Name: _____

Exam #2: Part 1 of 2 - Answers

COMP.3010 – Organization of Programming Languages; April 22, 2019 – Dr. Wilkes

Note: *This exam is closed book and notes, except for one 8.5x11" sheet of paper with handwritten notes (no photocopies) & the printed OCaml Language reference sheet.*

Multiple Choice Questions – 5 points each: Circle the correct answer.

1. In C, an expression such as `(3.14159 + 5)` is an example of:
   a. Type compatibility
   b. Type equivalence
   c. Type inference
   d. None of the above
2. In C, an expression such as `(int) 95.7` is an example of:
   a. Non-converting type cast
   b. Type coercion
   c. Type conversion
   d. None of the above
3. In OCaml, *unification* is a mechanism used in support of:
   a. Type compatibility
   b. Type equivalence
   c. Type inference
   d. None of the above
4. In C, the ability to declare an array with struct elements, or a struct with array fields, is an example of:
   a. Dynamic typing
   b. Orthogonality
   c. Static typing
   d. Strong typing
   e. None of the above
5. In C++, the use of virtual functions is an example of:
   a. Dynamic typing
   b. Orthogonality
   c. Static typing
   d. Strong typing
   e. None of the above
6. (**CIRCLE ALL THAT APPLY**) Which of the following best describe the type system in OCaml?
   a. Dynamically typed using name equivalence
   b. Dynamically typed using structural equivalence
   c. Statically typed using name equivalence
   d. Statically typed using structural equivalence
   e. Strongly typed using name equivalence
   f. Strongly typed using structural equivalence

+5 points for (d) only, or for both (d) and (f)
+3 points for (f) only

Name: _____

7. In OCaml, the expression "`[| 1; 2; 3 |]`" represents:
   a. An array with the three integer elements 1, 2, & 3.
   b. A list with the three integer elements 1, 2, & 3.
   c. A record with the three integer elements 1, 2, & 3.
   d. A tuple with the three integer elements 1, 2, & 3.
   e. A variant with the three integer elements 1, 2, & 3.
   f. None of the above.
8. In OCaml, the expression "`let a = !y in ...`" means:
   a. Local variable `a` is defined as the logical negation of the Boolean variable `y`.
   b. Local variable `a` is defined as the inverse of the integer variable `y`.
   c. Local variable `a` is defined as the value accessed by the reference variable `y`.
   d. Local variable `a` is defined as the value of the element of an array at index `y`.
   e. None of the above.

True/False Questions – 2 points each: Write T or F beside each question.

1. *Option* types allow a programmer to specify that a value (e.g., the return value of a function) is a valid or invalid value. **T**
2. C++ supports both the *denotational* and *abstraction-based* view of types. **T**
3. Programming languages that are highly orthogonal tend to be harder to learn and use than languages that are less orthogonal. **F**
4. The real number type is an example of a discrete type that is not a scalar type. **F**
5. Support for a large number of coercions between different types may decrease the ease of use and understandability of a programming language. **T**
6. Functional languages make extensive use of side effects. **F**
7. Functional languages allow programs to be manipulated with the same mechanisms used to manipulate data. **T**
8. The *lambda calculus* was the inspiration both for functional languages, and new features in C++ and Java. **T**
9. Most functional languages do not support structured function returns. **F**
10. Garbage collection is an essential feature of most functional language systems. **T**

Name: _____

Exam #2: Part 2 of 2 - Answers

COMP.3010 – Organization of Programming Languages; April 22, 2019 – Dr. Wilkes

Note: *This exam is closed book and notes, except for one 8.5x11" sheet of paper with handwritten notes (no photocopies)) & the printed OCaml Language reference sheet.*

Short Answer Questions – 10 points each: Write your answer in the space provided.

1. Why does OCaml provide separate arithmetic operators for integer and floating-point values?
   - To prevent potential errors arising from coercions (implicit type conversions)
   - To help with type inference

2. a. Briefly explain the difference between *physical* and *structural* equality of **values** in OCaml.
   - Physical equality implies that the values being compared are the same object instance in memory (i.e., have the same address).
   - Structural equality implies that each element of the values being compared recursively has the same value, even if they are not the same object instances in memory.

   b. Briefly compare and contrast your answer in part (a) to the difference between *name* and *structural* equivalence of **types** in type systems.
   - Name type equivalence is similar to physical value equivalence: For two variables to have equivalent types under name type equivalence, the declarations of those variables must reference the same type name.
   - Structural type equivalence is similar to structural value equivalence: Two variables are considered to have equivalent types if each element of the types in their declarations are equivalent when compared recursively.

Name: _____

3. In Ada, one may define *derived* types such as `celsius_temp` and `fahrenheit_temp` in the code snippet below. Briefly explain how this capability is useful.

```
procedure ada_types is
type celsius_temp is new integer;
type fahrenheit_temp is new integer;
c : celsius_temp;
f : fahrenheit_temp;
begin
    c := 0;
    f := 32;

    -- c := f;      -- error
    -- f := c;      -- error
end ada_types;
```

The use of derived types can catch certain common errors such as mixing values of different physical units, as in the above example of trying to assign a Celsius temperature value to a variable that expects a Fahrenheit temperature value and vice versa.

4. In C++11 and later, the `auto` keyword was introduced to support type inference. C++11 also went a step further, and introduced the `decltype` keyword that can be used to match the type of any existing expression. This capability is particularly handy in templates, where it is sometimes impossible to provide an appropriate static type name; for example:

```
template <typename A, typename B>
void f(A a, B b) {
   // Declaration of sum - the type of sum is inferred when
   // this template function is instantiated
   decltype(a + b) sum;

   sum = a + b;
}
```

   a. If `A` and `B` are both `int`, what is the type of `sum`?

      If `A` and `B` are both type `int`, the expression `a+b` would also have type `int`, so `sum` will have type `int`.

   b. If `A` is `int` and `B` is `double`, what is the type of `sum`?

      If `A` is `int` and `B` is `double`, the expression `a+b` would have type `double`, so `sum` will have type `double`.