CSC208 02.02 Lab3

limhavin

February 2023

1 Lab 3 : Assumptions in Proofs

1.1 Problem 1: Narrowing Down the Possibilities

```
if y >= 1$:
     return 0          # Point A
 elif x >= 1:
     return 1          # Point B
 elif x == -1:
     return 2          # Point C
 elif x <= -5 or y >= 3:
     return 3          # Point D
 elif x == 0:
     return 4          # Point E
 elif x == y:
     return 5          # Point F
```

1.1.1 Identify the types of each of the parameters

name (String)
x,y (real numbers)

1.1.2 Describe the set of assumptions we know are true about the parameters inside the branch expression at each labeled point, i.e., when that branch's guard is $True$.

For f1(name) we assume that the input is a String that reads 'Alice,' for point A. For point B, the string reads 'Bob'. In point C, the string is 'Carol', in point D, all other strings will reach this point. In Point E, no strings can reach this, as all strings are either 'Alice' (the point A) or not (point D). Thus, nothing will reach this point and other return points will be earlier in the function.

For f2(x, y) to reach Point A requires that y is greater than or equal to 1. Reaching point B requires that the y is less than 1, and x is greater than or equal to 1.
For Point C, it requires that x is equal to negative 1, and y is less than 1.

Point D requires that x is equal to -5 while y is less than 1, or y is greater than 3 while x is less than 1 and isn't -1.
The fifth requires that x is equal to y, and the previous conditions are all false

1.1.3 Are each of the conditionals exhaustive? If not, describe what values are not covered by the conditional

In f1, only "elif name != 'Emily'" cannot be reached, but since name != Alice exists, it is insured that at least something is returned. Assuming the input is a string, the string can either be equal to 'Alice' or not, and therefore passing both of those points is impossible.
In f2, is not exhaustive. For example, let x is negative 4, and y is negative 2. None of the branches will be fulfilled as "True" in this case.

2 Problem 2: Capture

```
def report(hw_avg, quiz1, quiz2, quiz3):
if hw_avg > 90 and quiz1 > 90 and quiz2 > 90 and quiz3 > 90:
    return 'excelling'
elif hw_avg > 75 and quiz1 <= quiz2 and quiz2 <= quiz3:
    return 'progressing'
elif hw_avg < 60 or quiz1 < 60 or quiz2 < 60 or quiz3 < 60:
    return 'needs help'
else:
    return 'just fine'
```

2.0.1 Describe the preconditions (both in terms of types and general constraints) on the parameters.

All inputs (hw_avg, quiz1, quiz2, quiz3) need to be positive numbers larger than 0.

2.0.2 Describe the conditions under which report(hw_avg, quiz1, quiz2, quiz3) will return 'just fine'.

All hw_avg, quiz1, quiz2, quiz3 can't be higher than 90 at the same time.
At least one of the two latter quizzes (quiz2 and quiz3) must be lower than the prior quiz.
All hw_avg, quiz1, quiz2, quiz3 needs to be higher than 60.
For example, if hw_avg is 90, quiz1 is 100, quiz2 is 70, and quiz3 is 61, the report(hw_avg, quiz1, quiz2, quiz3) will return 'just fine'.

2.0.3 Are there any conditions under which the function reports an inappropriate string, given the arguments?

No, there aren't any conditions which would report any inappropriate string. The program is exhaustive, assuming the input is a string, as we have a else statement.

3 Problem 3: Looking Ahead

```
    def head(l):
    return l[0]

def tail(l):
    return l[1:]

def cons(x, l):
    return [x] + l

def list_append(l1, l2):
    if l1 == []:
        return l2
    else:
        return cons(head(l1), list_append(tail(l1), l2))
```

3.0.1 While the function is recursive, our substitutive model of computation is capable of modeling the behavior of this function. Thus, we can prove properties about append just like any other function. Prove this one as an exercise:

Claim (Null Is An Identity on the Left): for any list l, list_append([], l) = l.

Proof :
According to the *append* operation, which takes two lists (l1, l2), if the first input list is empty it automataically returns the second list (l2).

```
def list_append(l1, l2):
    if l1 == []:
        return l2
    else:
        return cons(head(l1), list_append(tail(l1), l2))

--> list_append([], l)
--> def list_append([], l):
    [[if [] == []:
        return l]]
    else:
```

```
        return cons(head([]), list_append(tail([]), l))
--> [[if l1 ==[]:
        return l2]]
--> [[if True:
        return l]]
--> [[return l]]
--> l
```

3.0.2 We say that an operation is commutative if we can swap the order of the objects involved, i.e., if x = y, then y = x and vice versa. This is true for some operations, e.g., integers and addition, but not other, e.g., integers and subtraction. It turns out that list_append is not commutative for lists! Prove this fact by way of a counterexample:

Claim (Append is Not Commutative): there exists lists l1 and l2 such that list_append(l1, l2) $\neq$ list_append(l2, l1).

Proof : To prove that the claim is True, we will use case analysis by using two lists l1 = [1,2], and l2 = [3,4].

Case 1 : Appending l2 to l1.

```
def list_append([1,2], [3,4]):
    [[if [1,2] == []:
        return [3,4]]]
    else:
        return cons(head([1,2]), list_append(tail([1,2]), [3,4]))
-->*[[return cons(head([1,2]), list_append(tail([1,2]), [3,4]))]]
-->* [[return cons(1, list_append([2], [3,4]))]]
-->* [[return cons(1, list_append([2], [3,4]))]]
-->* [[return [1] + list_append([2] + [3,4])]]
-->* [[return  [1] + return cons([2], list_append(tail([]), [3, 4]) ]]
-->* [[return [1] + ([2] + list_append([], [3, 4]))]]
-->* [[return [1] + ([2] + [3, 4])]]
--> [[return [1] + [2, 3, 4]]]
--> [[return [1, 2, 3, 4]]]
--> [1, 2, 3, 4]
```

Case 2 : Appending l1 to l2.

```
def list_append([3, 4], [1, 2]):
    [[if [3,4] == []:
        return [1,2]]]
    else:
        return cons(head([3,4]), list_append(tail([3,4]), [1,2]))
-->*[[return cons(head([3,4]), list_append(tail([3,4]), [1,2]))]]
```

```
-->* [[return cons(3, list_append([4], [1,2]))]]
-->* [[return cons(3, list_append([4], [1,2]))]]
-->* [[return [3] + list_append([4] + [1,2])]]
-->* [[return  [3] + return cons([4], list_append(tail([]), [1, 2]) ]]
-->* [[return [3] + ([4] + list_append([], [1, 2]))]]
-->* [[return [3] + ([4] + [1, 2])]]
--> [[return [3] + [4, 1, 2]]]
--> [[return [3, 4, 1, 2]]]
--> [3, 4, 1, 2]
```

As we can see, even though we have the same lists, because we put them in in different orders, we get a different outcome in each case.

3.0.3 Because list_append is not commutative, this means that just because the empty list is an identity on the left-hand side of the function, it is not necessarily an identity on the right-hand side. With a little bit of thought, though, we can convince ourselves this ought to be the case. However, proving this fact is deceptively difficult! Attempt to prove this claim:

```
    Claim (Nil Is An Identity on the Right): for any list l, list_append(l, []) = l
--> Proof: let l1 be a universal variable L, and l2 be an empty list, [].
--> list_append(L, [])
-->* Since list_append is recursive, cons works until list 1 is empty. and when it is emp
[[return cons(L, list_append([], []))]]
-->* [[return L + []]]
--> [[return L]]
--> L
--> After moving deconstructing and reconstructing L, all that remains is L.
--> Working through an example, let us let L = [1, 2, 3, 4]
[[list_append([1, 2, 3, 4], [])]]
-->* [[return cons([1], list_append([2, 3, 4], []))]]
-->* [[return cons([1], cons([2], list_append([3, 4], [])))]]
-->* [[return cons([1], cons([2], cons([3], list_append([4], []))))]]
-->* [[return cons([1], cons([2], cons([3], cons([4], list_append([], [])))))]]
-->* [[return cons([1], cons([2], cons([3], cons([4], []))))]]
-->* [[return cons([1], cons([2], cons([3], cons([4], []))))]]
-->* [[return cons([1], cons([2], cons([3], [4])))]]
-->* [[return cons([1], cons([2],[3, 4]))]]
-->* [[return cons([1],[2, 3, 4])]]
-->* [1, 2, 3, 4]
--> As shown, l1 is equal to the output, as the second value will always be [] (null), wh
Note how for every tail of the list, the list was also reconstructed (for each iteration
```

.

3.0.4 Finally, reflect on your experience. Answering the following questions in a few sentences each:

- Why were you able to directly prove the left-identity claim? A: Yes. As the list_append returns the universal variable l when l1 is null, it is more straight forward to use a universal variable to return l. Even if l is not a list, the left-identity claim still holds true.
- Why is the right-identity claim more difficult to prove? A: Because a universal variable cannot be split up then concatenated, it's less straight forward to prove such a claim, especially since it is claiming that "for *any list* l, list_append(l, []) $\equiv$ l".
- What information do you need in order to break the chain of infinite reason that you found in the previous proof? //We weren't able to confirm with our instructor that we got to the right point with the infinite loop of reasoning. However, if we were to guess, we would need to have a universal variable that could successfully go through the function that can account for *any list* l.