Rules:

- Please do not turn the page until everyone is ready.
- The exam is closed-book, closed-note, except for two sides of one 8.5x11in piece of paper.
- Please stop promptly at 8:30.
- You can rip apart the pages, but please write your name on each page.
- There are 100 points total, distributed very unevenly among 7 questions (most of which have multiple parts).

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit.
- The questions are not necessarily in order of difficulty.
- Skip around and focus on the questions worth more points.
- If you have questions, ask.
- Relax. You are here to learn.
For your reference (page 1 of 2):

\[
\begin{align*}
\text{ADD} & \quad H : e_1 \downarrow v_1 \quad H : e_2 \downarrow v_2 \\
\hline
\quad & \quad H : e_1 + e_2 \downarrow e_1 + e_2
\end{align*}
\]

\[
\begin{align*}
\text{MULT} & \quad H : e_1 \downarrow v_1 \quad H : e_2 \downarrow v_2 \\
\hline
\quad & \quad H : e_1 \ast e_2 \downarrow e_1 \ast e_2
\end{align*}
\]

\[
\begin{align*}
\text{IF1} & \quad H : e \downarrow i \quad H : s_1 \downarrow H' \\
\hline
\quad & \quad H : \text{if } e \text{ s}_1 s_2 \downarrow H'
\end{align*}
\]

\[
\begin{align*}
\text{IF2} & \quad H : e \downarrow 0 \quad H : s_2 \downarrow H' \\
\hline
\quad & \quad H : \text{if } e \text{ s}_1 s_2 \downarrow H'
\end{align*}
\]

\[
\begin{align*}
\text{WHILE} & \quad H : \text{if } e (s; \text{while } e) \text{ skip} \downarrow H' \\
\hline
\quad & \quad H : \text{while } e \text{ s} \downarrow H'
\end{align*}
\]

\[
\begin{align*}
e & ::= \lambda x. e | x | e e \\
v & ::= \lambda x. e
\end{align*}
\]

\[
\begin{align*}
e & ::= c | x | \lambda x : \tau. e | e e | \Lambda \alpha. e | e[\tau] \\
\tau & ::= \text{int} | \tau \rightarrow \tau | \alpha | \forall \alpha. \tau \\
v & ::= c | \lambda x : \tau. e | \Lambda \alpha. e \\
\Gamma & ::= \cdot | \Gamma, x : \tau | \Gamma, \alpha
\end{align*}
\]
Name:

\[ e ::= \lambda x. e | e e | c | (e, e) | e.1 | e.2 | A e | B e | \text{match } e \text{ with } A x = e \mid B x = e | \text{letrec } f x. e \]
\[ v ::= \lambda x. e | c | (v, v) | A v | B v | \{ l_1 = v, \ldots, l_n = v \} \]
\[ \tau ::= \text{int} | \tau \to \tau | \tau \ast \tau | \tau + \tau | \{ l_1 = \tau, \ldots, l_n = \tau \} \]

\[ \Gamma \vdash e : \tau \text{ and } \tau_1 \leq \tau_2 \]

\[ \begin{array}{l}
\Gamma \vdash e : \text{int} \quad \Gamma \vdash x : \Gamma(x) \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \\
\Gamma \vdash \lambda x. e : \tau_1 \to \tau_2 \\
\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2 \\
\Gamma \vdash \text{letrec } f x. e : \tau_1 \to \tau_2 \\
\Gamma \vdash e : \tau_1 \\
\Gamma \vdash e : \tau_2 \\
\Gamma, x : \tau_1 \vdash e : \tau_3 \\
\Gamma, y : \tau_2 \vdash e : \tau_3 \\
\Gamma \vdash \text{letrec } f x. e : \tau_1 \to \tau_2 \\
\Gamma \vdash A e : \tau_1 \to \tau_2 \\
\Gamma \vdash B e : \tau_1 \to \tau_2 \\
\Gamma \vdash (\text{match } e_1 \text{ with } A x = e_2 | B y = e_3) : \tau_3 \\
\Gamma \vdash \{ l_1 = v_1, \ldots, l_n = v_n \} : \{ l_1 = \tau_1, \ldots, l_n = \tau_n \} \\
\Gamma \vdash e.1 : \tau_1 \\
\Gamma \vdash e.2 : \tau_2 \\
\Gamma \vdash e.1 : \tau_1 \ast \tau_2 \\
\Gamma \vdash e.2 : \tau_1 \ast \tau_2 \\
\Gamma \vdash e.1 : \tau_1 + \tau_2 \\
\Gamma \vdash e.2 : \tau_1 + \tau_2 \\
\Gamma \vdash \{ (l_1, v_1, \ldots, l_n, v_n) \} : \{ n \} \\
\Gamma \vdash e.1 \vdash e.2 : \tau_3 \\
\Gamma \vdash e.1 \vdash e.2 : \tau_4 \\
\Gamma \vdash e.1 \vdash e.2 : \tau_5 \\
\Gamma \vdash e.1 \vdash e.2 : \tau_6 \\
\end{array} \]
1. (20 points) Suppose we add division to our IMP expression language. In Caml, the expression syntax becomes:

```caml
type exp =
  Int of int | Var of string | Plus of exp * exp | Times of exp * exp | Div of exp * exp
```

Our interpreter (not shown) raises a Caml exception if the second argument to `Div` evaluates to 0. We are ignoring statements; assume an IMP program is an expression that takes an unknown heap and produces an integer.

(a) Write a Caml function `nsz` (stands for “no syntactic zero”) of type `exp->bool` that returns false if and only if its argument contains a division where the second argument is the integer constant 0. Note we are not interpreting the input; `nsz` is not even passed a heap.

(b) If we consider division-by-zero at run-time a “stuck state” and `nsz` a “type system” (where true means “type-checks”), then:

   i. Is `nsz` sound? Explain.

   ii. Is `nsz` complete? Explain.

**Solution:**

```caml
let rec nsz e =
  match e with
  | Int _ -> true
  | Var _ -> true
  | Plus(e1,e2) -> nsz e1 && nsz e2
  | Times(e1,e2) -> nsz e1 && nsz e2
  | Div(e1,Int 0) -> false
  | Div(e1,e2) -> nsz e1 && nsz e2
```

The type system is not sound: It may accept a program that would get stuck at run-time. For example, `Div(3,x)` would get stuck for any heap that mapped `x` to 0.

The type system is complete: All programs it rejects will get stuck at run-time under any heap. That is because expression evaluation always evaluates all subexpressions, so the division-by-zero will execute. (Substantial partial credit for explaining that code that doesn’t execute leads to incompleteness. It just happens that IMP expressions do not have code that doesn’t execute.)
2. (20 points) Consider this Caml code. It uses `strcmp`, which has type `string->string->bool` and the expected behavior.

```caml
exception NoValue
let empty = fun s -> raise NoValue
let extend m x v = fun s -> if strcmp s x then v else m s
let lookup m x = m x
```

(a) What functionality do these three bindings provide a client?
(b) What types do each of the bindings have?
    (Note: They are all polymorphic and may have more general types than expected.)

Solution:

(a) They provide maps from strings to values (where the client chooses the type of the values). `empty` is the empty-map; calling `lookup` with it and any string raises an exception. `extend` creates a larger map from a smaller one (`m`) by having `x` map to `v` (shadowing any previous mapping for `x`) and otherwise using the map `m`.

(We didn’t ask how the code works: A map is represented by a Caml function from strings to values, so `lookup` is just function application. `extend` creates a new function that uses `m`, `x`, and `v` as free variables: If the string it is passed is not equal to `x`, then it just applies the smaller map `m` to `s`.)

(b) 
    `empty` : `'a -> 'b`
    `extend` : `(string -> 'a) -> string -> 'a -> (string -> 'a)`
    `lookup` : `( 'a -> 'b ) -> 'a -> 'b`
3. (16 points) When we added sums (syntax A e, B e, and match $e_1$ with A $x \rightarrow e_2$|B $y \rightarrow e_3$) to the λ-calculus, we gave a small-step semantics and had exactly two constructors.

(a) Give sums a large-step semantics, still for exactly two constructors. That is, extend the call-by-value large-step judgment $e \Downarrow v$ with new rules. (Use 4 rules.)

(b) Suppose a program is written with three constructors (A, B, and C) and match expressions that have exactly three cases:

$$\text{match } e_1 \text{ with } A \ x \rightarrow e_2 | B \ y \rightarrow e_3 | C \ z \rightarrow e_4$$

Explain a possible translation of such a program into an equivalent one that uses only two constructors. (That is, explain how to translate the 3 constructors to use 2 constructors and how to translate match expressions. Do not write inference rules.)

Solution:

(a)

$$\begin{align*}
&\frac{e \Downarrow v}{A \ e \Downarrow A \ v} & &\frac{e \Downarrow v}{B \ e \Downarrow B \ v} \\
&\frac{e_1 \Downarrow A \ v_1 \quad e_2[v_1/x] \Downarrow v_2}{\text{match } e_1 \text{ with A } x \rightarrow e_2 | B \ y \rightarrow e_3 | C \ z \rightarrow e_4} & &\frac{e_1 \Downarrow B \ v_1 \quad e_3[v_1/y] \Downarrow v_2}{\text{match } e_1 \text{ with A } x \rightarrow e_2 | B \ y \rightarrow e_3 | C \ z \rightarrow e_4}
\end{align*}$$

(b) One solution: Replace every $B \ e$ with $B(A \ e)$ and $C \ e$ with $B(B \ e)$. Replace every:

$$\text{match } e_1 \text{ with A } x \rightarrow e_2 | B \ y \rightarrow e_3 | C \ z \rightarrow e_4$$

with:

$$\text{match } e_1 \text{ with A } x \rightarrow e_2 | B \ q \rightarrow (\text{match } q \text{ with A } y \rightarrow e_3 | B \ z \rightarrow e_4)$$
4. (14 points) Consider a \( \lambda \)-calculus with *tuples* (i.e., “pairs with any number of fields”), so we have expressions \((e_1, e_2, \ldots, e_n)\) and \(e.i\) and types \(\tau_1 \ast \tau_2 \ast \cdots \ast \tau_n\). For each of our subtyping rules for records, explain whether or not an analogous rule for tuples makes sense.

**Solution:**

- The permutation rule does *not* make sense. Tuple fields are accessed by position so subsuming \(\text{string} \ast \text{int}\) to \(\text{int} \ast \text{string}\) would allow \(e.2\) to have type \(\text{string}\) when it should not.

- The width and depth rules *do* make sense for the same reasons as records: Forgetting about fields on the right means only that fewer expressions of the form \(e.i\) will type-check. Assuming tuple-fields are read-only just like record fields, covariant subtyping is correct.
5. (14 points) Assume a class-based object-oriented language as in class, and a program that contains the call \( \text{e.f((C)e1)} \) where \( e1 \) is a (compile-time) subtype of \( C \) and the whole call type-checks.

(a) If calls are resolved with static overloading, is it possible that removing the cast (i.e., changing the call to \( \text{e.f}(e1) \)) could cause the program to still type-check but behave differently? Explain.

(b) If calls are resolved with static overloading and we have multiple inheritance, is it possible that removing the cast \( C \) (i.e., changing the call to \( \text{e.f}(e1) \)) could cause the program to no longer type-check? Explain.

(c) If calls are resolved with multimethods, is it possible that removing the cast \( C \) (i.e., changing the call to \( \text{e.f}(e1) \)) could cause the program to behave differently? Explain.

Solution:

(a) Yes, it is possible. For example, suppose:
   - \( e2 \) has type \( A \), which is a subtype of \( C \).
   - \( e \) has type \( D \) and class \( D \) defines methods \( f(C) \) and \( f(A) \).

   Now removing the cast results in a different method being called.

(b) Yes, it is possible. For example, suppose:
   - \( e2 \) has type \( A \), which is a subtype of \( C \) and \( B \).
   - \( e \) has type \( D \) and class \( D \) defines methods \( f(C) \) and \( f(B) \), but not \( f(A) \).

   Now removing the cast results in an ambiguous call.

(c) No, it is not possible. The method called depends on the run-time types of the values that \( e \) and \( e1 \) evaluate to, and \( (C)e1 \) evaluates to the same value as \( e1 \).
6. (9 points) Here are two large-step interpreters for the untyped lambda-calculus. The one on the right uses parallelism. Recall Thread.join blocks until the thread described by its argument terminates. Only the lines between the (*----------*) comments differ.

```ml
type exp = Var of string | Lam of string*exp | Apply of exp * exp
let subst e1_with e2_for x = ... (* unimportant *)
exception UnboundVar

let rec interp e = let rec interp e =
  match e with match e with
  | Var _ -> raise UnboundVar | Var x -> raise UnboundVar
  | Lam _ -> e | Lam _ -> e
  | Apply(e1,e2) -> | Apply(e1,e2) ->
  (*----------*) (*----------*)
    let v2r = ref (Var "dummy") in
    let t = Thread.create (fun () -> v2r := interp e2) () in
    let v1 = interp e1 in
    Thread.join t;
    let v2 = !v2r in
  (*----------*) (*----------*)
  match v1 with match v1 with
    | Lam(x,e3) -> interp(subst e3 v2 x) | Lam(x,e3) -> interp(subst e3 v2 x)
    | _ -> failwith "impossible" | _ -> failwith "impossible"
```

(a) Describe an input to these functions for which the interpreter on the right would raise an exception and the interpreter on the left would not. (Note: Evaluation of expressions may not terminate.)

(b) Explain why moving the line “let v2r = ref (Var "dummy") in” out to the top-level (and removing the keyword “in”) would make the interpreter on the right behave unpredictably (even for inputs with no free variables).

Solution:

(a) An argument that applies an expression with an unbound variable to an expression that doesn’t terminate shows the difference. For example:

```ml
App(Var("x"),
    App(Lam ("x", App(Var "x", Var "x")),
          Lam ("x", App(Var "x", Var "x"))))
```

(b) Interpretation could lead to more than two threads running concurrently because of nested applications: An expression like App(App(e1,e2),App(e3,e4)) would lead to four threads, and using a shared reference leads to a race condition: The thread evaluating App(e1,e2) may not read the reference set by the thread evaluating e2 until another thread (e.g., the thread evaluating e4) sets the reference to hold another value.
7. (7 points) You can do this problem in one of Caml, C, C++, Java, or C#. Your choice does not really change the problem.

(a) Write a short program that will exhaust memory if there is no garbage collector but take almost no space if there is a garbage-collector.

(b) Write a short program that will exhaust memory even if there is a garbage collector. Create only small objects.

Solution:

(a) #include <stdlib.h>
    int main() {
        for(;;)
            malloc(4);
    }

(b) #include <stdlib.h>
    struct L { struct L * x; };
    struct L * p = NULL;
    int main() {
        for(;;) {
            struct L * q = malloc(sizeof(struct L));
            q->x = p;
            p = q;
        }
    }