Practice Midterm Exam II

CAS CS 320: Concepts of Programming Languages

Name:

BUID:

- ▷ You will have approximately 75 minutes to complete this exam. Make sure to read every question, some are easier than others.
- ▷ Do not remove any pages from the exam.
- ▷ Make very clear what your final solution for each problem is (e.g., by surrounding it in a box). We reserve the right to mark off points if we cannot tell what your final solution is.
- ▷ You must show your work on all problems unless otherwise specified. A solution without work will be considered incorrect (and will be investigated for potential academic misconduct).
- ▷ Unless stated otherwise, you should only need the rules provided **in the problem statement** for any derivation.
- ▷ We will not look at any work on the pages marked "*This page is intentionally left blank.*" You should use these pages for scratch work.

This page is intentionally left blank.

1 Higher-Order Programming

A. (5 points) Implement the function

```
val fold_left2 : ('acc -> 'a -> 'b -> 'acc) -> 'acc -> 'a list -> 'b list -> 'acc
```

so that `fold_left2 op base l r` is the result of left-associatively folding `op` over the lists `l` and `r` simultaneously, e.g.,

```
fold_left2 op base [a1;a2;a3;a4] [b1;b2;b3;b4]
```

is equivalent to

```
op (op (op (op base a1 b1) a2 b2) a3 b3) a4 b4
```

The behavior of the function is undefined if `l` and `r` are not the same length. **Your solution must be tail-recursive.** You *cannot* use any functions from the standard library.

B. (2 points) Recall that, given an $(m \times n)$ matrix A and an n-dimensional vector $\mathbf{v}$, the matrix-vector product $A\mathbf{v}$ is then m-dimensional vector whose i^{th} entry is $\sum_{j=1}^{n} A_{ij}\mathbf{v}_j$. In other words, the i^{th} entry is the dot product of the ith row of A and $\mathbf{v}$.

We can give an simple definition of matrix-vector multiplication using higher-order functions. We represent a vector as a list of floating-point numbers, and a matrix as a list of rows (each represented by lists of floating-point numbers).

```
type vec = float list
type mat = float list list

let op = assert false (* TODO *)
let base = assert false (* TODO *)
let mat_vec_mul (a : mat) (v : vec) : vec = List.map (List.fold_left2 op base v) a

let a = [[1. ; 2.] ; [3. ; 4.]]
let v = [5. ; 6.]
let _ = assert (mat_vec_mul a v = [(1. *. 5. +. 2. *. 6.); (3. *. 5. +. 4. *. 6.)])
```

Give definitions for `op` and `base` so that the above definition of `mat_vec_mul` is correct. Note that the behavior of this function is undefined if A and $\mathbf{v}$ are not the same length (i.e., if A has more or fewer rows than $\mathbf{v}$ has entries) or if A is not a valid matrix. *Hint:* The only standard library you're allowed to use are `(+.)` and `( *. )` (i.e., floating-point addition and multiplication).

2 Formal Grammar

A. (5 points) Consider the following OCaml-like grammar (with start symbol <p>).

```
<p>  ::= <s> <p> | <s>
<s>  ::= let <v> = <e>
<e>  ::= let <v> = <e> in <e> | fun <v> -> <e> | <e2>
<e2> ::= <e2> <e3> | <e3>
<e3> ::= x | y | z | 0 | 1 | 2 | ( <e> )
<v>  ::= x | y | z
```

Demonstrate that the sentence

```
let x = let y = 0 in y
let z = (fun x -> x) 1
```

is recognized by the above grammar by providing a parse tree for it.

B. (5 points) Suppose we wanted to allow expressions at the top-level in OCaml. Then we could write:

```
let x = "foo"
let y = "bar"
print_endline x
print_endline y
```

This would require something like the following modification to the grammar from the first part (only two production rules for <p> have been added).

<p> ::= <s> <p> | <s> | <e> <p> | <e>
<s> ::= let <v> = <e>
<e> ::= let <v> = <e> in <e> | fun <v> -> <e> | <e2>
<e2> ::= <e2> <e3> | <e3>
<e3> ::= x | y | z | 0 | 1 | 2 | (<e>)
<v> ::= x | y | z

Demonstrate that the above grammar is ambiguous by determining a sentence which has multiple parse trees. **Include both parse trees in your solution.**

3 Evaluation Strategies

Recall the big-step call-by-value (CBV) semantics for the λ-calculus:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}\ \textsc{fun} \qquad \frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v_2 \qquad e' = [v_2/x]e \qquad e' \Downarrow v}{e_1 e_2 \Downarrow v}\ \textsc{appCBV}$$

and the big-step call-by-name (CBN) semantics for the λ-calculus:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}\ \textsc{fun} \qquad \frac{e_1 \Downarrow \lambda x.e \qquad e' = [e_2/x]e \qquad e' \Downarrow v}{e_1 e_2 \Downarrow v}\ \textsc{appCBN}$$

A. (5 points) Give a closed[1] expression e_0 and a value v_0 such that $e_0 \Downarrow v_0$ is derivable using CBN semantics, but **not** derivable using CBV semantics. Your solution must include:

   (a) the CBN semantic derivation of $e_0 \Downarrow v_0$;

   (b) a one sentence description for why $e_0 \Downarrow v_0$ is not derivable using CBV semantics.

   Hint. Recall the expression $(\lambda x.xx)(\lambda x.xx)$.

[1] An expression is **closed** if it has no free variables.

B. (5 points) One notable feature of the semantic systems from the previous part is that they do not *evaluate under binders*. This is why we were able to ignore capture-avoidance when working with closed expressions. The *full* evaluation strategy (which does evaluate under binders, i.e., **note the difference in the FUN rule**) has the following semantic rules.

$$\frac{x \text{ is a variable}}{x \Downarrow x}\ \text{VAR} \qquad \frac{e \Downarrow v}{\lambda x.e \Downarrow \lambda x.v}\ \text{FUN} \qquad \frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v_2 \qquad e' = [v_2/x]e \qquad e' \Downarrow v}{e_1 e_2 \Downarrow v}\ \text{APP}$$

Determine a value v up to α-equivalence[2] such that

$$\lambda y.(\lambda x.\lambda y.x)y \Downarrow v$$

using the full evaluation strategy. **Also provide the derivation.** *Hint.* This will require α-renaming. You may choose any fresh variable you'd like. The definition of capture-avoiding substitution is given on the following page.

[2]This means that your choice of v can differ from ours in the naming of bound variables.

(Problem 3B Continued)

Capture-Avoiding Substitution:

$$[v/y]x = \begin{cases} v & x = y \\ x & \text{otherwise} \end{cases}$$

$$[v/y](\lambda x.e) = \begin{cases} \lambda x.e & x = y \\ \lambda z.[v/y][z/x]e & x \in FV(v), z \text{ is fresh} \\ \lambda x.[v/y]e & \text{otherwise} \end{cases}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

4 Linear Types

Recall the typing rules for the simply-typed lambda calculus (STLC):

$$\frac{(x:\tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{var} \qquad \frac{\Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x^{\tau_1}.e : \tau_1 \to \tau_2}\ \textsc{fun} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}\ \textsc{app}$$

One issue with STLC is that simple types don't tell us anything about *resource usage*. For example, given an expression of type $\tau_1 \to \tau_2$, we know it's a function that takes an argument of type τ_1, but we don't know *how many times* it's going to use that argument. Here's an alternative system called the **linear lambda calculus** (LLC) which attempts to address this *by enforcing that every variables is used exactly once.*

$$\frac{}{\{x:\tau\} \vdash x : \tau}\ \textsc{var} \qquad \frac{\Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x^{\tau_1}.e : \tau_1 \to \tau_2}\ \textsc{fun}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_2 \to \tau \qquad \Gamma_2 \vdash e_2 : \tau_2 \qquad \Gamma_1 \text{ and } \Gamma_2 \text{ are disjoint}}{\Gamma_1, \Gamma_2 \vdash e_1 e_2 : \tau}\ \textsc{app}$$

Recall that two contexts are **disjoint** if they don't declare any of the same variables, e.g., $\{x : \bot, y : \bot \to \bot\}$ and $\{z : \bot \to \bot\}$ are disjoint whereas $\{x : \bot, y : \bot \to \bot\}$ and $\{x : \bot \to \bot\}$ are not. Also note that, by "Γ_1, Γ_2" in the APP rule, we mean the combination of all declarations in both Γ_1 and Γ_2.

A. (10 points) Determine the type τ such that

$$\varnothing \vdash \lambda f^{\bot \to \bot \to \bot}.\lambda x^{\bot}.\lambda y^{\bot}.f\,x\,y : \tau$$

is derivable in LLC. Also provide the derivation.

B. (5 points) Determine an expression with no variable-shadowing (i.e., bound variables must be distinct) that's well-typed in STLC but not in LLC. To receive full credit you must provide:

   ▷ A derivation of $\varnothing \vdash e : \tau$ in STLC for some expression e and type τ.

   ▷ A one sentence justification of why $\varnothing \vdash e : \tau$ is not derivable in LLC.

   Hint. Think about how many times a function well-typed in LLC is allowed to use its argument. In particular, use the fact that if $\Gamma \vdash e' : \tau'$ is derivable in LLC, then the variables declared in Γ are *exactly* the free variables in e'.