Homework 4 solutions

Name: Hung Nguyen

B-ID: B01037287

Question 1

C uses nominal equivalence (two types are equivalent iff they have the same name) to compare types while TypeScript uses structural equivalence (two types are equivalent iff they consist of the same type constructor applied to structurally equivalent types).

T1 and T2 have different names, hence the code doesn't compile in C, but they do in TypeScript because they have the same structure.

Question 2

```
33 12
33 9
```

When `f()` is called, lexical `a` is initialized (line 5), dynamic `x` is initialized (line 6), then enters `g()`. When line 13 is executed, `a` = 33 because `g()` accesses the value of `a` defined in the outer scope at line 1. `x` = 12 because `g()` accesses the value of `x` defined in `f()` at line 6.

At line 20, `a` = 33 because `print()` accesses the value of `a` defined in the outer scope at line 1. `x` = 9 because `print()` accesses the value of `x` set recently by `f()` at line 9.

Question 3

```
watcher() ->
  receive
    { Client, watch, TO_WATCH } ->
      ToWatch ! { self(), watch },
      Client ! { self(), watching, ToWatch },
      watch_process(Client, ToWatch);

    { Client, stop } ->
      Client ! { self(), stopped },
      ok
  end.

watch_process(Client, ToWatch) ->
  receive
    { ToWatch, 'EXIT', Why } ->
        io:format("~p exited ~p~n", [ToWatch, Why])
  end.
```

Question 4

```
11
11
33
11
```

`c.f(sc)` prints 11 then returns an SC object. Because variable `c` is has type `C`, at compile time, `c.f(sc)` is referring to `f(C c)` in `C`. Then it's overridden because instance of `SC` is assigned to `c`, hence it's using `f(C c)` in `SC`. `sc.f(c.f(sc))` prints 11 then returns an SC object. Because `c.f(C c)` returns a `C` object due to the function signature, `f(C c)` is called, not `f(SC c)` even though runtime, `SC` is returned by the inner `c.f(sc)`.

`sc.f(sc)` prints 33 then returns an SC object. Because `sc` variable has type `SC`, `f(SC c)` is called. `c.f(sc.f(sc))` prints 11 then returns an SC object. Because at compile time, `f(C c)` of `C` is being referred. Then it's overridden because instance of `SC` is assigned to `c`, hence it's using `f(C c)` in `SC`.

Question 5

5a

Since SC extends C, SC inherits all fields of C.

```
class SC {
  class pointer    // offset 0
  char c1;         // offset 8
  char c2;         // offset 9 hole 6
  Object obj;      // offset 16
  boolean cond;    // offset 24 hole 3
  int val;         // offset 28
  int v;           // offset 32 hole 4
  Object other;    // offset 40
  pointer to vtbl; // offset 48
}
```

SC has size of 56 bytes.

5b

```
class SC {
  class pointer    // offset 0
  Object obj;      // offset 8
  Object other;    // offset 16
  pointer to vtbl; // offset 24
  int val;         // offset 28
  int v;           // offset 32
  char c1;         // offset 36
```

```
  char c2;          // offset 37
  boolean cond;     // offset 38 hole 1
}
```

SC has size of 40 bytes.

5c

Method table
f_C_C
g_C_SC
h_SC_SC

Question 6

- 1: Legal. Assigning a `C` instance to variable of type `C` is ok because they have the same type.
- 2: Legal. Assigning an `SC` instance to a variable of type `C` is ok because `SC` is a subclass of `C`.
- 3: Illegal. Assigning a `C` instance to a variable `SC` is not ok because class `C` is not class `SC`. `SC`, however, is `C`.
- 4: Legal. Because `sc` has static type `SC`, and `SC` is `C` due to polymorphism, `sc` can be assigned to `c` (type `C`).
- 5: Legal. Because `c1` has static type `C`, which is the same as `c2`'s type.
- 6: Illegal. Because `c2` has static type `C`, which is not the same as `sc`'s type (`SC`). `SC` is, however, `C`.

Question 7

- Haskell's pattern matching is used extensively in function definitions and case expressions, algebraic data types, lists, tuples, and more complex structures.
- Prolog's pattern matching is used to define facts and rules, predicates, which are matched against input data to infer relationships and solve queries.
- Erlang's pattern matching is widely used in function clauses and receive statements to handle message passing, concurrent and distributed systems, as well as for destructuring complex data types.
- Prolog emphasizes declarative programming, where patterns are used to define relationships and solve queries. Haskell and Erlang, on the other hand, focus on functional and concurrent programming paradigms, respectively, where pattern matching is used for defining functions and handling data structures.
- Haskell typically requires exhaustive pattern coverage in function definitions, while Prolog and Erlang provide more flexibility in this regard.
- Erlang and Haskell support guard clauses alongside pattern matching, allowing additional conditions to be checked.

Question 8

8a

Depends on the use case, each has their own pros and cons.

For call-by-value, a copy of the argument's value is passed, this might create overhead for large data structures, and no side effect is created. However, it gives predictable behavior since the original value of the argument is not modified.

For call-by-reference, a reference to the original value is passed to the function. Any modification to the argument would also affect the original value, allowing a flexible behavior and avoid overhead for large structures. However, it can create unintended side effects.

8b

True. we can use list comprehension in Erlang to define a generator.

8c

True. Example in JavaScript:

```
//Immediately Invoked Function Expression (IIFE)
var mod = (function() {  //anonymous function
  var local = ...;       //hidden in function scope
  function f(...) { ... }//hidden in function scope
  function g(...) { ... }//hidden in function scope
  function h(...) { ... }//hidden in function scope
  ...
  return {f, h};    //export
})();
mod.f(); mod.h();   //okay
mod.g(); mod.local; //error
```

8d

False. Parameters of overriding methods cannot be covariant with the corresponding parameters of the overridden methods. Only the return parameter can be covariant. Attempting to change the parameter type in the subclass would violate the signature requirement for overriding.

8e

Cost for using virtual functions in C++ is minimal but not negligible. Because a call to virtual function involves an array lookup. Additionally, it can have memory overhead because each object has to store their own vtable.