CSPB 3155 - Reckwerdt - Principles of Programming Languages

Started on	Tuesday, 13 August 2024, 1:43 PM
State	Finished
Completed on	Tuesday, 13 August 2024, 2:49 PM
Time taken	1 hour 5 mins
Grade	**42.33** out of 50.00 (**85**%)

Question 1

Partially correct

Mark 1.00 out of 2.00

Select all the scala functions below that will be compiled without an error and execute without a runtime error on all possible inputs (with valid types). Please ensure that only correct scala functions are selected and the incorrect ones are not.

Select one or more:

a.

```
def fun4(x: Int): Int = x match {
    case y if y >= 0 => y - 10
    case y if y <= 0 => y + 10
}
```

b.

```
def fun1(x: Int): Int = {

    if (x <= 20) { 45 }

}
```

c.

```
def fun3(x: Int): Int = x match {

    case y if y >= 0 => y + 10

    case _ => z - 10

}
```

d.

```
def fun2(x: Int): Int = x match {

    case y if y >= 0 => y + 10

    case _ => x - 10

}
```

Your answer is partially correct.

You have correctly selected 1.

The correct answers are:

```
def fun2(x: Int): Int = x match {

    case y if y >= 0 => y + 10

    case _ => x - 10

}
```

,

```
def fun4(x: Int): Int = x match {
    case y if y >= 0 => y - 10
    case y if y <= 0 => y + 10
}
```

Question **2**

Partially correct

Mark 2.25 out of 3.00

Match the descriptions for functions that process lists with the code that will actually carry them out.

Note: dot product of two lists of same size is defined as

[e1, ..., en] . [f1, ..., fn] = (e1 * f1 + e2 * f2 + ... + en * fn)

Dot product of two lists of same size. Description of dot product above.	lst1.zip(lst2) . foldLeft (0) { (acc, elt) => acc + elt._1 * elt._2 } ✔
Sum of square of each element in the list.	lst.map(x => x * x) ✖
Select all odd elements from list.	lst.filter(_ %2 == 1) ✔
Double of each element in the list.	lst.map(x => 2 * x) ✔

Your answer is partially correct.

You have correctly selected 3.

The correct answer is:

Dot product of two lists of same size. Description of dot product above. → lst1.zip(lst2) . foldLeft (0) { (acc, elt) => acc + elt._1 * elt._2 },

Sum of square of each element in the list. → lst.foldLeft (0) { (acc, elt) => acc + elt * elt },

Select all odd elements from list. → lst.filter(_ %2 == 1),

Double of each element in the list. → lst.map(x => 2 * x)

Question **3**

Partially correct

Mark 3.75 out of 5.00

Consider the Lettuce environment:

```
Extend( "x", NumValue(-20),
    Extend ("y", NumValue(20),
      Extend( "z", BoolValue(false),
       Extend ( "f", Closure( "y",  y - z , env1 ),
         Extend ( "y", NumValue(15),
            EmptyEnvironment
            )))))
```

env1 is the environment: *Extend("z", NumValue(30), EmptyEnvironment)*

Match expressions given in concrete syntax to their values that they evaluate under the environment above.

* Notes: In the closure corresponding to "f", read "y - z" as Minus(Ident("y"), Ident("z"))

x + y	0	✔
(x < y) && x	true	✖
(x >= y) && x	false	✔
f(y)	-10	✔

Your answer is partially correct.

You have correctly selected 3.
The correct answer is:
x + y → 0,

(x < y) && x → error,

(x >= y) && x → false,

f(y) → -10

Question **4**

Correct

Mark 4.00 out of 4.00

Consider the following situation for an environment and memory store:

```
ENVIRONMENT:
{ x : NumValue(0.0),
  y : Reference(0),
  z : Reference(1)
}
```

```
STORE:
0 -> Closure("fish", Plus(Ident("fish"), NumValue(5.0)), EmptyEnvironment)
1 -> NumValue(2.0)
```

Now consider the following questions:

(A) In the above environment and store, what is the value of Lettuce expression **DeRef(z)(DeRef(y))** :

Read: The function denoted by **DeRef(z)** applied to the argument denoted by **DeRef(y)**

ERROR ✔

(B: 2 points) In the above environment and store, what is the value of Lettuce expression **DeRef(y)(x)** :

Read: The function denoted by **DeRef(y)** applied to the argument denoted by **x**

NumValue(5.0) ✔

(C: 2 points) In the above environment and store, what is the value of Lettuce expression **DeRef(y)(DeRef(z))** :

Read: The function denoted by **DeRef(y)** applied to the argument denoted by **DeRef(z)**

NumValue(7.0) ✔

(D: 2 points) In the above environment and store, if we execute Lettuce expression **DeRef(y)(DeRef(z))** what changes, if any, occur to the memory store: No changes occur ✔

Question **5**

Correct

Mark 6.00 out of 6.00

Suppose we evaluate the following Lettuce expression starting with the empty environment.

```
let y = 10 in (* let binding for y # 1 *)
```

```
let f = function (x) x - y in
 let y = 20 in  (* let binding for y # 2 *)
   let g = function (y) f(y) in
      g(y)
```

Let `env` be the environment when the function call `g(y)` in the very last line is about to be executed.

Match each of the questions below to the correct answer.

Comments are written within (* and *) are not part of the expression.

What value does the entire program evaluate to?	NumValue(10) ✔
The body of the function f refers to identifier "y". What value will this identifier have if function f were called?	NumValue(10) ✔
What does lookup(env, "x") yield?	error ✔
What does lookup(env, "f") yield?	Closure("x", x-y , { y -> 10 }) ✔
What does lookup(env, "y") yield?	NumValue(20) ✔

Your answer is correct.

The correct answer is:
What value does the entire program evaluate to? → NumValue(10),

The body of the function f refers to identifier "y". What value will this identifier have if function f were called? → NumValue(10),

What does lookup(env, "x") yield? → error,

What does lookup(env, "f") yield? → Closure("x", x-y , { y -> 10 }),

What does lookup(env, "y") yield? → NumValue(20)

Question **6**

Partially correct

Mark 4.00 out of 6.00

In class, we presented a short circuit semantics for the and (&&) and or (||) operators in Lettuce in line with other programming languages such as Scala, Python and C.

Suppose wish to implement the and operator (&&) in Lettuce **without** using short circuit semantics using the following semantic rule instead:

$$\frac{\textbf{eval}(e1,\sigma) = v_1, v_1 \in \text{Boolean},\ \textbf{eval}(e2,\sigma) = v_2, v_2 \in \text{Boolean}}{\textbf{eval}(And(e1,e2),\sigma) = v_1 \text{ and } v_2}$$

Here a and b is the propositional logic "and" of **a, b**.

We assume that the semantics of the cases that generate errors are the same as before, as recalled below:

$$\frac{\textbf{eval}(e1,\sigma) = v_1, v_1 \notin \text{Boolean}}{\textbf{eval}(And(e1,e2),\sigma) = \textbf{error}}$$

$$\frac{\textbf{eval}(e2,\sigma) = v_2, v_2 \notin \text{Boolean}}{\textbf{eval}(And(e1,e2),\sigma) = \textbf{error}}$$

The short circuit semantics for the or operator remain unchanged.

Select all expressions that yield different values when evaluated with the alternate semantics presented above versus when evaluated with the short circuit semantics from class.

Select one or more:

- [x] a. ✖
```
let x = 10 in
   let y = 20 in
      (x - 10/0 == 10/0) && (x <= y)
```

- [ ] b.
```
let x = 10 in
   let y = 20 in
      (x >= y) && (y <= x)
```

- [x] c. ✔
```
let z = newref(45) in
  let foo = function (x)  assignref(z, x)  in
     let x = 10 in
        let y = 20 in
           let v = (x >= y) && ( foo(x) <= 20) in
               deref(z)
```

- [ ] d.
```
let x = 10 in
   let y = 20 in
      (x >= y)
```

- [x] e. ✔
```
let x = 10 in
   let y = 20 in
      (x >= y) && (x - 10/0 == 10/0)
```

Your answer is partially correct.

You have selected too many options.

The correct answers are:

```
let x = 10 in
   let y = 20 in
      (x >= y) && (x - 10/0 == 10/0)
```

,

```
let z = newref(45) in
  let foo = function (x)  assignref(z, x)  in
     let x = 10 in
        let y = 20 in
           let v = (x >= y) && ( foo(x) <= 20) in
               deref(z)
```

Question 7

Correct

Mark 6.00 out of 6.00

Suppose we modified the semantics of Let bindings as follows:

$$\frac{x \notin \mathbf{domain}(\sigma),\ \mathtt{eval}(\mathtt{e1},\sigma) = v_1, v_1 \neq \mathbf{error}}{\mathtt{eval}(\mathtt{Let("x",\ e1,\ e2)},\sigma) = \mathtt{eval}(\mathtt{e2},\sigma \circ \{x \mapsto v_1\})}$$ (rule let-binding-ok)

The rule let-binding-ok allows us to evaluate the let binding in the usual manner presented in class as long as the identifier x being bound is not already in the environment σ.

The error cases include:

$$\frac{x \in \mathbf{domain}(\sigma)}{\mathtt{eval}(\mathtt{Let("x",\ e1,\ e2)},\sigma) = \mathbf{error}}$$ (rule let-error-shadow)

$$\frac{x \notin \mathbf{domain}(\sigma),\ \mathtt{eval}(\mathtt{e1},\sigma) = \mathbf{error}}{\mathtt{eval}(\mathtt{Let("x",\ e1,\ e2)},\sigma) = \mathbf{error}}$$ (rule let-error-lhs)

Match each of the expressions below to their values.

let x = 10 in let x = 20 in x	error ✔
let x = (let x = 20 in x + 20) in x	40 ✔
let f = function (x) x + 10 in let x = 15 in f(x)	25 ✔
let x = 0/0 in let y= 20 in y	error ✔
let x = 10 in let y = 20 in x + y	30 ✔

Your answer is correct.

The correct answer is:
let x = 10 in let x = 20 in x → error,

let x = (let x = 20 in x + 20) in x → 40,

let f = function (x) x + 10 in let x = 15 in f(x) → 25,

let x = 0/0 in let y= 20 in y → error,

let x = 10 in let y = 20 in x + y → 30

Question **8**

Partially correct

Mark 4.50 out of 6.00

Select all lettuce programs below **which are well-typed**: I.e, the type checker does not yield a typeerror when the program is checked.

Make sure that no ill-typed programs are selected.

Select one or more:

- [ ] a.

```
let x : num = 0/0 in
  x /0 + 0 - 0/0
```

- [x] b.

```
let f : (num => bool) => bool = function (g: num => bool) g(10)  in
  let h : num => bool =  function (x: num) x <= 10 in
    f (  h ) ✔
```

- [ ] c.

```
let f : (num => num) => num  = function (g: num => num) g(10)  in
  let h : num => bool =  function (x: num) x <= 10 in       f (  h ) (3)
```

- [x] d.

```
let rec infLoop: num => num = function (x: num) infLoop(x+1) in
 infLoop(0) ✔
```

- [ ] e.

```
let y: num  = 10 in
let f : num => num = function (x : num) x - 10 in
  let g : num => bool = function (y: num) f(y) >= 20 in
    g(y) + 5
```

- [x] f. ✔

```
let y: num = 10 in
let f : num => num = function (x : num) x - 10 in
  let g : num => num  = function (y: num) f(y) - 20  in
    g(y) + 5
```

Your answer is partially correct.

You have correctly selected 3.
The correct answers are:

```
let y: num = 10 in
let f : num => num = function (x : num) x - 10 in
  let g : num => num  = function (y: num) f(y) - 20  in
    g(y) + 5
```

,

```
let f : (num => bool) => bool = function (g: num => bool) g(10)  in
  let h : num => bool =  function (x: num) x <= 10 in
    f (  h ),

let x : num = 0/0 in
  x /0 + 0 - 0/0
 ,

let rec infLoop: num => num = function (x: num) infLoop(x+1) in
 infLoop(0)
```

Question **9**

Correct

Mark 5.00 out of 5.00

Consider the following scala definitions:

```
/* The parent class of them all */
abstract class Animal

/* some traits */
trait SwimsInWater
trait WalksInLand

/* some classes */
class Reptile extends Animal
class Fish extends Animal with SwimsInWater
class Amphibian extends Animal with SwimsInWater with WalksInLand
class Frog extends Amphibian
class Nemo extends Fish
// .. other class definitions are possible ..

/* some functions */
def analyzeSwimSpeed[T] ( T animal ): Double = { /* .. code not relevant ... */ }
def analyzeLandSpeed[T] ( T animal) : Double = { /* .. code not relevant ... */ }
def printAnimalGenusAndSpecies[T] ( T animal ) : String = { ... }
```

We wish to constraint the type parameter T for the function analyzeSwimSpeed to achieve the following properties. Match the specification against the constraint on T that achieves it in Scala.

The function applies only to classes that inherit from Fish (including Fish itself)	T < : Fish ✔
Function applies only to classes that inherit from subclasses of Animal with the trait SwimsInWater applied?	T < : Animal with SwimsInWater ✔
Function only applies to Amphibian or its parent classes but not to any other classes that inherit from Amphibian	T >: Amphibian ✔
Function applies only to objects that inherit from Animal and WalksInLand applied	T < : Animal with WalksInLand ✔

Your answer is correct.

The correct answer is:
The function applies only to classes that inherit from Fish (including Fish itself) → T < : Fish,

Function applies only to classes that inherit from subclasses of Animal with the trait SwimsInWater applied? → T < : Animal with SwimsInWater,

Function only applies to Amphibian or its parent classes but not to any other classes that inherit from Amphibian → T >: Amphibian,

Function applies only to objects that inherit from Animal and WalksInLand applied → T < : Animal with WalksInLand

Question **10**

Partially correct

Mark 5.83 out of 7.00

Here is a scala function:

```
def myfun ( x: Int, y: String) : String = {
        if (x <= 0) {
                 y + x.toString
        } else {
                myfun( x - 1, "hello") +  myfun( x - 2 ,  "world" )
       }
}
```

Fill in the missing part of the CPS transformation of the above function:

```
def myfun_cps[T]( x : Int, y : String, k : ???1 => T) : T = {
      if ( x < = 0) {
              ???2
       } else {
              myfun_cps ( ???3 , "hello",
                          (v1) => { myfun_cps( ???4, ???5,
                                               (v2) => { ???6 } )
                                  } )
       }
 }
```

???1	String	✔
???2	k(y + x.toString)	✔
???3	x - 1	✔
???4	x - 2	✔
???5	"world"	✔
???6	v1 + v2	✖

Your answer is partially correct.

You have correctly selected 5.

The correct answer is:

???1 → String,

???2 → k(y + x.toString),

???3 → x - 1,

???4 → x - 2,

???5 → "world",

???6 → k(v1+v2)