Homework Assignment 1
Programming Languages
CS471, Spring 2017
Due: Tues., Jan 23 midnight

Purpose: To examine the basic notions of programming languages via a review of C and gentle use of scripting languages, Ruby and Python.
Links for References and Software
Read Scott: Chapter 1 and review CYU 1-29

Instructions: requirements for submission of homework

Assignment

1. Consider the following **C** program program with multiple errors as indicated:

```
#include <stdio.h>
int a1, char b2; // error
INT arr[10];  // error
long fr; nx; //error
long i#rate = 99; //error
int * p;

int main (void){
      int struct = 8 ;  //error
      int x;
      arr[1] = x;
      *p = 100; // error
      struct = chg(a); //error
      return struct; //error
}
int chg(long  fr)  {
     if (fr = 0) {  //error
          x = 10; //error
          arr[10] = 10; //error
        } else {
             arr[1] = 1;
        }
    return arr[1];
}
```

Label each error as either: lexical, syntactical, static semantic, dynamic semantic, uncheckable error or logical error. To compile type: gcc -Wall -std=c99 YOU NEED TO JUSTIFY YOUR ANSWER IN 1 OR 2 SENTENCES.

2. Download and compile gcd.c to assembly code on your machine: Use the command `gcc -S gcd.c`

   Although C is considered a high-level language it really is only a few steps up from assembly. You should be able to recognize a simple correspondence between the C code and the compiled assembly code. Annotate the generated assembly code (you may copy and paste it in this file and annotate by line): Look for the key points, there is no need to label each instruction.

   In particular, you should recognize and label the following:
   1. The creation of the stack frame.
   2. Argument storage and updates.
   3. Local variable storage and updates.
   4. Translation of the loop.
   5. Some form of return result, and an exit from the function.

3. Do exercise 1.4 pg 38 Below is a modification of the code given in the textbook.

```
int gcdM(int i, int j) {
       while ( i != j) {
           if (i > j) {
              i = i % j;
           } else {
              j = j % i;
           }
        }
        return i;
    }
```

   Does this program always compute the same result as problem 2 above? If gcdM() does not give the same results as gcdI(), can you fix it? Under what circumstances would you expect one or the other to be faster?

4. Download and compile gcd_full.c: Use the command `gcc -Wall gcd_full.c -o gcd` (it's always a good idea to compile with -Wall). You may run it with

any integer input `./gcd 56 14`

The function **gcdI** is obviously an iterative implementation of a gcd function.
1. Add a new function, **gcdF**, that gcd function using a *functional style*: You should recursively call **gcdF** and use no loops. Adjust the main function to call **gcdF** instead.
2. Briefly discuss the computation complexity of both implementations of power (big-oh notation); one sentence will do.
3. In a language where you have the flexibility to implement an algorithm iteratively and recursively why would you choose one over the other?

5. Statically typed languages have their advantages, but sometimes it really is simpler to use a dynamic language.
The following is a simple implementation of factorial in Python: fact.py

```
#! /usr/bin/env python
import sys

def fact(n):
  if n == 0:
    return 1
  else:
    return n * fact(n-1)

if len(sys.argv) != 2:
  print("%s usage: [NUMBER]" % sys.argv[0])
  exit()

print(fact(int(sys.argv[1])))
```

The following is a simple implementation of factorial in Ruby: fact.rb

```
def fact(n)
  return 1 if n == 0
  return n * fact(n-1)
end

if ARGV.length != 1
  puts "fact.rb usage: [NUMBER]"
  exit
end
```

```
    puts fact(ARGV[0].to_i)
```

Rewrite **gcdI** and **gcdF** from pow_full.c in Python and Ruby (they will syntactically and semantically be very similar).

In particular:
(1) Create a file named gcd_full.py that contains an implementation of gcdI and gcdF in Python.
(2) Create a file named gcd_full.rb that contains an implementation of gcdI and gcdF in Ruby.

Make sure to place the code in this file as well.

6. Page 38 exercise 1.5. In your local implementation of C, what is the limit on the size of integers? (You can use /usr/include/limits.h to figure out the range of values of type int.) What happens in the event of arithmetic overflow? What are the implications of size limits on the portability of programs from one machine/compiler to another? How do the answers to these questions differ for Java? for Python? for Ada?

7. Page **38** problem 1.1 a-e.

8. True or False. Some programming languages are more "powerful" then others. Explain your answer.

9. [Rosen,*Discrete Mathematics and Its Applications* 5th ed., pg 273 #48] "Consider the following inductive definition of a version of **Ackermann's** function. This function was named after Wilhelm Ackermann, a German mathematician who was a student of the great mathematician David Hilbert. Ackermann's function plays an important role in the theory of recursive functions and in the study of complexity of certain algorithms involving set unions." In the early days of imperative languages, Ackermann's function was used to measure the recursion capability of a compiler.

   The Ackermann Number N of the compiler as the largest N for which

   ack (3,N)

   gives an answer without a stack overflow. In earlier decades a variation had been used as one of the benchmarking algorithms. (There are several different variants of this function. All are called Ackermann's functions and their values do NOT have to agree!)

ack(*m,n*) =	***n* + 1**	**if *m* = 0**
ack(*m,n*) =	**ack(*m* - 1, 1)**	if *n* = 0 and *m*
ack(*m,n*) =	**ack(*m*-1, ack(*m, n*-1))**	**if n >0 and m**

a. **What is the value of ack(1,0) ? ______**
b. **What is the value of ack(0,3) ? ______**
c. Create a file named ack.c and implement **ack** in C. Try and find the largest Ackermann number for your program.
d. Create a file named ack.py and implement **ack** in Python. Try and find the largest Ackermann number for your program.
e. Create a file named ack.rb and implement **ack** in Ruby. Try and find the largest Ackermann number for your program