1	50/
2	40/
3	20/
4	20/
5	35/
6	25/
7	30/
8	30/
9	25/
10	25/

	300/

17 May 2010

CMSC 331 Final Exam Spring 2010

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 but adding some might help us understand your code.

1. True/False (50 pts: 25*2)

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

T F Scheme uses dynamic typing whereas Python uses static typing.

T F In Python, types are associated with values and not with variables.

T F If a Python function terminates without explicitly returning a value, it returns the value None.

T F A disadvantage of dynamic type checking is that type-mismatch errors are generally not detected until runtime and then only if the program is thoroughly tested.

T F Scheme's syntax eliminates the need for precedence rules.

T F Scheme does not have any infix operators.

T F The Scheme interpreter optimizes tail-recursion as iteration.

T F A macro in Scheme is the only way that users can define new special forms.

T F A lambda expression in Python can only have one expression in its body.

T F Scheme functions look up the values of non-local variables in the environment of their caller.

T F Scheme's call-with-current-continuation can be used to define new mechanisms to control execution flow.

T F The Scheme-in-scheme interpreter does not really implement a lexical scanner or a parser.

T F The Scheme-in-python interpreter uses a recursive descent parser.

T F Python does not optimize tail-recursion as iteration.

T F In Scheme, any value that is not considered as the False is interpreted as True.

T F Python's lists data structure is mutable.

T F Every Scheme procedure or function returns a value.

T F In a Python dictionary, all of the values must be unique.

T F The keys stored in a Python dictionary must be immutable data structures.

T F The elements of a Python tuple must be immutable data structures.

T F In Python, the expression *range(5)[1:]* returns the list [1,2,3,4].

T F Namespaces in Python are roughly the equivalent of environments in Scheme.

T F Because Python tuples are immutable, their elements must be of the same type.

T F Scheme's *delay* special form and *force* function can be used to implement streams.

T F Optimizing tail recursion in Python is not important because Python has so many iterative control structures.

2. General multiple-choice questions (40 pts: 10*4)

Circle the letters for **all** of the correct answers.

1. Which of the following Python expressions would be interpreted as true when evaluated: (a) #t; (b) True (c) 0; (d) [ ]; (e) ‘’; (f) {}; (g) 1; (h) not(-1) .

2. Which phrase best describes the variable scoping used in Python? (a) wide scoping; (b) narrow scoping; (c) lexical scoping; (d) dynamic scoping; (e) structured scoping; (f) functional scoping; (g) static scoping; (h) none of the above.

3. In the Scheme interpreters we studied in class, which of the following items are a part of the representation of a procedure definition**:** (a) the procedure’s name; (b) the procedure’s parameter names; (c) the procedure’s code; (d) the environment in which the procedure was defined; (e) the procedure’s type; (f) storage for the procedure’s arguments.

4. In the Scheme interpreters we studied in class, a variable environment is a list of frames. Each frame is a data structure that records: (a) only names of local variables; (b) the names of local variables and their current values; (c) only the values of local variables; (d) the function whose call created the frame; (e) a list of global variables and their values.

5. Scheme macros are typically used to define: (a) new special forms; (b) tail-recursive functions; (c) functions that don’t evaluate all of their arguments; (d) functions that may evaluate their arguments in an idiosyncratic way; (e) new syntactic constructs.

6. While a recursive descent parser can be natural and efficient for parsing Scheme programs, one drawback in using it in Python is that: (a) It is difficult to transform it into a grammar with no left recursion; (b) Python has a limited stack and does not optimize tail-recursion as iteration; (c) the implementation in Python is obscure; (d) it is slow compared to using the trampolining style of programming.

7. A type safe programming language is one that catches type errors (errors due to a mis-match of types between data and operations) before execution. Which of the following languages are generally considered to be type safe? (a) Scheme; (b) Java; (c) C; (d) C++; (e) Python.

8. Which of the following languages use dynamic typing for variables? (a) Lisp; (b) Scheme; (c) Java; (d) JavaScript; (e) Python; (f) C.

9. When evaluated, a lambda special form always returns (a) a function of one or more arguments; (b) a continuation; (c) a promise; (d) a special form; (e) a function; (f) an s-expression.

10. Which of the following are true of Scheme’s **call/cc** function? (a) it is a special form; (b) it calls a function with a special argument that represents what should be computed next.; (c) it is used to implement lazy evaluation; (d) it returns a closure; (e) it returns a continuation.

3. Static vs. dynamic scoping (20: 5/5/5/5)

Scheme uses static or lexical scoping for variables. We looked at how we could modify the Scheme meta-circular interpreter of that it used dynamic scoping. Consider the following Scheme session.

```
> (define x 10)
> (define (f1 x)
    (set! x (+ x x))
    (f2))
>(define (f2) (print x))
> (f1 100)
?????
```

(a) what would be printed in place of ????? in a normal Scheme that uses static scoping?

(b) what would be printed in place of ????? in a modified Scheme that uses dynamic scoping?

Now consider this session:

```
> (define x 10)
> (define (f1 x)
    (define (f2) (print x))
    (set! x (+ x x))
    (f2))
> (f1 100)
?????
```

(c) what would be printed in place of ????? in a normal Scheme that uses static scoping?

(d) what would be printed in place of ????? in a modified Scheme that uses dynamic scoping.

4. Slices in Python (20 pts: 5/2/2/2/2/2/2/2 + 1 free!)

(a) To which Python data structures is the slicing operation applicable.

(b) In the following Python interactive session, show what would be printed for the missing values. If the expression would result in an error, write *error*

```
>>> "abcdefghij"[1:3]
```

```
>>> "abcdefghij"[2:2]
```

```
>>> "abcdefghij"[2:1]
```

```
>>> L = ['a','b','c','d','e','f']
>>> L[0:1] = ['x','y']
>>> L
```

```
>>> L2 = L[:]
>>> L2
```

```
>>> L2[0:2] = ['foo']
>>> L2
```

```
>>> L
```

5. Python regular expressions (35 pts, 5/5/5/20)

Write regular expression patterns to match US social security numbers. Your patterns should match an entire string, leaving no characters unmatched. Your answers should be able to support the sample session in the box below and to the right. Please give your answers as assignments to variables P1, P2 and P3, e.g., P1 = r'…your re pattern goes here…'.

(a) Write a simple Python regular expression, P1, that matches a SSN like "123-45-6789", i.e., three digits, a dash, two digits, a dash, four digits.

(b) Write another regular expression, P2, that matches SSNs where the separators between the numeric groups are optional and can a mixed sequence dashes or white space characters.

(c) Write a final version, P3, that extends P2 and defines three capturing match groups, one for each of the three "fields" in the SSN.

Python RE symbols

^	beginning of the string
$	end of the string
+	one or more times
?	at most one time
*	zero or more time
(...)	a capturing group
(?:...)	a noncapturing group
\t	a tab
\n	a newline character
{n}	n times
{n, m}	a range at least n and at most m
[...]	a character class
.	any character
\s	whitespace
\d	a digit
\w	an alphanumeric or underscore
\b	a word boundary

Example session

```
>>> import re
>>> re.match(P1, "123-45-6789").group()
'123-45-6789'
>>> print re.match(P1, "123-45-67890")
None
>>> print re.match(P1, "123 45 6789")
None
>>> print re.match(P1, "123-x5-6789")
None
>>> re.match(P2, "123-45-6789").group()
'123-45-6789'
>>> re.match(P2, "123 45   6789").group()
'123 45 6789'
>>> re.match(P2, "123456789").group()
'123456789'
>>> print re.match(P2, "1234567890")
None
>>> re.match(P3, "123-45-6789").groups()
('123', '45', '6789')
>>> re.match(P3, "123456789").groups()
('123', '45', '6789')
>>> re.match(P3, "123 45 6789").groups()
('123', '45', '6789')
>>> print re.match(P3, "123-45-67890")
None
```

(5d) Fill in the following table. For each row, assume that we evaluate the expression

```
mo = re.match(pattern,string)
```

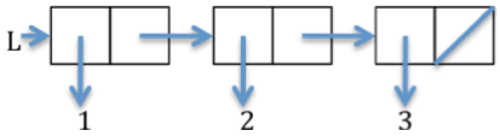
The variable *mo* will be matched to the result of the match, i.e., either *None* if the match failed or a *match object* if it succeeded. A match object has attributes and methods that can provide the details of the match.

Fill in the missing values for *Match?* (i.e., does the pattern matched the string, *mo.group()* (i.e., what part of the string matched the pattern) and *mo.group(1)* (i.e., which is the first of the pattern's match groups). Enter N/A in a cell if no answer is appropriate for it or it generates an error.

Pattern	**String**	**Match?**	**mo.group()**	**mo.group(1)**
*a+(p)**	*apple*	*Yes*	*app*	*p*
(\d*).	12345			
(123|abc)	123abc			
\w+\s(\w*)	foo bar x			
(.*)y$	xyxyx			
a.([bc]+)	abcbcbcbc			

6. Destructive assignment in Scheme (25 pts, 5/5/5/5/5)

Scheme has two destructive assignment functions that can change the pointers in a mutable cons cell: **set-mcar!** and **set-mcdr!.** As of version R6RS of Scheme, these functions can only be applied to mutable pairs and lists – i.e., those constructed with functions like *mcons* and *mlist*. Assume we evaluate (require scheme/mpair) and then each of the following six expressions are evaluated in order. After each is evaluated, show what will be printed for L and draw a box and pointer diagram of the structure of mpairs. Be sure to show any structure sharing. Recall that mutable lists are printed with curly braces and not parentheses. Recall also that circular lists are printed with tags to indicate sharing. For example, an mpair whose mcar is 1 and mcdr points to itself is printed as *#0={1 . #0#}*

Expression	Value of L	Diagram for L
`(define L (mlist 1 2 3))`	`{1 2 3}`	
`(set-mcar! L (mcar (mcdr L)))`		
`(set-mcdr! L (mcdr (mcdr L)))`		
`(set! L (mcons 1 L))`		
`(set-mcdr! L L)`		
`(set-mcar! L L)`		

7. Splitting lists with Scheme (30, 15/15)

(a) Define a Scheme function, *odds*, that takes a list and returns every other one, starting with the first. See the example to the right below.

```
> (odds '( ) )
()
> (odds '(a))
(a)
> (odds '(a b))
(a)
> (odds '(a b c))
(a c)
> (odds '(a b c d e f g h))
(a c e g)
```

(b) Using the function odds and any of the primitive functions if, null?, cond, car, cdr and cons, write a function evens that takes a list and returns the elements in even positions, starting with the second. Pay attention to the end conditions.

```
> (evens '( ))
( )
> (evens '(a))
( )
> (evens '(a b))
(b)
> (evens '(a b c))
(b)
> (evens '(a b c d e f g h))
(b d f h)
```

8. Splitting lists with Python (30, 15/15)

(a) Define a Python function, *odds*, that takes a list and returns every other element, starting with the first. See the example to the right below.

```
>>> odds([])
[ ]
>>> odds(['a'])
['a']
>>> odds(['a','b'])
['a']
>>> odds(['a','b','c'])
['a', 'c']
>>>
odds(['a','b','c','d','e','f','g','h'])
['a', 'c', 'e', 'g']
```

(b) Using the Python function odds and any of the primitives if, <, len(), return, write a function evens that takes a list and returns the elements in even positions, starting with the second. Pay attention to the end conditions.

```
>>> evens([])
[ ]
>>> evens(['a'])
[ ]
>>> evens(['a','b'])
['b']
>>> evens(['a','b','c'])
['b']
>>> evens(['a','b','c','d','e','f','g','h'])
['b', 'd', 'f', 'h']
```

9. Explain this! (25: 10/15)

(a) Your classmate wrote the simple module shown to the right and put it in a file *grp.py*. When she used it, she got the following error. Explain why this happened. (Don't just paraphrase the error message, but explain why Python was confused.)

```
group = [ ]
size = 0

def add(new):
    group.append(new)
    size = size + 1
```

```
>>> import grp
>>> grp.add('alice')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "grp.py", line 6, in add
    size = size + 1
UnboundLocalError: local variable 'size' referenced before assignment
```

(b) She modified the file as shown in the second box, re-started Python, and entered the following:

```
group = [ ]
size = 0

def add(new):
    global size
    group.append(new)
    size = size + 1
```

```
>>> import grp
>>> group = grp.group
>>> size = grp.size
>>> add = grp.add
```

She added lines 2-4 above to make her life easier by adding aliases for the variables and function in the grp module, so that she could refer, for example, to *grp.group* as just *group*. It seemed to work at first, but when she checked on the group and it's size, she was surprised and confused. Explain what's going on.

```
>>> group
[ ]
>>> size
0
>>> add('alice')
>>> group
['alice']
>>> size
0
>>> grp.group
['alice']
>>> grp.size
1
```

10. Tail recursion (25)

Here's a simple way to compute the length of a proper list in scheme:

```
(define (length l) (if (null? l) 0 (+ 1 (length (rest l)))))
```

The function is not tail-recursive since it must recurse to the end of the list before starting to add up the numbers. Write a tail-recursive version using the following template. Length will still take a single argument, the list, and returns the result of calling the tail recursive version, length-tr, which might have additional or different arguments than length.

```
(define (length l) (length-tr ... ))
(define (length-tr ... ) ... )
```