((\sigma_1 \to \sigma_2) \to \sigma_1) \to (\sigma_1 \to \sigma_2) \to \sigma_2$
1. (10 points)
   For each of the following type formulas, provide a closed lambda term in the simply typed lambda calculus that has that type.

2. $\sigma \to \sigma$:
   $$\lambda x : \sigma. x$$

3. $(\sigma_1 \to \sigma_2) \to \sigma_3 \to (\sigma_1 \to \sigma_2)$:
   $$\lambda x : \sigma_1 \to \sigma_2. \lambda y : \sigma_3. x$$

4. $(\sigma_1 \to \sigma_2) \to (\sigma_2 \to \sigma_3) \to (\sigma_1 \to \sigma_3)$:
   $$\lambda f. \sigma_1 \to \sigma_2. \lambda g : \sigma_2 \to \sigma_3. \lambda x : \sigma_1. g(f x)$$
Question 4. (15 points)
Prove the following type assertion by presenting a type derivation using the typing axioms and inference rules for the simply-typed lambda calculus.

1. $x : \sigma, y : \sigma \rightarrow \tau \vdash y \ x : \tau$

   Use T-APP to derive subtrees for $y$ and $x$. Use T-VAR to conclude these types bindings exist in the type environment.

2. $x : \sigma, y : \sigma \vdash (\lambda y : \sigma \rightarrow \tau. y \ x) : (\sigma \rightarrow \tau) \rightarrow \tau$

   Use T-ABS to derive subtrees for $y$ and $(yx)$. Use T-APP to derive a subtree for the application. Use T-VAR to conclude type bindings for $x$ and $y$ introduced by the abstraction.

3. $\phi \vdash (\lambda x : \sigma \rightarrow \sigma. \lambda y : \tau. x) : \tau \rightarrow \sigma \rightarrow \sigma$

   Not typable.
Question 5. (15 points)

Prove the following lemma: If Γ ⊢ e : σ then every free variable of e appears in Γ.

If you choose to use induction, you must clearly state the method of induction chosen, the base case, the induction hypothesis, the induction steps, and when you rely on the hypothesis.

By induction on typing derivations.

Base case. By assumption Γ ⊢ x : σ. Since any proof ends with x : σ ⊢ x : σ and x is the only free variable, the Lemma holds trivially.

Induction steps:

1. (Var): Suppose Γ, x : τ ⊢ e : σ. By IH, FV(e) ∈ Γ. Clearly, FV(e) ∈ Γ ∪ {x : τ}.

2. (Intro): Suppose Γ ⊢ λ x : τ.e : (τ → σ) follows from Γ, x : τ ⊢ e : σ. By IH, FV(e) ∈ Γ ∪ {x : τ}. Since FV(λ x : τ.e) = FV(e) − {x}, all free variables of (λ x : τ.e) ∈ Γ.

3. (Elim): Suppose Γ ⊢ (e₁ e₂) follows from Γ ⊢ e₁ : τ → σ and Γ ⊢ e₂ : σ. By IH, FV(e₁) ∈ Γ and FV(e₂) ∈ Γ. Thus, FV(e₁ e₂) = FV(e₁) ∪ FV(e₂), and all FV(e₁ e₂) ∈ Γ.
Question 6. (20 points)

Some call-by-value languages provide explicit constructs to support lazy evaluation. One common approach is to define operations called delay and force:

- The delay procedure is used together with the procedure force to implement lazy evaluation or call by need. The expression: delay(expression) returns an object called a promise which at some point in the future may be asked (by the force procedure) to evaluate expression, and deliver the resulting value.
- The expression force(expression) evaluates expression to a promise, and returns the value of the promise. If no value has been computed for the promise, then a value is computed and returned.

Consider the incorporation of delay and force into the simply-typed λ calculus.

1. (5 points) Show the typing rules for delay and force. You may assume the existence of a new type constructor prom such that τ prom denotes the type of a promise holding a value of type τ.

2. (5 points) Using a small-step operational semantics, formalize the informal description given above for the simply typed λ-calculus.

3. (10 points) The type preservation theorem can be stated as:

   If ⊢ e : τ and e → e' then ⊢ e' : τ

One way to prove this theorem is by induction on the derivation of e → e'. Show the cases for the new constructs in this proof.