CMSC 331 Final Exam Fall 2013

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, enter T (true) or F (false). (2 points each)

1. _______In Perl, a function takes a fixed number of arguments, set at compile-time.
2. _______Haskell's method of strong type checking means that type-mismatch errors are generally detected at compile-time, avoiding certain kinds of errors.
3. _______The "assert" statement in C and C++ is useful for an informal sort of program verification.
4. _______ Modern functional languages like Haskell, which feature lazy evaluation, have performance advantages compared to older languages like Lisp.
5. _______In Haskell and Scheme, the read-eval-print loop is what reads commands from the user, performs them, and show the user the results.
6. _______If a function in Haskell has only one formal parameter, then it's a recursive function.
7. _______When Haskell complains about type mismatch, the problem could be as simple as misspelling the name of a function.
8. _______Currying refers to taking the results of one function and using it as the input to another function.
9. _______In Perl, wrapping a regular expression in parentheses is used to "remember" the matched string for use, for example, in an assignment statement.
10. _______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.
11. _______Perl and Haskell have at least one feature in common: there is usually more than one way to do something.
12. _______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.
13. _______The C++ language allows for explicit definition of constructor and destructor functions.
14. _______Using lambda expressions, in a language like Haskell, one can define functions and use them without giving them a specific name.
15. _______A list comprehension is one way that Haskell provides for data structures that are essentially of infinite length.

2. Fill in the Blanks (20 points, 2 points each)

1. In UNIX, the __________ command is used to make a file executable.

2. Recursive descent is an example of what is known as ______________ parsing.

3. In _________________ arrays, the subscripts *may* be integers but other types may be used as well.

4. In Perl, it is sometimes useful to understand whether a variable is being used in the list or __________ context.

5. Alan Turing's work on early computers played an important role in the Battle of the Atlantic during World War _________.

6. In Haskell, the __________ class restriction requires that the input argument(s) have a defined less than (<) function. (Capitalization counts on this problem.)

7. In Python, a copy operation is said to be ______________ if it uses pointers, rather than allocating new storage, as it performs the copying.

8. In some languages including Haskell, _________________ functions are functions that take other functions as arguments.

9. When writing functions in Haskell, it's important to name functions so that they don't conflict with functions already defined in the _____________________.

10. In several languages, a _________________ expression is used to define a function that might otherwise not have a name.

3. Understanding Lisp and Scheme (10 points)

Consider the following Scheme program, which compiles and runs to completion on GL.

```
#lang scheme
; scheme problem for final exam

(define (aFunc lis)
  (cond
   ((null? lis) 0)
   ((not (list? (car lis)))
    (cond
     ((null? (car lis)) (aFunc (cdr lis)))
     (else (+ 1 (aFunc (cdr lis))))))
   (else (+ (aFunc (car lis)) (aFunc (cdr lis))))))

(when #t
      (printf "Hello, world!~n")
      (printf "~a~n" (aFunc '()))
      (printf "~a~n" (aFunc '((1 2) (3 4 5))))
)
```

(5 points) What does this program do?

(5 points) What is the output of this program?

4. Understanding Perl (10 points)

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. The following is a *recursive* version of the maxint function in Perl. Fill in the blanks.

```
#!/usr/bin/perl
print "Recursive version of maxint function!\n";  # a short Perl example

use strict;  # an example of a pragma
use warnings;

_____  ________ {
   if (______ == 0) {      # check for empty list
        0;
    } else {
        _____ $first = _________;   # save first element of input list
        my @newList = @_;          # make a copy of the input list
        ________ @newList;         # remove its first element
        my $max1 = maxint(@newList);    # recursion
        if (___________ > __________) {
            ___________
        } else {
            ___________
        }
    }
}

my @aList = (7,3,4,5);
my $max1 = maxint(@aList);
print "maxint of @aList is $max1\n";
```

5. Functional programming (20 points)

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

Once again, 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 YOUR CHOICE of either Lisp (Scheme) or 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. There may be several ways to do this, but make sure the function maxint is called only once. YOUR CHOICE of Lisp or Haskell.

6. Programming in Haskell (10 points)

Consider the function productSeries, defined as:

productSeries(k) = (1/1) * (1/2) * (1/3) * …* (1/k) for any positive integer k**.**

Implement this function in Haskell (not LISP) using some appropriate combination of recursion or high order functions. Again, there may be several ways to do this.

LISP Cheat Sheet – condensed version

```
(defun helloWorld ()
  ;; prints the standard message
  (print "Hello World"))

(defun predicateDemo ()
  (print (member 3 (list 1 2 3 4 5 6)))  ; prints (3 4 5 6)
  (print (symbolp 'foo)) ; should be TRUE
  (print (eq 4 (+ 2 2)))  ; good for atoms
  (print (equal '(1 2 3) '(1 2 3)))  ; good for other things
  (print (null ()))  ; should be TRUE
  (print (listp '(1 2 3)))  ; should be TRUE
  (print (listp '3))  ; should be FALSE
)

;; basic lisp manipulation
(defun listDemo()
  (print (cons 'a 'b)) ; creates a dotted pair
  (car '(this is a list))
  (cdr '(this is another list))
  (cons 'quick '(brown fox))
  (append '(1 2) '(3 4))
  (print (reverse '(1 2 3)))
)

;; every language needs an if statement
(print (if (= (+ 1 2) 3) 'yup 'nope))

;; the cond statement does more!
(defun condDemo()
  (let ((a 2))
    (cond ((eq a 1) (princ "a is 1"))
          ((eq a 3) (princ "a is 3"))
          (t        (princ "a is something else")))))

;; example of mapcar
(mapcar #'sqrt '(1 2 3 4 5))
(mapcar (function sqrt) '(1 2 3 4 5))
; both return (1 1.4142135 1.7320508 2 2.236068)

;; example of apply
(apply #'append '((10 20) (30) (40 50)))
; returns (10 20 30 40 50)
```

Perl Cheat Sheet – condensed version

```
#!/usr/bin/perl
print "Hello, world!\n";  # a short Perl example
# Perl statements end with a semicolon
# ordinary variables begin with $
my $nDocs = 0;
# perl is great for working with strings
# strings are delimited with either single quotes or double quotes
my $bString = "Inside double quotes, usual escapes apply";
# period . is the string concatenation operator
my $aLongString = $aString."\nconcatenated with the period operator\n".
"and the x factor for repetition\n$bString\nand interpolation!";

# LOTS of pattern matching operators, including s for substitute
$aString = "Do not do that!\n";
print $aString;
$aString =~ s/Do not/Don't/;
print $aString;

# nothing special about Boolean variables
# if it's zero, it's false, otherwise it's true
my $aBoolean = $fred lt $barney;

# control structures include if
# although we just defined $aBoolean, we can always test
if (defined($aBoolean) && $aBoolean) {
    print "fred is less than barney\n";
} else {
    print "fred is NOT less than barney\n";
}

# Perl supports lists and arrays, closely related concepts
# subscripts start at zero normally, as in C
my @fred;
$fred[0] = 2.8;
$fred[1] = "Wilma!";
$fred[2] = 'dino' x 2;

# push and pop add or delete elements from the end of a list
# use $fs to access an individual element of the list
# but we can use @fs to refer to the whole list
#
# sometimes we consider what Perl expects, i.e. list vs. scalar context
#
my @fs = qw/fred wilma barney betty/;
push @fs, qw/bambam pebbles/;
# shift and unshift delete or add elements to the start of a list
unshift @fs, "dino";
# list elements are separated by blanks when interpolated
print "@fs\n";
my @sf = reverse(@fs);
printf "Print the list in reverse @sf\n";

# simple echo of a given input file (default to STDIN) to STDOUT
sub echoDemo {
    # note use of @_ in both the scalar and list contexts in this if
    my $XLSfile = "default.xls";
    if (@_ == 1) {
      # list of arguments in list context
      ($XLSfile) = @_;
    } else {
      print "usage: echoDemo(inputFile=STDIN[,XLSfile=default.xls]\n";
    }

    my $word;
    my $nTerms=0;

    # Perl has built-in hash functions
    my %aHashTable = ();  # will be used in example below

    while (my $inputLine = <STDIN>) {
      chomp $inputLine;  # get rid of trailing newline
      print STDOUT "$inputLine\n";   #  I/O looks very C-like, eh?

      # recall that array names begin with @
      # split $inputLine into an array of blank-separated words

      my @words=split(" ", $inputLine);
      foreach $word (@words) {
          $aHashTable{$word} += 1;

          # to practice with regular expression matching, see if word
          # would be a good password, i.e. having at least one
          # digit, one lower case letter, and one upper case letter

          # make sure it finds a good password when run on itself XYzzy18
          if ($word =~ /[A-Z]/ && $word =~ /[a-z]/ && $word =~ /[0-9]/) {
            print "$word would be a good password.\n";
          }
      }
    }
    for $word (sort(keys(%aHashTable))) {
      $nTerms++;
      # but let's make each word lower-case
      # and make a variable wordlc local to this block
      my $wordlc = $word;
      $wordlc =~ tr/A-Z/a-z/;  # tr stands for translate
      printf STDOUT "%s, %d\n", $wordlc, $aHashTable{$word};
    }
}

#
# working with files and directories, and patterns
my @files = <*>;
foreach my $myFile (@files) {
    if (-f $myFile and -r $myFile) {   # see if it's a readable file
      my ($nl, $nw, $nch) =
          `wc $myFile` =~ /([0-9]+)\s+([0-9]+)\s+([0-9]+)/;
      print "file $myFile has $nl lines, $nw words and $nch characters\n";
    }
}
```