CS 320 Final Exam

Total: 100 pts

CS 320 Course Staff

Name: Nathan Mull

BU ID: 12345678

Instructions

- Please write your name and BU ID.
- This exam has 6 problems.
- Please read all problems before writing your solutions. You may find some problems easier than others.
- The total duration of the exam is 2 hrs.
- The typing/semantic rules that you need for each problem are provided **on the same page**. For these rules, sides conditions appear in boxes.

This page is intentionally left blank.

1 OCaml Programming

Recall the definition of rose trees:

```
type 'a rtree = Node of 'a * 'a rtree list
```

In this problem, you can use any of the following higher-order functions:

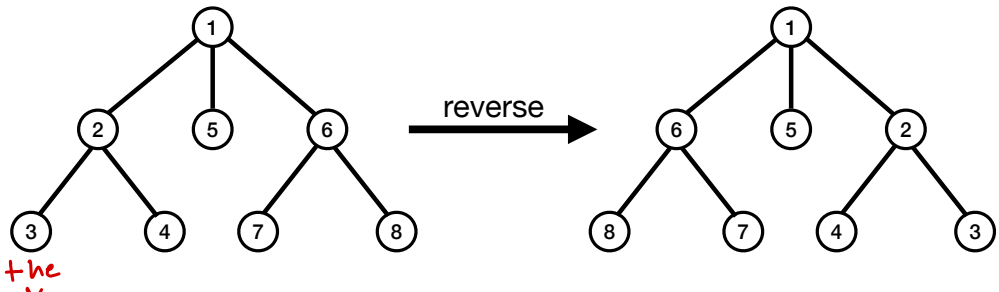
```
val List.map : ('a -> 'b) -> 'a list -> 'b list
val List.filter : ('a -> bool) -> 'a list -> 'a list
val List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
val List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

You cannot use any other functions from the standard library.

Problem 1 (20 pts) *Implement the function*

```
val reverse : 'a rtree -> 'a rtree
```

so that it reverses the order of nodes ~~in the tree~~, i.e., `reverse t` *is the mirror image of* `t` *about the y-axis:*



the

For example, following assertion holds:

```
let n x l = Node (x, l)
let test =
  reverse (n 1 [n 2 [n 3 []; n 4 []]; n 5 []; n 6 [n 7 []; n 8 []]])
  =       n 1 [n 6 [n 8 []; n 7 []]; n 5 []; n 2 [n 4 []; n 3 []]]
let _ = assert test
```

Solution.

```
let rev l = List.fold_left (fun xs x -> x :: xs) [] l
let rev =  (* alternative *)
  let rec go l r =
    match r with
    | [] -> l
    | x::xs -> go (x :: l) xs
  in go []
let rec rererse (Node (x, l)) =
  Node (x, rev (List.map rererse l))
```

Solution. (Contd.)

2 Formal Grammar

Consider the following grammar for an toy imperative language with references.

```
<p>  ::= <w> = <e> <p> | <w> = <e> | <e>
<e>  ::= <e2> <o> <e> | <e2>
<e2> ::= & <w> | <w> | ( <e> ) | 1 | 2 | 3
<o>  ::= + | - | * | /
<w>  ::= * <w> | x | y | z
```

This grammar expresses that a program (<p>) is a sequence of assignment statements (<w> = <e>) ending with an optional expression (<e>).[1] For example, the following

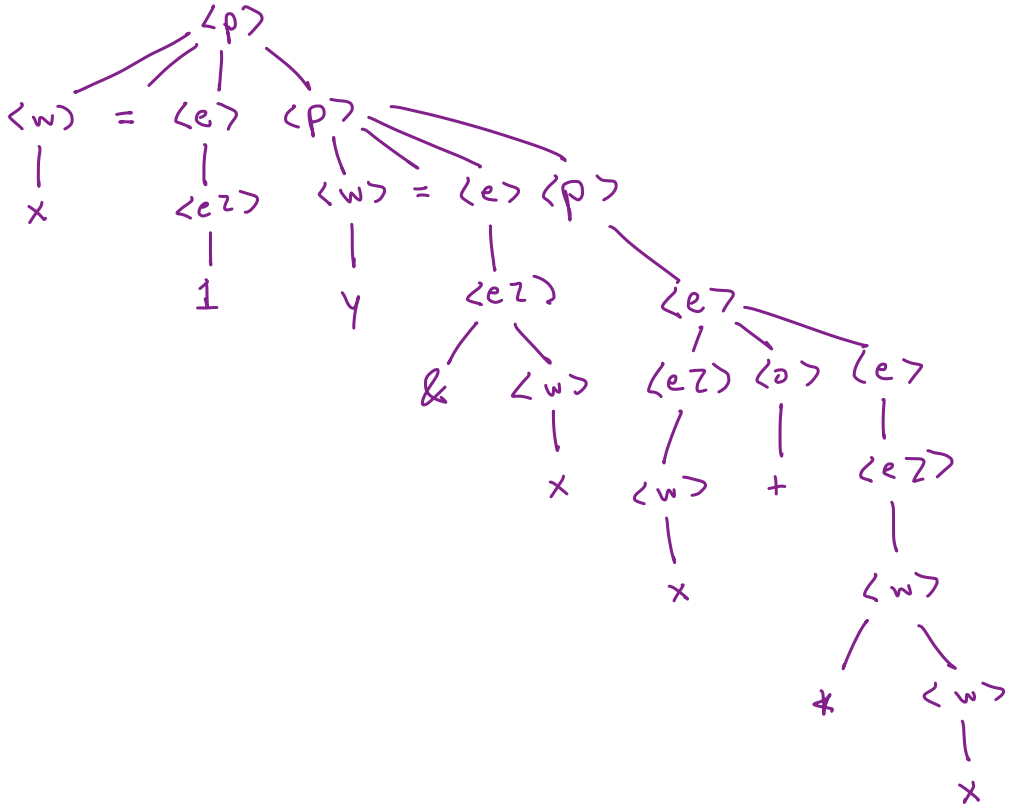
```
x = 1
y = &x
x + *y
```

is a well-formed program according to this grammar.[2]:

Problem 2 (15 pts) *Draw a parse tree for the following program according to the above grammar.*

```
x = 1
y = &x
x + *y
```

Solution.



[1] This is similar to how Rust works.

[2] The meaning of this program doesn't matter for this problem, but we might understand this program to be saying: x is assigned to 1, and then y is assigned to a reference to x, and then the output of the program is x plus the value that y refers to, i.e., y is *dereferenced*.

Repeated Grammar:

```
<p>  ::=  <w> = <e> <p> | <w> = <e> | <e>
<e>  ::=  <e2> <o> <e> | <e2>
<e2> ::=  & <w> | <w> | ( <e> ) | 1 | 2 | 3
<o>  ::=  + | - | * | /
<w>  ::=  * <w> | x | y | z
```

Repeated Program:

```
x = 1
y = &x
x + *y
```

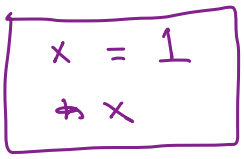
Solution. (Contd.)

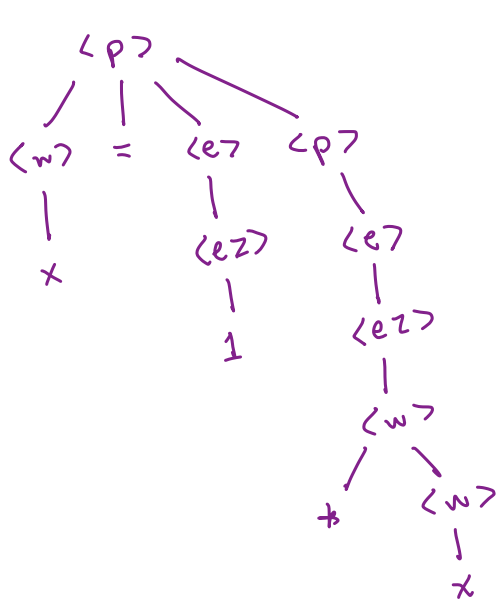
Repeated Grammar:

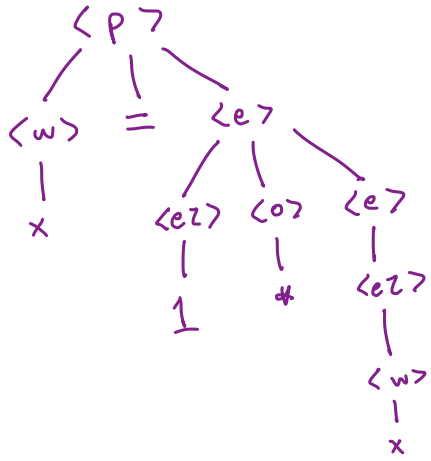
```
<p>  ::= <w> = <e> <p> | <w> = <e> | <e>
<e>  ::= <e2> <o> <e> | <e2>
<e2> ::= & <w> | <w> | ( <e> ) | 1 | 2 | 3
<o>  ::= + | - | * | /
<w>  ::= * <w> | x | y | z
```

Problem 3 (10 pts) *Demonstrate that the above grammar is ambiguous by giving a program that has multiple parse trees. Write down the program and draw two distinct parse trees for it.*

Solution.







3 Message Passing Semantics

Recall the following constructs for sending and receiving messages on channels:

- '$\mathsf{let}\ (tx, rx) = \mathsf{new\_channel}\ \mathsf{in}\ e$' creates a new channel with two ends: the variable tx is the sender's end, which can be used to send messages on the channel, and the variable rx is the receiver's end, which can be used to receive messages on the channel;
- '$\mathsf{send}\ e_1\ \mathsf{on}\ tx\ \mathsf{in}\ e$' evaluates the expression e_1 and sends its value on tx;
- '$\mathsf{let}\ x = \mathsf{recv}\ rx\ \mathsf{in}\ e$' receives a value on rx and binds it to variable x.

In this problem, you'll be looking at the *semantics* of message passing. We take judgments to be of the form $\langle \mathcal{E}, e\rangle \Downarrow v$, where the environment $\mathcal{E}$ not only contains variable-value bindings, but also channel-buffer bindings. Formally, $\mathcal{E}$ is additionally allowed to have bindings of the form $(tx, rx) \mapsto l$ where l is a list of values. Whenever a message is sent on tx, it gets added to the back of this list, and whenever a message is received on rx, it gets removed from the front of the list (i.e., the list acts as a queue).

The semantic rules for these constructs are as follows:

$$\frac{\langle \mathcal{E}[(tx, rx) \mapsto [\,]], e\rangle \Downarrow v}{\langle \mathcal{E}, \mathsf{let}\ (tx, rx) = \mathsf{new\_channel}\ \mathsf{in}\ e\rangle \Downarrow v}\ \textsc{NewChan}$$

$$\frac{\langle \mathcal{E}, e_1\rangle \Downarrow v_1 \qquad \boxed{\mathcal{E}((tx, rx)) = \ell} \qquad \langle \mathcal{E}[(tx, rx) \mapsto \ell \mathbin{@} [v_1]], e\rangle \Downarrow v}{\langle \mathcal{E}, \mathsf{send}\ e_1\ \mathsf{on}\ tx\ \mathsf{in}\ e\rangle \Downarrow v}\ \textsc{Send}$$

$$\frac{\boxed{\mathcal{E}((tx, rx)) = v_1 :: \ell} \qquad \langle \mathcal{E}[(tx, rx) \mapsto \ell][x \mapsto v_1], e\rangle \Downarrow v}{\langle \mathcal{E}, \mathsf{let}\ x = \mathsf{recv}\ rx\ \mathsf{in}\ e\rangle \Downarrow v}\ \textsc{Recv}$$

Additional Rules Needed:

$$\frac{\boxed{\mathcal{E}(x) = v}}{\langle \mathcal{E}, x\rangle \mathcal{E} v}\ \textsc{Var} \qquad \frac{\boxed{n \text{ is an integer literal}}}{\langle \mathcal{E}, n\rangle \Downarrow \mathtt{n}}\ \textsc{IntLit}$$

Problem 4 (20 pts) *Provide a derivation demonstrating that the following expression evaluates to* 5.

```
let (tx, rx) = new_channel in
send 5 on tx in
let x = recv rx in
x
```

Solution.

$$\dfrac{\dfrac{}{\langle \{(tx, rx) \mapsto [\,], x \mapsto 5\}, x\rangle \Downarrow 5}}{\dfrac{\dfrac{}{\langle \{(tx, rx \mapsto [\,]\}, 5\rangle \Downarrow 5} \qquad \langle \{(tx, rx) \mapsto [5]\}, \mathsf{let}\ x = \mathsf{recv}\ rx\ \mathsf{in}\ x\rangle \Downarrow 5}{\langle \{(tx, rx) \mapsto [\,]\}, \mathsf{send}\ 5\ \mathsf{on}\ tx\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{recv}\ rx\ \mathsf{in}\ x\rangle \Downarrow 5}}{\langle \emptyset, \mathsf{let}\ (tx, rx) = \mathsf{new\_channel}\ \mathsf{in}\ \mathsf{send}\ 5\ \mathsf{on}\ tx\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{recv}\ rx\ \mathsf{in}\ x\rangle \Downarrow 5}$$

Repeated Rules:

$$\frac{\boxed{\mathcal{E}(x) = v}}{\langle \mathcal{E}, x \rangle \mathcal{E} v}\ \textsc{Var} \qquad \frac{\boxed{n \text{ is an integer literal}}}{\langle \mathcal{E}, n \rangle \Downarrow \texttt{n}}\ \textsc{IntLit} \qquad \frac{\langle \mathcal{E}[(tx, rx) \mapsto [\,]], e \rangle \Downarrow v}{\langle \mathcal{E}, \mathsf{let}\ (tx, rx) = \mathsf{new\_channel}\ \mathsf{in}\ e \rangle \Downarrow v}\ \textsc{NewChan}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \qquad \boxed{\mathcal{E}((tx, rx)) = \ell} \qquad \langle \mathcal{E}[(tx, rx) \mapsto \ell \mathbin{@} [v_1]], e \rangle \Downarrow v}{\langle \mathcal{E}, \mathsf{send}\ e_1\ \mathsf{on}\ tx\ \mathsf{in}\ e \rangle \Downarrow v}\ \textsc{Send}$$

$$\frac{\boxed{\mathcal{E}((tx, rx)) = v_1 :: \ell} \qquad \langle \mathcal{E}[(tx, rx) \mapsto \ell][x \mapsto v_1], e \rangle \Downarrow v}{\langle \mathcal{E}, \mathsf{let}\ x = \mathsf{recv}\ rx\ \mathsf{in}\ e \rangle \Downarrow v}\ \textsc{Recv}$$

Repeated Expression:

```
let (tx, rx) = new_channel in
send 5 on tx in
let x = recv rx in
x
```

Solution. (Contd.)

This page is intentionally left blank.

4 Polymorphism

Problem 5 (10 pts) *Determine an OCaml expression with the following type:*

```
(('a -> 'b) -> 'b) -> 'c) -> 'a -> 'c
```

Solution.

```
fun f x -> f (fun g -> g x)
```

Problem 6 (25 pts) *Determine the principle type of the following expression.*

$$\mathsf{fun}\ x \to \mathsf{fun}\ y \to x\ y\ y$$

To receive full credit, you must provide:

1. *A derivation of the judgment* $\cdot \vdash \mathsf{fun}\ x \to \mathsf{fun}\ y \to x\ y\ y : \tau \dashv \mathcal{C}$ *for some type* τ *and constraints* $\mathcal{C}$*. You may write your derivation in compact form.*
2. *A step-by-step demonstration of the solving of constraints* $\mathcal{C}$ *using the unification algorithm we discussed in class.*
3. *The principle type of the above expression, clearly demarcated in your solution.*

This problem requires the following rules:

$$\frac{(x:\sigma) \in \Gamma \qquad \boxed{\sigma \text{ is a monotype}}}{\Gamma \vdash x : \sigma \dashv \varnothing}\ \textsc{VarMono} \qquad \frac{\boxed{\alpha \text{ is fresh}} \qquad \Gamma, x:\alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \mathsf{fun}\ x \to e : \alpha \to \tau \dashv \mathcal{C}}\ \textsc{Fun}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \boxed{\alpha \text{ is fresh}}}{\Gamma \vdash e_1 e_2 : \alpha \dashv \mathcal{C}_1, \mathcal{C}_2}\ \textsc{App}$$

Solution.

(1) $\cdot \vdash \mathsf{fun}\ x \to \mathsf{fun}\ y \to x\ y\ y : \boxed{\alpha \to \beta \to \delta}$ (τ) $\dashv \boxed{\eta \doteq \beta \to \delta,\ \alpha \doteq \beta \to \eta}$ (C)

- $\{x:\alpha\} \vdash \mathsf{fun}\ y \to x\ y\ y : \beta \to \delta \dashv \eta \doteq \beta \to \delta,\ \alpha \doteq \beta \to \eta$
  - $\{x:\alpha, y:\beta\} \vdash x\ y\ y : \delta \dashv \eta \doteq \beta \to \delta,\ \alpha \doteq \beta \to \eta$
    - $\{x:\alpha, y:\beta\} \vdash x\ y : \eta \dashv \alpha \doteq \beta \to \eta$
      - $\{x:\alpha, y:\beta\} \vdash x : \alpha \dashv \emptyset$
      - $\{x:\alpha, y:\beta\} \vdash y : \beta \dashv \emptyset$
    - $\{x:\alpha, y:\beta\} \vdash y : \beta \dashv \emptyset$

(2) ~~$\eta \doteq \beta \to \delta$~~, ~~$\alpha \doteq \beta \to \eta$~~, ~~$\beta \to \delta$~~

$$S = \{\eta \mapsto \beta \to \delta,\ \alpha \mapsto \beta \to \beta \to \delta\}$$

alternative:

~~$\alpha \doteq \beta \to \eta$~~, ~~$\eta \doteq \beta \to \delta$~~

$$S' = \{\alpha \mapsto \beta \to \eta,\ \eta \mapsto \beta \to \delta\}$$

(3) $S(\alpha \to \beta \to \delta) = (\beta \to \beta \to \delta) \to \beta \to \delta$

$$\boxed{\forall \beta.\, \forall \delta.\, (\beta \to \beta \to \delta) \to \beta \to \delta}$$ Principle type

Rules Repeated:

$$\frac{(x : \sigma) \in \Gamma \qquad \boxed{\sigma \text{ is a monotype}}}{\Gamma \vdash x : \sigma \dashv \varnothing}\ \textsc{VarMono} \qquad \frac{\boxed{\alpha \text{ is fresh}} \qquad \Gamma, x : \alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \mathsf{fun}\ x \to e : \alpha \to \tau \dashv \mathcal{C}}\ \textsc{Fun}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \boxed{\alpha \text{ is fresh}}}{\Gamma \vdash e_1 e_2 : \alpha \dashv \mathcal{C}_1, \mathcal{C}_2}\ \textsc{App}$$

Expression Repeated:

$$\mathsf{fun}\ x \to \mathsf{fun}\ y \to x\ y\ y$$

Solution. (Contd.)

This page is intentionally left blank.