CS 3100 – Models of Computation – Fall 2011 – Notes for L2

August 27, 2011

1 Preliminaries

Please get set up to learn (or refresh on) Unix; plenty of resources are available on the web (*e.g.*, `http://www.ee.surrey.ac.uk/Teaching/Unix/`). Then please get set up to use Python (available as `python3` on our CADE machines).

A very nice tutorial on Python (with plenty of good exercises) appears at `http://www.python-course.eu/index.php`. In fact, this is one page where I found a lot of useful material about Python3. Some additional details are at `http://www.python.org` and `http://docs.python.org/py3k/`.

2 A Tour of the Python Subset of Interest

We shall use a limited subset of Python initially, and we describe this subset in the order that best matches upcoming material.

2.1 Indentation

The indentation rules of Python are very important to understand. Read about recommended indentation styles (and coding-styles in general) at `http://www.python.org/doc/essays/styleguide.html`.

2.2 Characters

Characters are entered as follows against the Python prompt `>>>`:

```
>>> 'a'
'a'
>>> "a"
'a'
>>> """a"""
'a'
>>> "'"
"'"
```

```
>>> '"'
'"'
>>> ord('a')
97
>>> chr(97)
'a'
>>> chr(ord('a')+1)
'b'
```

2.3 Strings

Strings are entered as follows. Note how "raw" strings are entered and the convention pertaining to \, esp. at the end of a string.

```
>>> t='abcdef'

>>> t
'abcdef'

>>> t[::2]
'ace'

>>> t[::-1]
'fedcba'

>>> t='abcdef'

>>> t[2:]
'cdef'

>>> t[:2]
'ab'

>>> t[:2] + t[2:]
'abcdef'

>>> t[:]
'abcdef'

>>> t[:2] + 'z' + t[3:]
'abzdef'

>>> "aaa"
'aaa'

>>> "a\a"
'a\x07'

>>> "a\a"
'a\x07'

>>> "a"*5
"aaaaa"
```

```
>>> r"a\a"
'a\\a'

>>> s=r"a\a"
>>> s
'a\\a'

>>> s=="a\a"
False

>>> s==r"a\a"
True

>>> s[0]
'a'

>>> s[1]
'\\'

>>> s[2]
'a'

>>> s[0:]
'a\\a'

>>> s[1:]
'\\a'

>>> s[2:]
'a'

>>> t=r"a\b"

>>> t
'a\\b'

>>> t[::2]
'ab'
```

```
>>> for x in "abcd":
      print(x)
.. prints a, b, c, d

>>> len("abcd")
4

s= """hello, this is
... a multi-line sentence.
... """

>>> s
s
'hello, this is \na multi-line sentence.\n'
>>>
```

About raw strings ending in backslash, see

http://stackoverflow.com/questions/647769/..
why-cant-pythons-raw-string-literals-end-with-a-single-backslash

2.4 Lists

```
>>> lst = ['a', 'b']
```

```
>>> lst[0]
'a'
>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> n09 = [x for x in range(10)]
>>> n09
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> n09[0]=100
>>> n09
[100, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> n09 = [x for x in range(10)]
>>> n09
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> alias = n09
>>> alias
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> n09[0] = 100

>>> alias
[100, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> n09 = [x for x in range(10)]

>>> n09[2:]
[2, 3, 4, 5, 6, 7, 8, 9]
>>> acopy = n09[:]  # Make a deep copy

>>> acopy
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> n09[0] = 100
>>> acopy
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> f = lambda x: x+1

>>> f
f
<function <lambda> at 0x1005eb5a0>

>>> list(map(f, acopy))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> ["a"*i + "b"*j + "c"*k for i in range(3) for j in range(3)
        for k in range(3) if ((i != 1) or (j == k)) ]

['', 'c', 'cc', 'b', 'bc', 'bcc', 'bb', 'bbc', 'bbcc', 'a', 'abc', 'abbcc', 'aa',
   'aac', 'aacc', 'aab', 'aabc', 'aabcc', 'aabb', 'aabbc', 'aabbcc']

>>> f = lambda x, y: x+y

>>> acopy
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> functools.reduce(f, acopy)
functools.reduce(f, acopy)
45

>>> odd = lambda x: x%2 == 1

>>> odd(2)
False

>>> odd(3)
True

>>> list(filter(odd,acopy))
```

```
[1, 3, 5, 7, 9]

>>> [[1,2],3][0]
[1,2]

>>> [[1,2],3][0][1]
2

>>> [a,b] = [1,2]

>>> a
1

>>> b
2
```

2.5 Sequences

```
(1,2)
(1, 2)
>>> (1,2)[0]
1

>>> ((1,2),3)[0][0]
1

>>> ((1,2),3)[0][1]
2
```

```
>>> ((1,2),3)[1]
3

>>> (a,b) = (1,2)

>>> a
1

>>> b
2
```

```
>>> mixList = [ [], (1,2), (), "aa", ("aa", "bb", 2), 12, [1, (1,2)]]

>>> list(zip([1,2], [4,5]))
[(1, 4), (2, 5)]
```

2.6 Sets and Dicts (or Hash-maps)

Set representation involves hashing (as with Dicts), and so the members of a set must be "hashable." Thus unfortunately we can't have sets of sets or sets of lists, but we can have lists of sets or lists of lists. It is also good to disambiguate whether we are making sets or dicts—for instance, instead of just saying `{}`, say `set({})` or `dict({})`. We can also iterate over items of a set as well as build sets using comprehension. However, the moment we

```
>>> set({})
set()

>>> dict({})
{}

>>> {1,2,3,4,1,2,3}
{1, 2, 3, 4}

>>> {'a', (1,2), 'b', (1,2)}
```

```
{(1, 2), 'a', 'b'}

>>> (1,2) in {(1, 2), 'a', 'b'}
True

>>> d1 = {11 : 1122, 2234 : 33, 34343 : 'ee', "adf" : 222 }

>>> d1.keys()
dict_keys(['adf', 2234, 11, 34343])

>>> d1.values()
dict_values([222, 33, 1122, 'ee'])

>>> d1.items()
dict_items([('adf', 222), (2234, 33), (11, 1122), (34343, 'ee')])

>>> list(d1.items())
[('adf', 222), (2234, 33), (11, 1122), (34343, 'ee')]

>>> set(d1.items())
{('adf', 222), (11, 1122), (34343, 'ee'), (2234, 33)}

>>> d1['adf']
222

>>> d1.update({'aaaaa' : 11111, 'bbbbb' : 99999 })

>>> d1
d1
{'adf': 222, 34343: 'ee', 'aaaaa': 11111, 2234: 33, 11: 1122, 'bbbbb': 99999}
```

2.7 Variable Scoping, Function Definitions

Most of your work for this class will consist of definition short functions in a file and importing the files. You can place files in sub-directories also. To import everything from a file `foo` say `from foo import *`, and for importing from sub-directory `bar` of file `foo`, say `from foo.bar import.*`. After that, you can do `dir()` to list all the names in the scope (top-level global variables and function names, mainly) or `dir("foo")` to know the names in `foo`, or `dir(["foo", "bar"])` for all the names in `foo` and `bar`. After declaring the function name and arguments, one has to (as a matter of style) provide a documentation string, which automatically gets incorporated into the `help(..)` command. All these ideas are now illustrated.

```
# Try all these commands yourself, assuming that you have a file called lang.py and nfa.py
# If not, put in some functions yourself. Also import ply-3.4

from lang import *             # Import all names from lang.py
from nfa import *              # Similar to the above
dir("lang")                    # List names in lang.py
dir()                          # List of all names available to Python
dir(["lang", "nfa"])           # Names in these two modules
import ply.yacc                # Import from ply/yacc.py
import ply.yacc as yacc        # Alias the module as 'yacc' here, including for dir(yacc)

def cat(L1,L2):
    """Concat two languages
    cat(a,b) --> set(['abab', 'bc22', 'ab11', 'ab22', 'bcab', 'bc11']) where a=set(['ab', 'bc']) b=set(['11', 'ab', '22'])
    """
    return set([x+y for x in L1 for y in L2])
```

```
# Now you can type help(cat) and see the above documentation listed. Using triple-quotes allows the newlines to be preserved.
```

2.8 Reading and Writing from Keyboard, to Console, from/to Files

There are so many ways to print things, either to files or to the standard output. *Pretty much all methods of printing to the console also work with files.* Here are examples of dealing with files. The rest of the document deals mostly with console I/O.

Reading from a file: `fHandle = open(fileNameString)`, then `fHandle.read()`. You'll also find additional examples of console I/O as part of asg2.pdf which is already posted.

Writing to a file: `fHandle = open(fileNameString, 'w')`, then `print(stringToPrint, file=fHandle)`. You'll also find additional examples of console I/O as part of asg2.pdf which is already posted.

```
print(33) - prints 33 and puts a \n

print(33, end='') - prints 33 and ends the line with a '' i.e. does not put a \n
```

One could use `print` statements accompanied by format specifiers, or one could form strings to be printed and print them. Here is an example of a print with format involved. This example prints two items—a string and a number. The minimum field width is specified as 10 for the string and 9 for the number.

```
print('{0:10} --> {1:9d}'.format("history", 12))

This gives
history    -->        12
```

Another way to print things in a list separated by a given separator is shown below. This takes a list of things and converts each to a string, with *each* item separated by a tab. To that string, we attach two more tabs, and a tail string `tailstr`.

```
  print("separator".join(map(str, list_of_things)) + "\t\t" + tailstr)

For example,

print("~\b~".join(map(str, [{'a'}, 10, {'a':10}, {1:20}, []])) + " - " + " >>")

produces

  {'a'}~10~{'a': 10}~{1: 20}~[] -  >>
```

The cool thing is that the `format` modifier function applid to strings formats the string in a given way (much like Unix's `sprintf`). Here is an interesting example putting these ideas together:

```
fm = lambda x: '{0:3d}'.format(x)  # each item is the "zeroth" item, hence "0:"

list(map(fm, [1,2,30,400]))

produces

['  1', '  2', ' 30', '400']
```

2.9 Higher Order Formatter Functions

```
>>> num_in_width = lambda w: (lambda x: ('{0:' + str(w) + '}').format(x))

>>> num_in_width(4)
<function <lambda> at 0x1005f12f8>

>>> w4 = num_in_width(4)

>>> w5 = num_in_width(5)

>>> w3 = num_in_width(3)

>>> list(map(w3, [1, 20, 300, 4000]))

['  1', ' 20', '300', '4000']

>>> list(map(w4, [1, 20, 300, 4000]))

['   1', '  20', ' 300', '4000']

>>> list(map(w5, [1, 20, 300, 4000]))

['    1', '   20', '  300', ' 4000']
```

Now we can write a function to print Pascal's triangle by adjusting the print width depending upon the magnitude expected.

2.10 Illustration

Let us write a recursive program to generate the rows of Pascal's triangle. Then let us modify this program to print the rows of the Pascal's triangle.

Let's study an example Pascal's triange:

```
                              1

                          1       1

                      1       2       1

                  1       3       3       1

              1       4       6       4       1

          1       5      10      10       5       1

      1       6      15      20      15       6       1

  1       7      21      35      35      21       7       1
```

From this, we can think of an algorithm to derive the rows, starting from the first row which is just `1`

- Take the previous row and make two copies, one with a 0 to the left and one with a 0 to the right; for example `0 1` and `1 0`. Add the rows. You now obtain the next row!

- Do this N times.

Example of how this idea can be developed:

```
L = [1]

>>> La = [0]+L

>>> Lb = L + [0]

>>> zip(La, Lb)

>>> list(zip(La, Lb))
[(0, 1), (1, 0)]

>>> f = lambda x: x[0]+x[1]

>>> list(map(f, zip(La, Lb)))
[1, 1]
```

Based on this study, we can develop a "row to new row" function:

```
def row2nrow(row):
    return list(map(lambda x: x[0]+x[1], zip([0] + row, row + [0])))
```

Based on this, we can finally present the whole solution:

```
def pascal(N):
    return dopasc([[1]], N)

def dopasc(Ll, N):
    return Ll[::-1] if N==1 else dopasc( [ row2nrow(Ll[0]) ] + Ll, N-1)

def row2nrow(row):
    return list(map(lambda x: x[0]+x[1], zip([0] + row, row + [0])))
```

Let's see how things work:

```
>>> pascal(3)

[[1], [1, 1], [1, 2, 1]]

>>> pascal(5)

[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]
```

OK things seem to be working; but how to nicely format things so that we can obtain

```
def pascal(N):
    pascRows = dopasc([[1]], N)
    for line in list(map(lambda x: prRow(x, N), pascRows)):
        print(line)

def prRow(L, N):
    sp = 4 if N >= 5 else 3
```

```
    n = len(L)
    side_space = (" " * sp) * (N-n+1)
    mid_item = (" " * sp).join( list(map(num_in_w(sp), L)) )
    return "\n" + side_space + mid_item + side_space

def dopasc(Ll, N):
    return Ll[::-1] if N==1 else dopasc( [ row2nrow(Ll[0]) ] + Ll, N-1)

def row2nrow(row):
    return list(map(lambda x: x[0]+x[1], zip([0] + row, row + [0])))

def num_in_w(w):
    return (lambda x: ('{0:' + str(w) + '}').format(x))
```

3 Strings and Languages

With some functional programming under our belts, we can now embark on studying one of the first topics—namely, *strings* and *languages*. I'll try my best to introduce Python encodings of concepts in sf font.

3.1 Alphabet

An *alphabet*, Σ, is a finite set of *symbols*. In most cases, symbols will be characters. Alphabets are always non-empty. In Python, we will employ sets of strings, each of length 1 to denote alphabets.
Example: `Sigma = {'a', 'b', 'c'}`

3.2 Strings and Languages

Strings are a **finite** concatenation of symbols (a finite sequence of characters). Strings are read *left-to-right*. We write the empty string ε.

- Languages are sets of strings – (**finite**) strings for emphasis (for the last time!).
- **Languages can be infinite!** In fact, most languages that we study will be infinite.
- So remember:

  Almost all languages that we deal with are *infinite sets of finite strings.*

Two examples:

$Mylang = \{\varepsilon, aa, abc\}$

$La\_lt\_b = \{a^i b^j \mid i, j \geq 0, and\ i < j\}$

By saying $i, j > 0$ we will tacitly assume that i and j are natural numbers. Since for every natural number p there is a natural number $q > p$, we can easily conclude that La_lt_b is infinite.

Note that most languages discussed in Automata theory are infinitary. When we "simulate" these languages in Python, we will have to obtain finite approximations. This is a reasonable compromise in my opinion. The use of Python helps you understand *roughly* what is going on; for details, refer to the math!
In Python, we will employ sets of strings to denote languages. We will use `''` to denote ε.

```
Mylang = {'', 'aa', 'abc'} is a language over the above Sigma

La_lt_b_9 = { "a"*i + "b"*j for i in range(10) for j in range(10) if i < j }

>>> La_lt_b_9  # Approximated to <= 9 i's and j's

{'aaaaaabbbbbbbbb', 'abbbbb', 'aaabbbbbbbb', 'aabbbbb', 'aabbb',
'aaaabbbbbbbbb', 'bbbbbbbb', 'aaabbbbbbbbb', 'aaabbbbbbb', 'aaabbbbb',
'bbbbbbb', 'abb', 'aaaaaaaabbbbbbbbb', 'abbbbbb', 'aaaabbbbbb', 'bbb',
'aaaabbbbbbb', 'aabbbbbbb', 'aaaabbbbb', 'aaaaabbbbbbbbb',
'aabbbbbbbbb', 'abbbbbbbbb', 'aaaaaabbbbbbbb', 'abbb', 'bb',
'aaaaaabbbbbbb', 'abbbbbbb', 'aaaaaaabbbbbbbb', 'aabbbbbb',
'aaaaabbbbbbbb', 'aaaaabbbbbbb', 'bbbbbbbbb', 'bbbbb', 'aaabbbb', 'b',
'abbbb', 'abbbbbbbb', 'bbbbbb', 'bbbb', 'aaabbbbbb', 'aaaabbbbbbbb',
'aaaaabbbbbb', 'aabbbbbbbb', 'aaaaaaabbbbbbbbb', 'aabbbb'}
```

Strings and languages are *fundamental* to computer science theory. We will primarily be studying *patterns* described by strings. Many operations are defined on strings: (i) One can take substrings of a given string; (ii) Strings may also be concatenated or reversed. More operations on strings are defined later.

Drill problems:

- Write a one-line set comprehension to generate the set of all substrings of `s="abc"`. You can assume that `s="abc"` is a statement issued prior to your set comprehension.
- Is this true in Python for two strings p and q? Explain!

  `p+q == (q[::-1] + p[::-1])[::-1]`

3.3 Operations on Languages

We now list some of the basic operations on languages. Some of these are constant operations (meaning, they denote constant languages).

- *Empty Language (or Zero Language):* $\emptyset$ *or* $\{\}$
  We call it the "zero" language because it is like the 0-element for concatenation.

```
def Phi():
    """This is the ZERO language for concatenation (viewed as multiplication)
    """
    return set() # Don't write it as {} because this is ambiguous - dict/set
```

- *Unit Language:* $\{\varepsilon\}$
  We call it the "unit" language because it is like the unit element for concatenation.

```
def Unit():
    """This is the UNIT language for concatenation (viewed as multiplication)
    """
    return {""} # Set with epsilon
```

- *Concatenation:* $L_1L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$

```
def cat(L1,L2):
    """Concat two languages
    cat(a,b) --> set(['abab', 'bc22', 'ab11', 'ab22', 'bcab', 'bc11'])
    where a=set(['ab', 'bc']) b=set(['11', 'ab', '22'])
    """
    return set({x+y for x in L1 for y in L2})
```

- *Exponentiation:* $L^0 = \{\varepsilon\}$ *and* $L^n = LL^{n-1}$

```
def exp(L,n):
    """Exponentiate a language
    exp(a,2) --> set(['abab', 'bcab', 'bcbc', 'abbc']) where  a = set(['ab', 'bc'])
    """
    return Unit() if n == 0 else cat(L, exp(L, n-1))
```

- *Union:* $L_1 \cup L_2 = \{x \mid x \in L_1 \vee x \in L_2\}$

```
def lunion(L1,L2):
    """Language union
    """
    return L1 | L2
```

- *Star:* We introduce *Star* in stages.

  First let us introduce L_n^*: $L_n^* = L^n \cup L_{n-1}^*$, with $L_0^* = \{\varepsilon\}$

  i.e.,

  $L_n^* = L^n \cup L^{n-1} \cup \ldots \cup L^1 \cup L^0$ The following encodes L_n^*.

```
def star(L,n):
    """Star a language, bounding the iteration to the given n
    star(a,2) --> set(['abab', 'bcbc', 'ab', 'abbc', '', 'bc', 'bcab']),
    where  a = set(['ab', 'bc'])
    """
    return Unit() if n == 0 else exp(L,n) | star(L,n-1)
```

  Now $L^* = L_\infty^*$, *i.e.*, $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$

- *Reverse:* $rev(L) = \{rev(s) \mid s \in L\}$

```
def revl(L):
    """Reverse a language.
       revl(set(['ab', 'bc'])) --> set(['cb', 'ba'])
    """
    return set(map(lambda x: x[::-1], L))
```

- *Complementation:* Complementation of any set is with respect to a "universe" (or universal set). For language complementation, the universe is Σ^*. Now define the complementation of a language L with respect to that universe:

$$\overline{L} = \{x \mid x \in (\Sigma^* \setminus L)\}.$$

  Again, language complements can be (and usually are) infinitary. For "simulating it in Python," we need to bound complements:

```
def lcomplem(L,alph,m):
    """Complement L relative to alphabset alph. alph is also given as a set of strings.
       We subtract from the "star up to m" of the alphabet alph, the language L.
    """
    return star(alph,m) - L
```

- *Homomorphism on a string:* Given a string belonging to Σ^* (a "string over Σ^*"), a function h from domain Σ^* to range Γ^* is called a *homomorphism* if it respects two conditions:
  - $h(\varepsilon) = \varepsilon$
  - $h(xy) = h(x)h(y)$

```
def homos(S,f):
    """String homomorphism wrt lambda f
       homos("abcd",hm) --> 'bcde'  where hm = lambda x: chr( (ord(x)+1) % 256 )
    """
    return "".join(map(f,S))
```

- *Homomorphism on a language:* Given a homomorphism from Σ^* to range Γ^*, it can be applied to a language $L \subseteq \Sigma^*$ to produce a language $G \subseteq \Gamma^*$, and is defined in the obvious manner:
  $h(L) = \{h(x) \mid x \in L\}$

```
def homol(L,f):
    """Language homomorphism wrt lambda f.
       homos("Hello there", rot13) --> 'Uryy|-\x81ur\x7fr'
       where rot13 = lambda x: chr( (ord(x)+13) % 256 )
    """
    return set(map(lambda S: homos(S,f), L))
```

- *Intersection:* $L_1 \cap L_2 = \{x \mid x \in L_1 \wedge x \in L_2\}$

```
def lint(L1,L2):
    """Language intersection
    """
    return L1 & L2
```

- *Language Subtraction:* $L_1 \setminus L_2 = \{x \mid x \in L_1 \wedge x \notin L_2\}$

```
def lminus(L1,L2):
    """Language subtraction. Can do it as L1.difference(L2) also.
    """
    return L1 - L2
```

- *Symmetric difference:* $(L_1 \setminus L_2) \cup (L_2 \setminus L_1)$

```
def lsymdiff(L1,L2):
    """Language symmetric difference.
      lsymdiff(a,b) where a=set({'ab', 'bc'}) b=set({'11', 'ab', '22'}) --> set({'11', '22', 'bc'})
    """
    return L1 ^ L2
```

3.4 Lexicographic Orders

"Lexicon" means "dictionary" and so "lexicographic order" means "order as in a dictionary." Is `pig` before `poblano` in a dictionary? Yes, but Why? Answer: Note that we see `i` before `o` and so quit bothering the comparison of `g` and `b`. This is the basic idea behind lexicographic ordering.

Mathematically speaking, two strings s and t are in lexicographic order $s \leq t$ if $s[i] == t[i]$ holds up to a point, and we find $s[i+1] \leq t[i+1]$. We don't care what holds beyond $i+1$.
`lexlt2` for two strings `s` and `t` can be defined as follows:

```
def lexlt2(s, t):
    if (s==""):
        return True
    if (t==""):
```

```
        return False
    if (s[0] < t[0]):
        return True
    return (s[0] == t[0]) & lexlt2((s[1::], t[1::]))
```

We can also define `lexlt(p)` for a pair of strings `p` as follows (the indexing becomes a bit trickier; first you index to get strings out of a pair; then index the strings themselves):

```
def lexlt(p):
    if (p[0]==""):
        return True
    if (p[1]==""):
        return False
    if (p[0][0] < p[1][0]):
        return True
    return (p[0][0] == p[1][0]) & lexlt((p[0][1::], p[1][1::]))
```

Drill problem: Given the following languages

```
L1 = {"abacus", "bandana", "pig", "cat", "dodo", "zulu", "physics"}

L2 = {"dog", "zebra", "zzxyz", "pimento"}
```

Define a function that lists the set of pairs from L1 and L2 which are in the lexicographically less-than order.

3.5 Numeric Order

One problem with lexicographic ordering is the following. Suppose you are given $\Sigma = \{a, b\}$ and are asked to list all the strings in Σ^* in lexicographic order such that a string containing b is listed in a finite amount of time, what will your reaction be? The answer to this question motivates the notion of a numeric order.

According to the numeric order, one lists all strings of a length-group in lexicographic order before moving on to the next length group. We process all strings in length-group 0 (how many strings are in that group?), then move on to length-group 1, and so on.

Drill Problem: Here is a function that generates the Nth string in numeric order. Your bonus reward is to de-obfuscate this function! (It is not obfuscated although it may appear so; you just gotta know your binary numbers.)

```
from math import *

def nthnumeric(N):
    """Assume that Sigma is {a,b}. Produce the Nth string in numeric order, where N >= 0.
    Idea : Given N, get b = floor(log_2(N+1)) - need that many places; what to
    fill in the places is the binary code for N - (2^b - 1) with 0 as a and 1 as b.
    """
    if(N==0):
        return ''
    else:
        width = floor(log(N+1, 2))
        tofill = int(N - pow(2, width) + 1)
        relevant_binstr = bin(tofill)[2::] # strip the 0b leading string
        len_to_makeup = width - len(relevant_binstr)
        return "a"*len_to_makeup + homos(relevant_binstr, lambda x: 'b' if x=='1' else 'a')
```

3.6 Illustration of Homomorphisms

We now present two illustrations of homomorphisms (you are asked to verify that they are homomorphisms).

3.6.1 Changing labels so that 'dot' can print it

In printing DFA using `dot`, I had to change state names to conform to a legal `dot` syntax. Here is the mapping I devised, where by "dotsan" I mean "dot's sanity."

Drill Problem: Is `dot_san_str` a homomorphism? Argue formally!

```
def dotsan_map(x):
    """A homomorphism? Let the student answer!
    """
    if x in {  "{",   " ",   "'",   "}" }:
        return ""
    elif x == ",":
        return "_"
    else:
        return x

def dot_san_str(S):
    """Make dot like strings which are in set of states notation.
    """
    return homos(S, dotsan_map)
```

3.6.2 Parsing without first Lexing

In parsing a programming language such as C, the parser appeals to a *tokenizer* that first recognizes the structure of *tokens* (sometimes called *lexemes*) such as integers, floating-point numbers, variable declarations, keywords, and such. The tokenizer pattern-matches according to regular expressions, while the parser analyzes the structure of the token stream using a context free grammar. Suppose one wants to get rid of the tokenizer and write a context free grammar up to the level of individual characters. Such a grammar can be obtained by substituting the character stream corresponding to each token in place of each token in a *modular* fashion, according to a homomorphism. For example, if `begin` were to be a keyword in the language, instead of treating it as a token *keyword*, one would introduce additional productions of the form

```
begin -> b e g i n
```

Thanks to the *modular* nature of the substitutions, it can be shown that the resulting grammar would also be context-free. There is a formal result to this effect that CFLs remain CFLs under homomorphisms.

3.7 Prefix-closure

A language L is said to be prefix-closed if for every string $x \in L$, every prefix of x is also in L. If we are interested in *every* run of a machine, then its language will be prefix-closed. This is because a physically realizable string processing machine must encounter substrings of a string before it encounters the whole string. *Prefix closure* is an operation that 'closes' a set by putting in it all the prefixes of the strings in the set. For instance, given the set of strings $\{a, aab, abb\}$, its *prefix closure* is $\{\varepsilon, a, aa, ab, aab, abb\}$.

Drill Problem: We have already asked you to write a Python function to generate the prefix-closure of a language. Please refer to that exercise, and ... do it!