Homework 4

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

Each problem below has a time associated with it. This is the amount of time that we expect a thoroughly-prepared student would take to solve this problem on an exam. We understand, however, that as you learn these concepts, these questions may take more time to solve, and we don't want to overwhelm you with homework. For that reason, only **required** questions need to be completed when you submit this homework. That said, the exams will cover all materials covered (a) in class, (b) from class projects, (c) in the required homework problems, and (d) in the optional homework problems, so at a minimum, please review the optional problems and/or use them as exam prep materials.

You must turn in a PDF file with your answers via Gradescope - you may include both typed and hand-written solutions, so long as they are legible and make sense to our TAs. Make sure to clearly label each answer with the problem number you're solving so our TAs can more easily evaluate your work.

For homework #4, you must complete the following **required** problems:

- PYTH9: Lambdas (4 min)
- PYTH10: Closures (4 min)
- PYTH11: List Comprehensions (12 min)
- HASK17: Algebraic Data Types, Recursion (15 min)
- HASK19: Tail Recursion (10 min)
- DATA1: Static vs Dynamic Typing (10 min)
- DATA2: Strong vs Weak Typing (11 min)

- DATA3: Casting and Conversions (15 min)

For homework #4, you may optionally complete the following additional problems:

- HASK18: Algebraic Data Types, Recursion (20 min)

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(_____)
```

Your answer:

PYTH10: Closures (4 min)

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

Your Answer:

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)
```

Your answer:

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.

Your answer:

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"}`.

Your answer:

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}`.

Your answer:

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`.

Your answer:

`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`.

Your answer:

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

```
#import <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.**

Your answer:

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.

Your answer:

HASK18 (optional): 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`.

Your answer:

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**.

```haskell
sumSquares :: Integer -> Integer
sumSquares 0 = _________
sumSquares n = ___________________________________
```

Part B:

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

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?

Your answer:

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.

Your answer:

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.

Your answer:

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?

Your answer:

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?

Your answer:

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?

Your answer:

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?

Your answer:

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.

Your answer:

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.

Your answer: