CMSC 330 Final Exam Spring 2022 Solutions

Q2. PL Concepts

Q2.1. Checking physical equality for cyclic data structures using references in OCaml is not possible. T/F

Q2.2. `Rc<RefCell<T>>` and `RefCell<Rc<T>>` can be used interchangeably in Rust. T/F

Q2.3. An invariant, or an explicit logical assertion, is required to implement property-based tests for a given function. T/F

Q2.4. Which of following explains the role of shrinking in Property-based Testing?

- Shows the part of the counterexample that caused the test to fail
- The process by which the next arbitrary example is generated
- Simplifies the counterexample as much as possible

Q2.5. Which of the following is the closest to interfaces in Java?

- Types in OCaml
- Traits in Rust
- Structs in Rust
- Objects in Ruby

Q2.6. Which of the following is true about tail recursion in OCaml? Select all that apply.

- Tail recursion involves starting recursion from the tail of the list
- Tail recursion enables us to write more efficient recursive functions
- Both `fold_left` and `fold_right` are tail recursive functions
- All recursive functions can be rewritten as tail recursive functions

Q2.7. Lambda calculus allows us to implement which of the following features? Select all that apply.

- Defining variables
- Type checking
- Arithmetic operations
- Recursion

Q3. Ruby: Regular Expressions

Q3.1. As part of your unpaid internship this summer, you are tasked with implementing a banking system that processes all transactions at the end of the day. Suppose all transactions are stored safely in a text file as lines with the following format: `<Vendor ID>, <Charge>`.

We will define a valid line as follows:

- The **Vendor ID** consists of one or more uppercase letters.
- Commas should also be followed by a single space.
- The **Charge** consists of a sequence of one or more digits. The charge may be preceded by a negative sign if the amount needs to be deducted from the balance.

Examples of Valid Lines:

```
MUSIC, -10
RENT, -1500
REFUND, 500
```

`ACDC, 1000`

Write a regex that will exactly match transactions as described above.

`/^[A-Z]+, -?\d+$/`

Q3.2. **Recap:** A Fibonacci sequence is a sequence of numbers starting with 0 and 1, where each term is the sum of the two numbers preceding it. For example, the Fibonacci sequence of length 6 is `0,1,1,2,3,5`.

Is it possible to write a regex to **exactly** match all Fibonacci sequences of length ≤ 20 and no other strings? Justify your answer.

Yes. Since the Fibonacci sequences are finite and known, we can concatenate them together using | or write a regex that looks like `/0(,1(,1(,2(,3(,5(...))?)?)?)?)?)?/`. Note that writing the regex was not required for full credit.

Q4. Ruby: Coding

The Chinese Zodiac is represented by 12 zodiac animals, the Rat, Ox, Tiger, Rabbit, Dragon, Snake, Horse, Goat, Monkey, Rooster, Dog, and Pig, and every year is associated with one of the zodiac animals. For example, 2022 is the year of the Tiger.

Suppose we have a file **zodiac.txt** containing the zodiac animal associated with every year from 1981 to 2028. Each line in the file will have the format: `<Zodiac Animal>, <Years>`. Assume that all lines in the file are valid and there are no duplicate entries for any zodiac animal.

We will define a line in **zodiac.txt** as follows:

- The **Zodiac Animal** starts with an uppercase letter, followed by one or more lowercase letters.
- Commas should always be followed by a single space.
- The **Years** are a sequence of four-digit numbers, separated by a comma and a space.
- You can assume that there will only be 4 years in the sequence.

Contents of zodiac.txt:

```
Rat, 2020, 1996, 1984, 2008
Ox, 1997, 2009, 1985, 2021
Tiger, 1998, 1986, 2022, 2010
Rabbit, 1999, 2011, 2023, 1987
Dragon, 1988, 2000, 2012, 2024
Snake, 2025, 1989, 2013, 2001
Horse, 1990, 2026, 2014, 2002
Goat, 2015, 2027, 1991, 2003
Monkey, 2004, 2016, 2028, 1992
Rooster, 1981, 2017, 2005, 1993
Dog, 2018, 1982, 1994, 2006
Pig, 1983, 2019, 2007, 1995
```

You will have to implement four functions, described below:

1. **`initialize(filename):`** Reads the file and parses its contents. You may store the contents of the file in any data structure you prefer, as long as the other three functions described below work as expected.

2. **`get_years(zodiac):`** Returns a **sorted** array of all the years (as integers) associated with the inputted zodiac. In the case of invalid input, return `nil`. For example: Cat does not belong to the zodiac, but Rat does.

3. **`get_zodiac(year):`** Returns the zodiac associated with the inputted year. In the case of invalid input, return `nil`.

4. **`predict_zodiac(year)`**: While our file only contains zodiac animals for years between 1981 and 2028, we want to predict what the zodiac animal will be in the future. Each zodiac animal is repeated every 12 years. For example, 2010, 2022, and 2034 are all years associated with the zodiac Tiger. Implement the function to predict the zodiac animals for years after 2028.

Examples:

```
z = ChineseZodiac.new('zodiac.txt')
z.get_years('Rat')
=> [1984, 1996, 2008, 2020]
z.get_years('Cat')
=> nil
z.get_zodiac(2016)
=> 'Monkey'
z.predict_zodiac(2035)
=> 'Rabbit'
```

Prompt:

```
class ChineseZodiac
  # Part 1
  def initialize(filename)
    # You may use this block to define your data structures
    @zodiac = Hash.new
    File.readlines(filename).each do |line|
      if line =~ /^([A-Z][a-z]+), (\d{4}), (\d{4}), (\d{4}), (\d{4})$/
        @zodiac[$1] = [$2.to_i, $3.to_i, $4.to_i, $5.to_i].sort
      end
    end
  end

  # Part 2
  def get_years(zodiac)
    @zodiac[zodiac]
  end

  # Part 3
  def get_zodiac(year)
    res = nil
    @zodiac.each {|k, v| if v.include? year then res = k end }
    res
  end

  # Part 4
  def predict_zodiac(year)
    while year > 2028
      year -= 12
    end
    @zodiac[year]
  end
end
```

Q5. OCaml: Typing

- For all following three sub-questions, your expression's most general type must be the one given i.e. OCaml infers the expression's type exactly as the one given.
- You are **not allowed to use type annotations**.
- All pattern matching **must be exhaustive**.
- No other warnings should be raised.

Q5.1. Give an expression of the following type: `(int * string * int) list`

```
[(1, "hello", 1)]
```

Q5.2. Give an expression of the following type: `('a -> 'b) -> 'a -> 'b -> bool`

```
fun f a b -> f a = b
```

Q5.3. Give an expression of the following type: `'a -> 'a -> bool list`

```
fun a b -> [a = b]
```

Q6. OCaml: Coding

- You are not allowed to use any external modules such as `List`, etc.
- You are not allowed to use imperative OCaml
- For questions that use higher-order functions, we have provided you the following definitions:

```
let rec map f xs = match xs with
| [] -> []
| x :: xt -> (f x)::(map f xt)

let rec fold_left f a xs = match xs with
| [] -> a
| x :: xt -> fold_left f (f a x) xt

let rec fold_right f xs a = match xs with
| [] -> a
| x :: xt -> f x (fold_right f xt a)
```

Q6.1. Write a function **`get_house`** which takes a catalogue of products at Capsule Corp and your budget and returns a list of houses within the specified budget. The catalogue is defined as a list of `string * int` tuples. Only items which exactly match `"House"` should be returned.

Examples:

```
get_house [("House", 100); ("PS5", 500); ("House", 150)] 100
=> [("House", 100)]
get_house [("Waterproof Socks", 100); ("Fireworks", 100); ("Soap", 200)] 100
=> []
get_house [("House", 70); ("House", 200); ("House", 90); ("House", 140)] 140
=> [("House", 70); ("House", 90); ("House", 140)]
get_house [("House", 100); ("Venti", 64); ("Gates of Babylon", 99)] 200
=> [("House", 100)]
```

Notes:

- Order does not matter in the resulting list.

- You may choose to implement this function recursively or using higher-order functions.

Prompt:

```
let rec get_house lst budget =
  fold_left (fun a (name, price) -> if name = "House" && price <= budget then
  a @ [(name, price)] else a) [] lst
```

Q6.2. Implement the function **`abbreviate`**, which takes a sentence, consisting of words separated by a single space, as input and returns an abbreviated string consisting of only the first letter in each word. However, if the input string contains only a single word, the word is simply returned without getting abbreviated.

Examples:

```
abbreviate "" => ""
abbreviate "Fate" => "Fate"
abbreviate "Among Us" => "AU"
abbreviate "The World God Only Knows"  =>  "TWGOK"
abbreviate "Dragon Ball Z" => "DBZ"
abbreviate "Kanojo Mo Kanojo" => "KMK"
```

Fill in the blanks to complete the implementation below.

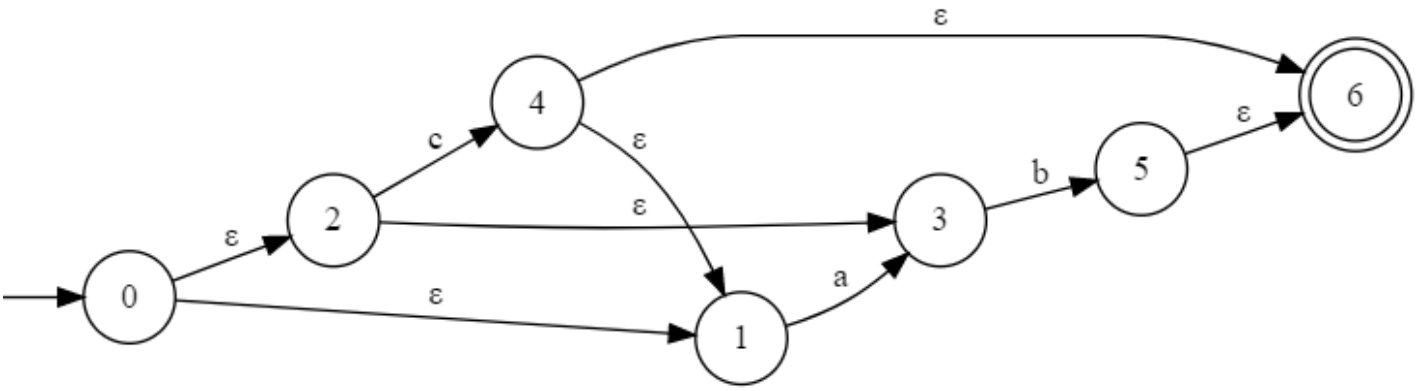
```
let split str = String.split_on_char ' ' str
let first str = String.sub str 0 1
let abbreviate str =
  let lst = split str in
  match lst with
  | [] -> ""
  | [h] -> h
  | _ -> fold_left (fun a x -> a ^ (first x)) "" lst
```

Note: The function split converts a sentence into a list of words and the function first extracts the first character from a string. For example,

```
split "Hello Again World" => ["Hello"; "Again"; "World"]
first "World" => "W"
```

Q7. Regex, NFA and DFA

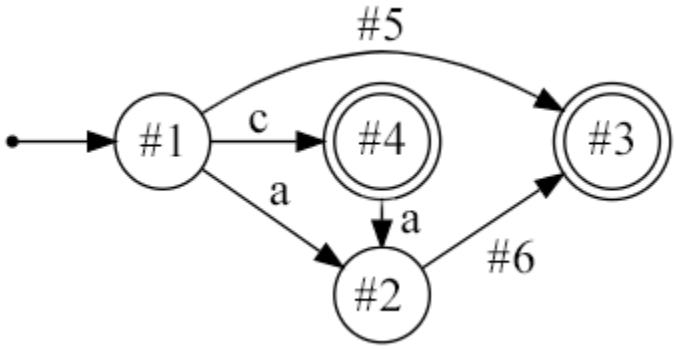
Consider the following NFA.



Q7.1. Which of the following strings are accepted by the NFA?

- abc
- ab
- cb
- cab

Q7.2. Use subset construction - the NFA to DFA algorithm covered in class - to fill in the blanks on the DFA so that the given NFA and DFA are equivalent.



Note: You can put more than one symbol in each blank using commas.

Blank #1: `0,1,2,3`

Blank #2: `3`

Blank #3: `5, 6`

Blank #4: `1,4,6`

Blank #5: `b`

Blank #6: `b`

Q8. CFGs and Parsing

Q8.1 Consider the following CFG.

$$\begin{aligned} &\mathrm{S} \rightarrow \mathrm{aSa} \mid \mathrm{T} \\ &\mathrm{T} \rightarrow \mathrm{bT} \mid \epsilon \end{aligned}$$

Justify why this above CFG cannot be parsed via recursive descent.

It cannot be parsed because of the symmetry in `aSa`.

Detailed Explanation: Consider the string `aaa`. While this string is not valid, the symmetry causes the parser to not be able to differentiate between valid and invalid input i.e., the parser "thinks" `aaa` is valid, until it tries to consume an `a` which does not exist and crashes. Initially, the parser looks ahead to see the first `a`, consumes it, and recursively calls `parse_S` again. This repeats for the second and third `a`. However, when `parse_S` is called for the third time, all three `a` have been consumed and it sees nothing i.e., ϵ. It tries to consume the final `a` in `aSa` and crashes since there is no `a` left to consume.

Q8.2. Consider the following CFG.

$$\begin{aligned} &\mathrm{S} \rightarrow \mathrm{ayaT} \mid \mathrm{aST} \mid \epsilon \\ &\mathrm{T} \rightarrow \mathrm{kaT} \mid \mathrm{toT} \mid \mathrm{yaU} \mid \mathrm{U} \\ &\mathrm{U} \rightarrow \epsilon \end{aligned}$$

Which of the following strings will the CFG accept? Select all that apply.

- `ayaka`
- `aaka`
- `ato`
- `aka`
- `atoa`
- `ayato`

Q8.3. Consider the following CFG.

$$\begin{aligned} &\mathrm{S} \rightarrow \mathrm{SaA} \mid \mathrm{a} \\ &\mathrm{A} \rightarrow \mathrm{aS} \mid \mathrm{bST} \mid \epsilon \\ &\mathrm{T} \rightarrow \mathrm{cT} \mid \epsilon \end{aligned}$$

Show two ways by which the string "aaaa" can be derived from the CFG.

```
S -> SaA -> aaA -> aaaS -> aaaa
S -> SaA -> SaAaA ->SaAaAaA -> aaAaAaA -> aaaAaA -> aaaaA -> aaaa
```

Q8.4. Define a CFG that describes the language.

$$\mathrm{a}^{\mathrm{x}}\mathrm{b}^{\mathrm{y}}\mathrm{c}^{\mathrm{z}} \text{ where } \mathrm{y} = 2\mathrm{x} + 3\mathrm{z},\ \mathrm{x} \geq 0 \text{ and } \mathrm{z} \geq 0.$$

Note: To represent ϵ in the CFG, you can either copy and paste the symbol **ϵ**, type the word epsilon or just type the letter **e**.

```
S -> TU
T -> aTbb | ϵ
U -> bbbUc | ϵ
```

Q9. Operational Semantics

Using the given rules, fill in the blanks the complete the derivation below.

$$\frac{}{A;\ \text{true} \Rightarrow \text{true}} \qquad \frac{}{A;\ \text{false} \Rightarrow \text{false}}$$

$$\frac{A;\ e_1 \Rightarrow \text{true}}{A;\ (\text{not } e_1) \Rightarrow \text{false}} \qquad \frac{A;\ e_1 \Rightarrow \text{false}}{A;\ (\text{not } e_1) \Rightarrow \text{true}}$$

$$\frac{A;\ e_1 \Rightarrow \text{true} \qquad A;\ e_2 \Rightarrow v_1}{A;\ (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \Rightarrow v_1} \qquad \frac{A;\ e_1 \Rightarrow \text{false} \qquad A;\ e_3 \Rightarrow v_1}{A;\ (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \Rightarrow v_1}$$

$$\frac{A;\ e_1 \Rightarrow v_1 \qquad A;\ e_2 \Rightarrow v_2 \qquad v_3 \text{ is } v_1 \,||\, v_2}{A;\ (e_1 \,||\, e_2) \Rightarrow v_3}$$

$$\frac{A;\ e_1 \Rightarrow v_1 \qquad A;\ e_2 \Rightarrow v_2 \qquad v_3 \text{ is } v_1 \,\&\&\, v_2}{A;\ (e_1 \,\&\&\, e_2) \Rightarrow v_3}$$

$$\dfrac{\dfrac{A;\ \text{false} \Rightarrow \text{false}}{A;\ (\#1) \Rightarrow \text{true}} \qquad \dfrac{\dfrac{A;\ \text{true} \Rightarrow \text{true} \qquad A;\ \text{false} \Rightarrow \text{false} \qquad (\#3)}{A;\ (\#2) \Rightarrow \text{true}} \qquad \dfrac{A;\ \text{false} \Rightarrow \text{false} \qquad A;\ \text{true} \Rightarrow \text{true} \qquad (\#5)}{A;\ (\#4) \Rightarrow \text{false}}}{A;\ (\text{if } (\#2) \text{ then } (\#4) \text{ else true}) \Rightarrow \text{false}}}{A;\ (\text{if } (\#1) \text{ then } (\text{if } (\#2) \text{ then } (\#4) \text{ else true}) \text{ else false}) \Rightarrow \text{false}}$$

Blank #1: `not false`

Blank #2: `true || false`

Blank #3: `true is true || false`

Blank #4: `false && true`

Blank #5: `false is false && true`

Q10. Lambda Calculus

To represent λ, you may either copy and paste the symbol **λ** or just type the characters **L** or **** in your solutions.

Q10.1 Consider the following lambda calculus expression.

```
(λx. x λx. y x λy. x y) x λx. x x
```

Make parentheses explicit in the above expression.

```
((λx. (x (λx. ((y x) (λy. x y))))) x) (λx. (x x)))
```

Give a valid α-conversion for the expression, such that there are no duplicate variables

```
(λa. a λb. y b λc. b c) x λd. d d
```

Q10.2. Consider the following encodings,

`true = (λx. λy. x)`

`false = (λx. λy. y)`

`not = (λx. x false true)`

`xor = (λx. λy. x (not y) y)`

Prove that `xor true false = true`.

```
xor true false
= (λx. λy. x (not y) y) true false
= (λy. true (not y) y) false
= true (not false) false
= (λx. λy. x) (not false) false
= (λy. (not false)) false
= not false
= (λx. x false true) false
= false false true
= (λx. λy. y) false true
= (λy. y) true
= true
```

Q11. Rust

Q11.1. Consider the following Rust code.

```
1 let a = String::from("Cryo");
2 {
3   let b = a;
4   {
5     let c = b;
6     let d = &c;
7   }
8   // HERE
9 }
```

Who owns the string "Cryo" at the line **HERE**? Justify your answer.

Nobody owns “Cryo” since on Line 5, it is moved/ownership transferred from b to c and then at Line 7, c is dropped from scope.

Q11.2. Consider the following Rust code.

```
1 let mut a = String::from("Pyro");
2 let b = &a;
3 let mut c = a;
4 let d = &mut c;
5 let e = &c;
6 let f = &e;
```

Assuming that the above code is not subject to non-lexical lifetimes, this code violates two borrowing rules. Identify both violations along with the line number on which the rules are violated.

Violation 1: Move after borrow on Line 3

Violation 2: Immutable borrow after mutable borrow on Line 5

Q11.3. Consider the following Rust code.

```
use std::ops::Deref;
struct MyBox<T>(T);
impl<T> MyBox<T> {
   fn new(x:T)-> MyBox<T> {
      MyBox(x)
   }
}
impl<T> Deref for MyBox<T> {
   type Target = T;
   fn deref(&self) -> &T {
      &self.0 //returns data
   }
}
impl<T> Drop for MyBox<T>{
   fn drop(&mut self){
      println!("My rust is rusty");
   }
}
fn main() {
   let x = "ocaml";
   let y = MyBox::new(x);
   let z = MyBox::new("rust");
   println!("x == *y is {}", x == *y);
   println!("{}", *z)
}
```

The following code compiles and prints the following results:

```
x == *y is true
rust
My rust is rusty
My rust is rusty
```

Why is the 3rd and 4th line of the output the same? Select the option that justifies this behavior the best.

- x and z were dropped
- x and y were dropped
- x, y, and z were dropped
- y and z were dropped
- Nothing was dropped