Name:	Roll No:	

Programming Language Concepts

Quiz 2, II Semester, 2023–2024

20 February, 2024

1. Consider the following Rust functions.

(i)
```
fn fact1 (n : i32) -> i32{
  let mut i = 1;
  let mut fact = 1;
  while i <= n {
    fact = fact * i;
    i = i + 1;
  }
  return fact;
}
```

(ii)
```
fn fact2 (n : i32) -> i32{
  let mut i = 1;
  let fact = 1;
  while i <= n {
    let fact = fact * i;
    i = i + 1;
  }
  return fact;
}
```

(iii)
```
fn fact3 (n : i32) -> i32{
  let mut i = 1;
  let fact = 1;
  while i <= n {
    let fact = fact * i;
    let i = i + 1;
  }
  return fact;
}
```

(iv)
```
fn fact4 (n : i32) -> i32{
  let fact = 1;
  while n > 0 {
    let fact = fact * n;
    n = n - 1;
  }
  return fact;
}
```

Fill in each entry in the following table with **Yes** or **No**.

	Compiles	*Runs*	*Terminates*	*Correct answer*
`fact1`	Y	Y	Y	Y
`fact2`	Y	Y	Y	N
`fact3`	Y	Y	N	N
`fact4`	N	N	N	N

Explanations

(i) No problems.

(ii) `fact` redeclared inside the loop has a different scope from `fact` initialized to `1` initially. The return value is the outer `fact`, which is always `1`.

(iii) The `i` being tested in the while condition is the outer `i`. The `i` incremented inside the loop is a fresh variable with a different scope and the increment is "lost" each time the loop ends. This results in an infinite loop.

(iv) This code does not compile. To update `n` inside the function, the parameter should be tagged as `mut`.

... Question 2 on reverse

2. Consider the following Rust functions.

(i)
```
fn maxlen1(s1 : String, s2 : String)
                        -> String {
  if s1.len() > s2.len() {s1}
  else {s2}
}

fn main1(){
  let x = String::from("Python");
  let y = String::from("Java");
  let z = maxlen1(x,y);
  println!("maxlen1({}, {}) is {}",
            x,y,z);
}
```

(ii)
```
fn maxlen2(s1 : String, s2 : String)
      -> (String,String,String) {
  let s3 = if s1.len() > s2.len()
           {s1} else {s2};
  return(s1,s2,s3);
}

fn main2(){
  let x = String::from("Python");
  let y = String::from("Java");
  let (x,y,z) = maxlen2(x,y);
  println!("maxlen2({}, {}) is {}",
            x,y,z);
}
```

(iii)
```
fn maxlen3(s1 : String, s2 : String)
     -> (String,String,String) {
  let s3 = if s1.len() > s2.len()
           {s1.clone()}
           else {s2.clone()};
  return(s1,s2,s3);
}

fn main3(){
  let x = String::from("Python");
  let y = String::from("Java");
  let (x,y,z) = maxlen3(x,y);
  println!("maxlen3({}, {}) is {}",
            x,y,z);
}
```

(iv)
```
fn maxlen4(s1 : &str, s2 : &str)
                    -> &str {
  if s1.len() > s2.len() {s1}
  else {s2}
}

fn main4(){
  let x = String::from("Python");
  let y = String::from("Java");
  let z = maxlen4(&x,&y);
  println!("maxlen4({}, {}) is {}",
            x,y,z);
}
```

Fill in each entry in the following table with **Yes** or **No**.

	Compiles	*Runs*	*Terminates*	*Correct answer*
`maxlen1,main1`	N	N	N	N
`maxlen2,main2`	N	N	N	N
`maxlen3,main3`	Y	Y	Y	Y
`maxlen4,main4`	N	N	N	N

Explanations

(i) Ownership of the strings `x` and `y` is transferred to `maxlen1()`, so they are undefined in `main1()` after the call to `maxlen1()`.

(ii) Within `maxlen2()` the assignment to `s3` moves the ownership of either `s1` or `s2` to `s3`. There are only two string objects in scope at the return statement.

(iii) Since we are cloning `s1` or `s2` to assign to `s3`, this code works fine.

(iv) `maxlen4()` returns a reference corresponding to one of its two arguments. To avoid dangling references, Rust requires us to annotate the lifetimes of the arguments and the return value.