CMSC 331 Final Exam Fall 2012

Name: ________________________________

UMBC username:____________________________

You have two hours to complete this closed book exam. Use the backs of these pages if you need more room for your answers. Describe any assumptions you make in solving a problem. We reserve the right to assign partial credit, and to deduct points for answers that are needlessly wordy. Skim through the entire exam before beginning to get a sense of where best to spend your time. If you get stuck on one question, go on to another and return to the difficult question later. Comments are not required for programming questions but adding some might help us understand your code.

1. True/False (30 points)

For each of the following questions, circle T (true) or F (false). (2 points each)

T F 1. Scheme uses static scope whereas ordinary Lisp uses dynamic scope.

T F 2. If a Perl function terminates without explicitly returning a value, then it's a recursive function.

T F 3. Haskell's method of type checking means that type-mismatch errors are generally not detected until runtime and then only if the program is thoroughly tested.

T F 4. The "assert" statement in C and C++ is useful for an informal sort of program verification.

T F 5. Most modern programming languages use call by value and call by value-result.

T F 6. Modern functional languages like Haskell can be compiled as well as interpreted.

T F 7. In Haskell and Scheme, the read-eval-print loop causes user programs to read their input data before doing any other I/O.

T F 8. If a function in Haskell has only one formal parameter, then it's a recursive function.

T F 9. When Haskell (ghc or hugs) complains about type mismatch, the problem could be as simple as misspelling the name of a function.

T F 10. In an event-driven program, the main program handles each event itself, for the sake of performance.

T F 11. If a finite-state machine has even only one state in which two outbound arcs have the same symbol attached, then the machine is non-deterministic.

T F 12. Perl and Haskell have at least one feature in common: there is usually more than one way to do something.

T F 13. Regular expressions can be converted into finite state machines, and vice-versa.

T F 14. One of the advantages of functional programming languages like Haskell is that once the program compiles, it is more likely to run correctly, i.e. without logic errors.

T F 15. The C++ language allows for explicit definition of constructor and destructor functions.

2. General multiple-choice questions (15 points)

Indicate the **one, best** answer for each problem (3 points each).

_____ 1 A common way to define a programming language's syntax is to use a (a) binary tree; (b) box and pointer diagram; (c) context free grammar; (d) LRRL grammar

_____ 2 Attribute grammars are typically used to (a) Handle left-recursion. (b) Describe language features which context-free grammars alone cannot. (c) Prove program correctness. (d) Compile grammars into efficient tables.

_____ 3 The main difference between a sentence and a sentential form is (a) there is no difference; (b) a sentence contains only terminal symbols but a sentential form can contain some non-terminal symbols; (c) sentential forms are a subset of sentences but the converse is not true; (d) sentential forms have no attributes but a sentence does.

_____ 4 In Haskell, the high order functions include (a) car and cdr; (b) map and filter; (c) head and tail; (d) log and sqrt.

_____ 5 In C++, what is the relation between classes and objects: (a) Objects must be declared before their classes can be used; (b) Objects describe the behavior of their classes; (c) All objects belong to exactly one class; (d) A class is an instantiated object. (e) Classes describe types and objects are the values of those types.

3. Writing C++ functions (25 points)

Consider the rational class described on class handouts, and summarized in the attached C++ summary (aka the C++ cheat sheet). Suppose we wanted to add a function to multiply two objects belonging to the rational class, using the familiar * operator.

(5 points) How would such a function be declared in C++?

(10 points) How would such a function be defined (or implemented) in C++ code? Show the body of the function, between the curly braces:

{

}

(10 points) Suppose we wanted to add a function called productSeries, where productSeries(k) is (1/1) * (1/2) * (1/3) * …* (1/k) for any positive integer k. For example, productSeries(1) =1, and productSeries(4) = 1*(1/2)*(1/3)*(1/4) = 1/24. The function productSeries would be implemented as follows. Fill in the blanks. (2 points each.)

```
rational productSeries( int k) {

  __________ prod(1,1), t;

  int denom;

  assert (k >= _______);

  for (denom = 1; denom  ______ k; denom++) {

    t = rational(1,___________);

    prod = prod * t;

  }

  _________ prod;
}
```

4. Functional programming (30 points)

```
> maxint([])
0
> maxint([3])
3
> maxint([3, 2, 4, 9, 1])
9
```

Consider a function maxint that takes a list of positive integers and returns 0 if the list is empty and the largest value in the list if it is not empty. Examples of this function's behavior are shown in the box to the right.

(a) (10 points) Write a recursive version of this function in Haskell.

(b) (10 points) Given a list L and the function maxint as described above, write a function AllBut-Max that returns a list consisting of only those elements of L that are less than the max. Make sure the function maxint is called only once.

(c) (10 points) As in the C++ question, define productSeries(k) as (1/1) * (1/2) * (1/3) * …* (1/k) for any positive integer k**.** Write a Haskell version of this function using

(5 points) recursion

(5 points) high-order function(s)

Haskell Cheat Sheet – condensed version

```
{- short version of Notes on Haskell.  -}
{- compiles and executes correctly using HUGS on gl    -}
and1 :: Bool -> Bool -> Bool
and1 x y = x==True && y==True

and2 :: Bool -> Bool -> Bool   -- two Bool input args
and2 True True = True          -- and pattern matching
and2 _ _       = False         -- with underscore as a wildcard

fact :: Integer -> Integer
fact 0 = 1
fact n = n*fact(n-1)

fact2 n
 | (n==0) = 1
 | otherwise = n*fact2(n-1)

listLen1 :: [a] -> Int
listLen1 []     = 0
listLen1 (x:xs) = 1 + listLen1(xs)

-- here's another way to do listLen
listLen2 :: [a] -> Int
listLen2 = sum . map (const 1)

sumList2 aList = foldr (+) 0 aList

sumSeries n = foldr (+) 0 (map (\x -> x^2) [1..n])

positives aList = filter (\x -> x > 0) aList

splitlist::[a] -> ([a],[a])
splitlist aList = (part1, part2)
  where
    n = length(aList)
    n2 = n `div` 2
    n1 = n - n2
    part1 = take n1 aList
    part2 = drop n1 aList

demohw3 = do
  let aList = [1,-2,4,-6,9]
  putStrLn("aList is "++show(aList))
  putStrLn("sum of square from 1 to 3 should be 14: "++show(sumSeries(3)))
  let (part1,part2) = splitlist [1]
  putStrLn("part1 of splitlist [1] is "++show(part1))
  putStrLn("part2 of splitlist [1] is "++show(part2))
  let (part1,part2) = splitlist(aList)
  putStrLn("part1 of splitlist aList is "++show(part1))
  putStrLn("part2 of splitlist aList is "++show(part2))
  putStrLn(“positive elements of aList “++show(positives(aList)))

main = do
  demohw3
```

C++ Cheat Sheet – condensed version

```
// C++ Cheat Sheet for CMSC 331 Final Exam
// this program compiles and executes correctly using g++ on gl

#include <iostream>      // the built-in stream I/O package
using namespace std;
#include <stdlib.h>
#include <assert.h>      // I like C's assert macro

// class rational, inspired by Budd's data structures book, chap. 2
class rational {
public:
  /* constructors have the same name as the class itself */
  rational ();
  rational (int,int);

  /* accessor functions */
  int numerator () const;
  int denominator () const;

  /* assignment operator(s), note overloading */
  void operator = (const rational &);

  /* arithmetic */
  friend rational operator + (const rational &, const rational &);

  /* comparison */
  friend bool operator == (const rational &left, const rational &right);

  /* output */
  friend ostream & operator << (ostream &, const rational &);

private:
  int top;            // should we make sure that top can be pos or neg?
  int bottom;         // and make sure bottom is positive
};

// Constructors
rational::rational()        // make zero the default value
{
  top = 0;  /* could also say this.top = 0 */
  bottom = 1;
}

rational::rational(int numerator, int denominator)
{
  assert( denominator != 0 );     // because that's bad
  top = numerator;           // note no collision with functions of same
name
  bottom = denominator;
}

// Accessors
int rational::numerator() const { return top; }
int rational::denominator() const { return bottom; }

// Assignment
void rational::operator = (const rational &right)
```

```
{
  top = right.numerator();
  bottom = right.denominator();
}

// Arithmetic
rational operator + (const rational &left, const rational &right)
{
  rational result(
              left.numerator() * right.denominator() +
              right.numerator() * left.denominator(),
              left.denominator() * right.denominator());
  return result;
}

// Comparison
bool operator == (const rational &left, const rational &right)
{
  return (left.numerator() * right.denominator() ==
        left.denominator() * right.numerator());
}

// a simple output function
ostream & operator << (ostream & ostr, const rational &r)
{
  ostr << r.numerator() << "/" << r.denominator();
}

int main()
{
  cout << "Hello, world!" <<endl;

  // Test the different constructors (and output)
  rational w, t(3,1), x, y(3,4), z(-2,3), u(y);
  cout << "w = " << w << endl;
  cout << "t = " << t << endl;
  cout << "y = " << y << endl;
  return 0;
}
```