CS131 Spring 23 - Final (Solutions)

June 15th, 2023

Full Name (**LAST**, FIRST): _____

Student ID: _____

Signature: _____

Problem #1: Haskell	/12
Problem #2: Haskell	/16
Problem #3: Memory management	/14
Problem #4: Parametric poly/duck typing	/10
Problem #5: Prolog	/8
Problem #6: Binding semantics	/15
Problem #7: Error Handling	/8
Problem #8: OOP	/6
Problem #9: This n that	/9
Problem #10: Data musings	/8
Problem #11: OOP Typing	/ 9
Total	**/115**

This is the answer key for v1 of the exam. Unless otherwise noted, the answers are the same for the second version of the exam.

1. Haskell – chewing on choose (12 points)

The binomial coefficient (or “choose” function) is an important tool in math and computer science. As a refresher, the binomial coefficient can be written as a function of two variables n and k. Mathematically, we’d say

```
binomial(n, k) = n! / [k!(n-k)!]
```

Throughout this problem, we’ll build up a neat set of tools to work with binomials!

For all subproblems where you write a function, you must include a type annotation. Unless otherwise stated, use `Integer` when representing a numeric type.

Part A (3 points)

a. Write **two** functions:

- A function called ***fact*** that takes in one integer and returns its factorial. For numbers that are less than 0, return the number 0; you may (or may not) recall that 0! = 1.
- A function called ***binom*** that takes in two numbers, n and k, and returns the binomial coefficient as defined above – using ***fact***.

You may assume that n >= k >= 0 (i.e., your solution does not have to work on negative n or k). For example:

```
fact 0 = 1
fact 4 = 24
binom 1 1 = 1
binom 6 4 = 15
```

Answer:

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n - 1)

binom :: Integer -> Integer -> Integer
binom n k = (fact n) `div` ((fact k) * (fact (n - k)))
--- or
binom n k = div (fact n) ((fact k) * (fact (n - k)))
```

Part B (3 points)

Write a function called ***binomials*** that takes in one number, n, and returns a list of each binomial coefficient for k = [0, n], inclusive, in order. You must use a list comprehension and your helper function from Part A. For example:

```
binomials 1 = [1, 1]
binomials 4 = [1, 4, 6, 4, 1]
```

You may assume that n >= 0 (i.e., your solution does not have to work on negative n). Your solution must be no more than 3 lines long (including the type annotation).

Answer:

```
binomials:: Integer -> [Integer]
binomials n = [binom n k | k <- [0 .. n] ]
--- or
binomials n = map (binom n) [0..n]
```

Part C (2 points)

"Pascal's triangle" is a beautiful (infinite) mathematical object that involves binomials. We can represent Pascal's triangle as a list of lists, where the i^{th} list is the result of calling binomials i (starting from i = 0).

First, fill in the blank for the following infinite list comprehension, which should generate Pascal's triangle. You must use use the *binomials* function you defined in Part B.

Here's how your comprehension might be used:

```
ghci> take 5 pascalsTriangle
[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]
```

```
pascalsTriangle :: [[Integer]]
pascalsTriangle = [ ____________________________________ ]
```

Answer:

```
pascalsTriangle = [binomials i | i <- [0..]]
```

Part D (4 points)

Finally, write a function ***hasDivisible*** that takes in one argument n, and returns *only* the rows of Pascal's Triangle that have at least one element evenly divisible by that number. For example:

```
ghci> take 2 (hasDivisible 3)
[[1, 3, 3, 1], [1, 4, 6, 4, 1]]

ghci> take 2 (hasDivisible 2)
[[1, 2, 1], [1, 4, 6, 4, 1]]
```

You may assume that n >= 0 (i.e., your solution does not have to work on negative n). You may use a helper function if needed.

Answer:

```
helper :: Integer -> [Integer]  -> Bool
helper n list = and (map (\x -> x `mod` n == 0) list)

hasDivisible :: Integer -> [[Integer]]
hasDivisible n = filter (helper n) pascalsTriangle

--- or

helper :: Integer -> [Integer]  -> Bool
helper n list = ((filter (\x -> (mod x n) == 0) list) /= [])

hasDivisible :: Integer -> [[Integer]]
hasDivisible n = filter (helper n) pascalsTriangle
```

2. Haskell – parsing parsers (16 points)

For your class projects, we provided you a parser. In this problem, we'll implement some parser-related functionality – and see how functional programming makes writing parsers simple! For all functions you write, you must include a type annotation.

Part A (8 points)

In almost all languages, keeping track of parentheses is an important task. Write a predicate function ***hasBalancedParens*** that takes in a list of strings (i.e., tokens in a program), and returns true if and only if the parentheses in the string are "balanced". There are 2 requirements:

1. The number of opening and closing parentheses should be the same (including 0).
2. Every closing parenthesis must have exactly one matching open parenthesis to its left.

```
hasBalancedParens ["hello world"] = true
hasBalancedParens ["print", "(", "3", "+", "5", ")"] = true
hasBalancedParens ["print", "(", "3", "+", "5"] = false
hasBalancedParens ["print", "(", "3", "+", "5", ")", ")"] = false
hasBalancedParens ["print", ")", "3", "+", "5", "("] = false
```

Write your answer below. Hint: you almost certainly want to write a helper function.

Answer:

```
hasBalancedParens :: [String] -> bool
hasBalancedParens p =
  let
    bal opens [] = opens == 0
    bal opens (x:xs)
     | x == "(" = bal (opens + 1) xs
     | x == ")" = if opens == 0 then False else bal (opens - 1) xs
     | otherwise = bal opens xs
  in
     bal 0 p
```

Part B (8 points)
Expression evaluation in programming languages is often represented as a tree. Sometimes, you can apply optimizations to these trees to make computation faster. One such optimization is zero-product replacement, where you replace any term multiplied by the constant 0 with 0 itself.

Consider this ADT, which only supports addition and multiplication:

```
data Expr = Multiply Expr Expr | Add Expr Expr | N Integer
```

We could represent the following prefix expression:

(* (+ 3 5) (+ 0 0)) as: `(Multiply (Add (N 3) (N 5)) (Add (N 0) (N 0)))`

Write a function called ***zeroOptimizer*** that takes in an *Expr* and returns an equivalent *Expr*, but with zero-product replacement. That is, whenever one or both of a Multiply expression's children starts out as a number with a value of zero, or has been converted by a recursive call to zeroOptimizer into a number with a value of zero, replace the Multiply variant with `(N 0)`, e.g.:

```
ghci> zeroOptimizer (Multiply (N 5) (Add (N 0) (N 0)))
(Multiply (N 5) (Add (N 0) (N 0)))

ghci> zeroOptimizer (Add (Multiply (Add (N 1) (N 2)) (N 0)) (N 6))
(Add (N 0) (N 6))

ghci> zeroOptimizer (Multiply (Add (N 3) (N 5)) (Multiply (N 0) (N 0)))
(N 0)
```

Write your answer below. Hint: you almost certainly want to write a helper function.

```
isN0::Expr -> Bool
isN0 (N 0) = True
isN0 _ = False

zeroOptimizer :: Expr -> Expr
zeroOptimizer (N v) = N v
zeroOptimizer (Add a b) = Add (zeroOptimizer a) (zeroOptimizer b)
zeroOptimizer (Multiply a b) =
  let newA = zeroOptimizer a
      newB = zeroOptimizer b
  in if isN0 newA || isN0 newB
       then N 0
       else Multiply newA newB
```

3. Memory Management (14 points)

All of these subproblems are about memory management in real life. The questions are moderately open-ended, so any reasonable and well-justified answer will receive full marks.

Part A (4 points)

Apple designed the Swift programming language to, in part, make it easier to develop user-facing applications like iOS and macOS apps. When deciding on Swift's memory management approach, the language designers picked "Automatic Reference Counting" (ARC): an implementation of the reference counting algorithm we discussed in class.

For the device user's experience – particularly in runtime performance – why might Swift's designers pick reference counting over a tracing algorithm (e.g. mark-sweep, mark-compact)? - Your answer must reference both approaches, and you must limit your answer to five sentences.

Write your answer here:

Answer - both parts required:

First, reference counting occurs incrementally, where each object is deallocated as soon as the reference count reaches zero. In contrast, mark-sweep and mark-compact happen in bulk. So, reference counting frees less memory more frequently, which will pause execution for smaller amounts of time – reducing the length of UI lag in the average case. While there may be cascades of objects being freed in some cases with reference counting, this would be the outlier case.

Second, reference counting happens immediately (and deterministically), while tracing algorithms occur non-deterministically (to the programming language). So, it's harder to make guarantees about how frequent or long UI lag will be or when it will occur within the execution of the program.

Part B (6 points)

In many practical applications, the amount of memory used by a program is the most important constraint to optimize for (as opposed to performance). For example, many embedded systems – like the microcontroller on an air conditioning unit – have very little onboard memory.

Let's assume we're writing a program and can choose the memory management scheme we use. Consider the four types of memory management that we have discussed in class:

- Manual memory management (like in C++)
- Reference Counting
- Mark and Sweep
- Mark and Compact

Rank each memory management approach from best to worst in terms of memory overhead. In particular, assume that your program is correctly implemented (i.e. it has no bugs); justify your ranking with a concise argument touching on either average-case or worst-case memory usage.

To receive full credit, you must discuss all four approaches.

Answer:

If we focus simply on the memory overhead used by the algorithm (as opposed to any fragmentation that might occur), this would be our answer:

- 1st: manual memory mgmt: this has the least overhead since there is no extra memory required to support garbage collection, reference counts or mark-and-sweep tags/queues, so assuming a program is correctly implemented, manual memory management will use the least amount of additional memory
- 2nd: a close tie between mark and sweep and reference counting (either is OK)
    - mark and sweep which requires a simple flag/bit to indicate whether an object has been marked in the marking phase; this is stored along with each object. This bit might be stored in an existing byte/word that serves multiple purposes, so it may not incur much additional overhead. A queue is also needed to process the items (there are clever ways to avoid using a separate queue).
    - reference counting requires a reference count for each object, which will typically be a processor word (8 bytes on a modern computer)
- 4th: mark and compact since it must allocate half of the memory for a compaction space

If we focus on fragmentation and average amount of memory that is unusable, the following would be our answer:

- 1st: manual memory mgmt: still the best, since there's no overhead for reference counts or tags or compaction memory. It's subject to fragmentation, but coalescing of adjacent blocks into larger free blocks at the time of deletion reduces this for the most part.

- 2nd: reference counting ensures memory is freed immediately in the average case, so it will ensure that more memory is available for use at all times. Still subject to the same fragmentation (and coalescing) described for manual memory management.
- 3rd: mark and sweep frees memory only when there is memory pressure, which means that on average, more memory is tied up and unusable (e.g., for other processes) until there's enough memory pressure in a process to force a mark-and-sweep operation. Fragmentation (and coalescing) is the same as for manual memory management and reference counting
- 4th: mark and compact is still last since it must allocate half of the memory for a compaction space. Even though this addresses fragmentation, given the 50% memory overhead used for the compaction phase, with virtually all fragmentation scenarios except for the most pathological, mark and compact will use more memory than the other schemes.

Part C (4 points)

Consider a language which uses mark and sweep garbage collection, has **mutable** variables, and supports concurrency. What issues might the garbage collection system run into when running programs that use concurrency, and how might we address these issues? Limit your answer to five sentences.

Answer:

Object references/pointers might experience race conditions (e.g., where two threads are modifying the same object reference at the same time) and it's unclear if a given object reference is valid/invalid at the time a process is paused to perform a mark and sweep operation. In a mark-and-sweep implementation that doesn't account for this, this leads to (a) memory leaks due to objects that are not garbage collected, and even worse, it can lead to (b) garbage collection of objects that should not be garbage collected (resulting in dangling object references).

The solution to this problem is to sleep all threads at key points when no object references are in the process of being modified before performing the mark-and-sweep process. Simply putting all threads to sleep without taking into account whether object references are in the middle of modification is not sufficient. Another possible (but less efficient) answer is to use a global interpreter lock (GIL) like Python uses, which ensures that no two threads will modify object references at the same time, such that when a mark-and-sweep operation occurs there is no chance of race conditions.

4 Parametric Polymorphism and Duck Typing (10 points)

Part A (6 points)

Between C++ templates and Java-like generics, is one approach “more conservative” than the other? In other words, does either approach disallow code at compile-time that could technically be valid code? Justify your answer with an example (it does not need to be actual code).

Answer:

Unbounded generics (the default if you don't specify a bounding type) are more conservative than templates, since you can't use any class-specific operations that are not supported by all types of objects/data types. You must limit its operations on parameterized variables to things like assignment/instantiation/comparison. Even bounded generics are more conservative, because the generic can only use operations associated with the specific bounding type. But with templates, the parameterized code can use *any* operations it likes (as long as during compilation, those operations end up being compatible with the parameterized type). For example, given a class X with method Y(), if we used X with a generic function, it could not call Y() from an unbounded generic. But if we used X with a template, calling Y() would be allowed.

Part B (4 points)

Now consider a **bounded** generic function in a language like Java and a function that uses duck typing in a language like Python, both of which accept an object reference as a parameter. Which one is more likely to result in **runtime errors** due to an invalid method being called through the object reference? Justify your answer.

Answer:

The function using duck typing will be more likely to result in runtime errors. Why? A function using duck typing can call any method it likes, and the validity of the method call can only be determined at runtime. In contrast, a bounded generic can only call methods associated with the bounded type, so those operations are guaranteed to be safe at runtime (with the exception of downcasts, which may result in a runtime type check error).

5. Prolog (8 points)

Consider the following Prolog program that defines a knowledge base of courses, their prerequisites, and courses that are required by the major:

```
prerequisite(cs31, cs32).
prerequisite(cs32, cs33).
prerequisite(cs33, cs35l).
prerequisite(cs33, cs131).
prerequisite(cs33, cs151b).

major_requirement(computer_science, cs131).
major_requirement(electrical_engineering, cs151b).
```

Write a Prolog predicate called ***is_major_prereq(course_name, major)*** that determines if a given course is *directly or indirectly* a prerequisite course for a course required by a major. You must ensure correct syntax for full points.

```
Example query:  is_major_prereq(cs31, computer_science)
Expected output: true     % cs31 -> cs32 -> cs33 -> cs131

Example query:  is_major_prereq(cs35l, electrical_engineering)
Expected output: false   % cs35l isn't a prereq for any EE major reqts.
```

Answer:

```
is_major_prereq(Course, Major) :-
    major_requirement(Major, RequiredCourse),          % line A
    is_prereq(Course, RequiredCourse).                 % line B

is_prereq(Course, RequiredCourse) :-        % helper - may be named anything
    prerequisite(Course, RequiredCourse).

is_prereq(Course, RequiredCourse) :-    % name must be same as other helper
    prerequisite(Course, IntermediateCourse),        % line C
    is_prereq(IntermediateCourse, RequiredCourse).   % line D

%%%% this version of is_prereq is also ok

is_prereq(Course, RequiredCourse) :-    % name must be same as other helper
    prerequisite(Course, IntermediateCourse),              % line C
    is_prereq(IntermediateCourse, RequiredCourse) ; % semicolon is "or"
```

```
    prerequisite(Course, RequiredCourse).


%%%% Other possible solution:

prerequisite(Course, RequiredCourse):-
   prerequisite(Course, IntermediateCourse),
   prerequisite(IntermediateCourse, RequiredCourse).

is_major_prereq(Course, Major) :-
    major_requirement(Major, RequiredCourse),          % line A
    prerequisite(Course, RequiredCourse).              % line B

%%% Other possible solution:

is_major_prereq(C,M) :- major_requirement(M,D), prerequisite(C,D).
is_major_prereq(C,M) :- prerequisite(C,D), is_major_prereq(D,M).
```

6. Binding Semantics (15 points)

Consider this program in a hypothetical language called !Brewin:

```
struct Dog:
  String name
  int bark

def foo(Dog d) -> void:
  d.bark = 20
  d = Dog{"Koda", 5}
  d.bark = 10

def bar() -> int:
  print("bar")
  return 1 + 2

def main() -> void:
  var d1 = Dog{"Kippy", bar()}
  var d2 = d1
  foo(d1)
  // print results
  print("d1.name = ", d1.name, "d1.bark = ", d1.bark)
  print("d2.name = ", d2.name, "d2.bark = ", d2.bark)
```

Part A (3 points)

What would this program print if the language were using **object reference semantics** for all variable definitions/assignments (i.e., d1 and d2) and **pass-by-object reference** for all parameters (e.g., d)?

Answer v1:

bar
d1.name = Kippy d1.bark = 20
d2.name = Kippy d2.bark = 20

Answer v2:

foo
d1.name = Koda d1.bark = 11
d2.name = Koda d2.bark = 11

Part B (3 points)

What would this program print if the language were using **value semantics** for all variable definitions/assignments (i.e., d1 and d2) and **pass-by-reference** for all parameters (e.g., d)?

Answer v1:

```
bar
d1.name = Koda d1.bark = 10
d2.name = Kippy d2.bark = 3
```

Answer v2:

```
foo
d1.name = Skip d1.bark = 21
d2.name = Koda d2.bark = 7
```

Part C (3 points)

What would this program print if the language were using **value semantics** for all variable definitions/assignments (i.e., d1 and d2) and **pass-by-value** for all parameters (e.g., d)?

VERSION #1 OF THE EXAM (integrity.)

Answer v1:

```
bar
d1.name = Kippy d1.bark = 3
d2.name = Kippy d2.bark = 3
```

Answer v2:

```
foo
d1.name = Koda d1.bark = 7
d2.name = Koda d2.bark = 7
```

Part D (3 points)

What would this program print if the language were using **object reference semantics** for all variable definitions/assignments (i.e., d1 and d2) and **pass-by-reference** for all parameters (e.g., in the call foo(d1), d would be a reference to the object reference d1)?

Answer v1:

```
bar
```

d1.name = Koda d1.bark = 10
d2.name = Kippy d2.bark = 20

Answer v2:

foo
d1.name = Skip d1.bark = 21
d2.name = Koda d2.bark = 11

Part E (3 points)

At the moment the program reaches the *"// print results"* line (before the print statements execute), what would this program have printed if the language were **lazy** like Haskell and used **name semantics** for all variable definitions/assignments and **pass-by-name** for all parameters?

Answer v1:

// NO BAR PRINTED/nothing printed

Answer v2:

// NO FOO PRINTED/nothing printed

7. Exception Handling (8 points)

Consider the following Python program:

```
def foo(x, y):
  print('Alpha')
  print(x / y)
  print('Beta')

def bar(x, y):
  print('Gamma')
  foo(x, y)
  print("Delta")

def main():
  bar(10, "foo")
  bar(10, 0) # Line Q
```

Part A (2 points)

Prior to any exception that terminates the program, what does this program print? Why?

Answer v1:

- 2 points for:
  Gamma
  Alpha
- 0 points for everything else

Answer v2:

Kappa
Alpha

Part B (6 points)

Assume that Python has only three types of exceptions which can be caught: *ZeroDivisionError*, *TypeError*, and *Exception*. Make the smallest/simplest possible change to the above program by adding exception handler(s) such that it prints out the following **lines** *before* it terminates by throwing an exception on line Q:

```
Gamma
Alpha
Phew!
Delta
Gamma
Alpha
ZeroDivisionError: division by zero      # thrown by line Q
```

You MUST NOT make any other changes beyond adding exception handler(s) which somehow print "Phew!". You **MUST NOT** use any ***finally*** **clauses** or **return** statements in your handler.

Show the updated code here (show only the functions that you changed):

Answer v1:

```
def bar(x, y):
  print("Gamma")
  try:
    foo(x,y)
  except TypeError:
    print("Phew!")
  print("Delta")
```

Answer v2:

```
def bar(x, y):
  print("Kappa")
  try:
    foo(x,y)
  except TypeError:
    print("Whoa!")
  print("Delta")
```

8. Object Oriented Programming (6 points)

Consider the following C++ program:

```
class A {
public:
  virtual void foo() { cout << "A"; }
  void bar() { foo(); c++; }
  void bletch() { cout << c; }
private:
  static int c;  // class variable
};

class B : public A {
public:
  virtual void foo() { cout << "B"; }
};

int A::c = 0;  // c starts out at zero

int main() {
  A* v[2] = { new A(), new B() };

  for (int i = 0; i < 2; i++) {
    v[i]->bletch();
    v[i]->foo();
    v[i]->bar();
  }

  for (int i = 0; i < 2; i++)
    v[i]->bletch();
}
```

What is the output of this program?

Answer v1:

0AA1BB22

Answer v2:

5BB6DD77

9. This n That (9 points)

Part A (3 points)

Consider the following code in C++:

```
int a() { cout << "A"; return 4; }
int b() { cout << "B"; return 3; }
int c() { cout << "C"; return 2; }
int d() { cout << "D"; return 1; }
int e() { cout << "E"; return 5; }

int main() {
  if (a() > 10 || (b() > 2 || c() > 7) || d() < 7 && e() > 0)
    cout << "F";
}
```

What does it print (choose one)?

A. ABDEF
B. ADF
C. ABCD
D. ABF
E. ADEF

Answer: ABF (D on exam v1, B on exam v2)

Part B (3 points)

Given the *foo* function in Haskell below:

```
foo _ []     = []
foo f (x:xs) = f x : foo f xs
```

Which of the following is a valid type signature of the foo function (choose one)?

A. foo :: (a -> b) -> [a] -> b
B. foo :: (a -> b) -> ([a] -> [b])
C. foo :: (a -> b) -> ([a] -> b)
D. foo :: ([a] -> [b]) -> ([a] -> [b])
E. foo :: ((a -> b) -> [a]) -> [b]

VERSION #1 OF THE EXAM (integrity.)

Answer: foo :: (a -> b) -> ([a] -> [b]) (B on exam v1, E on exam v2)

Part C (3 points)

What does this Python code print out?

```
def foo(x):
 print("B")
 for i in x:
   yield i
   print("C")

a = foo([1,2,3])
print("A")
print(next(a))
print(next(a))
```

What does it print (choose one)?

A. B, A, 1, C, 2
B. B, A, 1, C, 2, C, 3
C. A, B, 1, C, 2
D. A, B, 1, C, 2, C
E. B, A, 1, C, 2, C

Answer: A, B, 1, C, 2 (C on exam v1, A on exam v2)

10. Data musings (8 points)

Consider the following code excerpt from an hypothetical language, which compiles without errors. When you run it, it executes without any errors up until it throws an exception on line Q:

```
func print_ages(Animal arr[]): void {
  for (for i = 0; i < NUM_ANIMALS; ++i)
    print(arr[i].age)
}

func main(): void {
  var NUM_ANIMALS = 10
  Animal pack[NUM_ANIMALS]    // An array of Animal object references
  pack[0] = new Dog(name: "Koda", age: 1)
  pack[1] = new Dog(name: "Kippy", age: 7)
  print_ages(arr: pack)
  pack[0].age = -10     // Line Q
}
```

Part A (2 points)

What can we conclude about the scoping approach used by this language? How do we know this?

Answer: This language uses dynamic scoping. We know this because the value of NUM_ANIMALS referenced in print_ages would be out of scope if lexical scoping were to be used.

Part B (2 points)

Assuming that this language only creates new objects when the *new* command is explicitly used (e.g., `new Dog(name: "Koda", age: 1)`), what can we conclude about the typing approach(es) used by this language? Why?

Answer: This language is (a) statically-typed (or gradually-typed language) and (b) weakly-typed. We know it is statically-typed because the types of variables are given (e.g., Animal arr[]). We know it is weakly typed because the function enumerates through uninitialized items in the array in slots 2-9 (everything past the first two slots which have object references to valid Dog objects) and accesses the invalid memory that is pointed to without errors/exceptions.

Part C (2 points)

What can we conclude about the Dog class given that the program throws an exception during execution on line Q?

Answer: The Dog class must be using a "mutator method", "setter method" or "property method" (any of the terms are acceptable) to change the age field. The method throws an exception when an invalid/negative age is specified.

Part D (2 points)

Does this program perform any conversions or casts? If so, list those that you find, detailing whether they are widening or narrowing.

Answer: Yes the assignments of the pack[0] and pack[1] to new Dog objects are performing casts (NOT conversions or coercions). These casts are widening because they convert from a subtype (Dog) to a supertype (Animal).

11. OOP (9 points)

Imagine a language where a derived class could indicate that one or more of the public methods defined in its base class will be converted to private methods in the derived class, e.g.:

```
class Animal {
public:
   virtual void eat() { … }
   virtual void sleep() { … }
   virtual void poop_in_public() { … } // ok for animals to do…
};

class Person: public Animal {
public:
  string get_name() const;
private:
   void poop_in_public() { … }        // but not appropriate for people
};
```

This means that for example, if you created a Person object p in your main() function, you could not call p.poop_in_public(), because this method is private in Persons.

Part A (3 points)

Describe the typing relationship between the Animal class and the Person class, using terms like subtype, supertype, etc. Explain your reasoning.

Answer:

The types of the two classes are unrelated, or Person is not a subtype of Animal, or Animal is not a supertype of Person (any of the above is sufficient). Why? Because to be a subtype, the type must implement all of the operations (at a minimum) supported by the supertype. But in this case, the derived class implements a subset of the operations of the supertype, since it doesn't support the poop_in_public() operation.

Part B (3 points)

Assuming the language were statically typed, could we use polymorphism with these two classes? Explain your reasoning.

Answer:

No, we cannot use **subtype** polymorphism. Why? Since the two classes have unrelated types (Person is not a subtype of Animal), we cannot pass an object of a Person type to a function that accepts an Animal type. That said, we can use parametric/ad-hoc polymorphism (e.g., templates and unbounded generics). **Bounded** parametric polymorphism is NOT fine, so we couldn't use a Person with a generic bounded by the Animal type).

Part C (3 points)

What category of inheritance (that we learned in class) would be semantically most similar to this method-hiding approach, from the perspective of typing? Why?

Answer:

This would be most similar to implementation inheritance (aka private inheritance). Why? Since in implementation inheritance, the derived class hides all of the public interfaces of the base class, and the derived class is therefore not a subtype of the base class. In the above question, we hide only one method (instead of all the methods of the base class) but the result is the same. The derived class is no longer a subtype of the base class's type.