Practice Final Examination

CAS CS 320: Principles of Programming Languages

April 25, 2025

Name: Nathan Mull

BUID: 12345678

- ▷ You will have approximately 120 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 dishonesty).
- ▷ Unless stated otherwise, you should only need the rules provided **in that problem** for your derivations.
- ▷ 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.

1 Grammatical Ambiguity

Let $\mathcal{G}$ denote the following grammar.

```
<e>  ::= if true then <e> else <e> | <e2>
<e2> ::= <e2> + <e3> | <e3>
<e3> ::= 1 | ( <e> )
```

A. Demonstrate that $\mathcal{G}$ recognizes the following sentence by writing a derivation of it. For partial credit, you can draw a parse tree instead.

```
if true then 1 else 1 + 1
```

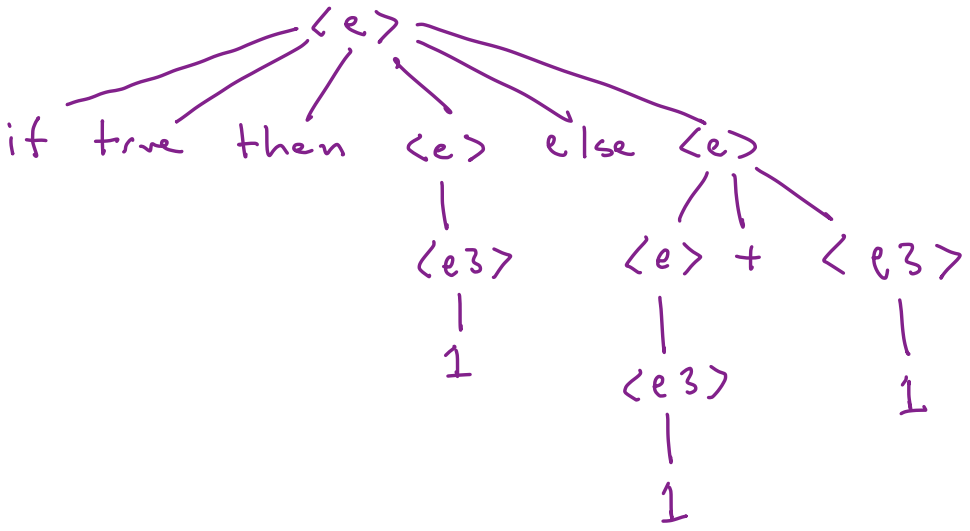
<e>
if true then <e> else <e>
if true then <e2> else <e>
if true then <e3> else <e>
if true then 1 else <e>
if true then 1 else <e2>
if true then 1 else <e2> + <e3>
if true then 1 else <e3> + <e3>
if true then 1 else 1 + <e3>
if true then 1 else 1 + 1

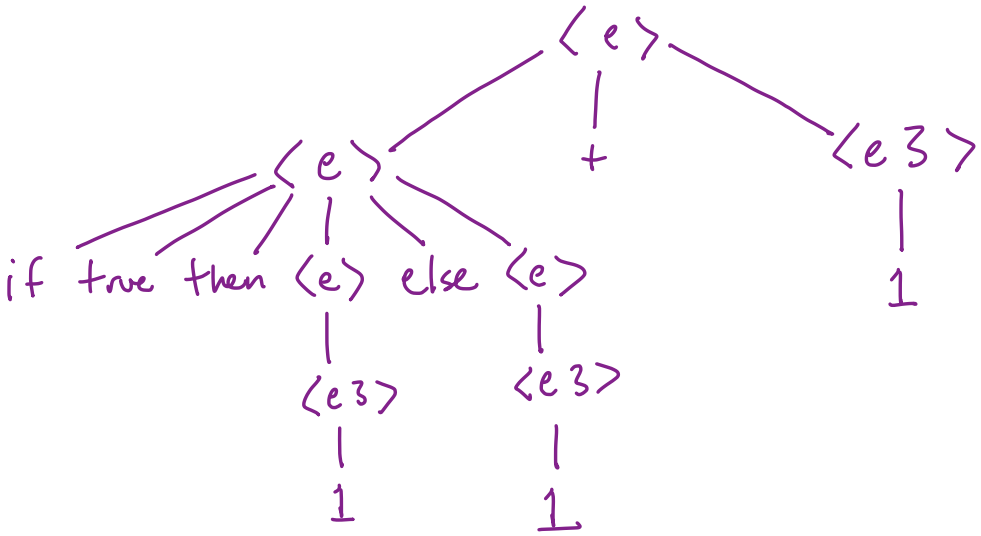
B. Suppose we wanted to simplify $\mathcal{G}$ by getting rid of the nonterminal symbol <e2>. Let $\mathcal{G}'$ denote the following grammar.

<e> ::= if true then <e> else <e> | <e> + <e3> | <e3>
<e3> ::= 1 | (<e>)

Demonstrate that this grammar is ambiguous by determining a sentence with two distinct parse trees in $\mathcal{G}'$. In addition, draw two parse trees for this sentence.

if true then 1 else 1 + 1





C. One issue with the grammar $\mathcal{G}$ is that it does not recognize sentences like the following.

```
1 + if true then 1 else 1
```

OCaml accepts this sentences and gives it the value `2`. Write down a *single* production rule that we can add to $\mathcal{G}$ so that it allows an if-expression to appear on the right-hand side of an addition operator, and so that *the resulting grammar remains unambiguous*. (**Hint.** The left-hand side does not necessarily have to be the number `1`, it could be `1 + 1` or `(1 + if true then 1 else 1)`. So think about what kinds of expressions can be on the left-hand side of the addition operator, if the right-hand side is an if-expression.)

⟨e⟩ ::= ⟨e2⟩ + if true then ⟨e⟩ else ⟨e⟩

Note that ⟨e⟩ ::= ⟨e2⟩ + ⟨e⟩ does not work. It makes the grammar ambiguous

2 Bizarre Type Systems

A. In all the type systems we've studied, the expression `fun x → x x` is not well-typed. Roughly speaking, this is because the `x` that appears in the body of the expression needs to have *two types at once*; it needs to behave like a function, but also as an argument to a function. *Intersection types* allow us to express this. Consider the following type system:

$$\texttt{<e>} ::= \texttt{<v>} \mid \texttt{fun <v> → <e>} \mid \texttt{<e> <e>}$$
$$\texttt{<ty>} ::= \texttt{empty} \mid \texttt{<ty> → <ty>} \mid \texttt{<ty> ∩ <ty>}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \text{(var)} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \texttt{ → } e : \tau_1 \texttt{ → } \tau_2}\ \text{(fun)} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \texttt{ → } \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ \text{(app)}$$

$$\frac{\Gamma \vdash e : \tau_1 \cap \tau_2}{\Gamma \vdash e : \tau_1}\ \text{(inter-left)} \qquad \frac{\Gamma \vdash e : \tau_1 \cap \tau_2}{\Gamma \vdash e : \tau_2}\ \text{(inter-right)}$$

We've introduced a new type constructor $\tau_1 \cap \tau_2$ which we read as "simultaneously τ_1 and τ_2." The rules (inter-left) and (inter-right) allow us to use something that is simultaneously a τ_1 and a τ_2 in settings where we need just a τ_1 or just a τ_2. (The type `empty` is the empty type, a type with no rules or constructors.)

Demonstrate that the expression `fun x → x x` is well-typed in the empty context in this type system by writing a typing derivation. Your derivation should be in standard form, and all inferences should be labeled with rule names.

$$\dfrac{\dfrac{\dfrac{}{x : \bot \cap (\bot \to \bot) \vdash x : \bot \cap (\bot \to \bot)}\,(\text{var})}{\{x : \bot \cap (\bot \to \bot)\} \vdash x : \bot \to \bot}\,(\text{inter-right}) \qquad \dfrac{\dfrac{}{\{x : \bot \cap (\bot \to \bot)\} \vdash x : \bot \cap (\bot \to \bot)}\,(\text{var})}{\{x : \bot \cap (\bot \to \bot)\} \vdash x : \bot}\,(\text{inter-left})}{\{x : \bot \cap (\bot \to \bot)\} \vdash x\ x : \bot}\,(\text{app})}{\bullet \vdash \lambda x.\ x\ x : (\bot \cap (\bot \to \bot)) \to \bot}\,(\text{fun})$$

B. It can sometimes be frustrating that we cannot type certain expressions, even though we know they *should* be okay with respect to computation. For example, consider the following expression:

```
fun x → (fun y → x) (x x)
```

Even though we cannot apply `x` to itself, this shouldn't matter because the result of the application is never used. Consider the following type system:

```
<e> ::= <v> | fun <v> → <e> | <e> <e>
<ty> ::= empty | <ty> → <ty> | whatever
```

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ (\text{var}) \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \texttt{ → } e : \tau_1 \texttt{ → } \tau_2}\ (\text{fun}) \qquad \frac{\Gamma \vdash e_1 : \tau_2 \texttt{ → } \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ (\text{app})$$

$$\frac{}{\Gamma \vdash e : \texttt{whatever}}\ (\text{whatever})$$

Demonstrate the following judgment is derivable in the above system. Your derivation should be in standard form, and all inferences should be labeled with rule names.

$$\cdot \vdash \texttt{fun x → (fun y → x) (x x) : empty → empty}$$

$$\dfrac{\dfrac{\dfrac{}{\{x : \bot, y : W\} \vdash x : \bot}\ (\text{var})}{\{x : \bot\} \vdash \lambda y : x : W \to \bot}\ (\text{fun}) \qquad \dfrac{}{\{x : \bot\} \vdash x\,x : W}\ (\text{whatever})}{\{x : \bot\} \vdash (\lambda y.\, x)\,(x\,x) : \bot}\ (\text{app})}{\cdot \vdash \lambda x.\, (\lambda y.\, x)\,(x\,x) : \bot \to \bot}\ (\text{fun})$$

3 Linear Algebra?

A. A list `l` of type `'a list list` represents a valid $m \times n$ matrix (where $m \geq 0$ and $n \geq 0$) if `l` has m elements and every element of `l` has n elements. If this holds, then `l` represents a row in the matrix it represents. Note that `l` may be empty, or may contain empty lists.

Implement the function `mk_matrix` so that `mk_matrix l m n` is `Some a` if if `l` represents a valid $m \times n$ matrix `a`, and is `None` otherwise. You may use any function from the OCaml standard library.

```
type matrix = {
  entries : 'a list list;
  rows : int;
  cols : int;
}

val mk_matrix : 'a list list -> int -> int -> 'a matrix option
```

```
let rec length l =
  match l with
  | [] -> 0
  | _ :: l -> 1 + length l

let rec for_all f l =
  match l with
  | [] -> true
  | x :: l -> f x && forall f l

let mk_matrix l r c =
  if length l = m
     && for_all (fun l -> length l = n) l
  then Some { entries = l ; rows = r ; cols = c }
  else None
```

B. Implement the function `elem` so that `elem m (i, j)` is `Some a` if `(i, j)` is in-bounds for the matrix `m` and is `None` otherwise. You may use any function from the OCaml standard library.

```
val elem : 'a matrix -> (int * int) -> 'a option
```

```
let rec nth_opt l n =
    match l with
    | [] -> None
    | x :: l -> if n=0 then Some x else nth_opt l (n-1)

let elem a (i,j) =
    match nth_opt a i with
    | Some row -> nth_opt row j
    | None -> None
```

4 Constraint-Based Inference

One useful function that does not appear in OCaml standard library is the composition operator.

```
let (>>) (f : 'a -> 'b) (g : 'b -> 'c) : 'a -> c' = fun x -> g (f x)


let _ = assert ((((+) 1) >> ((+) 2)) 3 = 6)
```

Alternatively, we could add the operator '>>' as a primitive to our language from mini-project 3. Write the constraint-based inference rule for this operator. In other words, fill in the question marks (???) in the following (to be clear, there should be more than one premise).

$$\frac{???}{\Gamma \vdash e_1 \texttt{ >> } e_2 : ??? \dashv ???}\ (\text{comp})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv C_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv C_2 \qquad \alpha, \beta, \gamma \text{ are fresh}}{\Gamma \vdash e_1 \texttt{ >> } e_2 : \alpha \to \beta \dashv \tau_1 \doteq \alpha \to \gamma,\ \tau \doteq \gamma \to \beta,\ C_1,\ C_2}$$

5 Principle Types

Determine the principle type of the expression `fun x → k (x + 1)` in the context $\{\texttt{k} : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha\}$. You must show all your work. This means (1) constructing a derivation, (2) solving an unification problem, and (3) determining the principle type.

$\{k : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha\} \vdash \lambda x.\ k\ (x+1) : \gamma \to \eta \dashv \delta \to \varepsilon \to \delta \doteq int \to \eta,\ \gamma \doteq int,\ int \doteq int$

$\{k : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha,\ x : \gamma\} \vdash k\ (x+1) : \eta \dashv \delta \to \varepsilon \to \delta \doteq int \to \eta,\ \gamma \doteq int,\ int \doteq int$

$\{k : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha,\ x : \gamma\} \vdash k : \delta \to \varepsilon \to \delta \dashv \emptyset$

$\{k : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha,\ x : \gamma\} \vdash x + 1 : int \dashv \gamma \doteq int,\ int \doteq int$

$\{k : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha,\ x : \gamma\} \vdash x : \gamma \dashv \emptyset$

$\{k : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha,\ x : \gamma\} \vdash 1 : int \dashv \emptyset$

~~$\delta \to \varepsilon \doteq \delta \doteq int \to \eta$~~

~~$\gamma \doteq int$~~

~~$int \doteq int$~~

~~$\delta \doteq int$~~

~~$\varepsilon \to \delta \doteq \eta$~~

int

$$S = \left\{ \begin{array}{l} \gamma \mapsto int \\ \delta \mapsto int \\ \eta \mapsto \varepsilon \to int \end{array} \right\}$$

$S\tau = S(\gamma \to \eta) =$

$[\varepsilon \to int / \eta]\,[int/\delta]\,[int/\gamma]\,(\gamma \to \eta) =$

$\forall \varepsilon.\ int \to \varepsilon \to int$

6 Closures

One neat thing about closures is that they can be used to simulate data structures like lists. Consider the following OCaml code.

```
type int_list = int -> int option

let nil = fun n -> None
let cons x xs = fun n -> if n = 0 then Some x else xs (n - 1)

let l = cons 1 (cons 2 (cons 3 nil))
let _ = assert (l (-1) = None)
let _ = assert (l 0 = Some 1)
let _ = assert (l 1 = Some 2)
let _ = assert (l 2 = Some 3)
let _ = assert (l 4 = None)
```

First, we consider a slight modification of our semantic rules.

$$\frac{n \text{ is an integer}}{\langle\ \mathcal{E}\ ,\ n\ \rangle\ \Downarrow\ n}\ \text{int-eval} \qquad \frac{x \text{ is a variable}}{\langle\ \mathcal{E}\ ,\ x\ \rangle\ \Downarrow\ \mathcal{E}(x)}\ \text{var-eval}$$

$$\frac{}{\langle\ \mathcal{E}\ ,\ \texttt{fun}\ x\ \texttt{->}\ e\ \rangle\ \Downarrow\ (\!|\ \mathcal{E}[e]\ ,\ \texttt{fun}\ x\ \texttt{->}\ e\ |\!)}\ \text{(fun-eval)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle\ \Downarrow\ (\!|\ \mathcal{E}'\ ,\ \texttt{fun}\ x\ \texttt{->}\ e\ |\!) \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle\ \Downarrow\ v_2 \qquad \langle\ \mathcal{E}'[x \mapsto v_2]\ ,\ e_2\ \rangle\ \Downarrow\ v}{\langle\ \mathcal{E}\ ,\ e_1\ e_2\ \rangle\ \Downarrow\ v}\ \text{(app-eval)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle\ \Downarrow\ v_1 \qquad \langle\ \mathcal{E}[x \mapsto v_1]\ ,\ e_2\ \rangle\ \Downarrow\ v}{\langle\ \mathcal{E}\ ,\ \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \rangle\ \Downarrow\ v}\ \text{(let-eval)}$$

The only difference here is the environment $\mathcal{E}[e]$ for the closure in the (fun-eval) rule. We define $\mathcal{E}[e]$ to be the subset of mappings $(x \mapsto v)$ of $\mathcal{E}$ such that x is a free variable of e. In other words, $\mathcal{E}[e]$ small subset of the environment $\mathcal{E}$ required to evaluate e. For example,[1]

$$\{\texttt{x} \mapsto 1, \texttt{y} \mapsto 2, \texttt{z} \mapsto 3\}[\texttt{x + y}] = \{\texttt{x} \mapsto 1, \texttt{y} \mapsto 2\}$$

The problem continues on the next page.

[1] To be clear, this is meant to make the problem simpler. There are fewer things to keep track of within closures.

Using the semantics from the previous page, determine the value of the following expression. You don't need to give a derivation. But you should show your work and/or justify your answer.

```
let nil = fun n -> None in
let cons x xs = fun n -> if n = 0 then Some x else xs (n - 1) in
cons 3 (cons 10 nil)
```

$(\{x \mapsto 3, xs \mapsto (\{x \mapsto 10, xs \mapsto (\emptyset, \text{fun } n \to \text{None})\}, \text{fun } n \to \text{if } n=0 \text{ then Some } x \text{ else } xs\ (n-1))\}, \text{fun } n \to \text{if } n=0 \text{ then Some } x \text{ else } xs\ (n-1))$

Each call of cons evaluates to a closure whose env. has x mapped to the 'cons'ed element.