Name: ________________________________

**Instructions:** Answer all questions in the space provided. Extra blank pages are provided in the back. Partial credit will be given where appropriate. Reference material are given as attachments.

Maximum score: 200

Attained total score: ________________________________
Question 1. (50 points)

A closure is a data structure consisting of a piece of code and a representation of an environment used to hold values of the code’s free variables. For example, the program given on the left below has a closure-converted representation shown on the right:

```
let val x = 1
  val y = 2
  val z = 3
  val f = λw. x + y + w
in f 100
end
```

A closure thus eliminates free variables in the body of an abstraction without resorting to substitution.

1. (5 points) What is the closure converted representation of the following program?

```
let val y = 1
  in if true then λx. x + y
    else λz. z
  end
```

2. (10 points) Consider the language $\lambda^C$ that extends the simply-typed $\lambda$-calculus (equipped with tuples and product types) with closure values. Present the grammar for this language, along with necessary typing and (small-step) evaluation rules.
3. (5 points) Explain why the program given in (1) above is not well-typed in $\lambda^C$.

4. (10 points) Explain why extending $\lambda^C$ with existential types would make this program well-typed. Give the explicit translation of this program using existential operators to demonstrate your answer.

5. (10 points) Explain how the definition of application must change to support properly-typed closures as existential types. If the above program fragment was bound to $f$, show the translation of $f(13)$ using existential types.

6. (10 points) What problems or issues (if any) do you foresee extending this approach to support typed closure-conversion in the context of a language that supports predicative (ML-style) polymorphism?
Question 2. (35 points)

Consider the ML datatype:

\[
\text{datatype } \text{tree} = \text{Leaf} \text{ of int} | \text{Unary} \text{ of tree} | \text{Binary} \text{ of tree} \times \text{tree}
\]

1. (5 points) Give a definition of \text{tree} as a recursive data type.

2. (5 points) Using an iso-recursive formulation (i.e., assuming existence of \text{fold} and \text{unfold} operations), provide definitions for the constructors \text{Leaf}, \text{Unary}, and \text{Binary}.

3. (10 points) On your representation, use an iso-recursive formulation (i.e., assume existence of \text{fold} and \text{unfold} operations) to write a function \text{count: tree} \rightarrow \text{int} in ML-syntax that counts the number of leaves in a tree.

4. (15 points) State progress and preservation theorems for the simply-typed \(\lambda\)-calculus equipped with iso-recursive types. How would you go about proving these theorems?
Question 3. (40 points)

Many programming languages such as Scheme or Python support *dynamic* values, values whose types are not known statically at compile-time.

Values of type **Dynamic** are built using the `dynamic` construct. The result of evaluating expression `dynamic e : T` is a pair of a value `v` and type-tag `T` where `v` is the result of evaluating `e`.

The `typecase` construct is used to examine the tag of a **Dynamic** value. Thus, the expression:

\[
\lambda x : \text{Dynamic}.
\text{typecase } x \text{ of }
\begin{align*}
(i : \text{Nat}) & \Rightarrow i + i \\
\text{else} & \Rightarrow 0
\end{align*}
\text{end}
\]

applied to `dynamic 1 : \text{Nat}` evaluates to `2`.

Consider a language whose terms and types are generated by the following grammar:

\[
e ::= x | \text{wrong} | \lambda x : \tau. e | e(e) | \text{dynamic } e : \tau | \text{typecase } e \text{ of } \ldots (x : \tau) \Rightarrow e \ldots \text{else } e \\
\tau ::= c | \tau_1 \rightarrow \tau_2 | \text{Dynamic}
\]

where `c` ranges over base types (e.g., numbers, Booleans, etc.), and `wrong` corresponds to a runtime type error.

1. (5 points) Give a one sentence comparison between dynamic types and disjoint unions.

2. (5 points) Write a program fragment whose evaluation yields `wrong`.
3. (10 points) Define evaluation rules for \texttt{typecase} and \texttt{dynamic} using any style of operational semantics you wish.

4. (10 points) Define type rules for \texttt{dynamic} and \texttt{typecase}.

5. (10 points) Consider an extension to typecase that allows the introduction of \textit{type variables}. The expression:
\begin{verbatim}
  typecase e sel of 
  \quad (X_1, \ldots, X_n) (x : \tau) => e 
  \ldots
\end{verbatim}

introduces type variables $X_1 \ldots X_n$ that have scope over the entire branch (both $\tau$ and $e$).

Using this extended version of \texttt{typecase}, write a function that applies its first argument to its second argument, after checking that the application is correctly typed. (Hint: both arguments must be passed as dynamic values.)
**Question 4 (25 points)**

1. (5 points) Show that the type assigned to a term by the algorithmic subtyping rules can *decrease* during evaluation by finding two terms \( s \) and \( t \) with algorithmic types \( \sigma \) and \( \tau \) such that \( s \to^* t \) and \( \tau <: \sigma \) but \( \sigma \not\ll: \tau \).

2. (5 points) In the absence of a **Bot** type, is there a type that is a subtype of every other type? Is there an arrow type that is a supertype of every other arrow type?

3. (15 points)

   For each lettered type \( \sigma \) from the left-hand column below, indicate the number of every type \( \tau \) on the right-hand column such that \( \sigma <: \tau \). If no matching term exists, write “None” for that letter.

   \[
   \begin{array}{ll}
   \sigma & \tau \\
   (a) \{ \text{a: } \{ \text{ }, \text{b: } \{ \text{x: Nat} \} \} \} & (1) \text{ (Bot } \to \text{ Top } \to \text{ Bot} \\
   (b) \text{ (Nat } \to \text{ Nat } \to \text{ Bool} \} & (2) \text{ Top } \to \text{ Nat} \\
   (c) \{ \text{a: Top } \to \{ \text{a: Top } \} \} & (3) \{ \text{b: } \{ \} \} \\
   (d) \{ \{ \text{a:Top} \} \to \{ \} \} \to \{ \text{b:Top} \} & (4) \{ \text{b: Nat} \} \\
   (e) \{ \text{b: Top } \to \text{ Nat} \} & (5) \{ \{ \text{a:Nat} \} \to \{ \text{a:Top} \} \} \to \{ \} \\
   & (6) \{ \text{b: } \{ \} \to \text{ Top} \} \\
   & (7) \{ \{ \text{a:Top} \} \to \{ \text{a:Nat} \} \} \to \{ \} \\
   & (8) \{ \} \to \{ \text{a: Top} \} \to \{ \} \\
   & (9) \text{ (Top } \to \text{ Bot } \to \text{ Top} \\
   & (10) \{ \text{a: Top, b: } \{ \} \} \to \{ \} 
   \end{array}
   \]
Question 5. (25 points)

1. (9 points) For each of the following ML terms, write a corresponding term in System F. You may assume the availability of tuples and projections. You may further assume the availability of base types and lists.

   (a) \texttt{fn x => x + let fun f (x) = x in f(2) end}

   (b) \texttt{let fun twice(f,x) = f(f(x)) in twice(fn y => y + 1,2)}

   (c) \texttt{let val f = fn x => x in f [f(0)] end}

2. (8 points) Give a one sentence explanation to describe each of the following terms. For each term, state whether it could be encoded in ML.

   (a) \(\lambda x : \forall t.t \rightarrow \forall u.t.x\int\text{int}\)

   (b) \((\lambda f : \forall t.t \rightarrow t \text{\textit{list}}.f[\text{int}\text{list}],f[\text{int}].0) (\Lambda t.\lambda x : t.\text{\textit{cons}}(x,\text{\textit{nil}}))\)

3. (8 points) Give a one sentence explanation for why the following terms are not well-typed.

   (a) \(\Lambda t.\lambda x : t.\Lambda t.x\)

   (b) \(\Lambda t.\lambda x : t.\text{\textit{cons}}(x,\text{\textit{nil}})[\text{\textit{int}}][\text{\textit{int\textit{list}}}]\)
Question 6. (25 points)

Prove the following theorem:

Theorem: [Minimality of Algorithmic Subtyping]. If $\Gamma \vdash e : \sigma$ then $\Gamma \vdash e : \tau$ for some $\tau <: \sigma$, where $\vdash$ defines the least relation closed under the algorithmic typing rules.

For your proof, you may use the following lemma:

Lemma [Inversion of Subtyping].

1. If $\sigma <: \tau_1 \rightarrow \tau_2$ then $\sigma$ has the form $\sigma_1 \rightarrow \sigma_2$ with $\tau_1 <: \sigma_1$ and $\sigma_2 <: \tau_2$.

2. If $\sigma <: \{l_i : \tau_i^{i \in 1,...,m}\}$, then $\sigma$ has the form $\{k_j : \sigma_j^{j \in 1,...,m}\}$ with at least labels $\{l_i^{i \in 1,...,n}\}$ and with $\sigma_j <: \tau_i$ for each common label $l_i = k_j$. 