CMSI 386 Homework #5

Zane Kansil & Edward Bramanti

December 3, 2013

3. Write a tail-recursive function to compute the minimum value of an array in Python, C, JavaScript, and Go.

   Python:

```
a = [1,2,3]
b = [21,7,12,2]
c = [1, 12, -1, 0]
d = [1, 20, 500, 6, -500, 40, -500, 67]


def findMinValue(a, i = 0, sofar = None):
    if i == len(a) - 1:
        return a[i] if a[i] < sofar else sofar
    elif sofar == None:
        sofar = a[i]
    elif a[i] < sofar:
        sofar = a[i]

    return findMinValue(a, i + 1, sofar)

print findMinValue(a)
print findMinValue(b)
print findMinValue(c)
print findMinValue(d)
```

C:

```
#include <stdio.h>

int minValueHelper (int * anArray, int length, int i, int sofar) {
    if (i == length - 1) {
        return  (anArray[i] < sofar) ? anArray[i] : sofar;
    }
    else if (anArray[i] < sofar) {
        sofar = anArray[i];
    }

    return minValueHelper(anArray, length, i + 1, sofar);
}

int findMinValue(int * anArray, int length) {
    return minValueHelper(anArray, length, 0, anArray[0]);
}

int main() {
    int a[3] = {1, 2, 3};
    int b[4] = {21,7,12,2};
    int c[4] = {1, 12, -1, 0};
    int d[8] = {1, 20, 500, 6, -500, 40, -500, 67};
    printf("%d\n", findMinValue(a, 3));
    printf("%d\n", findMinValue(b, 4));
    printf("%d\n", findMinValue(c, 4));
    printf("%d\n", findMinValue(d, 8));
}
```

JavaScript:

```
var a = [1,2,3];
var b = [21,7,12,2];
var c = [1, 12, -1, 0];
var d = [1, 20, 500, 6, -500, 40, -500, 67];

function findMinValue(a, i, sofar) {
    if (i === undefined) i = 0;
    if (i === a.length - 1) {
        return (a[i] < sofar) ? a[i] : sofar;
    }
    else if (sofar === undefined) {
        sofar = a[i]
    }
    else if (a[i] < sofar) {
        sofar = a[i]
    }

    return findMinValue(a, i + 1, sofar)
}

alert(findMinValue(a));
alert(findMinValue(b));
alert(findMinValue(c));
alert(findMinValue(d));
```

Go:

```
//http://play.golang.org/p/0gyqEAuqhn

package main

import "fmt"

func main() {
    a := []int{1,2,3}
    b := []int{21,7,12,2}
    c := []int{1, 12, -1, 0}
    d := []int{1, 20, 500, 6, -500, 40, -500, 67}

    findMinValue(a, 3)
    findMinValue(b, 4)
    findMinValue(c, 4)
    findMinValue(d, 8)
}


func findMinValue(a []int, length int) {
    minValueHelper(a, length, 0, a[0])
}

func minValueHelper(a []int, length int, i int, sofar int) int {
    if (i == length - 1) {
        if a[i] < sofar {
            fmt.Println(a[i])
            return a[i]
        } else {
            fmt.Println(sofar)
            return sofar
        }
    } else if (a[i] < sofar) {
        sofar = a[i]
    }

    return minValueHelper(a, length, i + 1, sofar)
}
```

4. Here's some code in some language that looks exactly like C++. It's sort of like Go, also, except the pointer types are kind of backwards. It is defining two mutually recursive types, A and B.

```
struct A {B* x; int y;};
struct B {A* x; int y;};
```

   Suppose the rules for this language stated that this language used structural equivalence for types. How would you feel if you were a compiler and had to typecheck an expression in which an A was used as a B? What problem might you run into?

5. Write a program in C++, JavaScript, Python, Ruby, Scala, or Clojure that determines the order in which subroutine arguments are evaluated.

   We decided to write our subroutine in Javascript.

```
//  http://jsfiddle.net/72SGF/1/
var x = 10;

function determine(a, b) {
    alert(x);
}

function half() {
    x = x / 2;
}

function addFour() {
    x = x + 4;
}

determine(addFour(), half());
```

6. Consider the following (erroneous) program in C:

```
void foo() {
    int i;
    printf("%d ", i++);
}
int main() {
    int j;
    for (j = 1; j <= 10; j++) foo();
}
```

Local variable i in subroutine foo is never initialized. On many systems, however, the program will display repeatable behavior, printing 0 1 2 3 4 5 6 7 8 9. Suggest an explanation. Also explain why the behavior on other systems might be different, or nondeterministic.

First, `main()` is placed on the stack. Then, `foo()` is called, placing it on top of `main()` in the stack. Somewhere in the stack space of `foo()`, `int i` is declared, but not initialized. If you initialize a variable without a value, then it seems to not change anything about the specific stack space that it occupies. Then, i is incremented. Each time `foo()` is called, the new stack frame ends up in the exact same storage space as the previous call. Due to the fact that initializing a variable (`int i`) does not change the stack space that it occupies, the incrementations to `int i` of the previous `foo()` call are preserved.

On other systems where it will not naturally print `0 1 2 3 4 5 6 7 8 9`, it may be because `int i` is not being reliably initialized to the same stack space within `foo()`.

8. In some implementations of an old language called Fortran IV, the following code would print a 3.

```
call foo(2)
print* 2
stop
end
subroutine foo(x)
    x = x + 1
    return
end
```

Can you suggest an explanation? (Hint: Fortran passes by reference.) More recent versions of the Fortran langauge don't have this problem. How can it be that two versions of the same language can give different results even though parameters are officially passed "the same way". Note that knowledge of Fortran is not required for this problem.

The earlier version of Fortran passes all values by reference. In the sample program the value 2 is passed by reference into subroutine foo which increments it's parameter by 1. In this case, 2 is not a variable but hard-coded integer value. What seems to be happening is that when 2 is incremented as a result of pass by reference, ALL hard-coded 2's are incremented.

This has the unfortunate effect of causing the `print* 2` statement to print `3` instead. The more recent version of Fortran probably solves this by passing by unique reference, protecting other hard-coded variables from being changed as a side effect of a subroutine.

10. Explain what is printed under (a) call by value, (b) call by value-result, (c) call by reference, (d) call by name.

```
x = 1;
y = [2, 3, 4];
sub f(a, b) {b++; a = x + 1;}
f(y[x], x);
print x, y;
```

11. I've written a simple JavaScript queue type that does not use encapsulation. Can we achieve encapsulation using the module system in node.js? If so, implement it. If not, state why not.

12. EXTRA CREDIT: It is certainly possible to make a Person class, then subclasses of Person for different jobs, like Manager, Employee, Student, Monitor, Advisor, Teacher, Officer and so on. But this is a bad idea, even though the IS-A test passes. Why is this a bad idea and how should this society of classes be built?

    It is better to favor object composition to construct the related subclasses. One reason for this is because these sorts of subclasses tend to have a lot of overlapping properties, whilst potentially having few values that they all have in common with a shared superclass.

    When inheritance is used to implement a society, multi-inheritance becomes a necessity to compose classes with similar properties. This requirement is exaserbated if the classes are to inherit from a relatively basic superclass. Also, not all languages support multi-inheritance. Without object composition, complex inheritance heirarchies can also arise where unneccessary.

    Consider the set of classes Person, Employee, Manager and Consultant. We can reason that a heirarchy of Person $>$ Employee $>$ Manager is reasonable. However a Consultant cannot cleanly inherit from Employee or Manager. They do not directly work for the firm so they do not have their own parking spot. Ignoring parking spot, it is also improper to have Consultant inherit bonus from Manager, because a Consultant is not a Manager and therefore a suggestive inheritance does not make sense.

```
class Person:
    name
    age

class Employee < Person:
    salary
    parking_space

class Manager < Employee:
    bonus

class Consultant < Person:
    salary
    bonus
```

Person, Worker, Manager and Consultant should instead be composed of the properties name, age, salary, parking_space and bonus. No direct relations need to be implied, the classes are just clearly constructed with what they need.

```
propset Human:
    name
    age

propset Worker:
    salary

propset ParkingSpot:
    parking_space

propset Perks:
    bonus

class Person:
    endow Human

class Worker:
    endow Human, Worker, ParkingSpot

class Manager:
    endow Human, Worker, ParkingSpot, Perks

class Consultant:
    endow Human, Worker, Perks
```

13. Write in Java, Python, JavaScript, and C++, a module with a function called nextOdd (or next_odd or next-odd depending on the naming conventions of the language's culture). The first time you call this subroutine you get the value 1. The next time, you get a 3, then 5, then 7, and so on. Show a snippet of code that uses this subroutine from outside the module. Is it possible to make this module hack-proof? In other words, once you compile this module, can you be sure that malicious code can't do something to disrupt the sequence of values resulting from successive calls to this function?