1	40/
2	40/
3	25/
4	30/
5	36/
6	40/
7	20/
8	15/
9	50/
296/	

17 December 2010

CMSC 331 Final Exam Fall 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 (40 pts: 20*2)

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

T F Most programming languages are implemented as interpreters these days. F

T F Static type checking adds some execution-time overhead, but improves reliability of programs. F

T F A grammar G is ambiguous if there is more than one parse tree for at least one sentence in the language defined by G. T

T F Lexical scanners are usually specified using regular expressions. T

T F Axiomatic semantics define the meaning of statements in a programming language by translating them into statements in another programming language. F

T F While every BNF grammar can be rewritten in EBNF, not every EBNF grammar can be rewritten in BNF. F

T F Both Scheme and python use dynamic typing. T

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

T F Every Python function returns a value. T

T F An advantage of static typing is that a compiler can detect many type errors. T

T F Scheme's syntax eliminates the need for **both** precedence and associatively rules. T

T F Using the yield function in a Python function makes it a generator. T

T F Scheme and Python both use dynamic scoping. F

T F Since Python doesn't optimize tail-recursive calls, a recursive function can run out of stack space. T

T F A decorator in Python allows one to write a function that can return a stream of values on demand. F

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

T F While the number of elements in a Python tuple can not be changed, you can replace an element with a new value. F

T F A Prolog fact is actually a rule with a trivial condition of true. T

T F Scheme's delay special form returns a closure. T

T F Prolog uses backtracking to search for a solution to a query. T

2. Grammars I (40)

Consider the grammar to the right where uppercase letters indicate non-terminals and lowercase letters indicate terminals. Which of the following sentences are in the language defined by this grammar? If the sentence is in the language, show a parse tree.

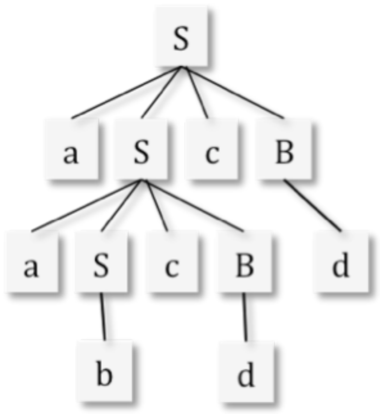
```
S -> a S c B
S -> A
S -> b
A -> c A
A -> c
B -> d
B -> A
```

(a) acccbd

This is not in the language.

(b) aabcdcd

S(a, S(a, S(b), c, B(d)), c, B(d))

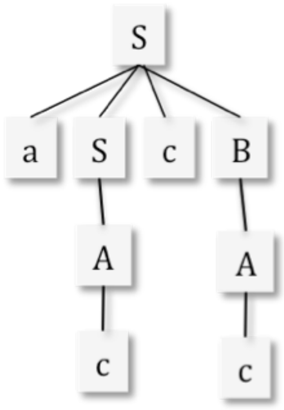


(c) acd

This is not in the language.

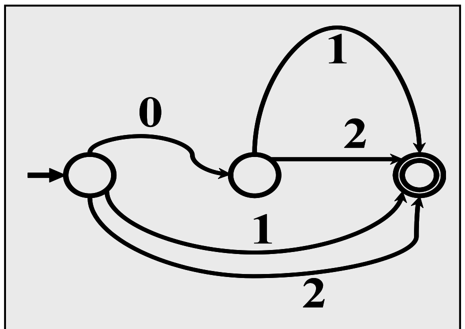
(d) accc

S(a, S(A(c)), c, B(A(c)))



3. Deterministic finite state automaton (25)

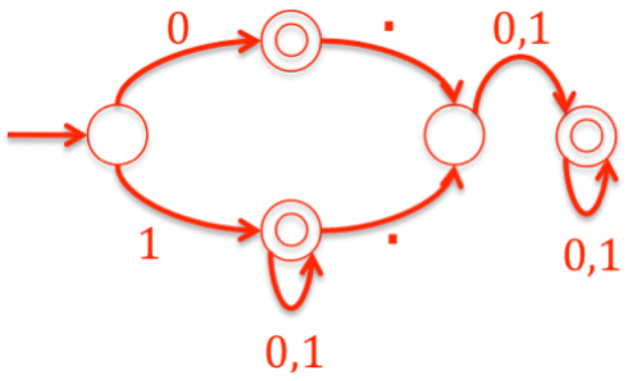
Draw a deterministic finite automaton (DFA) that accepts a binary fraction (BF) like "101.01" or "101" or "0.1". A BF can be a binary integer with no fractional part or a true fraction with an integer part (which can be zero), a decimal point and a fractional part (which can be zero). No leading zeros are allowed, i.e., if the BF starts with a zero, it must be immediately followed by a decimal point and a fractional part or followed by nothing. Any number of trailing zeros are OK. Empty integer (".01") or fractional ("10.") parts are not allowed.



This example shows a DFA for the regular expression **`0?[12]`**

- These are legal binary fractions: 0, 1, 101, 0.0, 0.00, 1.0, 1.00, 0.101, 10.01.
- These are illegal: 010, 01.01, .101, 101. 0.

Use the conventions in the example DFA in the figure above:
- There is only one start state and it has an arc pointing to it without a node at its tail.
- There is a least one accepting state. Accepting states are marked as such by having a double circle.



4. Python regular expressions (30 pts, 5/5/5/15)

Write regular expression patterns to match a US telephone number. 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 = "… pattern goes here…".

(a) Write a simple Python regular expression, P1, that matches a telephone number like "1-800-555-1212" or "410-455-1000", i.e., an optional "1-" prefix, an area code of three digits, a dash, an exchange of three digits, a dash, and four final digits. An additional constraint is that neither the area code nor exchange can start with a zero or a one, e.g., "***1***23-456-6789" and "234-**0**55-7890' are illegal. The other digits are unconstrained.

(b) Write another regular expression, P2, that matches phone numbers where the separators between the numeric groups are optional and can be either a single space, dash or period.

(c) Write a final version, P3, that extends P2 by defining three capturing match groups for the area code, the exchange and the final four digits.

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
{n}	n times
{n, m}	a range at least n and at most m
[...]	a character class
.	any character
\s	whitespace
\S	non-whitespace
\d	a digit
\D	a non-digit
\w	an alphanumeric or underscore
|	or

Example session

```
>>> from re import match
>>> match(P1, "410-455-0522")
<_sre.SRE_Match object at 0x1004ca990>
>>> match(P1, "410-000-3522")
>>> match(P1, "123-455-3522")
>>> match(P1, "1-410-800-3522")
<_sre.SRE_Match object at 0x1004e7558>
>>> match(P2, "1 410 800 3522")
<_sre.SRE_Match object at 0x1004ca990>
>>> match(P2, "1-8005551212")
<_sre.SRE_Match object at 0x1005213f0>
>>> match(P2, "18005551212")
<_sre.SRE_Match object at 0x1005216c0>
>>> match(P3, "1-800-555-1212").groups()
('800', '555', '1212')
>>> match(P3, "1-800.555.1212").groups()
('800', '555', '1212')
>>> match(P3, "800 555 1212").groups()
('800', '555', '1212')
>>> match(P3, "18005551212").groups()
('800', '555', '1212')
```

```
P1 = "(1-)?[2-9]\d\d-[2-9]\d\d-\d\d\d\d$"
```

```
P2 = "(1[\-. ]?)?[2-9]\d\d[\-. ]?[2-9]\d\d[\-. ]?\d\d\d\d$"
```

```
P3 = "(?:1[\-. ]?)?([2-9]\d\d)[\-. ]?([2-9]\d\d)[\-. ]?(\d\d\d\d)$"
```

(4d) 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.

Recall that if we evaluate *m = re.match(pattern, string)* then

- *m* will be *None* if *pattern* does not match *string*
- *m.group(0)* is the portion of *string* that *pattern* matched
- *m.group(1)* is the portion of *string* that the first "match group" in *pattern* matched. A match group is part of pattern that is inside parentheses. Each of he problems below has just one match group.

Fill in the missing values for *Match?*, *mo.group()*, and *mo.group(1)*. Enter N/A in a cell if no answer is appropriate for it or it generates an error.

Pattern	String	Match?	mo.group(0)	mo.group(1)
*a+(p)**	*apple*	*Yes*	*app*	*p*
a*(.*)	aaaa	**Yes**	**'aaaa'**	**''**
.+(a+)	aaaa	**Yes**	**'aaaa'**	**'a'**
\D*(\d+)\D.*	eat 100 pies	**Yes**	**'eat 100 pies'**	**'100'**
.*\((.*)\).*	(1 (2 3) 4)	**Yes**	**'(1 (2 3) 4)'**	**'2 3) 4'**
([01]*)[01][01]	0110011100;	**Yes**	**'0110011100'**	**'01100111'**

5. Manipulating lists in Scheme (36 pts, 6*6)

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,

X	***How to construct X using only cons and null and integers***	***How to return the value 2 in X using only car and cdr***
(1 2)	(cons 1 (cons 2 null))	(car (cdr X))
(0 1 2)	(cons 0 (cons 1 (cons 2 null)))	(car (cdr (cdr X)))
((1)(2))	(cons (cons 1 null)(cons (cons 2 null) null))	(car (car (cdr X)))
((1 2))	(cons (cons 1 (cons 2 null)) null)	(car (cdr (car X)))
(1 . 2)	(cons 1 2)	(cdr X)
(((2)) 1)	(cons (cons (cons 2 null) null) (cons 1 null))	(car (car (car X)))
(1 ((2)))	(cons 1 (cons (cons (cons 2 null) null) null))	(car (car (car (cdr X))))

6. Zip and unzip in Python (40, 10/10/10/10)

(a) Write a simple two-argument version of the built-in Python function **zip** that takes two lists of equal length and returns a list of tuples, where the ith tuple is composed of the ith elements of the two lists. Use iteration rather than recursion, map, or list comprehensions.

```
>>> zip([ ], [ ])
[ ]
>>> zip([1], [2])
[(1, 2)]
>>> z = zip([1,2,3],[10,20,30])
>>> z
[(1, 10), (2, 20), (3, 30)]
>>> unzip(z)
([1, 2, 3], [10, 20, 30])
>>> unzip([ ])
([ ], [ ])
```

```
def zip (list1, list2):

   zipped = []
   for i in range(len(list1)):
      zipped.append( (list1[i], list2[i]) )
   return zipped
```

(b) Write the zip function using map, rather than iteration, recursion or list comprehensions. Hint: You'll find the function **lambda x,y: (x,y)** useful. It takes two args and returns a tuple of them.

```
def zip (list1, list2):

  return map(lambda x,y:(x,y), list1, list2)
```

(c) Write a function unzip that reverses the zip process, returning a tuple of two lists comprising the first tuple elements and the second, respectively. Use iteration rather than recursion, map, or list comprehensions.

```
>>> zip([ ], [ ])
[ ]
>>> zip([1], [2])
[(1, 2)]
>>> z = zip([1,2,3],[10,20,30])
>>> z
[(1, 10), (2, 20), (3, 30)]
>>> unzip(z)
([1, 2, 3], [10, 20, 30])
>>> unzip([ ])
([ ], [ ])
```

```
def unzip (lst):

   l1 = []
   l2 = []
   for (x,y) in zipped:
      l1.append(x)
      l2.append(y)
   return (l1, l2)
```

(d) Write a function unzip that reverses the zip process, returning a tuple of two lists comprising the first tuple elements and the second, respectively. Use map or list comprehensions rather than iteration or recursion. Hint: think about how to build a list of just the first tuple elements from the list and then do the same thing for just the second elements from the tuples in the list. If you can do both of those, then just return a tuple of their results. For map, you'll find something like **lambda x: x[0]** helpful.

```
def unzip (lst):

   return map(lambda x: x[0], lst), map(lambda x: x[1], lst)

-- or –

  return [x[0] for x in lst], [x[1] for x in lst]
```

7. Zip and unzip in Scheme (20: 10/10)

(a) Write a Scheme version of zip that takes two arguments that are lists of equal length and returns a list of two-element lists where the ith two-list has the ith elements of the first and second input lists. You can do this with a recursive function or use map. *Hint: The recursive version is easy, but using map is even easier.*

```
> (zip '() '())
()
> (zip '(1) '(a))
((1 a))
> (zip '(1 2 3) '(10 20 30))
((1 10) (2 20) (3 30))
> (unzip '())
(() ())
> (unzip '((1 a)))
((1) (a))
> (unzip '((1 a) (2 b)))
((1 2) (a b))
> (unzip '((1 10)(2 20) (3 30)))
((1 2 3) (10 20 30))
```

```
(define (zip list1 list2)

  (if (null? list1)
      null
      (cons (list (car list1) (car list2))
              (zip (cdr list1) (cdr list2))))
-- or –
  (map list list1 list2)
-- or –
  (map (lambda (x y) (list x y)) list1 list2)
)
```

(b) Write a Scheme version of unzip that takes one argument that is a list of two-lists. It returns a list of two list, the first containing all of the first elements of the input tuples and the second with all of the second elements. You can do this with a recursive function or use map. *Hint: It is very, very easy if you use map. Consider that (map first '((a 1 x) (b 2 y)(c 3 z)) returns (a b c).*

```
(define (unzip list_of_2lists)

  (list (map first list_of_2lists)
        (map second list_of_2lists))

-- or –

(list (map car list_of_2lists)
        (map cadr list_of_2lists))


)
```

8. Explain this! (15: 10/5)

(a) Your classmate wrote the simple function shown to the right intending that it would take a list of numbers and return a tuple with two lists of numbers: the negative input numbers and the positive ones. She was surprised by what it did.

```
def split (in_list):
   neg = pos = [ ]
   for x in in_list:
      if x<0:
         neg.append(x)
      else:
         pos.append(x)
   return (neg, pos)
```

```
>>> split([ ])
([ ], [ ])
>>> split([1,2,3])
([1, 2, 3], [1, 2, 3])
>>> split([1, -2, 3, -4])
([1, -2, 3, -4], [1, -2, 3, -4]
```

(a) Explain why the function does not work as she intended it to.

Both *neg* and *pos* refer to the same object representing an empty list. Since append is a "destructive" operation (i.e., it modifies the list it is called with rather than returning a new one, any change to this list will effect both names.

(b) How can she fix the program?

Replace the line "neg = pos = [ ]" with the two lines

```
pos = []
neg = []
```

9. Little functions (50: 5*10)

For each of the following functions, describe what it does in a sentence. Assuming that my_list is [1,2,3,4] show what the function returns when called with my_list. We've done the first one as an example.

`def f0(ints):`
`   # ints is a list of integers`
`   n = 0`
`   for i in range(len(ints)):`
`      n += ints[i]`
`   return n`
	f0 returns the sum of the integers in the list ints. f0(my_list) returns 10.

`def f1(ints):`
`   # ints is a list of integers`
`   n = 0`
`   for i in range(1, len(ints)):`
`      if ints[i] > ints[n]:`
`         n = i`
`   return ints[n]`
	f1 returns the largest integer in the list ints. f1(my_list) returns 4.

`def f2(ints):`
`   # ints is a list of integers`
`   for i in range(1, len(ints)):`
`      if ints[i] < ints[i - 1]:`
`         return False`
`   return True`
	f2 returns True if the integers in ints are in ascending order and False otherwise. f2(my_list) returns True.

`def f3(ints):`
`   # ints is a list of integers`
`   flag = False`
`   for i in range(len(ints)):`
`      flag = flag or (ints[i] < 0)`
`   return flag`
	f3 returns True if any of the integers in ints is less than zero and False otherwise. f3(my_list) returns False.

`def f4(ints):`
`   # ints is a list of integer`
`   for i in range(1, len(ints)):`
`      ints[i] += ints[i - 1]`
`   return ints`
	f4 returns a list where the ith element of the list is equal to the sum of the elements of ints to its left. f4(my_list) returns [1,3,6,10]

`def f5(ints):`
`   # ints is a list of integers`
`   n = len(ints) - 1`
`   for i in range(len(ints) / 2):`
`      ints[i], ints[n-i] = ints[n-i], ints[i]`
`   return ints`
	f5 returns a list that is the reverse of its input. f5(my_list) returns [4,3,2,1].