HW4

Due date: 12/26

Turn in your code for the starred (sub)problems.

1* [Variable arguments in C/C++] (15%)

Write the following C/C++ function to find the maximum of a variable number of integral and/or floating-point arguments.

```
double max(const char* typestr,double x,...);
```

Comments

1 The parameter typestr is a string of d's and f's.
For each d in the typestr, there should be an argument of type bool, char, short, or int. For each f in the typestr, there should be an argument of type float or double.

2 There must have at least one integral or floating-point argument passed to the parameter x.

3 The value of the function is of double type.

For examples,

max("") ⇒ error; need at least one number

max("ddffdd",29,'a',(short)255,34.56f,78.9,254,true) ⇒ 255.0

Observe that the 2nd argument 29 is bound to the 2nd parameter x and thus has no corresponding d or f in the typestr.

Comment

Recall that for variable-argument function calls, the compiler performs *default argument promotions* on the arguments, including

a) floating-point promotion, i.e. float → double

b) integral promotion, i.e. bool/char/short → int

In other words, within the function max, apply var_arg() only to int or double. It makes no sense to apply it to other types, such as bool, char, short, or float.

2 [Variable arguments in Scheme]

Scheme also supports variable arguments.

For example, + has any number of arguments; max has at least one argument.

(+) ⇒ 0

(+ 2 3/4 5.6) ⇒ 8.35

(max) ⇒ error

(max 3) ⇒ 3

(max 2 3/4 5.6) ⇒ 5.6

λ-expression for any number of arguments

Syntax: (lambda args body)

Semantics: The parameter args is bound to a list of arguments.

Example

(define f (lambda args args))

(f) ⇒ ()

(f 1 2 3 4 5) ⇒ (1 2 3 4 5)

λ-expression for at least one argument

Syntax: (lambda (x . args) body)

Semantics: The parameters x and args are bound to the first argument and a list of remaining arguments, respectively.

Example

(define f (lambda (x . args) x))

(define g (lambda (x . args) args))

(f) ⇒ *error*

(f 1 2 3 4 5) ⇒ 1

(g 1 2 3 4 5) ⇒ (2 3 4 5)

a)* Define a ***recursive*** function my+ that behaves the same way as the built-in +.

Hint: Use the apply function (10%)

(apply f '(x1 x2 … xn)) = (f x1 x2 … xn)

For example,

(apply + '(1 2 3 4 5)) ⇒ 15

Note: Do not define my+ as

(define my+ (lambda args (apply + args)))

As it uses the built-in multiple-argument additive operator + and isn't recursive.

This problem asks you to simulate the behavior of the built-in multiple-argument additive operator +. Put differently, you shall define my+ as a recursive function and use only binary addition (i.e. assume that the built-in + is binary).

b)* Define a ***recursive*** function mymax that behaves the same way as the built-in max. (10%)

c) Compare the variable-argument mechanisms of C/C++ and Scheme for type safety. (5%)

3 [Operand evaluation order]

Consider the following C++ program

```
#include <iostream>
using namespace std;
int k;
int f()    // compute k!, where k is global
{
        int r=1;
        for (int i=2;i<=k;i++) r*=i;
        return r;
}
int c(int m,int n)    // compute m!/(n!*(m-n)!)
{
        return (k=m,f()) / ( (k=n,f()) * (k=m-n,f()) );    //*
}
int main() { cout << c(5,2) << endl; }
```

a) How many ways are there to evaluate the expression in the starred line? (5%)

b) What is the output of this program under VC++? under GNU C++? under clang++? Explain. (10%)

Hint:
```
bsd2 > clang++ file.cpp
bsd2> g++47 file.cpp
```

4 [Tail-recursive optimization]

Given the following ML functions

```
val sumr = foldr op+ 0;
val suml = foldl op+ 0;
```

Recall from HW4 that sumr = suml. However, which is better and why? (5%)

5 [Tail-recursive optimization]

Consider

```
void qsort(int l,int h)
{
    if (l<h) {
        int m=partition(l,h);
        qsort(l,m-1);
        qsort(m+1,h);
    }
}
```

a) What is the worst-case *space* complexity of this qsort function? Brief explanations suffice. (5%)

b) Rewrite it by the technique of tail-recursive optimization to *minimize* the use of stack space. (10%)

Hint: The order of the two recursive calls can be reversed, i.e. each can be made tail-recursive. The question is: to save space, which one should be made tail-recursive?

c) What is the worst-case space complexity of the optimized qsort function of part b)? Brief explanations suffice. (5%)

d) What can you say about the worst-case time complexity before and after optimization? Brief explanations suffice. (5%)

6 [Last-call optimization]

Consider the following C++ program

```
bool even(int n) { if (n==0) return true; else return odd(n-1); }
bool odd(int n) { if (n==0) return false; else return even(n-1); }
int main() { cout << even(3); }
```

a) Draw the contents of the runtime stack at the point when the recursion reaches its end, i.e. when the boundary condition n==0 becomes true.

Be sure to indicate the values of parameters, instruction pointers, and dynamic pointers. Note: There are no static pointers in C/C++. (5%)

b) Repeat a), but this time assumes that the program is compiled by a C++ compiler that does last-call optimization. (5%)

Hint: The functions even and odd share the same AR.