CS4215 Programming Language Implementation

You have 45 minutes to complete the exam. Use a B2 pencil to fill up the provided MCQ form. Leave Section A blank. Fill up Sections B and C.

After finishing, place the MCQ sheet on top of the question sheet and leave both on the table, when you exit the room.

Question 1: Which one of the following statements about the programming language processor families of assemblers, hardware emulators, compilers and decompilers is false?

1 [A] Assemblers are translators.

1 [B] Hardware emulators are translators.

1 [C] Compilers are translators.

1 [D] Decompilers are translators.

Answer 1:

1 [B] Hardware emulators are interpreters, not transators.

All other mentioned language processor families contain translators.

Question 2: Consider two alternative implementations of a language X using a processor Y.

- Compile the programs from X to the machine code of Y.
- Interpret the programs written in X using an interpreter written in the machine code of Y.

Which statement characterizes the relationship between the interpretation-based implementation and the compiler-based implementation of X.

2 [A] Interpretation is always faster than running compiled code.

2 [B] Interpretation is always slower than running compiled code.

2 [C] Interpretation avoids the compilation step.

2 [D] Compilation avoids the execution of any machine code.

2 [E] The compilation cannot perform type checking of programs written in X.

Answer 2:

2 [C] It lies in the nature of an interpreter to not compile the program in a separate step. There may be internal "compilation" happening, such as in just-in-time compilation, but that would be interleaved with interpretation, and not in a "compilation step".

Question 3: Assume a given programming language L with a typing relation $\cdots \vdash \cdots : \cdots$ and one-step evaluation $\mapsto$. Furthermore, we assume a given set $V \subset L$ called values. Assume that L is type-safe with respect to $\cdots \vdash \cdots : \cdots$, $\mapsto$ and the notion of values given by V.
Which of the following statements is false?

3 [A] There may be a well-typed expression $E \in L$ such that there is no value $v \in V$ with $E \mapsto^* v$.

3 [B] There may be a non well-typed expression $E \in L$ such that there is a value $v \in V$ with $E \mapsto^* v$.

3 [C] There may be a well-typed expression $E \in L$ such that $E \notin V$ and there is no expression E' with $E \mapsto E'$.

3 [D] There may be a well-typed expression $E \in L$ such that $E \notin V$ and there is a value v with $E \mapsto v$.

3 [E] None of the above.

Answer 3:

3 [C] This statement contradicts the progress property of type-safety.

All other statements are correct.

Question 4: What is the result of evaluating the following simPL expression, according to the *dynamic semantics* (based on substitution)?

```
if 1 < 2 then (fun {int -> int} x -> x + 1 end (1 + (5 - 3) / 0))
else 4 end
```

4 [A] `(fun {(int -> int)} x -> (x + 1) end  (1 + (2 / 0)))`

4 [B] `4`

4 [C] `(1 + (2 / 0)) + 1`

4 [D] `(fun {(int -> int)} x -> (x + 1) end  (1 + ((5 - 3) / 0)))`

4 [E] `if 1 < 2 then (fun {int -> int} x -> x + 1 end (1 + ((5 - 3) / 0))) else 4 end`

Answer 4:

4 [A] The division by 0 gets stuck; neither the addition nor the application can proceed.

Question 5: What is the result of evaluating the following simPL expression, according to the *dynamic semantics* (based on substitution)?

```
((fun {int -> int -> int} x ->
     fun {int -> int} y ->
        x + y
     end
  end
  true)
 4)
```

5 [A] `true`

5 [B] `4`

5 [C] `(fun {int -> int} y -> true + y end 4)`

5 [D] `((fun {int -> int -> int} x -> fun {int -> int} y -> x + y end end true) 4)`

5 [E] `(true + 4)`

Answer 5:

5 [E] The applications proceed, according to the dynamic semantics. At the end, addition gets stuck because the first argument is a non-integer value.

Question 6: What is the result of evaluating the following simPL expression according to the denotational semantics of simPL?

```
(recfun f {int -> int} x ->
   (fun {int -> int} y ->
      if x=y then (f x - 1) else y end
    end
    2)
 end
 0)
```

6 [A] 2

6 [B] 1

6 [C] 0

6 [D] infinite loop

6 [E] None of the above.

Answer 6:

6 [A]

Question 7: What is the result of type-checking the following simPL program, using the type checker that was implemented in Lab Task 4?

```
if true then 3 else 4 / 0 end
```

7 [A] `int -> int`

7 [B] Type error

7 [C] `int`

7 [D] Runtime error

7 [E] `3`

Answer 7:

7 [C]

Question 8: What is the result of type-checking the following simPL program, using the type checker that was implemented in Lab Task 4?

```
if true then 3 else 4 / (3 - 3) end
```

8 [A] `3`

8 [B] Type error

8 [C] Runtime error

8 [D] `int -> int`

8 [E] `int`

Answer 8:

8 [E]

Question 9: The type checker that we implemented in the lab tasks flags out the first type error it finds, using Java's exception handling mechanism. More realistic type checkers try to report as many type errors as possible to the user. What is the best way to modify the existing type checker to achieve this goal?

9 [A] Change the class `TypeError` such that it prints out all error messages encountered during type checking.

9 [B] Change the function `check` such that it returns data structure that contains a type along with a list of type errors.

9 [C] Change the `main` function in the class `simPLtypechecker.typecheck`, and run the function `check` again, whenever an exception occurs during type checking.

9 [D] This is not possible using the type checking approach chosen for the language simPL.

9 [E] Change the class `TypeEnvironment` to print out an error message whenever a variable is not found.

9 [F] Change the class `Type` to print out the incorrect types and continue the type checking without exception.

Answer 9:

9 [B] This is the only way to capture multiple type errors in the current structure of the type checker.

Question 10: Consider a new instruction "`MYST`" (for "mystery"), which comes (like "GOTO") with an `ADDRESS`, referring to the index of an instruction in the instruction array. We decide to implement the instruction as follows:

```
case OPCODES.MYST: {
        int x = (((IntValue) os.pop()).value);
        int y = (((IntValue) os.pop()).value);
        if (x > y) {
           pc = pc + 1;
        } else {
           pc = ((MYST)i).ADDRESS;
        }
        break;
}
```

What is the result of executing this instruction with an operand stack after we first push a value v on the operand stack, and then a value w?

10 [A] The instruction will compare the integer values of v and w, and jump to the given `ADDRESS` if v is greater than w, and execute the instruction after `MYST`, otherwise. The instruction removes both v and w from the operand stack.

10 [B] The instruction will compare the integer values of v and w, and jump to the given `ADDRESS` if w is greater than v, and execute the instruction after `MYST`, otherwise. The instruction removes both v and w from the operand stack.

10 [C] The instruction will compare the integer values of v and w, and jump to the given `ADDRESS` if w is less than or equal to v, and execute the instruction after `MYST`, otherwise. The instruction removes both v and w from the operand stack.

10 [D] The instruction will compare the integer values of v and w, and jump to the given `ADDRESS` if v is greater than w, and execute the instruction after `MYST`, otherwise. The operand stack remains unchanged.

10 [E] The instruction will compare the integer values of v and w, and jump to the given `ADDRESS` if w is greater than v, and execute the instruction after `MYST`, otherwise. The operand stack remains unchanged.

Answer 10:

10 [C]

Question 11: What will be the result of modifying our recursive interpreter, based on the denotational semantics of simPL, by replacing the function `extend` in the class `Environment` with the following function:

```
public Environment extend(String v, Value d) {
   if (containsKey(v)) {
      new ErrorValue();
   } else {
      Environment e = (Environment)clone();
      e.put(v,d);
      return e;
   }
```

11 [A] Variables cannot be redefined in nested function definitions and let-expressions.

11 [B] Variable occurrences must be distinct; you cannot write `(f x x)` any longer.

11 [C] All variable declarations have to use different variable names.

11 [D] There is no effect of this change observable at runtime.

11 [E] None of these.

Answer 11:

11 [A] In simPL, the function `extend` will extend the environment that represents the variable bindings of global variables of functions by a binding of parameters to values. So if we prevent variables from being defined if they are already present in an environment, we are preventing redefinition of variables.

Question 12: Let us say the operand stack *os* of the virtual machine for the language simPL has the form

$$\langle 10, 20, 30 \rangle$$

with 10 on top of the stack, and the environment e is of the form

$$e = \emptyset[\texttt{x} \leftarrow 3][\texttt{y} \leftarrow 4]$$

What is the value on top of the operand stack after execution of the following instruction sequence?

```
[LDS x,
PLUS,
TIMES]
```

12 [A] 230

12 [B] 70

12 [C] 320

12 [D] 260

Answer 12:

12 [D] `LDS` pushes the value 3, `PLUS` pops 3 and 10 and pushes 13, and `TIMES` pops 13 and 20 and pushes 260.