CS2030S

Programming Methodology II

Recitation 07

Q1

Pure Functions

Q1

Pure

Recap: Pure Functions

Definition

`f : X -> Y`

where `X` is the domain and `Y` is the codomain. Requirements:

1. `f(x)` is deterministic
    - `f(x) = y` everywhere
    - Also called *referential transparency*
2. `f(x)` has no side-effects
    - No print to screen
    - No write to files
    - Not throwing exceptions
    - No change/mutation to fields
    - No change/mutation to arguments

Q1

Pure
Code A

Code A

Question

Consider the function on the right. Is the function a pure function?

Pure Functions

- Deterministic *($f(x) = y$ everywhere)*
- No Side Effects *(print, write, exceptions, etc)*

throws

```java
int f(int i) {
  if (i < 0) {
    throw new IllegalArgumentException();
  } else {
    return i + 1;
  }
}
```

runtime exception

	Choice	Comment	
A	yes	*NO: it may throw an exception*	✗
B	no	*YES: it may throw an exception*	✓

Q1

Pure
Code A
Code B

Code B

Question

Consider the function on the right. Is the function a pure function?

Pure Functions

- Deterministic *($f(x) = y$ everywhere)*
- No Side Effects *(print, write, exceptions, etc)*

```java
int g(int i) {
  System.out.println(i);
  return i + 1;
}
```

	Choice	Comment	
A	yes	*NO: printing is a side-effect*	✗
B	no	*YES: printing is a side-effect*	✓

Q1

Pure Functions

- Deterministic *($f(x) = y$ everywhere)*
- No Side Effects *(print, write, exceptions, etc)*

Code C

Question

Consider the function on the right. Is the function a pure function?

current time

```java
int h(int i) {
  Random rand = new Random();
  return rand.nextInt() + 1;
}
```

seed = t

nextInt

	Choice	Comment	
A	yes	*NO: it is non-deterministic*	✗
B	no	*YES: it is non-deterministic*	✓

Q1

Pure
Code A ✗
Code B ✗
Code C ✗
Code D ✓

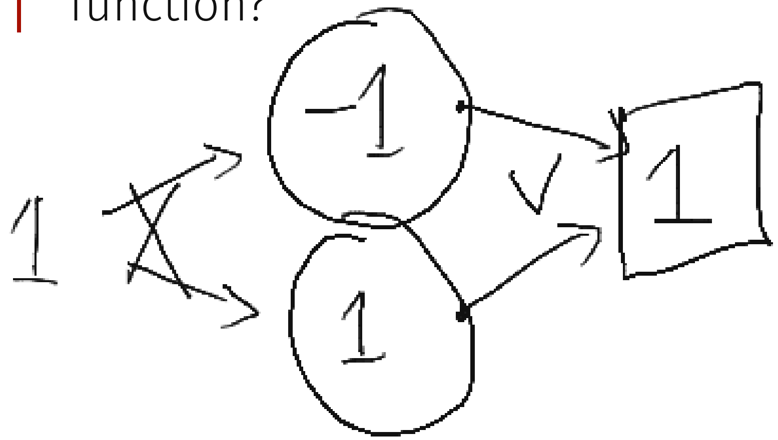
Code D

Question

Consider the function on the right. Is the function a pure function?

```
int k(int i) {
  return Math.abs(i);
}
```

```
if (i<0) {
  -i;
} else {
  i;
}
```



Pure Functions

- Deterministic *($f(x) = y$ everywhere)*
- No Side Effects *(print, write, exceptions, etc)*

	Choice	Comment	
A	yes	*YES: no side-effect & deterministic*	✔
B	no	*NO: no side-effect & deterministic*	✖

Q2

Lambda

Anonymous Function

param → <expr>

param → { <body>;
return <expr>; }

(par1, par2) → { ____ }

$P \longrightarrow R$

Lambda

Recap: Anonymous Function

1. Start with Interface

```
@FunctionalInterface
interface Immutator<R, P> {
  R invoke(P p);
}
```

Q2

Lambda

Lambda

Recap: Anonymous Function

2. Create Anonymous Class

```
@FunctionalInterface
interface Immutator<R, P> {
  R invoke(P p);
}
```

R P

```
Immutator<Integer, String> len = new Immutator<>() {
  @Override
  Integer invoke(String p) {
    return p.length();
  }
}
```

R P

Q2

Lambda

Lambda

param → <expr>

Recap: Anonymous Function

3. Syntactic Sugar

```java
@FunctionalInterface
interface Immutator<R, P> {
  R invoke(P p);
}
```

param → <expr>

```java
Immutator<Integer, String> len = new Immutator<>() {
  @Override
  Integer invoke(String p) {
    return p.length();
  }
}
```

⟷

```java
len = p -> p.length();
```

();

Integer

Q2

BiFunction

Functions with 2 Parameters

Java SE 17 & JDK 17

Module java.base
Package java.util.function

Interface BiFunction<T,U,R>

Type Parameters:

T - the type of the first argument to the function

U - the type of the second argument to the function

R - the type of the result of the function

All Known Subinterfaces:

`BinaryOperator`<T>

`(t, u) -> r` $\Longleftrightarrow$ `t -> u -> r`

Q2

Lambda
BiFunction
Part A

Part A

takes in x

returns function F_2

takes in z

returns a result

Question

Consider the following lambda expression

$F =$

```
x -> (y -> (z -> f(x, y, z)))
```

$F(x) = F_2$

where x, y, and z are some type *T* and f returns a value of type *R*.
What kind of lambda expression is this?

takes in y

returns function F_3

	Choice	Comment	
A	Uncurried Function	*NO: **f** is the uncurried version*	✗
B	Curried Function	*YES: this is the curried version of **f***	✓
C	Partial Function	*NO: that only happen on partial application*	✗
D	Binary Function	*NO: binary function takes in 2 arguments*	✗
E	Unary Function	*YES: unary function takes in 1 argument*	✓

Q2

Lambda
BiFunction
Part A
Part B

$Im\langle L, Int\rangle$

$\to Im\langle Im\langle L_2, Int\rangle, Int\rangle$

$Im\langle Im\langle Im\langle Int, Int\rangle, Int\rangle, Int\rangle$

interface Immutator<R,P> {
R invoke(P p);
}

Part B

```
x -> y -> z -> f(x, y, z)
```

P   L   L2

Question

Suppose that *T* and *R* are `Integer` and `f(x, y, z)` is given by `x + y + z`. For simplicity, we write `Im` for `Immutator` and `Int` for `Integer`. What is the type of the above lambda expression using `Immutator`?

$L = Im\langle L_2, Int\rangle$

$L_2 = Im\langle Int, Int\rangle$

A: $((x \to y) \to z) \to v$

B: $x \to y \to z \to v$

$x \to (y \to (z \to v))$

	Choice	Comment	
A	`Im<Int,Im<Int,Im<Int,Int>>>`	*NO: be careful with the direction*	✗
B	`Im<Im<Im<Int,Int>,Int>,Int>`	*YES: return type first*	✓

Q3

Tail Recursion

Q3

TCO : tail-call optimization

Tail Recursion

Tail Recursion

Math

$$\sum_{i=0}^{n} i$$

Code

```
fact(n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
```

```java
static long sum(long n, long result) {
  if (n == 0) {
    return result;
  } else {
    return sum(n - 1, n + result);
  }
}
```

→ no recursion

→ recursion is last step

Q3

Tail Recursion
Iteration

Iteration

sumR

```
static Compute<Long> sum(long n, long s) {
  if (n == 0) {
    return new Base<>(() -> s);
  } else {
    return new Recursive<>(
      () -> sum(n - 1, n + s)
    );
  }
}
```

summer

```
static long summer(long n) {
  Compute<Long> result = sum(n, 0);

  while (result.isRecursive()) {
    result = result.recurse();
  }

  return result.evaluate();
}
```

Q3

Tail Recursion
Iteration
Compute

Interface

Compute< T >

sumR

```
static Compute<Long> sum(long n, long s) {
  if (n == 0) {
    return new Base<>(() -> s);
  } else {
    return new Recursive<>(
      () -> sum(n - 1, n + s)
    );
  }
}
```

summer

```
static long summer(long n) {
  Compute<Long> result = sum(n, 0);

  while (result.isRecursive()) {
    result = result.recurse();
  }

  return result.evaluate();
}
```

- Base<T>
- Recursive<T>

Q3

Tail Recursion
Iteration
Compute
- isRecursive

Compute< T >

Design

- `boolean isRecursive()`

sumR

```
static Compute<Long> sum(long n, long s) {
  if (n == 0) {
    return new Base<>(() -> s);
  } else {
    return new Recursive<>(
      () -> sum(n - 1, n + s)
    );
  }
}
```

- `Base<T>`
- `Recursive<T>`

summer

```
static long summer(long n) {
  Compute<Long> result = sum(n, 0);

  while (result.isRecursive()) {
    result = result.recurse();
  }

  return result.evaluate();
}
```

Q3

Tail Recursion
Iteration
Compute
- *isRecursive*
- *recurse*

Compute< T >

sumR

```
static Compute<Long> sum(long n, long s) {
  if (n == 0) {
    return new Base<>(() -> s);
  } else {
    return new Recursive<>(
      () -> sum(n - 1, n + s)
    );
  }
}
```

- `Base<T>`
- `Recursive<T>`

Design

- `boolean isRecursive()`
- `Compute<T> recurse()`

summer

```
static long summer(long n) {
  Compute<Long> result = sum(n, 0);

  while (result.isRecursive()) {
    result = result.recurse();
  }

  return result.evaluate();
}
```

Q3

Tail Recursion
Iteration
Compute
- isRecursive
- recurse
- evaluate

Compute< T >

Long

sumR

```
static Compute<Long> sum(long n, long s) {
  if (n == 0) {
    return new Base<>(() -> s);
  } else {
    return new Recursive<>(
      () -> sum(n - 1, n + s)
    );
  }
}
```

summer

```
static long summer(long n) {
  Compute<Long> result = sum(n, 0);

  while (result.isRecursive()) {
    result = result.recurse();
  }

  return result.evaluate();
}
```

Design

- `boolean isRecursive()`
- `Compute<T> recurse()`
- `T evaluate()`

- Base<T> → Base (Producer<Long> p)
- Recursive<T> → Recursive (Producer<Compute<Long>> p)

Q3

Tail Recursion
Iteration
Compute
Code
- Producer

Code

Producer< T >

```
interface Producer<T> {
  T produce();
}
```

```
interface Constant<T> {
  T init();
}
```

Design

- `boolean isRecursive()`
- `Compute<T> recurse()`
- `T evaluate()`

Q3

Tail Recursion
Iteration
Compute
Code
- Producer
- Compute

Code

Compute< T >

```
interface Compute<T> {
  boolean isRecursive();
  Compute<T> recurse();
  T evaluate();
}
```

Design

- `boolean isRecursive()`
- `Compute<T> recurse()`
- `T evaluate()`

```
jshell> /exit
|    Goodbye
```