CMPSC 461: Programming Language Concepts
Assignment 5 Solution

Problem 1 [15pt] Consider the following implementation of function power.

```
int power (int x, int n) {
    if (n <= 0) {
        return 1;
    }
    else {
        return x*power(x, n-1);
    }
}
```

a) (3pt) Briefly explain why this implementation is not tail recursive.

Solution: This implementation is not tail recursive since the implementation multiplies the x with the result of the recursive call. In other words, recursive call is not the last thing in the implementation.

b) (6pt) Give a tail recursive implementation in the C language. You might need to define a helper function.

Solution:

```
int power(int x, int n) {
    power_help(x, n, 1);
}

int power_help (int x, int n, int prev) {
    if (n <= 0)
        return prev;
    else
        return power_help(x, n-1, x*prev);
}
```

c) (6pt) Give an equivalent loop-based implementation of the same function in the C language.

Solution:

```
int power(int x, int n) {
    power_help(x, n, 1);
}

int power_help (int x, int n, int prev) {
    while (true) {
        if (n <= 0)
            return prev;
        else {
            n = n-1;
            prev = x*prev;
        }
    }
}
```

Problem 2 [5pt] Consider the following (erroneous) program in the C language:

```
void foo( ) {
    int i;
    printf("%d ", i--);
}

void bar( ) {
    int i=5;
}

int main( ) {
    bar();
    int j;
    for (j=1; j<=5; j++)
        foo();
    return 0;
}
```

In this piece of code, local variable `i` in `foo` is never initialized. On many systems, however, the program will display repeatable behavior: printing 5 4 3 2 1. Suggest an explanation.

Solution: The activation records for different instances of foo will occupy the same space in the stack. Hence, although the local variable i is never initialized, it might "inherit" a value from the previous instance of the routine. In this example, the initial value comes from function bar, which previously occupies the same space on the stack. Then i will decrease by one in each iteration after that.

Problem 3 [18pt] Consider the following pseudo-code. For each of the following parameter passing mechanisms, write down all outputs produced by the entire program (note that `print p, q` outputs both p and q). Assume parameters are evaluated left-to-right and their values are copied back left-to-right, where appropriate.

```
int p = 3;
int q = 5;
int f(int b, int c) {
    b = 2 * c;
    c = 1 + p;
    return b + c;
}
print f(p,q)
print p, q
```

1. (4.5pt) Call by Value
2. (4.5pt) Call by Reference
3. (4.5pt) Call by Value-Return
4. (4.5pt) Call by Name

Solution:

1. 14, 3, 5
2. 21, 10, 11

3. 14, 10, 4

4. 21, 10, 11

Problem 4 [12pt] Consider the following code snippet with exceptions. Note that the main function calls `foo` in the nested try block.

```
void foo () {
  try {
    throw new Exception1();
    print ("A");
    throw new Exception2();
    print ("B");
  }
  catch(Exception1 e1) {
    print "handler1";
  }
  print ("C");
  throw new Exception1();
}

void main () {
  try {
    try {
      foo();
      print ("D");
    }
    catch(Exception1 e1) { print "handler2"; }
    print ("E");
  }
  catch(Exception2 e2) { print "handler3"; }
}
```

a) (4pt) Write down what will be printed out by the program and briefly justify your answer.

Solution:It will print out:
handler 1
C
handler 2
E

b) (8pt) Instead of the "replacement" semantics of exception handling used in modern languages (i.e., the semantics introduced in lecture), a very early design of exception handling introduced in the PL/I language uses a "binding" semantics. In particular, the design dynamically tracks a sequence of "catch" blocks that are currently active; a catch block is active whenever the corresponding try block is active. Moreover, whenever an exception is thrown, the sequence of active "catch" blocks will be traversed (in a first-in-last-out manner) to find a matching handler. Furthermore, execution will resume at the statement following the one that throws exception, rather than the next statement after the matching "catch" block as we have seen in the "replacement" semantics.

Write down what will be printed out by the program if the language uses the "binding" semantics, and briefly justify your answer.

Solution:It will print out:
handler1
A
handler3
B
C
handler2
D
E