CMSC330 - Organization of Programming Languages
Spring 2025- FinalSolutions

CMSC330 Course Staff
University of Maryland
Department of Computer Science

Name: _____

UID: _____

I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination

Signature: _____

Ground Rules

- Please write legibly. **If we cannot read your answer you will not receive credit.**
- You may use anything on the accompanying reference sheet anywhere on this exam
- Please remove the reference sheet from the exam
- You may not leave the room or hand in your exam within the last 10 minutes of the exam
- If anything is unclear, ask a proctor. If you are still confused, write down your assumptions in the margin
- Do no take photos of this exam or share this exam in anyway shape or form
- If you need extra space, the last page is for scratch work. Make a note for the grader to check the scratch page if you want it graded.
- NOTE: there is a page for scratch work at the end of the exam. You may use this freely, just don't tear it off.

Question	Points
P1.	10
P2.	5
P3.	6
P4.	4
P5.	6
P6.	4
P7.	10
P8.	10
P9.	8
P10.	6
P11.	8
P12.	8
P13.	10
P14.	5
EC	2
Total	100 + 2

Problem 1: Concepts

[Total 10 pts]

	true	false
Every CFG can be represented by an NFA	T	**F**
If $A <: B$ and $B <: C$ then $A <: C$	**T**	F
In Rust, every variable is immutable by default	**T**	F
OCaml is dynamically typed because it uses type inference	T	**F**
Regular expressions are necessary to code a parser.	T	**F**
Safe Rust will never produce runtime errors.	T	**F**
By fixing all logic bugs, you have a secure program.	T	**F**
Higher order functions are specific to OCaml.	T	**F**
Property based testing cannot be used in conjunction with unit testing.	T	**F**
In OCaml, anywhere we use map, we could use fold instead.	**T**	F
If $A <: B$ and $B <: C$ then $A <: C$	**T**	F
Every CFG can be represented by an NFA	T	**F**
In Rust, every variable is immutable by default	**T**	F
OCaml is dynamically typed because it uses type inference	T	**F**
Regular expressions are necessary to code a parser.	T	**F**
Safe Rust will never produce runtime errors.	T	**F**
By fixing all logic bugs, you have a secure program.	T	**F**
Higher order functions are specific to OCaml.	T	**F**
Property based testing cannot be used in conjunction with unit testing.	T	**F**
In OCaml, anywhere we use map, we could use fold instead.	**T**	F
Safe Rust will never be the source of a buffer overflow attack	**T**	F
If $A <: B$ and $C <: A$ then $A <: C$	T	**F**
Rust and OCaml are both statically typed	**T**	F
A parser will not accept meaningless programs	T	**F**

Every Regular Expression has an equivalent DFA	T	F
A Context Free Grammar could describe strings of length 6 which include balanced parenthesis	T	F
When writing an evaluator, the specification of the grammar is more important than the operational semantics	T	F
Regular Expressions can only be found in OCaml	T	F
You could theoretically implement project 4 in lambda calculus	T	F
(λx.x x)(λx.x x) can be beta reduced to (λx. x x)	T	F

Problem 2: Regex

[Total 5 pts]

Which of the following strings are accepted by the regular expression below?

$$\lambda^*\delta\sigma|[\omega\beta]\{2\}$$

Select `NONE` if none of the first five (5) options match.

(A) $\delta\sigma$ (B) $\omega\beta\omega\beta$ (C) $\lambda\lambda\sigma\beta\beta$ (D) $\omega\beta$ (E) $\delta\omega\lambda$ (F) NONE

Problem 3: Regex

[Total 5 pts]

Which of the following strings are accepted by the regular expression below?

$$\lambda^*\delta\sigma|[\omega\beta]\{2\}$$

Select `NONE` if none of the first five (5) options match.

(A) $\delta\sigma$ (B) $\omega\beta\omega\beta$ (C) $\lambda\lambda\sigma\beta\beta$ (D) $\omega\beta$ (E) $\delta\omega\lambda$ (F) NONE

Problem 4: Regex

[Total 5 pts]

Which of the following strings are accepted by the regular expression below?

$$\lambda^*\delta\sigma?[\omega\beta]\{2\}$$

Select `NONE` if none of the first five (5) options match.

(A) $\lambda\lambda\sigma\beta\beta$ (B) $\omega\beta\omega\beta$ (C) $\delta\sigma$ (D) $\omega\beta$ (E) $\delta\omega\lambda$ (F) NONE

Problem 5: Property Based Testing

[Total 6 pts]

Consider the following incorrect `mapip` function for a list of i32s in Rust. It is meant to apply the function to each i32 element in a Vec, modifying the input Vec and not returning anything new.

```
fn mapip<F: Fn(i32) -> i32>(f: F, v: &mut Vec<i32>) -> Vec<i32> {
    let mut r = v.clone();
    for mut item in &mut r {
        *item = f(*item);
    }
    r
}
```

Consider the following property *p* about the `mapip` function:

p : A Vec should have different values before and after it is passed into `mapip`.

Using a **correct** implementation of `mapip`, this property *p* should hold true for all valid inputs?

(●) Yes ( ) No

Using **our** implementation of `mapip`, this property *p* should hold true for all valid inputs?

(●) Yes ( ) No

Suppose I encode this property in Rust as the following:

```
    #[test]
     fn test<F: Fn(i32) -> i32>(f: F, v: Vec<i32>) {
         let initial = v.clone();
         mapip(f,v);
         assert_eq!(v, initial);
     }
```

The above `test` function is a valid encoding of the property *p*.

( ) Yes (●) No

Problem 6: Property Based Testing

[Total 6 pts]

Consider the following incorrect `mapip` function for a list of i32s in Rust. It is meant to apply the function to each i32 element in a Vec, modifying the input Vec and not returning anything new.

```
fn mapip<F: Fn(i32) -> i32>(f: F, v: &mut Vec<i32>) -> Vec<i32> {
    let mut r = v.clone();
    for mut item in &mut r {
        *item = f(*item);
    }
    r
}
```

Consider the following property *p* about the `mapip` function:

p : A Vec should have different values before and after it is passed into `mapip`.

Using a **correct** implementation of `mapip`, this property *p* should hold true for all valid inputs?

(●) Yes ( ) No

Using **our** implementation of `mapip`, this property *p* should hold true for all valid inputs?

(●) Yes ( ) No

Suppose I encode this property in Rust as the following:

```
#[test]
 fn test<F: Fn(i32) -> i32>(f: F, v: Vec<i32>) {
     let initial = v.clone();
     mapip(f,v);
     assert_eq!(v, initial);
 }
```

The above `test` function is a valid encoding of the property *p*.

( ) Yes (●) No

Problem 7: Property Based Testing

[Total 6 pts]

Consider the following incorrect `mapip` function for a list of i32s in Rust. It is meant to apply the function to each i32 element in a Vec, modifying the input Vec and not returning anything new.

```
fn mapip<F: Fn(i32) -> i32>(f: F, v: &mut Vec<i32>) -> Vec<i32> {
    let mut r = v.clone();
    for mut item in &mut r {
        *item = f(*item);
    }
    r
}
```

Consider the following property *p* about the `mapip` function:

p : A Vec should have different values before and after it is passed into `mapip`.

Using a **correct** implementation of `mapip`, this property *p* should hold true for all valid inputs?

(●) Yes ( ) No

Using **our** implementation of `mapip`, this property *p* should hold true for all valid inputs?

(●) Yes ( ) No

Suppose I encode this property in Rust as the following:

```
#[test]
 fn test<F: Fn(i32) -> i32>(f: F, v: Vec<i32>) {
     let initial = v.clone();
     mapip(f,v);
     assert_eq!(v, initial);
 }
```

The above `test` function is a valid encoding of the property p.

Yes ● No

Problem 8: CFG Creation

[Total 4 pts]

Which Context Free Grammar(s) are equivalent to the regex: `c*(ab)+d`?

Select all that apply.

(A)
$$\begin{aligned} U &-> MUD|abU \\ M &-> cM|\epsilon \\ D &-> d|\epsilon \end{aligned}$$

(B)
$$\begin{aligned} U &-> MDC \\ D &-> abD|ab \\ M &-> cM|\epsilon \\ C &-> d|\epsilon \end{aligned}$$

(C)
$$\begin{aligned} U &-> MDC \\ D &-> abD|ab \\ M &-> cM|\epsilon \\ C &-> dC|\epsilon \end{aligned}$$

(D)
$$\begin{aligned} U &-> MUD|abU \\ M &-> c|M|\epsilon \\ D &-> d|\epsilon \end{aligned}$$

Problem 9: CFG Creation

[Total 4 pts]

Which Context Free Grammar(s) are equivalent to the regex: `c*(ab)+d`?

Select all that apply.

(B) $U->\ MDC$, $D->\ abD|ab$, $M->\ cM|\epsilon$, $C->\ d|\epsilon$

(A) $U->\ MUD|abU$, $M->\ cM|\epsilon$, $D->\ d|\epsilon$

(D) $U->\ MUD|abU$, $M->\ c|M|\epsilon$, $D->\ d|\epsilon$

(C) $U->\ MDC$, $D->\ abD|ab$, $M->\ cM|\epsilon$, $C->\ dC|\epsilon$

Problem 10: CFG Creation

[Total 4 pts]

Which Context Free Grammar(s) are equivalent to the regex: `[ab]+(c?d)*`

Select all that apply.

(A) $S->\ abS|abT$
$T->\ cdT|dT|\epsilon$

(B) $S->\ aS|bS|aT|bT$
$T->\ dT|cdT|\epsilon$

(C) $S->\ TF$
$T->\ aT|bT|a|b$
$F->\ UF|\epsilon$
$U->\ cd|d$

(D) $S->\ abS|abT|T$
$T->\ cdT|dT|\epsilon$

Problem 11: OCaml and Rust Typing

[Total 6 pts]

Give the type of the expression. If there is a type error, put "ERROR"

```
(* Ocaml *)
fun x ->
    let (a,b) = x in
    fun y ->
    let a = (a+1, b > true) in
    (a::[y])
```

int * bool $\Longrightarrow$ $int * bool \Longrightarrow (int * bool) list$

```
// Rust
{
    let a = if false {
        true > false
    } else {
        false
    };
    let b = true;
    (a, b)
}
```

(bool, bool)

Problem 12: Ocaml and Rust Evaluation

[Total 4 pts]

Evaluate the following expressions. If there is a compilation error, put "ERROR"

```
(* Ocaml *)
let rec f x = match x with
  [] -> 3
 |x::xs -> List.fold_left x (f xs) [1;2;3] in

f [(fun a b -> a + b)]
```

9

```
// Rust
fn f1(x: i32, y: i32) -> i32 {
    x + y
}

fn f2(x: i32, y: i32) -> i32 {
    x * y
}
...
{
    let mut x = vec![-4, -2, 3];
    let mut a = true;
    for i in x.iter_mut() {
        if a {
            *i = f1(*i, *i);
            a = false;
        } else {
            *i = f2(*i, *i);
            a = true;
        }
    }
    x
}
```

[-8, 4, 6]

Problem 13: OCaml and Rust Typing

[Total 6 pts]

Give the type of the expression. If there is a type error, put "ERROR"

```rust
// Rust
{
    let a = if false {
        true > false
    } else {
        false
    };
    let b = true;
    (a, b)
}
```

(bool, bool)

```ocaml
(* Ocaml *)
fun x ->
    let (a,b) = x in
    fun y ->
    let a = (a+1, b > true) in
    (a::[y])
```

int * bool $\Longrightarrow$ $int * bool \Longrightarrow (int * bool) list$

Problem 14: Ocaml and Rust Evaluation

[Total 4 pts]

Evaluate the following expressions. If there is a compilation error, put "ERROR"

```ocaml
(* Ocaml *)
let rec f x = match x with
  [] -> 3
 |x::xs -> List.fold_left x (f xs) [1;2;3] in

f [(fun a b -> a + b)]
```

9

```rust
// Rust
fn f1(x: i32, y: i32) -> i32 {
    x + y
}

fn f2(x: i32, y: i32) -> i32 {
    x * y
}
...
{
    let mut x = vec![-4, -2, 3];
    let mut a = true;
    for i in x.iter_mut() {
        if a {
            *i = f1(*i, *i);
            a = false;
        } else {
            *i = f2(*i, *i);
            a = true;
        }
    }
    x
}
```

[-8, 4, 6]

Problem 15: OCaml and Rust Typing

[Total 6 pts]

Give the type of the expression. If there is a type error, put "ERROR"

```
(* Ocaml *)
fun x ->
    let y = match x with
         [] -> x
        |(a,b)::xs -> (b,a)::xs
    in y
```

```
// Rust
{
    let x = true;
    let a = if x || false {
        4 + 2
    } else {
        false
    };
    let b = 6;
    a == b
}
```

Problem 16: Ocaml and Rust Evaluation

[Total 4 pts]

Evaluate the following expressions. If there is a compilation error, put "ERROR"

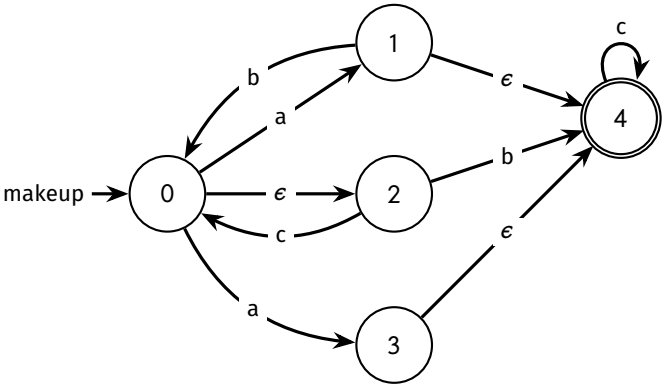
```
(* Ocaml *)
let rec f x = match x with
  [] -> []
 |x::xs ->
 (List.map (fun a -> a::[x]) [x])@f xs in

f [(fun a b -> a + b)]
```

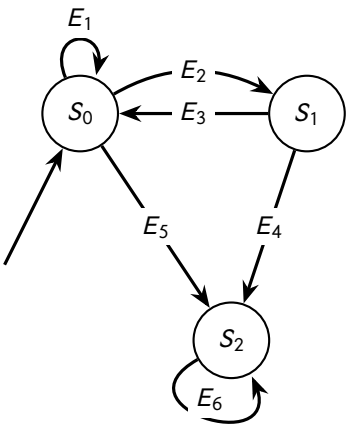
```
// Rust
fn f1(x: i32, y: i32) -> bool {
    x == y % 2
}
...
{
    let mut x = vec![(1,3),(2,4),(0,5)];
    let mut res = vec![];
    for i in x.into_iter() {
        match i{
            (a,b)=>res.push(f1(a,b))
        }
    };
    res
}
```

Problem 17: NFA to DFA

[Total 10 pts]



Convert the above NFA to the below DFA. You **MUST** show your work to get any credit. (You will need to remove the Garbage state)



Scratch Space:

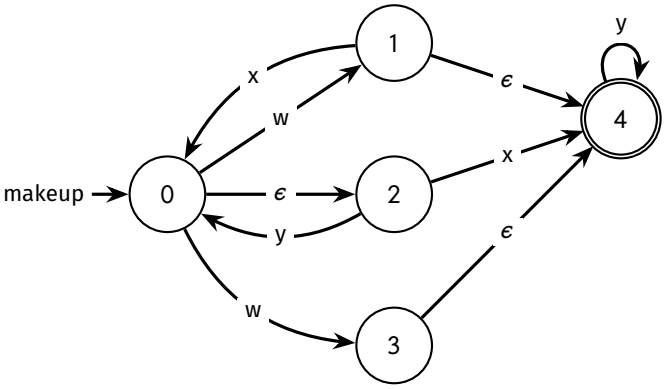
S_0:	0,3,4	S_1:	1,2,3,4	S_2:	4
E_1:	a	E_2:	b	E_3:	c
E_4:	a	E_5:	d	E_6:	a

(a) Which states are the final (accepting) states? Select all that apply [2 pts]

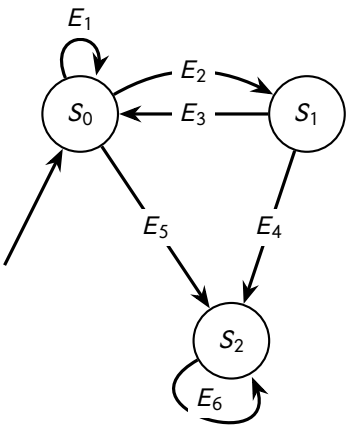
A State S_0 B State S_1 C State S_2

Problem 18: NFA to DFA

[Total 10 pts]



Convert the above NFA to the below DFA. You **MUST** show your work to get any credit. (You will need to remove the Garbage state)



Scratch Space:

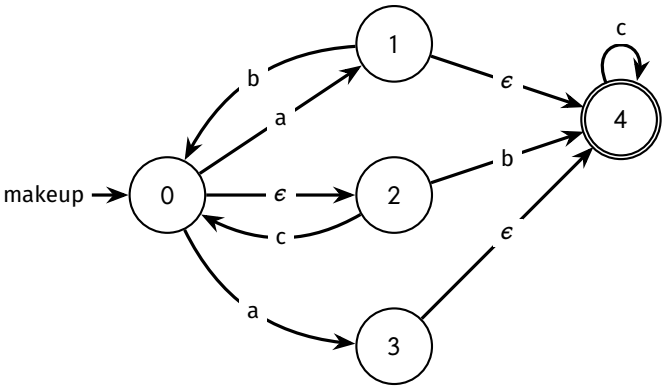
S_0:	0,3,4	S_1:	1,2,3,4	S_2:	4
E_1:	w	E_2:	x	E_3:	y
E_4:	w	E_5:	z	E_6:	w

(a) Which states are the final (accepting) states? Select all that apply [2 pts]

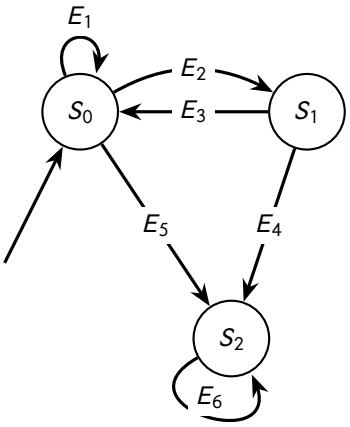
A State S_0 B State S_1 C State S_2

Problem 19: NFA to DFA

[Total 10 pts]



Convert the above NFA to the below DFA. You **MUST** show your work to get any credit. (You will need to remove the Garbage state)



Scratch Space:

S_0: S_1: S_2:

E_1: E_2: E_3:

E_4: E_5: E_6:

(a) Which states are the final (accepting) states? Select all that apply [2 pts]

A State S_0 B State S_1 C State S_2

Problem 20: Lexing, Parsing, Interpreting

[Total 10 pts]

Given the following CFG, and assuming the **Ocaml** type system and semantics, at what stage of language processing would each expression **fail**? Mark **'Valid'** if the expression would be accepted by the grammar and evaluate successfully. Assume the only symbols allowed are those found in the grammar.

$$E \rightarrow \text{let } S = E \text{ in } E \mid \text{let ref } S = E \text{ ; } E \mid R$$
$$R \rightarrow R \text{ <> } R \mid S := R \mid M$$
$$M \rightarrow M * M \mid M - M \mid S$$
$$S \rightarrow 1 \mid 2 \mid 3 \mid true \mid false \mid x \mid (E)$$

For all x ∈ S , x is a lowercase English character.

	Lexer	Parser	Evaluator	Valid
`let x = x <> false in x`	L	P	**E**	V
`true := 3`	L	P	**E**	V
`let 2 - 1 = 1 in 3`	L	**P**	E	V
`let ref a = 1 * 4; a := 5`	**L**	P	E	V
`if true then false else 3`	**L**	P	E	V

Problem 21: Lexing, Parsing, Interpreting

[Total 10 pts]

Given the following CFG, and assuming the **Ocaml** type system and semantics, at what stage of language processing would each expression **fail**? Mark **'Valid'** if the expression would be accepted by the grammar and evaluate successfully. Assume the only symbols allowed are those found in the grammar.

$$E \rightarrow \text{let } S = E \text{ in } E \mid \text{let ref } S = E \text{ ; } E \mid R$$
$$R \rightarrow R \text{ <> } R \mid S := R \mid M$$
$$M \rightarrow M * M \mid M - M \mid S$$
$$S \rightarrow 1 \mid 2 \mid 3 \mid true \mid false \mid x \mid (E)$$

For all x ∈ S , x is a lowercase English character.

	Lexer	Parser	Evaluator	Valid
`let 2 - 1 = 1 in 3`	L	**P**	E	V
`if true then false else 3`	**L**	P	E	V
`let ref a = 1 * 4; a := 5`	**L**	P	E	V
`true := 3`	L	P	**E**	V
`let x = x <> false in x`	L	P	**E**	V

Problem 22: Lexing, Parsing, Interpreting

[Total 10 pts]

Given the following CFG, and assuming the **Ocaml** type system and semantics, at what stage of language processing would each expression **fail**? Mark **'Valid'** if the expression would be accepted by the grammar and evaluate successfully. Assume the only symbols allowed are those found in the grammar.

$$E \rightarrow \text{let } S = E \text{ in } E \mid \text{let ref } S = E \text{ ; } E \mid R$$
$$R \rightarrow R \text{ <> } R \mid S := R \mid M$$
$$M \rightarrow M * M \mid M - M \mid S$$
$$S \rightarrow 1 \mid 2 \mid 3 \mid true \mid false \mid x \mid (E)$$

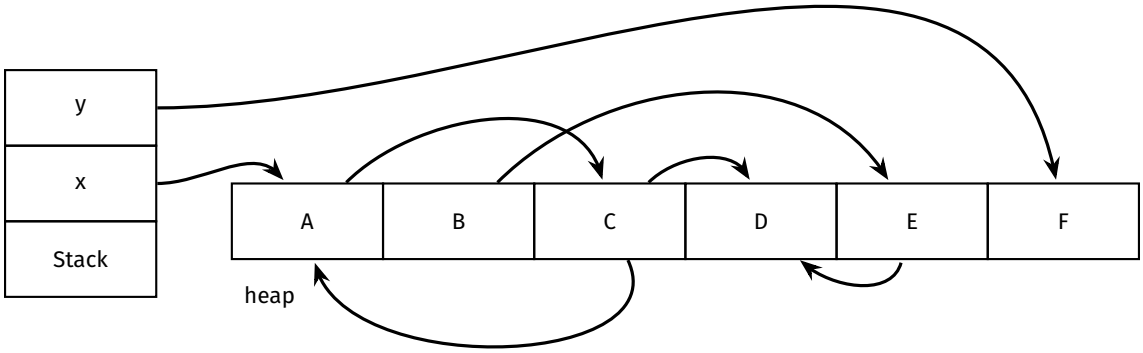
For all x ∈ S , x is a lowercase English character.

	Lexer	Parser	Evaluator	Valid
`1 * 2 * 3`	L	P	E	**V**
`let x = (1 != 2) in x`	**L**	P	E	V
`a := let x = 1 in 1`	L	**P**	E	V
`let a = true in a - 1`	L	P	**E**	V
`((true)`	L	**P**	E	V

Problem 23: Garbage Collection

[Total 6 pts]

Given the following memory diagram:



(a) Select each piece of memory that should be marked as freed after calling Mark and Sweep at the time of this diagram. [3 pts]

Free A (A) Free B (B) Free C (C) Free D (D) Free E (E) Free F (F) None (G)

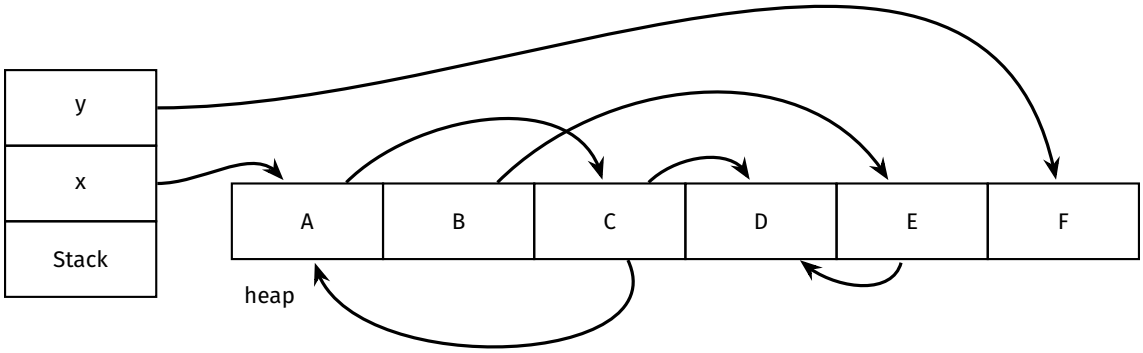
(b) Select each piece of memory that should be marked as freed at the time of this diagram using Reference Counting. [3 pts]

Free A (A) Free B (B) Free C (C) Free D (D) Free E (E) Free F (F) None (G)

Problem 24: Garbage Collection

[Total 6 pts]

Given the following memory diagram:



(a) Select each piece of memory that should be marked as freed after calling Mark and Sweep at the time of this diagram. [3 pts]

Free A (A) Free B (B) Free C (C) Free D (D) Free E (E) Free F (F) None (G)

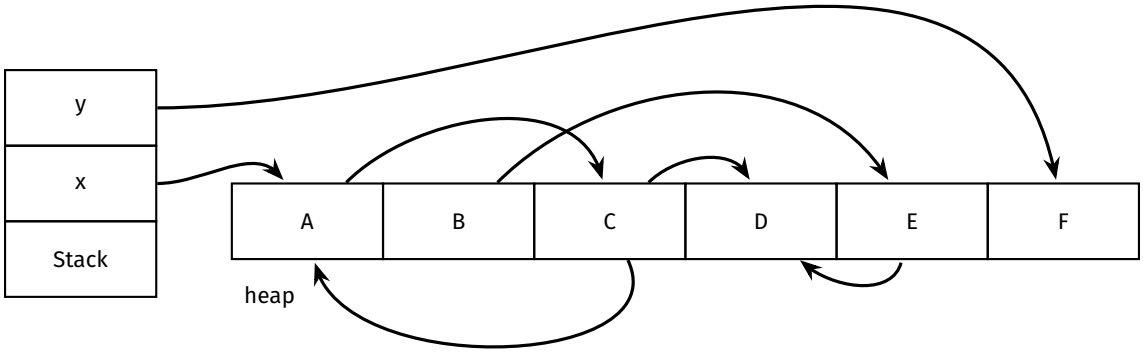
(b) Select each piece of memory that should be marked as freed at the time of this diagram using Reference Counting. [3 pts]

Free A (A) Free B (B) Free C (C) Free D (D) Free E (E) Free F (F) None (G)

Problem 25: Garbage Collection

[Total 6 pts]

Given the following memory diagram:



(a) Select each piece of memory that should be marked as freed after calling Mark and Sweep at the time of this diagram. [3 pts]

Free A (A) Free B (B) Free C (C) Free D (D) Free E (E) Free F (F) None (G)

(b) Select each piece of memory that should be marked as freed at the time of this diagram using Reference Counting. [3 pts]

Free A (A) Free B (B) Free C (C) Free D (D) Free E (E) Free F (F) None (G)

Problem 26: Lambda Calculus

[Total 8 pts]

(a) Reduce [3 pts]

Order the following statements to produce a correct reduction of the following expression. The reduction you choose MUST use **lazy evaluation**. Write the letter corresponding to each step in the boxes on the right. **Note that you may not need to fill every box.**

$$(\lambda c.\ a\ b\ (c\ a))\ ((\lambda a.\ c)\ (\lambda c.\ (\lambda a.\ b)))$$

A. $c\ a$

B. $a\ b\ (\lambda a.\ (\lambda c.\ (\lambda a.\ b))\ a)$

C. $a\ b\ ((\lambda a.\ c)\ (\lambda c.\ (\lambda a.\ b))\ a)$

D. $((\lambda a.\ c)\ (\lambda c.\ (\lambda a.\ b)))\ a$

E. $a\ b\ (c\ a)$

F. $a\ b\ (\lambda c.\ (\lambda a.\ b))$

G. $(\lambda c.\ a\ b\ (c\ a))\ ((\lambda c.\ (\lambda a.\ b)))$

H. $(\lambda c.\ a\ b\ (c\ a))\ \ c$

I. $(a\ b\ ((\lambda a.\ c)\ a))\ (\lambda a.\ b)$

J. $(a\ b\ ((\lambda a.\ c)\ a))\ (\lambda c.\ (\lambda a.\ b))$

B
F

(b) Free Variables: [3 pts]

Circle the free variables in the expression below:

$$(\lambda a.\ (\lambda b.\ \text{c})(\lambda c.\ c)(\text{b}\ \text{b}(\lambda c.\ \text{b})))$$

(c) Alpha Equivalence: [2 pts]

Which of the following are alpha equivalent to the following expression (from part b): $(\lambda a.\ (\lambda b.\ \text{c})(\lambda c.\ c)(\text{b}\ \text{b}(\lambda c.\ \text{b})))$? **Select all that apply.**

- (A) $(\lambda a.\ (\lambda a.\ c)(\lambda a.\ a)(b\ b(\lambda a.\ b)))$
- (B) $(\lambda b.\ (\lambda c.\ a)(\lambda a.\ a)(c\ c(\lambda a.\ c)))$
- (C) $(\lambda d.\ (\lambda f.\ c)(\lambda g.\ g)(b\ b(\lambda h.\ b)))$
- (D) $(\lambda c.\ (\lambda a.\ c)(\lambda b.\ b)(b\ b(\lambda a.\ b)))$

Problem 27: Lambda Calculus

[Total 8 pts]

(a) Reduce [3 pts]

Order the following statements to produce a correct reduction of the following expression. The reduction you choose MUST use **eager evaluation**. Write the letter corresponding to each step in the boxes on the right. **Note that you may not need to fill every box.**

$$(\lambda f.\ d\ e\ (f\ d))\ ((\lambda d.\ f)\ (\lambda f.\ (\lambda d.\ e)))$$

A. $f\ d$

B. $d\ e\ (\lambda d.\ (\lambda f.\ (\lambda d.\ e))\ d)$

C. $d\ e\ ((\lambda d.\ f)\ (\lambda f.\ (\lambda d.\ e))\ d)$

D. $((\lambda d.\ f)\ (\lambda f.\ (\lambda d.\ e)))\ d$

E. $d\ e\ (f\ d)$

F. $d\ e\ (\lambda f.\ (\lambda d.\ e))$

G. $(\lambda f.\ d\ e\ (f\ d))\ ((\lambda f.\ (\lambda d.\ e)))$

H. $(\lambda f.\ d\ e\ (f\ d))\ f$

I. $(d\ e\ ((\lambda d.\ f)\ d))\ (\lambda d.\ e)$

J. $(d\ e\ ((\lambda d.\ f)\ d))\ (\lambda f.\ (\lambda d.\ e))$

D
F

(b) Free Variables: [3 pts]

Circle the free variables in the expression below:

$$(\lambda d.\ (\lambda e.\ \text{(f)})(\lambda f.\ f)(\text{(e)}\ \text{(e)}(\lambda f.\ \text{(e)})))$$

(c) Alpha Equivalence: [2 pts]

Which of the following are alpha equivalent to the following expression (from part b): $(\lambda d.\ (\lambda e.\ \text{(f)})(\lambda f.\ f)(\text{(e)}\ \text{(e)}(\lambda f.\ \text{(e)})))$? **Select all that apply.**

- (A) $(\lambda d.\ (\lambda d.\ f)(\lambda d.\ d)(e\ e(\lambda d.\ e)))$
- (B) $(\lambda e.\ (\lambda f.\ d)(\lambda d.\ d)(f\ f(\lambda d.\ f)))$
- (C) $(\lambda d.\ (\lambda f.\ f)(\lambda g.\ g)(e\ e(\lambda h.\ e)))$
- (D) $(\lambda f.\ (\lambda d.\ f)(\lambda e.\ e)(e\ e(\lambda d.\ e)))$

Problem 28: Lambda Calculus

[Total 8 pts]

(a) Reduce [3 pts]

Order the following statements to produce a correct reduction of the following expression. The reduction you choose MUST use **eager evaluation**. Write the letter corresponding to each step in the boxes on the right. **Note that you may not need to fill every box.**

$$(\lambda m.\ j\ k\ (m\ j))\ ((\lambda j.\ m)\ (\lambda m.\ (\lambda j.\ k)))$$

A. $m\ j$

B. $j\ k\ (\lambda j.\ (\lambda m.\ (\lambda j.\ k))\ j)$

C. $j\ k\ ((\lambda j.\ m)\ (\lambda m.\ (\lambda j.\ k))\ j)$

D. $((\lambda j.\ m)\ (\lambda m.\ (\lambda j.\ k)))\ j$

E. $j\ k\ (m\ j)$

F. $j\ k\ (\lambda m.\ (\lambda j.\ k))$

G. $(\lambda m.\ j\ k\ (m\ j))\ ((\lambda m.\ (\lambda j.\ k)))$

H. $(\lambda m.\ j\ k\ (m\ j))\ \ m$

I. $(j\ k\ ((\lambda j.\ m)\ j))\ (\lambda j.\ k)$

J. $(j\ k\ ((\lambda j.\ m)\ j))\ (\lambda m.\ (\lambda j.\ k))$

H
E

(b) Free Variables: [3 pts]

Circle the free variables in the expression below:

$$(\lambda j.\ (\lambda k.\ \text{m})(\lambda m.\ m)(\text{k}\ \ \text{k}(\lambda m.\ \text{k})))$$

(c) Alpha Equivalence: [2 pts]

Which of the following are alpha equivalent to the following expression (from part b): $(\lambda j.\ (\lambda k.\ \text{m})(\lambda m.\ m)(\text{k}\ \ \text{k}(\lambda m.\ \text{k})))$? **Select all that apply.**

- (A) $(\lambda j.\ (\lambda j.\ m)(\lambda j.\ j)(k\ k(\lambda j.\ k)))$
- (B) $(\lambda k.\ (\lambda m.\ j)(\lambda j.\ j)(m\ m(\lambda j.\ m)))$
- (C) $(\lambda d.\ (\lambda f.\ m)(\lambda g.\ g)(k\ k(\lambda h.\ k)))$
- (D) $(\lambda m.\ (\lambda j.\ m)(\lambda k.\ k)(k\ k(\lambda j.\ k)))$

Problem 29: Operational Semantics

[Total 8 pts]

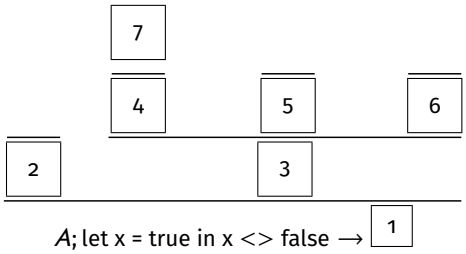
Consider the following rules of OCaml:

$$\frac{}{A; true \rightarrow true} \quad \frac{}{A; false \rightarrow false} \quad \frac{}{A; n \rightarrow n} \quad \frac{A(x) = v}{A; x \rightarrow v}$$

$$\frac{A; e_1 \rightarrow v_1 \quad A, x : v_1;\ e_2 \rightarrow v_2}{A; \text{let } x\ =\ e_1 \text{ in } e_2 \rightarrow v_2} \qquad \frac{A; e_1 \rightarrow v_1 \quad A; e_2 \rightarrow v_2 \quad v_3\ is\ v_1\ <>\ v_2}{A; e_1\ <>\ e_2 \rightarrow v_3}$$

Using OCaml as the metalanguage, prove the following sentence evaluates.
IMPORTANT: you must fill in the blanks to receive credit.

let x = true in x <> false



Scratch Space:

Blank 1: $true$

Blank 2: $A; true \rightarrow true$

Blank 3: $A, x : true; x <> false \rightarrow true$

Blank 4: $A, x : true; x \rightarrow true$

Blank 5: $A, x : true; false \rightarrow false$

Blank 6: $true\ is\ true\ <>\ false$

Blank 7: $A, x : true(x) = true$

Problem 30: Operational Semantics

[Total 8 pts]

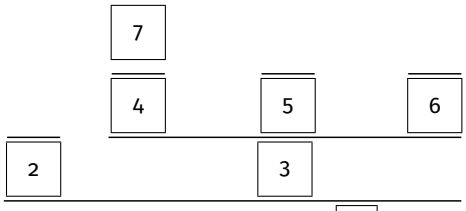
Consider the following rules of OCaml:

$$\frac{}{A; true \rightarrow true} \quad \frac{}{A; false \rightarrow false} \quad \frac{}{A; n \rightarrow n} \quad \frac{A(x) = v}{A; x \rightarrow v}$$

$$\frac{A; e_1 \rightarrow v_1 \quad A, x : v_1; \ e_2 \rightarrow v_2}{A; \text{let } x \ = \ e_1 \text{ in } e_2 \rightarrow v_2} \qquad \frac{A; e_1 \rightarrow v_1 \quad A; e_2 \rightarrow v_2 \quad v_3 \ is \ v_1 <> v_2}{A; e_1 \ <> \ e_2 \rightarrow v_3}$$

Using OCaml as the metalanguage, prove the following sentence evaluates.
IMPORTANT: you must fill in the blanks to receive credit.

let x = 1 in x <> 2



A; let x = 1 in x <> 2 $\rightarrow$ 1

Scratch Space:

Blank 1: 1

Blank 2: $A; 1 \rightarrow 1$

Blank 3: $A, x : 1; x <> 2 \rightarrow true$

Blank 4: $A, x : 1; x \rightarrow 1$

Blank 5: $A, x : 1; 2 \rightarrow 2$

Blank 6: $true \ is \ 1 \ <> \ 2$

Blank 7: $A, x : 1(x) = 1$

Problem 31: Operational Semantics

[Total 8 pts]

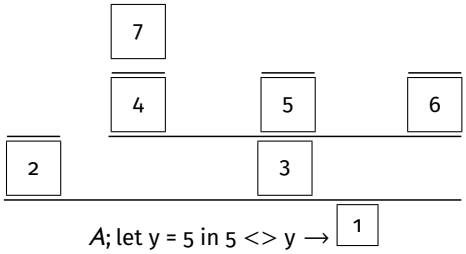
Consider the following rules of OCaml:

$$\frac{}{A; true \rightarrow true} \quad \frac{}{A; false \rightarrow false} \quad \frac{}{A; n \rightarrow n} \quad \frac{A(x) = v}{A; x \rightarrow v}$$

$$\frac{A; e_1 \rightarrow v_1 \quad A, x : v_1; \ e_2 \rightarrow v_2}{A; \text{let } x \ = \ e_1 \text{ in } e_2 \rightarrow v_2} \qquad \frac{A; e_1 \rightarrow v_1 \quad A; e_2 \rightarrow v_2 \quad v_3 \ is \ v_1 <> v_2}{A; e_1 \ <> \ e_2 \rightarrow v_3}$$

Using OCaml as the metalanguage, prove the following sentence evaluates.
IMPORTANT: you must fill in the blanks to receive credit.

let y = 5 in 5 <> y



A; let y = 5 in 5 <> y $\rightarrow$ 1

Scratch Space:

Blank 1: $bool$

Blank 2: $G \vdash 5 : int$

Blank 3: $G, x : int \vdash x <> 7 : bool$

Blank 4: $G, x : int \vdash x : int$

Blank 5: $G, x : int \vdash 7 : int$

Blank 6: $G, x : int(x) = int$

Blank 7:

Problem 32: Ownership and Lifetimes

[Total 8 pts]

```
1 fn main(){
2   let mut x = String::from("hello");
3   let y = &x;
4   println!("{},{}",x,y);
5 }
```

Does the code compile? (Y) Yes (F) No

If **no**, explain why not in one sentence:

```
1 fn main(){
2   let x = String::from("Hello");
3   let mut y = &x;
4   y.push_str(" world");
5   println!("{}",x);
6 }
```

Does the code compile? (Y) Yes (F) No

If **no**, explain why not in one sentence:

y is not borrowed as mutable. so cannot pushΩ_str

```
1 fn main(){
2   let mut x = String::from("Hello");
3   let _a = &x;
4   let y = &mut x;
5   y.push_str(" world");
6   println!("{}",x);
7 }
```

Does the code compile? (Y) Yes (F) No

If **no**, explain why not in one sentence:

```
1  fn function(s1: String,
2              s2: String,
3              f:bool)->usize{
4      if f {s1.len()} else{s2.len()}
5  }

6  fn main(){
7      let a = String::from("hello");
8      let b = a.clone();
9      let c = function(b,a,true);
10     println!("{} has length {}",a,c);
11 }
```

Does the code compile? (Y) Yes (F) No

If **no**, explain why not in one sentence:

a is no longer owner after function call

Problem 33: Ownership and Lifetimes

[Total 8 pts]

```
1 fn main(){
2   let mut x = String::from("hello");
3   let y = &x;
4   println!("{},{}",x,y);
5 }
```

Does the code compile? (Y) Yes (F) No

If **no**, explain why not in one sentence:

```
1 fn main(){
2   let x = String::from("Hello");
3   let mut y = &x;
4   y.push_str(" world");
5   println!("{}",x);
6 }
```

Does the code compile? (Y) Yes (F) No

If **no**, explain why not in one sentence:

y is not borrowed as mutable. so cannot pushstr

```
1 fn main(){
2   let mut x = String::from("Hello");
3   let _a = &x;
4   let y = &mut x;
5   y.push_str(" world");
6   println!("{}",x);
7 }
```

Does the code compile? (Y) Yes (F) No

If **no**, explain why not in one sentence:

```
1  fn function(s1: String,
2              s2: String,
3              f:bool)->usize{
4      if f {s1.len()} else{s2.len()}
5  }

6  fn main(){
7      let a = String::from("hello");
8      let b = a.clone();
9      let c = function(b,a,true);
10     println!("{} has length {}",a,c);
11 }
```

Does the code compile? (Y) Yes (F) No

If **no**, explain why not in one sentence:

a is no longer owner after function call

Problem 34: Ownership and Lifetimes

[Total 8 pts]

```
1 fn main(){
2   let x = String::from("Hello");
3   let mut y = &mut x;
4   y.push_str(" world");
5   println!("{}",x);
6 }
```

Does the code compile? (Y) Yes (F) No

If **no**, explain why not in one sentence:

a is no longer owner after function call

```
1 fn main(){
2   let mut x = String::from("hello");
3   let y = &mut x;
4   println!("{},{}",x,y);
5 }
```

Does the code compile? (Y) Yes (F) No

If **no**, explain why not in one sentence:

```
1  fn function(s1: String,
2              s2: String,
3              f:bool)->usize{
4      if f {s1.len()} else{s2.len()}
5  }

6  fn main(){
7      let a = String::from("hello");
8      let b = a.clone();
9      let c = function(b,a,true);
10     println!("{} has length {}",a,c);
11 }
```

Does the code compile? (Y) Yes (F) No

If **no**, explain why not in one sentence:

y is not borrowed as mutable. so cannot pushstr

```
1 fn main(){
2   let mut x = String::from("Hello");
3   let _a = &x;
4   let y = &mut x;
5   println!("{}",x);
6   y.push_str(" world");
7 }
```

Does the code compile? (Y) Yes (F) No

If **no**, explain why not in one sentence:

Problem 35: Ocaml Coding [Total 10 pts]

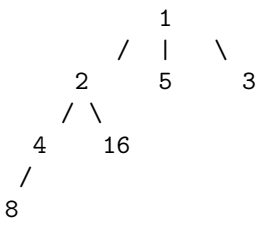
Restrictions: You are **not** allowed to use imperative OCaml; you can define recursive helper/helper functions below the function signatures we provided. You are not allowed to use any List module functions except the ones already given to you on the cheat sheet, otherwise you may use anything in `StdLib`.

Write a function called `largestsumpath` that, given a tree, returns a list of the path from the root to the node in the tree that gives the largest sum when each value on the path is added up.

If there are multiple tied sums, you can return any one path.
Return an empty list if the tree is empty.

```
type 'a tree = Petal | Stem of 'a * 'a tree list
```

Example tree t:




```
let t = Stem (1,
            [Stem (2,
                [Stem (4,
                    [Stem (8, [Petal])
                ]);
                Stem (6, [Petal])
            ]);
            Stem (3, [Petal]);
            Stem (5, [Petal])
            ])
```

Expected Output:
`largest_sum_path t = [1;2;16]`

Write your code on the next page.

```
let rec largest_sum_path t =

match t with
  Petal -> []
  |Stem(curr, children) ->
  match (List.map (fun x -> curr::(longest_path x)) children) with
[] -> failwith "not possible"
|x::xs -> List.fold_left (fun a x -> if List.length x > List.length a then x else a) x xs
```

Problem 36: Ocaml Coding

[Total 10 pts]

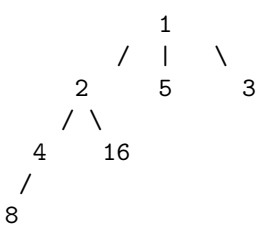
Restrictions: You are **not** allowed to use imperative OCaml; you can define recursive helper/helper functions below the function signatures we provided. You are not allowed to use any List module functions except the ones already given to you on the cheat sheet, otherwise you may use anything in `StdLib`.

Write a function called `largestsumpath` that, given a tree, returns a list of the path from the root to the node in the tree that gives the largest sum when each value on the path is added up.

If there are multiple tied sums, you can return any one path.
Return an empty list if the tree is empty.

```
type 'a tree = External | Internal of 'a * 'a tree list
```

Example tree t:




```
let t = Internal (1,
            [Internal (2,
                [Internal (4,
                    [Internal (8, [External])
                ]);
                Internal (6, [External])
            ]);
            Internal (3, [External]);
            Internal (5, [External])
            ])
```

Expected Output:
`largest_sum_path t = [1;2;16]`

Write your code on the next page.

```
let rec largest_sum_path t =

match t with
  Petal -> []
  |Stem(curr, children) ->
  match (List.map (fun x -> curr::(longest_path x)) children) with
[] -> failwith "not possible"
|x::xs -> List.fold_left (fun a x -> if List.length x > List.length a then x else a) x xs
```

Problem 37: Ocaml Coding

[Total 10 pts]

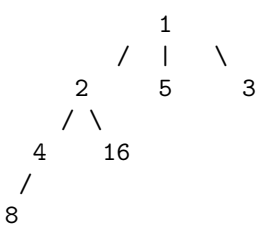
Restrictions: You are **not** allowed to use imperative OCaml; you can define recursive helper/helper functions below the function signatures we provided. You are not allowed to use any List module functions except the ones already given to you on the cheat sheet, otherwise you may use anything in `StdLib`.

Write a function called `largestsumpath` that, given a tree, returns a list of the path from the root to the node in the tree that gives the largest sum when each value on the path is added up.

If there are multiple tied sums, you can return any one path.
Return an empty list if the tree is empty.

```
    type int tree = Petal | Stem of int * int tree list
```

Example tree t:




```
let t = Stem (1,
            [Stem (2,
                [Stem (4,
                    [Stem (8, [Petal])
                ]);
                Stem (16, [Petal])
            ]);
            Stem (3, [Petal]);
            Stem (5, [Petal])
            ])
```

Expected Output:
`largest_sum_path t = [1;2;16]`

Write your code on the next page.

```
let rec largest_sum_path t =

match t with
  Petal -> []
  |Stem(curr, children) ->
  match (List.map (fun x -> curr::(longest_path x)) children) with
[] -> failwith "not possible"
|x::xs -> List.fold_left (fun a x -> if List.length x > List.length a then x else a) x xs
```

Problem 38: Rust Coding

[Total 5 pts]

Restrictions: You may not use Rust's built in map or fold for this function.

Recall the higher order function `fold` in OCaml. We are going to write a version of it in Rust, though we are restricting the types of the input vec and accumulator to u32s rather than generics for simplicity.

f in the type signature is a function that takes in 2 u32s (one an element of the Vec and one an accumulator) and returns a u32.

Note: You can do `foldleft` or `foldright` but `foldleft` is probably easier.

Example

```
fn add (e: u32, mut a : u32) -> u32 {
    a + e
}

fn main() {
    let mut v = vec![1,2,3];
    let mut a = 0;
    let x = fold (add, v, a);
    println!("{:?}", x); // prints 6
}
```

Write your code for fold below:

```
fn fold (f: impl Fn(u32, u32) -> u32, v: Vec<u32>, mut a: u32) -> u32 {
    for i in v {
        a = f(i, a);
    }
    a









}
```

Problem 39: Rust Coding

[Total 5 pts]

Restrictions: You may not use Rust's built in map or fold for this function.

Recall the higher order function `fold` in OCaml. We are going to write a version of it in Rust, though we are restricting the types of the input vec and accumulator to u32s rather than generics for simplicity.

f in the type signature is a function that takes in 2 u32s (one an element of the Vec and one an accumulator) and returns a u32.

Note: You can do `foldleft` or `foldright` but `foldleft` is probably easier.

Example

```
fn add (e: u32, mut a : u32) -> u32 {
    a + e
}

fn main() {
    let mut v = vec![1,2,3];
    let mut a = 0;
    let x = fold (add, v, a);
    println!("{:?}", x); // prints 6
}
```

Write your code for fold below:

```
fn fold (f: impl Fn(u32, u32) -> u32, v: Vec<u32>, mut a: u32) -> u32 {
    for i in v {
        a = f(i, a);
    }
    a

}
```

Problem 40: Rust Coding

[Total 5 pts]

Restrictions: You may not use Rust's built in map or fold for this function.

Consider the higher order function `filter`. `filter` works by taking in a function and an list, and returning a new list of all the values in the list which cause the function to return true. We will modify this slightly. We will first restrict the types of the input list (vector here), to u32. Second, we want it to occur in place so we don't have to return a new vector.

f in the type signature is a function that takes in a u32 (the element of the Vec) and returns a bool.

Example

```
fn pos (e: u32) -> bool {
    e > 0
}

fn main() {
    let mut v = vec![1,-2,3];
    myfilter(pos,&mut v);
    println!("{:?}", v); // prints [1,3]
}
```

Write your code for myfilter below:

```
fn myfilter (f: impl Fn(u32) -> bool, v: Vec<u32>){
    for i in v {
        a = f(i, a);
    }
    a









}
```

Problem 41: Extra Credit

[Total 2 pts]

For the following, you may only receive up to 2 points regardless of how many answers you give.

(a) Staff Stalking [1 pts]

What is your discussion TA's name and what is your discussion's section number?

(b) Staff Stalking [1 pts]

If your Discussion TA were an animal what would they be?

(c) Colon Parenthesis [1 pts]

Write a poem!

Problem 42: Extra Credit

[Total 2 pts]

For the following, you may only receive up to 2 points regardless of how many answers you give.

(a) Staff Stalking [1 pts]

What is your discussion TA's name and what is your discussion's section number?

(b) Staff Stalking [1 pts]

If your Discussion TA were an animal what would they be?

(c) Colon Parenthesis [1 pts]

Write a poem!

Problem 43: Extra Credit

[Total 2 pts]

For the following, you may only receive up to 2 points regardless of how many answers you give.

(a) Staff Stalking [1 pts]

What is your discussion TA's name and what is your discussion's section number?

(b) Staff Stalking [1 pts]

If your Discussion TA were an animal what would they be?

(c) Colon Parenthesis [1 pts]

Write a poem!

For Scratch Work - Do not tear off. Indicate the problem with your work

Cheat Sheet

OCaml

```
(* Map and Fold *)
(* ('a -> 'b) -> 'a list -> 'b list *)
let rec map f l = match l with
   [] -> []
  |x::xs -> (f x)::(map f xs)

(* ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a *)
let rec fold_left f a l = match l with
   [] -> a
  |x::xs -> fold_left f (f a x) xs

(* ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b *)
let rec fold_right f l a = match l with
   [] -> a
  |x::xs -> f x (fold_right f xs a)

(* 'a list -> int *)
List.length: Returns the length
(number of elements) of the given list.
```

```
(* Regex in OCaml *)
Re.Posix.re: string -> regex
Re.compile: regex -> compiled_regex

Re.exec: compiled_regex -> string -> group
Re.execp: compiled_regex -> string -> bool
Re.exec_opt: compiled_regex -> string -> group option

Re.matches: compiled_regex -> string -> string list

Re.Group.get: group -> int -> string
Re.Group.get_opt: group -> int -> string option
```

```
(* OCaml Function Types *)
:: -: 'a -> 'a list -> 'a list

@ -: 'a list -> 'a list -> 'a list

+, -, *, /  -: int -> int -> int
+., -., *., /. -: float -> float -> float

&&, || -: bool -> bool -> bool
not -: bool -> bool

^ -: string -> string -> string

=>,>,=,<,<= :- 'a -> 'a -> bool
```

Structure of Regex

$$
\begin{array}{rcl}
R & \rightarrow & \emptyset \\
 & | & \sigma \\
 & | & \epsilon \\
 & | & RR \\
 & | & R|R \\
 & | & R^* \\
\end{array}
$$

Regex

*	zero or more repetitions of the preceding character or group
+	one or more repetitions of the preceding character or group
?	zero or one repetitions of the preceding character or group
.	any character
$r_1 \vert r_2$	r_1 or r_2 (eg. a|b means 'a' or 'b')
[abc]	match any character in abc
[^r_1]	anything except r_1 (eg. [^abc] is anything but an 'a', 'b', or 'c')
[r_1-r_2]	range specification (eg. [a-z] means any letter in the ASCII range of a-z)
{n}	exactly n repetitions of the preceding character or group
{n,}	at least n repetitions of the preceding character or group
{m,n}	at least m and at most n repetitions of the preceding character or group
^	start of string
$	end of string
(r_1)	capture the pattern r_1 and store it somewhere (match group in Python)
\d	any digit, same as [0-9]
\s	any space character like \n, \t, \r, \f, or space

NFA to DFA Algorithm (Subset Construction Algorithm)

NFA (input): $(\Sigma, Q, q_0, F_n, \delta)$, DFA (output): $(\Sigma, R, r_0, F_d, \delta_n)$

$R \leftarrow \{\}$
$r_0 \leftarrow \epsilon - \text{closure}(\sigma, q_0)$
while $\exists$ an unmarked state $r \in R$ **do**
  mark r
  for all $a \in \Sigma$ **do**
    $E \leftarrow \text{move}(\sigma, r, a)$
    $e \leftarrow \epsilon - \text{closure}(\sigma, E)$
    if $e \notin R$ **then**
      $R \leftarrow R \cup \{e\}$
    end if
    $\sigma_n \leftarrow \sigma_n \cup \{r, a, e\}$
  end for
end while
$F_d \leftarrow \{r \mid \exists s \in r \text{ with } s \in F_n\}$

Rust

```
// Vectors
let vec = Vec::new();  // makes a new vector
let vec1 = vec![1,2,3]

vec.push(ele);  // Pushes the element 'ele'
                // to end of the vector 'vec'
// Strings
let string = String::from("Hello");

string.push_str(&str); // appends the str
                       // to string

vec.to_iter();  // returns an iterator for vec

vec.iter_mut(); // returns an iterator that
                // allows modifying each value.

string.chars()  // returns an iterator of chars
                // over the a string

iter.rev();       // reverses an iterators direction

iter.next();      // returns an Option of the next
                  // item in the iterator.

struct Building{   // example of struct
    name:String,
    floors:i32,
    locationx:f32,
    locationy:f32,
}

enum Option<T>{ Some(T); None } //enum Option type
option.unwrap();  // returns the item in an Option or
                  // panics if None
```