CMSC 331 Final Exam Fall 2011

Name: _________________________________
UMBC username:_____________________________

You have two hours to complete this closed book/notes 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.

1. True/False (40 pts: 20*2)
For each of the following questions, circle T (true) or F (false).

T  F  COBOL was most often used for Artificial Intelligence work.
T  F  FORTRAN was developed primarily for applications in science and engineering.
T  F  JavaScript is a portable subset of the Java programming language.
T  F  Some EBNF grammar cannot be rewritten in BNF.
T  F  Attribute grammars are more powerful than context free grammars because they support iteration in addition to recursion.
T  F  Every BNF grammar G that has left recursion can be converted into a grammar G’ that recognizes exactly the same language and has no left recursive rules in it.
T  F  If two languages have the same non-terminal symbols and different grammar rules, the languages cannot be identical (e.g., have the same strings in them).
T  F  Operator precedence rules determine which operators appear lower or higher in a parse tree.
T  F  Attribute grammars associate pre-conditions and post-conditions with each statement in a program.
T  F  A statically typed language usually requires a programmer to declare the types of variables.
T  F  Lexical scanners are usually specified using regular expressions.
T  F  Every non-deterministic finite automaton can be rewritten as a deterministic finite automaton.
T  F  A special form in Scheme is a function that evaluates its arguments in a non-conventional way.
T  F  Scheme’s cond is a special form, but Scheme’s if is a regular function.
T  F  Scheme has no infix operators.
T  F  Non-local variables in a Scheme function are assumed to refer to a variable in the environment in which the function was defined.
T  F  Non-local variables in a Python function are assumed to refer to a variable in the environment in which the function was called.
T  F  One difference between the Scheme functions set! and define is that set! never creates a new variable where as define can.
T  F  One difference between the Scheme functions set! and define is that define can only change a local variable’s value, whereas set! can change the value of a variable in an inherited context.
2. General multiple-choice questions (25: 8*3 + one free!)

Circle all of the correct answers for each of the following questions

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

2.2 The Unix LEX program is used to define the (a) rules for tokenizing the input for a parser; (b) lexical attributes for an attribute grammar; (c) the lexical structure of a language; (d) a bottom up shift-reduce parser.

2.3 Most modern programming languages use the following schemes for their variable scoping; (a) lexical (i.e., static) scoping; (b) dynamic scoping; (c) both lexical and dynamic scoping; (e) scope inferencing.

2.4 What distinguishes a purely "functional" programming language from an "imperative" one? (a) There are no variables in a purely functional language; (b) Operations with side effects and minimized in a pure functional programming language; (c) Pure functional programming languages are strongly typed; (d) There is no real difference, only a difference in the recommended coding style.

2.5 In a dynamically typed programming language (a) A variable's type is looked up in the environment of the caller; (b) Types are associated with objects and values, but not with variables or functions; (c) Many type errors are never caught, leading to unreliable programs; (d) Type errors are caught at run time and not when a program is compiled or loaded.

2.6 In both Scheme and Python, lambda expressions are used to (a) define macros; (b) create functions without giving them names; (c) create variables that are scoped dynamically; (d) define special forms.

2.7 Lazy evaluation is a technique that can be used to (a) execute special forms; (b) evaluate expressions only when their values are needed; (c) create infinite streams; (d) evaluate function arguments in parallel.

2.8 Which of the following are true about meta-circular evaluators i.e. interpreters for a programming language written in the programming language itself. (a) They can never evaluate themselves. (b) They can never implement every feature of the language. (c) They shed little light on how to implement the language. (d) If carefully implemented, they can evaluate themselves.
3. Ambiguous grammars (15 pts)

Consider the two grammars G1 and G2, each of which has a single non-terminal symbol, S.

G1: $S ::= x \ S \ z \ | \ x \ S \ y \ S \ z \ | \ a$

G2: $S ::= x \ S \ | \ x \ S \ y \ S \ | \ a$

(a) (5 pts) Which of these two grammars is ambiguous? Circle all of the ambiguous ones: G1 G2

(b) (10 points) For each ambiguous grammar, show that it is ambiguous by giving two different parse trees for a single sentence in the language defined by the grammar.
4. Regular expressions (20)

The following regular expression recognizes certain strings consisting of the letters x, y and z. Recall that the Kleene star (*) means zero or more repetitions of the pattern it follows, the vertical bar is an infix operator delimiting alternatives, parentheses are used for grouping patterns and white space does not matter.

\[ x \ ( (xy) \mid (xz) \) * \ z \]

For each of the following strings, circle the response to show whether it is IN or OUT of the language defined by this regular expression (3 points each)

3.1 [IN] [OUT] xxzz
3.2 [IN] [OUT] xyyx
3.3 [IN] [OUT] xz
3.4 [IN] [OUT] xyyxyxzz
3.5 [IN] [OUT] xyyzz

3.6 (10 points) Draw a deterministic finite automaton (DFA) for the language defined by this grammar using as few states as possible. Label each arc with the input required for that transition. Mark the initial state with an arc coming into it and mark all final states with a double circle.
5. Manipulating lists in Scheme (30 pts, 6*5)

Complete the table below. The first column shows a scheme structure made of pairs (i.e., cons cells) and integers. The second column gives a Scheme expression that when evaluated would return the structure in the first column. The third column is a Scheme expression that would return the value 2 if the variable X is bound to the structure in the first column. We’ve completed the first row for you,

<table>
<thead>
<tr>
<th>X</th>
<th>How to construct X using only cons and null and integers</th>
<th>How to return the value 2 in X using only car and cdr</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 2)</td>
<td>(cons 1 (cons 2 null))</td>
<td>(car (cdr X))</td>
</tr>
<tr>
<td>((1 2))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((1) (2))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1 (2))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((1) 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0 1 . 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1 ((2)))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6. Little Scheme functions (50: 5*10)

For each of the following functions, describe what it does in a sentence. You can assume that each function is always called with a proper list of integers. Assuming that mylist is (1 2 3 4), show what the function returns when called with mylist. We’ve done the first one as an example.

<table>
<thead>
<tr>
<th>Function Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(define (f0 ints) (apply + ints))</code></td>
<td>f0 returns the sum of the integers in the list ints. (f0 mylist) returns 10.</td>
</tr>
<tr>
<td><code>(define (f1 ints)</code></td>
<td></td>
</tr>
<tr>
<td><code>(if (null? (cdr ints))</code></td>
<td></td>
</tr>
<tr>
<td><code>(car ints)</code>)</td>
<td></td>
</tr>
<tr>
<td><code>(f1 (if (&gt; (car ints) (car (cdr ints)))</code></td>
<td></td>
</tr>
<tr>
<td><code>(cons (car ints) (cdr (cdr ints)))</code></td>
<td></td>
</tr>
<tr>
<td><code>(cdr ints)))</code></td>
<td></td>
</tr>
<tr>
<td><code>(define (f2 ints)</code></td>
<td></td>
</tr>
<tr>
<td><code>(or (null? ints) (null? (cdr ints))</code></td>
<td></td>
</tr>
<tr>
<td><code>(and (&lt;= (car ints) (car (cdr ints)))</code></td>
<td></td>
</tr>
<tr>
<td><code>(f2 (cdr ints)))</code></td>
<td></td>
</tr>
<tr>
<td><code>(define (f3 ints)</code></td>
<td></td>
</tr>
<tr>
<td><code>(member #t)</code></td>
<td></td>
</tr>
<tr>
<td><code>(map (lambda (x) (&lt; x 0)) ints))</code></td>
<td></td>
</tr>
<tr>
<td><code>(define (f4 ints)</code></td>
<td></td>
</tr>
<tr>
<td><code>(define (f4.1 ints x)</code></td>
<td></td>
</tr>
<tr>
<td><code>(if (null? ints)</code></td>
<td></td>
</tr>
<tr>
<td><code>null</code></td>
<td></td>
</tr>
<tr>
<td><code>(cons (+ x (car ints))</code></td>
<td></td>
</tr>
<tr>
<td><code>(f4.1 (cdr ints) (+ x (car ints))))</code></td>
<td></td>
</tr>
<tr>
<td><code>(f4.1 ints 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>(define (f5 ints)</code></td>
<td></td>
</tr>
<tr>
<td><code>(if (null? ints)</code></td>
<td></td>
</tr>
<tr>
<td><code>null</code></td>
<td></td>
</tr>
<tr>
<td><code>(append (f5 (cdr ints))</code></td>
<td></td>
</tr>
<tr>
<td><code>(cons (car ints) null)))</code></td>
<td></td>
</tr>
</tbody>
</table>
7. Functional Programming I (20)

A simple way to represent a vector of N dimensions in Scheme is as a list with N elements. Simple functions on vectors can be written concisely and easily using a functional programming style, e.g., with map, reduce, filter, etc. Adding two vectors is done by adding their values on respective dimensions. For example

> (define (vadd v1 v2) (map + v1 v2))
> (vadd '(1) '(-2))
(-1)
> (vadd '(2 -3) '(-3 -2))
(-1 -5)
> (vadd '(1 2 3) '(-2 -4 -8))
(-1 -2 -5)

Define the following Scheme functions on two vectors using just define, arithmetic functions (+, *, -, /), map, reduce and lambda. Simple versions of map and reduce are given above.

(a) Write a function `vdistance` that returns the distance between the endpoints or two vectors. The distance is computed by adding the squares of the difference between corresponding values, summing the results and taking the square root. For example, the distance between (1 2 3) and (4 5 6) is just sqrt[((1-3)**2 + (2-5)**2 + (3-6)**2)] or ~5.2

(define (vdistance v1 v2)
)

(b) Write a function `vdotproduct` that computes the dot product of two vectors. The dot product is computed by multiplying the corresponding values of the two vectors and then summing the results. For example, the dot product of (1 2 3) and (4 5 6) is just ( 1*3 + 2*5 + 3*6 ) or 32.

(define (vdotproduct v1 v2)
)
8. Dynamic vs. Static Variable Scope (20 pts)

(a) (10 pts) Briefly describe the difference between dynamic and static (or lexical) variable scoping.

(b) (10 pts) Consider evaluating the following Scheme expressions

```scheme
> (define X 100)
> (define (foo) (set! X (+ X 1)) X)
> (define (bar X) (set! X (+ 1 X)) (foo))
> (bar 10)
```

What will the last expression, `(bar 10)`, evaluate to if Scheme uses

- Dynamic scoping:
- Static scoping:
9. Functional programming II (20: 5/5/5/5)

Consider the Scheme function `narf` defined as follows:

```
(define (narf a b) (lambda (c) (a (b c))))
```

9.1 With what types of arguments should `narf` be called?

9.2 What type of object does `narf` return?

9.3 What will the following expression return?

```
(map (narf (lambda (x) (+ x 3)) (lambda (x) (* x x))) '(1 2 3 4))
```

9.4 What will the following expression return where `add1` is defined as
```
(lambda (x) (+ x 1))
```

```
((narf abs (narf add1 add1)) -100)
```