Programming Languages **Midterm 2 Solution** ECE326, Fall 2020

University of Toronto

Duration: 60 minutes
Examiner: Kuei (Jack) Sun

Please fill your student number, last and first name below and then read the instructions carefully.

Student Number: ____ ____ ____ ____ ____ ____ ____ ____ ____ ____

Last Name: _____

First Name: _____

Instructions

Examination Aids: Ruler and examiner approved aid sheet are allowed.

Do not turn this page until you have received the signal to start.

You may remove the aid sheet from the back of this test book. Do not remove any other sheets from this test book. Answer all questions in the space provided. No additional sheets are permitted. Use the blank space in last page as scratch space. Its content will not be marked.

This exam consists of 6 questions on 8 pages (including this page). The value of each part of each question is indicated. The total value of all questions is 45 marks.

For the written answers, explain your reasoning clearly. Be as brief and specific as possible. Clear, concise answers will be given higher marks than vague, wordy answers. Marks will be deducted for incorrect statements in an answer. Please write legibly!

Work independently.

MARKING GUIDE

Q1:	_____	(4)
Q2:	_____	(6)
Q3:	_____	(7)
Q4:	_____	(7)
Q5:	_____	(8)
Q6:	_____	(13)
Total:	_____	(45)

Question 1. Mut and Mutability [4 marks]

```
let mut a: i32 = 4;
let mut b: i32 = 2;
let x = &mut a;
let mut y = &b;
```

a) What are the types of the variables x and y? [2 marks]

Type of x is &mut i32.
Type of y is & i32.

b) Explain the difference between the variables x and y. [2 marks]

x is a mutable reference with immutable binding.
y is an immutable reference with mutable binding.

Can also say:

x is an immutable variable that borrows *a* mutably.
y is a mutable variable that borrows *b* immutably.

Question 2. Code Reuse [6 marks]

List 3 programming language constructs or features in Rust that promote code reuse. You must clearly explain how each of the construct or feature enables code reuse to get full marks. Note that the lecture notes only explicitly state two of such constructs or features. You must come up with the third one based on your understanding of what code reuse means.

```
[2 marks]
Generic programming enables code reuse across data types so same
algorithm for different data type do not need to be rewritten.

[2 marks]
Traits enable code reuse by supplying default implementation.

[2 marks]
Functions and methods enable code reuse so that programmers can call
a function instead of duplicating code that performs the same task.
```

Question 3. Method Resolution Order [7 marks]

Given the following class hierarchy:

```
class X: pass
class Y: pass
class Z: pass
class A(X): pass
class B(Y): pass
```

```
class C(X): pass
class D(Y, Z): pass
class H(A, B, C): pass
class I(C, D): pass
class J(H, I): pass
```

Compute the method resolution order for the class J using C3 linearization. You must show your steps for the classes H, I, and J to receive full marks. For classes X, Y, Z, A, B, C, D, just show the results.

```
// 1 mark for this part
L[X] = X, o          L[Y] = Y, o          L[Z] = Z, o
L[A] = A, X, o       L[B] = B, Y, o       L[C] = C, X, o
L[D] = D, Y, Z, o

// 2 marks for this part
L[H] = H, merge((A, X, o), (B, Y, o), (C, X, o), (A, B, C))
L[H] = H, A, merge((X, o), (B, Y, o), (C, X, o), (B, C))
L[H] = H, A, B, merge((X, o), (Y, o), (C, X, o), (C))
L[H] = H, A, B, Y, merge((X, o), (o), (C, X, o), (C))
L[H] = H, A, B, Y, C, merge((X, o), (o), (X, o))
L[H] = H, A, B, Y, C, X, o

// 2 marks for this part
L[I] = I, merge((C, X, o), (D, Y, Z, o), (C, D))
L[I] = I, C, merge((X, o), (D, Y, Z, o), (D))
L[I] = I, C, X, merge((o), (D, Y, Z, o), (D))
L[I] = I, C, X, D, merge((o), (Y, Z, o))
L[I] = I, C, X, D, Y, Z, o

// 1 mark for the work
L[J] = J, merge((H, A, B, Y, C, X, o), (I, C, X, D, Y, Z, o), (H, I))
L[J] = J, H, merge((A, B, Y, C, X, o), (I, C, X, D, Y, Z, o), (I))
L[J] = J, H, A, B, merge((Y, C, X, o), (I, C, X, D, Y, Z, o), (I))
L[J] = J, H, A, B, I, merge((Y, C, X, o), (C, X, D, Y, Z, o))

// 1 mark for the conclusion
Neither Y or C are good heads. Therefore, linearization failed.
```

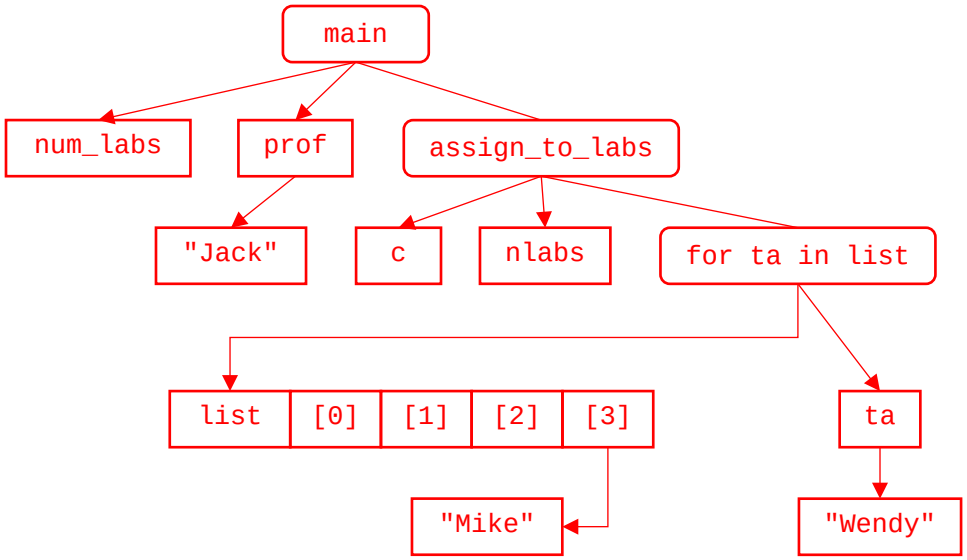
Question 4. Rust Ownership [7 marks]

```
struct Person { name: String }
impl Person {
    fn new(v: &str) -> Person { Person { name: v.to_string() } }
}

fn assign_to_labs(prof: &Person, list: Vec<Person>, nlabs: i32) {
    let mut c = 1;
    for ta in list {
        let name = &ta.name;
        println!("{} is the TA of Lab {}", name, c); /* here */
        c += 1;
    }
    for i in c..=nlabs {
        println!("{} is responsible for Lab {}", prof.name, i);
    }
}

fn main() {
    let num_labs = 6;
    let prof = Person::new("Jack");
    let tas = vec![ Person::new("Kia"), Person::new("Martiya"),
                    Person::new("Wendy"), Person::new("Mike") ];
    assign_to_labs(&prof, tas, num_labs);
}
```

Draw an ownership diagram in the form of a tree at the point when the program prints out “Wendy is the TA of Lab 3” (the line with the comment “here”). Do not draw references to borrowed objects.



Question 5. **Array Calculator** [8 marks]

Write a generic calculate function that takes two array slices, *v1* and *v2*, of equal length and an enum value of type Operator, and returns a new **vector** where each element is the result of performing one of the four the arithmetic operations specified by the enum value on the two elements from *v1* and *v2* at the same index.

For example, the output of `calculate(&[14, 9], &[7, 3], Operator::Divide)` should be `[2, 3]`. You must return a string literal that says "array size mismatch" as an error if *v1* and *v2* are not the same size. Also, you are required to use a match statement to receive full marks. Note: you may need to add some trait bounds to this function prototype.

```
fn calculate<T>(v1: &[T], v2: &[T], op: Operator)
        -> Result<Vec<T>, &'static str>
```

```
enum Operator { Plus, Minus, Times, Divide }

// 3 marks for correct trait bounds
fn calculate<T: Add<Output=T> + Sub<Output=T> + Mul<Output=T> +
    Div<Output=T> + Copy>(v1: &[T], v2: &[T], op: Operator)
    -> Result<Vec<T>, &'static str>
{
    // 1 mark for error checking
    if v1.len() != v2.len() {
        return Err("array size mismatch");
    }

    let mut v = Vec::new();

    // 1 mark for this loop
    for i in 0..v1.len() {

        // 2 marks for the match statement
        match op {
            Operator::Plus => v.push(v1[i] + v2[i]),
            Operator::Minus => v.push(v1[i] - v2[i]),
            Operator::Times => v.push(v1[i] * v2[i]),
            Operator::Divide => v.push(v1[i] / v2[i]),
        };
    }

    // 1 mark for creating a vector and returning it in Ok
    Ok(v)
}
```

Question 6. **Managed Attributes** [13 marks]

You are developing a package for adding type safety checks to the end users' custom-defined classes. In addition, it verifies that all fields of the user-defined classes are initialized in the constructor. The interface you provide requires that their classes inherit from Base, and that the fields they want to type check be specified as class attributes of type Field, For example:

```
class User(Base):
    name = Field(type=str)
    age = Field(type=int)
    height = Field(type=float)

bob = User(name="Bob", age=23, height=6.2)
tom = User(name="Tom", age="x", height=7.)   # error: age not an int
zoe = User(name="Zoe", height=5.6)     # error: field age is missing
```

The Field class has already been completed, as shown here. You may not change it for this question.

```
class Field:
    def __init__(self, type):
        self.type = type
```

a) Complete the `__init__` method of Base's metaclass so that every *class* that inherits from Base keeps track of a dictionary named `_fields` that maps field names to the corresponding field objects. Hint: read ahead to see what this dictionary is used for. [3 marks]

```
class Meta(type):
    def __init__(cls, name, bases, attrs):


        cls._fields = {}
        for col, val in attrs.items():
            if isinstance(val, Field):
                cls._fields[col] = val
```

b) Complete the `__init__`, `__setattr__`, and `__getattribute__`methods of the Base class. You must make sure your solution conforms to the following specification. [10 marks]

i. Raise AttributeError if a field is missing from the keyword arguments passed to the constructor.
ii. Raise TypeError if a field is set to a value that is not the type specified in the corresponding Field object, e.g. `foo.name = 123`.
iii. You may ignore arguments to `__init__` that do not have an associated Field object.
iv. You must allow non-managed attributes to be added to the instance. e.g. `foo.pk = 3`.

```
class Base(metaclass=Meta):
    def __init__(self, **kwargs):

        # 3 marks
        for name, field in type(self)._fields.items():
            if name not in kwargs:
                raise AttributeError
            setattr(self, name, kwargs[name])


    # 4 marks
    def __setattr__(self, name, value):
        try:
            field = type(self)._fields[name]
        except KeyError:
            return super().__setattr__(name, value)

        if not isinstance(value, field.type):
            raise TypeError

        super().__setattr__("_" + name, value)



    # 3 marks
    def __getattribute__(self, name):
        if name in type(self)._fields:
            return super().__getattribute__("_" + name)
        return super().__getattribute__(name)
```

[*Use the space below for rough work*]

END OF EXAMINATION