CSC 173 Project 3
Maryfrances Umeora and Carolina Lion He
NETIDs: mumeora & clionhe

Program Files Used

Proj3.lisp

How To Run Our Project

Navigate to where you stored my project lisp file, type `abcl` (or `java -jar abcl.jar`, whichever one works on your device) and then after the compiler loads, type `(load "Proj3").`

How The Code Runs (PLEASE READ)

So basically, we have three repl functions that are called depending on how many parameters my function has. Each repl function takes in two parameters: a message to print and the name of the function to run.

First we print the message, then have the user enter input to run the specified function.

After that, we ask if the user wants to test that same function again (the loop part) and if yes, we recursively call repl again with the same input, and if not, the code goes to the next function.

All of this will happen automatically (ie you will be prompted accordingly) once our .lisp file is loaded.

LIST FUNCTIONS

1. Append List

In this, we made good use of the list functions:

- car: which takes a list as an argument, and returns its first element,
- cdr: which takes a list as an argument, and returns a list without the first element and
- cons: which takes two arguments, an element and a list and returns a list with the element inserted in the first place.

If the first list is empty, we just return the second list. Else, we return the first element of the first list and the rest of that first list which we call recursively, along with the second list.

2. Reverse List

Basically for this one, we append the rest of the list, then the first element of the list. Of course, the "rest of the list" is called recursively.

3. Add Element

Here we are to add an element to the end of a list. Basically, we reverse the list, use cons to add our element to the front of our now-reversed list, then reverse it back.

4. Index Of

Index starts at 0. Return -1 if element is not in list.
So first, using out handy inlist function, if the element is not in the given list, we return -1. However, if it is, we check if the first element of the list is the element we're looking for. If it is, we return true. Else, we add 1 to the index and perform the check on the rest of the list.

SET FUNCTIONS

1. Set Membership

Return t if an element is in the set and nil if not.
Basically, we check if the set is empty. If it is, return false, else, recursively call it on the elements of the set to see if we'll ever get true.

2. Insert

If an element is already in the set, we just return the set. Else, we use the cons command, which takes two arguments, an element and a list and returns a list with the element inserted in the first place. Pretty simple.

3. Set Intersection

Return similar elements between two sets.
For this one, we take in two sets L1 and L2, then "create" another empty set named L3. If the first element in L1 is a member of L2, we insert that element into L3, and recall the function on the rest of L1. We keep doing this until L1 is null or empty, at which point we return L3.

4. Set Cardinality

Returns the number of unique elements in the set. Basically, the base case is 0, and for every other element in the set, we add one.
This code assumes that the input will always be a valid set ie no duplicates.
The teacher said it's reasonable to assume for this assignment that the inputs you give our functions are correct. If the function expects a set, then please enter a set without duplicates. Basically, if you enter (a a b c) we will return 4, but we expect that you enter (a b c) instead so we can return 3.

MATH FUNCTIONS

1. Absolute Value

Here is the pseudocode for how my absolute value function works.

```
int abs(n)
  if n >= 0
    return n;
  if n < 0
    return n/-1;
```

2. Factorial

So the java code for factorial would look something like this:

```
public static int factorial(int x)    {
    if (x == 0)  {
        return 1;
    }
    else
        return x*factorial(x-1);
}
```

So the lisp code must be

3. Fibonacci

The pseudocode for Fib would be

```
int nth-fibo(int n)  {
    if (n <= 1)
       return n;
    return fib(n-1) + fib(n-2);
}
```

4. Triangle Sides

Check if 3 integers can be the lengths of the two sides and the hypotenuse of a right triangle (in that order). The Pythagorean Theorem states that *c*, the length of the hypotenuse, is equal to the square root of the sum of the squares of the other two sides.

Thus, our triangle function looks like this.

```
right-tri(a, b, c)    {
  z = (a*a) + (b*b)
  if (c^2 == z)
     return true;
  else
     return false;
}
```

REQUIRED FUNCTIONS

1. Perfect Numbers

A number is perfect if the sum of its factors other than itself is equal to itself.
The best way to explain how we did this is to look at the equivalent Java code I made:

```
public static boolean perfect(int n) {
    if (sumOfFactors(n, 1, 0) == n)
        return true;
    else
        return false;
}

public static int sumOfFactors(int n, int i, int sum)   {
    if (i < n)    {
        if (n%i == 0)    {
            sum = sum + i;
            return sumOfFactors(n, i+1, sum);
        }
        else
            return sumOfFactors(n, i+1, sum);
    }
    return sum;
}
```

Helper functions: mod
Test: Perfect numbers include 6, 28 and 496.

2. Abundant Numbers

An abundant number's sum of factors other than itself is greater than the number.
Our function works the same as perfect, except we return true only when the recursive function gives a result greater than the initial number.

3. Deficient Numbers

A deficient number's sum of factors is less than itself. This works the same as perfect and abundant, only with "less than."

BONUS FUNCTIONS FOR EXTRA CREDIT

1. [MATH] Greatest Common Divisor

This one was pretty simple. If the first number is 0, then the other number is the gcd. Else, recursively call the function with the mod of the second number.

2. [MATH] Least Common Factor

The formula for lcm is (a*b)/gcd(a, b)

3. [MATH] Prime Number?

   I had created a sumOfFactors function to check the sum of all the factors of a number except itself. Since a prime number's only other factor should be 1, I check if the sum of its factors is 1.

4. [SET] Set Difference

   Given two sets, we return a list of the elements that are in the first set, but not the second one. For this function, we did this thing where our title function calls a different function which is recursive so we can return a third list, which is initially empty. The base case is that if the first list is empty, we return null. Else, for every element in the first list, we check if it is present in the second list.

   I'm assuming order doesn't matter.

5. [SET] Symmetric Difference

   GFG defines symmetric difference as " all elements of two arrays except common elements." It can easily be done by appending regular set difference with both the sets in different orders (ie everything in the first set that isn't in the second one and everything in the second set that isn't in the first set).

6. [SET] Subset or Equal

   If a list is a subset of another list, that means that everything in the first list must also be in the second list. That is, the set difference between the two of them must be null.

7. [SET] Superset or Equal

   For a set to be a superset of another, that means the second set has to be a subset of the first :)

8. [SET] UNION

   For this one, I simply append the two sets, then get rid of the duplicates.

9. [MATH] REMOVE DUPLICATES

   If the list is empty, then we just return it. Else, if and only if the first element in the list can also be found in the rest of the list (cdr), we call the function recursively on the rest of the list only, effectively getting rid of that first element.