Homework 4 Solutions

This homework covers the following topics from class:

- Data palooza, part 1
  - Data: Variables vs Values, Types, Typing Strategies (Static vs. Dynamic, Duck Typing)
- Data palooza, part 2
  - Typing Strategies, cont. (Gradual Typing, Weak vs. Strong Typing), Casting and Conversion

PYTH9: Map, Filter, Reduce, Lambdas (4 min)

(4 min.) Consider the following Python function that takes in a string `sentence` and a set of characters `chars_to_remove` and returns `sentence` with all characters in `chars_to_remove` removed. Fill in the blank so the function works correctly. **You may not use a separate helper function**.

Hint: You may find some of the concepts we learned in our functional programming discussions useful 😁.

Example:

`strip_characters("Hello, world!", {"o", "h", "l"})` should return `"He, wrd!"`

```
def strip_characters(sentence, chars_to_remove):
  return "".join(_____________________________________________________)
```

Solution:

```
def strip_characters(sentence, chars_to_remove):
  return "".join(filter(lambda x: x not in chars_to_remove, sentence))
```

PYTH10: Closures (4 min)

(4 min.) Show, by example, whether or not Python supports closures. Briefly explain your reasoning.

Solution:

We need to show that it is possible to bundle together function code with references to outside state (captured variables). We can use either a nested function or lambda for this. The following snippet from the REPL suffices:

```
>>> def foo():
...   b = 5
...   f = lambda x: x + b
...   return f
...
>>> foo()(0)
5
```

foo returns a closure that has captured b, indicating that closures must be present.

PYTH11: List Comprehensions (12 min)

Part A: (3 min.) Consider the following function that takes in a list of integers (either 0 or 1) representing a binary number and returns the decimal value of that number. Fill in the blanks in the list comprehension so the function works correctly.

Example:

`convert_to_decimal([1, 0, 1, 1, 0])` should return 22.

`convert_to_decimal([1, 0, 1])` should return 5.

```
from functools import reduce
def convert_to_decimal(bits):
  exponents = range(len(bits)-1, -1, -1)
  nums = [_____________ for _______, _______ in zip(bits, exponents)]
  return reduce(lambda acc, num: acc + num, nums)
```

Solution:

```
from functools import reduce
def convert_to_decimal(bits):
  exponents = range(len(bits)-1, -1, -1)
  nums = [bit * 2**x for bit, x in zip(bits, exponents)]
  return reduce(lambda acc, num: acc + num, nums)
```

Part B: (5 min.) Write a Python function named `parse_csv` that takes in a list of strings named `lines`. Each string in `lines` contains a word, followed by a comma, followed by some number (e.g. `"apple,8"`). Your function should return a new list where each string has been converted to a tuple containing the word and the integer (i.e. the tuple should be of type `(string, int)`). **Your function's implementation should be a single, one-line nested list comprehension**.

Example:

`parse_csv(["apple,8", "pear,24", "gooseberry,-2"]` should return `[("apple", 8), ("pear", 24), ("gooseberry", -2)]`.

Hint: You may find list unpacking useful.

Solution:

```
def parse_csv(lines):
  return [(x, int(y)) for x, y in [line.split(",") for line in lines]]
```

Part C: (2 min.) Write a Python function named `unique_characters` that takes in a string `sentence` and returns a set of every unique character in that string. **Your function's implementation should be a single, one-line set comprehension.**

Example:
`unique_characters("happy")` should return `{"h", "a", "p", "y"}`.

Solution:

```
def unique_characters(sentence):
  return {s for s in sentence}
```

Part D: (2 min.) Write a Python function named `squares_dict` that takes in an integer `lower_bound` and an integer `upper_bound` and returns a dictionary of all integers between `lower_bound` and `upper_bound` (inclusive) mapped to their squared value. You may assume `lower_bound` is strictly less than `upper_bound`. **Your function's implementation should be a single, one-line dictionary comprehension.**

Example:
`squares_dict(1, 5)` should return

`{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}`.

Solution:

```
def squares_dict(lower_bound, upper_bound):
  return {x: x**2 for x in range(lower_bound, upper_bound+1)}
```

HASK17: Algebraic Data Types, Recursion (30 min)

In this question, we'll examine how the choice of programming language/paradigm can affect the difficulty of a given task.

Part A: (5 min.) Using C++, write a function named `longestRun` that takes in a `vector` of booleans and returns the length of the longest consecutive sequence of `true` values in that vector.

Examples:
Given `{true, true, false, true, true, true, false}`, `longestRun(vec)` should return `3`.
Given `{true, false, true, true}`, `longestRun(vec)` should return `2`.

Solution:

```
int longestRun(vector<bool> vec) {
  int currentMax = 0;
  int currentRun = 0;
  for (bool element : vec) {
    if (element) currentRun++;
    else {
```

```
      currentMax = max(currentMax, currentRun);
      currentRun = 0;
    }
  }
  return max(currentMax, currentRun);
}
```

`Part B:` (10 min.) Using Haskell, write a function named `longest_run` that takes in a list of `Bool`s and returns the length of the longest consecutive sequence of `True` values in that list.

Examples:

`longest_run [True, True, False, True, True, True, False]` should return `3`.

`longest_run [True, False, True, True]` should return `2`.

Solution using helper parameters:

```
longest_run :: [Bool] -> Int
longest_run xs =
  let
    longest_run_helper [] current_run current_max =
      max current_run current_max
    longest_run_helper (x:xs) current_run current_max =
      if x
        then longest_run_helper xs (current_run+1) current_max
        else longest_run_helper xs 0 (max current_run current_max)
  in
    longest_run_helper xs 0 0
```

Fancy solutions using scanl courtesy of Timothy Gu:

```
longest_run :: [Bool] -> Int
longest_run l = maximum (scanl (\p x -> if x then p + 1 else 0) 0 l)
```

```haskell
longest_run :: [Bool] -> Int
longest_run = maximum . scanl (\p x -> if x then p + 1 else 0) 0
```

`Part C:` (10 min.) Consider the following C++ class:

```cpp
#include <vector>
using namespace std;

class Tree {
public:
  unsigned value;
  vector<Tree *> children;

  Tree(unsigned value, vector<Tree *> children) {
    this->value = value;
    this->children = children;
  }
};
```

Using C++, write a function named `maxTreeValue` that takes in a `Tree` pointer `root` and returns the largest value within the tree. If `root` is `nullptr`, return `0`. **This function may not contain any recursive calls.**

Solution:

```cpp
#import <queue>

unsigned maxValue(Tree *root) {
  unsigned currentMax = 0;
  queue<Tree *> nodesToExplore;
  nodesToExplore.push(root);

  while (!nodesToExplore.empty()) {
```

```
    Tree *current = nodesToExplore.front();
    nodesToExplore.pop();
    if (!current) continue;

    unsigned value = current->value;
    vector<Tree *> children = current->children;

    if (value > currentMax)
      currentMax = value;
    for (Tree *child : children)
      nodesToExplore.push(child);
  }

  return currentMax;
}
```

Part D: (5 min.) Consider the following Haskell data type:

```
data Tree = Empty | Node Integer [Tree]
```

Using Haskell, write a function named `max_tree_value` that takes in a `Tree` and returns the largest `Integer` in the `Tree`. Assume that all values in the tree are non-negative. If the root is `Empty`, return `0`.

Example:
`max_tree_value (Node 3 [(Node 2 [Node 7 []]), (Node 5 [Node 4 []])])` should return 7.

Solutions:

```
-- hand-coded recursive solution
max_tree_value :: Tree -> Integer
max_tree_value Empty = 0
max_tree_value (Node val []) = val
max_tree_value (Node val lst) = max val (max_aux lst)
```

```
  where
    max_aux [] = 0
    max_aux (x:xs) = max (max_tree_value x) (max_aux xs)
```

```
-- map-based solution
max_tree_value :: Tree -> Integer
max_tree_value Empty = 0
max_tree_value (Node value children) =
  case children of
    [] -> value
    xs -> max value (maximum (map max_tree_value xs))
```

HASK18: Algebraic Data Types, Recursion (20 min)

(20 min.) Super Giuseppe is a hero trying to save Princess Watermelon from the clutches of the evil villain Oogway. He has a long journey ahead of him, which is comprised of many events:

```
data Event = Travel Integer | Fight Integer | Heal Integer
```

Super Giuseppe begins his adventure with 100 hit points, and may never exceed that amount. When he encounters:
A `Travel` event, the `Integer` represents the distance he needs to travel. During this time, he heals for ¼ of the distance traveled (floor division).
A `Fight` event, the `Integer` represents the amount of hit points he loses from the fight.
A `Heal` event, the `Integer` represents the amount of hit points he heals after consuming his favorite power-up, the bittermelon.

If Super Giuseppe has 40 or fewer life points after an Event, he enters defensive mode.

While in this mode, he takes half the damage from fights (floor division), but he no longer heals while traveling. Nothing changes with Heal events. Once he heals **above** the 40 point threshold following an Event, he returns to his normal mode (as described above).

Write a Haskell function named `super_giuseppe` that takes in a list of `Events` that comprise Super Giuseppe's journey, and returns the number of hit points that he has at the end of it. If his hit points hit 0 or below at any point, then he has unfortunately Game Over-ed and the function should return -1.

Examples:
`super_giuseppe [Heal 20, Fight 20, Travel 40, Fight 60, Travel 80, Heal 30, Fight 40, Fight 20]` should return 10.
`super_giuseppe [Heal 40, Fight 70, Travel 100, Fight 60, Heal 40]` should return -1.

Solutions:

```haskell
-- applying each event recursively
super_giuseppe xs =
  let normal_step (Travel distance) hp =
        min 100 (hp + distance `div` 4)
      normal_step (Fight loss) hp = hp - loss
      normal_step (Heal amount) hp = min 100 (hp + amount)
      defensive_step (Travel _) hp = hp
      defensive_step (Fight loss) hp = hp - (loss `div` 2)
      defensive_step (Heal amount) hp = min 100 (hp + amount)
      stepper [] hp = hp
      stepper (x:xs) hp =
        let new_hp = if hp <= 40 then defensive_step x hp
                                 else normal_step x hp
        in if new_hp <= 0 then (-1) else stepper xs new_hp
  in stepper xs 100
```

or:

```haskell
-- using mutual recursion
super_giuseppe xs =
 let normal_mode [] hp = hp
     normal_mode ((Travel distance):xs) hp =
     normal_mode xs (min (hp + (distance `div` 4)) 100)
     normal_mode ((Fight loss):xs) hp
       | new_hp <= 0 = (-1)
       | new_hp <= 40 = defensive_mode xs new_hp
       | otherwise = normal_mode xs new_hp
       where new_hp = hp - loss
     normal_mode ((Heal amount):xs) hp =
       normal_mode xs (min (hp + amount) 100)
     defensive_mode [] hp = hp
     defensive_mode ((Travel _):xs) hp = defensive_mode xs hp
     defensive_mode ((Fight loss):xs) hp =
       let new_hp = hp - (loss `div` 2)
       in if new_hp <= 0 then (-1) else defensive_mode xs new_hp
     defensive_mode ((Heal amount):xs) hp =
       let new_hp = min (hp + amount) 100
       in if new_hp > 40 then normal_mode xs new_hp
                         else defensive_mode xs new_hp
 in normal_mode xs 100
```

HASK19: Tail Recursion (10 min)

Tail recursion is a type of recursion where the recursive call is the final operation in the function, meaning no additional computation occurs after the recursive call returns (including arithmetic operations, assignments, etc.). This allows the compiler or interpreter to optimize the recursive function by reusing the same stack frame for each recursive step, essentially compiling the recursive function into one that's iterative, avoiding the need to keep multiple frames in memory. As a result, tail recursion can handle deeper recursive

calls without running into stack overflow issues, making it more efficient in terms of memory usage compared to non-tail-recursive functions.

Here's an example of the factorial function written in a manner which is not tail recursive:

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

Here, the multiplication (*) occurs after the recursive call to `factorial`. This means that for each recursive call, the current stack frame (which holds the value of n) must be kept in memory until the recursive call completes, at which point the result is multiplied by n. With deep recursion, this leads to high memory consumption–O(n) in this case–because each recursive step requires its own stack frame.

Now, here's a tail-recursive version of the factorial function:

```haskell
factorialTail :: Integer -> Integer
factorialTail n = helper n 1
  where
    helper 0 acc = acc
    helper n acc = helper (n - 1) (acc * n)
```

In the tail-recursive version, we introduce a helper function with an accumulator (`acc`) that stores the running result of the factorial. Initially, the accumulator is set to 1. During each recursive call, we pass the updated value of `acc` (the product of the previous `acc` and the current n) to the next step, and once the base case (`n == 0`) is reached, the accumulator holds the final result. Since no operations occur after the recursive call, the compiler can optimize the recursion by using the same stack frame for each call. This effectively reduces

the recursion to a loop, eliminating the need for additional stack frames and making the process much more memory efficient.

Part A:

Fill in the blanks to complete this Haskell function, **`sumSquares`**, that calculates the sum of squares of the first `n` natural numbers, i.e., sumSquares(n)=12+22+...+n2\text{sumSquares}(n) = 1^2 + 2^2 + ... + n^2sumSquares(n)=12+22+...+n2. This version **should not use tail recursion**.

```
sumSquares :: Integer -> Integer
sumSquares 0 = ________
sumSquares n = _____________________________
```

Solution:

```
sumSquares :: Integer -> Integer
sumSquares 0 = 0
sumSquares n = n^2 + sumSquares (n - 1)
```

Part B:

Rewrite the sumSquares function to be tail-recursive. Introduce an accumulator to ensure that the recursive call is the last operation.

Solution:

To make it tail-recursive, we introduce an accumulator that holds the running sum.

```
sumSquaresTail :: Integer -> Integer
sumSquaresTail n = helper n 0
  where
    helper 0 acc = acc
```

```
    helper n acc = helper (n - 1) (acc + n^2)
```

In this version, the `helper` function takes two arguments: the current number `n` and the accumulator `acc`, which stores the sum of squares so far. The recursive call `helper (n - 1) (acc + n^2)` is the last operation, making it tail-recursive.

This version demonstrates better stack management and allows the function to scale more efficiently with larger inputs.

DATA1: Static vs Dynamic Typing (10 min)

We're such huge fans of "Classify That Language", so we've brought you a special episode in this homework.

Part A: (2 min.) Consider the following code snippet from a language that a former TA worked with in his summer internship:

```
user_id = get_most_followed_id(group) # returns a string
user_id = user_id.to_i
# IDs are zero-indexed, so we need to add 1
user_id += 1
puts "Congrats User No. #{user_id}, you're the most followed
user in group #{group}!"
```

Without knowing the language, is the language dynamically or statically typed? Why?

Solution:

This is Ruby. It is dynamically typed:

- **get_most_followed_id returns a string**
- **you could guess that .to_i is an integer conversion, so the type of user_id is changing in the program**
    - **but, even if you didn't know that...**
    - **we are performing integer addition to user_id, so at this point user_id has definitely changed types from string to integer**

Part B: Siddarth has created a compiler for a new language he's invented. He gives you the following code written in the language, with line numbers added for clarity:

```
00  func mystery(x) {
01    y = x;
02    print(f"{y}");
03    y = 3.5;
04    print(f"{y}");
05    return y;
06  }
07
08  q = mystery(10);
09  print(f"{q}");
```

Siddarth tells you that the program above outputs the following:

```
10
3
3
```

Part B.i: (2 min.) What can you say about the type system of the language? Is it statically typed or dynamically typed? Explain your reasoning.

Solution:

It is a statically typed language which is using type inference. Why? Because y's type remains an integer even when we assign it to a floating point value (3.5), which is why we're printing out 3 and not 3.5. Had the language been

dynamically typed, the variable y wouldn't have a fixed type, and once we assigned it to 3.5 it the variable y would refer to a value of 3.5, not 3.

Part B.ii: (2 min.) Is either casting or conversion being performed in this code? If so, what technique are being used on what line(s)? If you do identify any casts/conversions, explain for each if they are narrowing or widening.

Solution:

Yes, the code is using a conversion/coercion on line 3 where it converts 3.5 into an integer value of 3.

This is a narrowing conversion since it loses precision when it truncates the float to an integer.

Part B.iii: (2 min.) Assuming the language used the opposite type system as the one that you answered in part a, what would the output of the program be?

Solution:

10
3.5
3.5

Why? Because now the variables do not have types, so after the assignment on line 3, variable y now refers to a floating-point value. When mystery() returns, it also returns a floating point value, which is why we print out 3.5 on line 09 as well.

Part B.iv: (2 min.) Ruining likes the language so much that she built her own compiler for the language, but with some changes to the typing system. Siddarth wants to determine what typing system Ruining chose for her updated language, so he changed line 3 above to:

y = “3”;

and found that the program still runs and produces the same output. What can we say about the typing system for Ruining's language? Is it static or dynamic? Or is it impossible to tell? Why?

Solution:

There are two possible correct answers here:

- #1: Ruining's language must be dynamically typed. Why? Because the language allows the y variable to be assigned to two different types over time.
- #2: Ruining's language is statically typed and the string "3" undergoes an implicit conversion (aka coercion) into an int.

DATA2: Strong vs Weak Typing (11 min)

This problem explores the union data type:

Part A: (4 min.) Examine the following C++ code:

```
int main() {
  WeirdNumber w;
  w.n = 0;
  std::cout << w.n << std::endl;
  std::cout << w.f << std::endl;
  w.n = 123;
  std::cout << w.n << std::endl;
  std::cout << w.f << std::endl;
}
```

This prints out:

```
0
0
123
1.7236e-43
```

Why? What does this say about C++'s type system?

Solution:

This happens because C++ lets you access any type of the union at any time, even if it's not the one that's "active". `1.7236e-43` happens to be the floating-point representation encoded by the *bits* that 123 sets for an `int`.

In programming language terminology, this is called an untagged union.

Implicitly, this is an argument that C++ is weakly-typed.

- **`float`'s don't have a singular bit representation in C++: their precision depends on compiler flags, machine architecture, etc.!**
  - **We could even interpret this as an implicit cast**
- **Thus, this program does not have clear behavior across different environments; we could frame this as undefined behavior or as type-unsafe behavior.**
  - **In Carey's slides, this is probably in the "unsafe memory access" category, or an undefined type conversion**
- **There is also a more strict definition of "undefined behavior" in the C++ spec. Whether or not this is actually undefined behavior is not clear-cut (see: this SO post). Our interpretation of the C++11 spec on Unions (Section 9.5, page 220), is that this is undefined behavior.**

Part B: (7 min.) Zig is an up-and-coming language that in some ways, is a direct competitor to C and C++. Let's examine a similar union in Zig.

```
const WeirdNumber = union {
  n: i64,
  f: f64,
};

test "simple union" {
  var result = WeirdNumber { .n = 123 };
  result.f = 12.3;
```

```
}
```

```
test "simple union"...access of inactive union field
.\tests.zig:342:12: 0x7ff62c89244a in test "simple union"
(test.obj)
result.float = 12.3;
```

Assuming you haven't used Zig (but, crucially – you do know how to read error messages): what does this tell you about Zig's type system? How would you compare it to C++'s – and in particular, do you think one is better than the other?

Solution:

First, interpreting the error: it's indicative that Zig does not allow arbitrary access for any of the types that are part of the union. Instead, we can only use the "active" union field (in Zig's language). We can verify this in the Zig docs.

In programming language terminology, this is called a tagged union.

This leans towards a more strongly-typed type system. To compare and contrast this to C++:

- **Zig's approach is "safer": it requires us to explicitly tell the language when we want to access various union fields, and prevents undefined behavior. The idea is to prevent mistakes!**
- **C++'s approach is faster. There is some overhead when Zig checks union access; C++ does not have that overhead.**

Of course – there's a spectrum of valid opinions here, and we wouldn't ask you what the *correct* opinion is on an exam – just to compare and contrast.

DATA3: Casting and Conversions (15 min)

Part A: Consider the following C++ program:

```
01:  class Person {
02:  public:
03:    void say_hi() const { } };
04:  class Student : public Person { };

05:  double cast_away(long x) {
06:    int y = static_cast<int>(x);
07:    return y;
08:  }

09:  const Person *greet(const Student *p) {
10:    p->say_hi();
11:    return p;
12:  }

13:  int main() {
14:    Student arthur;
15:    const Person *p = greet(&arthur);
16:    short s = 10;
17:    double d = cast_away(s);
18:    cout << s + d;
19:    const Student *s = dynamic_cast<const Student *>(p);
20:  }
```

Identify all instances of casting and conversion in this code. When identifying a cast, explain whether it's an upcast or a downcast. When identifying a conversion, make sure to indicate whether it's a widening conversion or a narrowing conversion, and make sure to identify all promotions as well.

Solution:

Line 06: `static_cast<int>(x)` is an explicit narrowing conversion from `long` to `int`. While C++ calls this a cast via static_cast, it's actually creating a whole new value, so it's a conversion. Since this is narrowing, there is a chance that some information will be lost (e.g., if the long's value won't fit into an integer). This was arguably a poor choice of terminology from the C++ standards body.

Line 07: Returning an int value y where the function's return type is double is an implicit widening conversion, also known as a promotion.

Line 11: Returning a pointer to a Student object where the function's return type is Person * is an upcast, since we're casting from the subtype to the supertype.

Line 15: Passing a Student pointer to a function that accepts a const Student pointer results in an implicit const-cast. We're changing the const status of the object pointed to by the pointer, but no new value is being created so no conversion is taking place. So this is a cast because we're referring to the same object as before, just via a different type (a const type).

Line 17: Passing a short value to a function that accepts a long integer is an implicit widening conversion, also known as a promotion.

Line 18: Adding a short and a double requires the short value to first be implicitly converted (aka promoted) from a short value to a double value before the addition step occurs.

Line 19: Casting from a const Person pointer to a const Student pointer would be an explicit narrowing cast, aka a downcast.

Part B: Now consider this version of the program in Python:

```
01: class Person:
02:     def say_hi(self):
03:         pass
04:
05: class Student(Person):
06:     pass
07:
08: def cast_away(x):
09:     y = int(x)
10:     return float(y)
11:
12: def greet(p):
13:     p.say_hi()
14:     return p
15:
16: if __name__ == "__main__":
17:     arthur = Student()
18:     p = greet(arthur)
19:     s = 10
20:     d = cast_away(s)
21:     print(s + d)
```

```
22:     s = p
```

Identify all instances of casting and conversion in this code. When identifying a cast, explain whether it's an upcast or a downcast. When identifying a conversion, make sure to indicate whether it's a widening conversion or a narrowing conversion, and make sure to identify all promotions as well.

Solution:

Line 09: `y = int(x)` is an explicit conversion from `x` to `int`. In C++, this would correspond to `static_cast<int>(x)`, representing a narrowing conversion if `x` were originally a larger type. Given that we don't know the type of variable x when this code runs in the general case, we can't say whether it's a widening or narrowing conversion, or even legal.

Line 10: `return float(y)` The integer `y` is converted to a `float` before returning. This mirrors the widening conversion from `int` to `double` in C++. It is not a promotion because it is not implicit.

Line 21: `print(s + d)` In this operation, `s` (an integer) is implicitly converted to a `float` for addition with `d`, which is already a `float`. This is an implicit promotion.

One thing to notice is that since variables don't have types in dynamically-typed languages like Python, we don't have any casting. Moreover, conversions only happen when we do so explicitly (e.g., int(x)) or implicitly in expressions when one value must be promoted to the same type as another value for an operation to take place (e.g., s+d).