Computer Science 81
Homework 9
4/4/17

1, 2, 3, 4

1. [10 points] A **pairing function** is a function that maps any pair of natural numbers to a single natural number, such that the original numbers can be recovered from that number using separate recovery functions. Numerous pairing functions are known, but in this problem we will create a less known one, using techniques discussed above.
We'll use **pair** as the name of our pairing function, and **getx** and **gety** as the recovery functions. So we want, for all natural numbers x and y:

$$\mathbf{getx}(\mathbf{pair}(x, y)) = x$$
$$\mathbf{gety}(\mathbf{pair}(x, y)) = y$$

The specific pairing function of interest is based on an arbitrary n-ary representation of numbers, where $n > 1$. We'll use $n = 2$ for sake of discussion. To compute $\mathbf{pair}(x, y)$, first express x and y in binary. For example, in computing $\mathbf{pair}(6, 9)$, the binary representations would be 0110 and 1001. Next *interlace* those binary representations, starting with the least significant bits, so you don't have to worry about leading 0's. In this example, we would have 01 10 10 01 to show the interlacing phases as spaces, right-to-left. That is

```
0  1  1  0
 1  0  0  1 interlace
01 10 10 01
```

Then convert the interlaced representation, in this case 01101001, back to a number (105 decimal in this case), giving the value of the pairing function. Here is a table representing a few values of this function. The rows are x and the columns are y.

pair(x, y)	**0**	**1**	**2**	**3**	**4**	**5**
0	0	1	4	5	16	17
1	2	3	6	7	18	19
2	8	9	12	13	24	25
3	10	11	14	15	26	27
4	32	33	36	37	48	49
5	34	35	38	39	50	51

a. [5/10] Present Python code for the function **pair**.

b. [5/10] Present Python code for the companion recovery functions **getx** and **gety**.

You may exploit the code given above as an example for encoding and decoding.

■

2. [5 points extra credit] Do problem 1 using only numeric functions, no strings.

■

3. [20 points] In the lecture, we indicated how a Turing machine can be simulated using first-order logic. In this problem, we want to show how to simulate a Turing machine using strictly numeric functions (no strings). A key aspect of this simulation is using numbers to simulate tapes.

Suppose there are n tape symbols. Then we can encode any tape using n-ary encodings for the right and left parts of the tape. The left part of the tape will be encoded with the least-significant digit nearest the read/write head, so that the extension with blanks on the left appears as leading "zeroes". Similarly the right part of the tape will be encoded with the least-significant digit under the read/write, so that the extension with blanks on the right appears as leading "zeroes". In other words, when as a numeral, the right part would be in reverse, least-significant digit first.

a. [1/20] Construct in Python the function **first**(base, number) that returns the least-significant digit of number as a number. If the number argument is 0, then 0 should be returned. Example: $\textbf{first}(3, 25) = 1$, since $25 = 221_3$.

b. [1/20] Construct in Python the function **rest**(base, number) that returns the number represented by all but the least significant digit of number as a number. If the number argument is 0, then 0 should be returned. Example: $\textbf{rest}(3, 25) = 8(22_3)$.

c. [1/20] Construct in Python the function **cons**(base, least, number) that returns the number represented by putting least as the least significant digit of a new number with number as the rest of the digits. Example: $\textbf{cons}(3, 1, 8) = 25$.

When defined as indicated above, the functions **first**, **rest**, and **cons** automatically deal with the problem of extending the Turing machine tape with blanks as necessary, since blanks at the extremities of the tape and leading zeroes are the same thing.

3. cont. The transition rules for the Turing machine will be given as a list of 5-tuples, with the states, tape symbols, and moves encoded as numbers. For example, the triangle function, which uses tape symbols 0, 1, 2 with 0 as blank, would be encoded as follows (with motion 1 =left, 2 =right, 3 =none):

```
            # m(current-state, read, next-state, write, motion)
rules = [(1, 0, 0, 0, 3),
          (1, 1, 2, 2, 2),

          (2, 1, 2, 1, 2),
          (2, 0, 3, 0, 2),

          (3, 1, 3, 1, 2),
          (3, 0, 4, 1, 1),

          (4, 1, 4, 1, 1),
          (4, 0, 5, 0, 1),

          (5, 1, 5, 1, 1),
          (5, 2, 6, 2, 2),
          (5, 0, 0, 0, 3),

          (6, 1, 2, 2, 2),
          (6, 0, 7, 0, 1),

          (7, 2, 7, 1, 1),
          (7, 0, 8, 0, 2),

          (8, 1, 1, 0, 2)
         ]
```

Although we could further encode each 5-tuple as a single number, and even encode the entire list as a single number, we will not require it here.

d. [17/20] Construct the Python function **run(left, control, right, rules, base)** that simulates a Turing machine starting at the given state (left, control, right) using the indicated rules, with base as the number of tape symbols. The returned value should be the final state as a triple of the form (left, control, right), which occurs when there is no applicable rule.

Use encodeString to encode the initial tape, and decodeNumber to show the result as a string. However, it is required that you use only numeric functions to carry out the simulation apart from tracing, except that you can use list functions to search the rule set for a given rule. These numeric functions should be defined in Python. You may use **general recursion**, even though most functions, other than **run** itself, will be primitive recursive. The function run could be coded as an iteration, and this would necessarily be an indefinite iteration. Have the run function return a pair of numbers representing the left and right tapes. Then you can display the contents using decodeNumber.
Test your run function on the rule set above, using this call:

```
run(encodeString(3, ""), 1, encodeString(3, "111"), rules, 3)
```

$\blacksquare$

4. [5 points extra credit] Create a second version of the **run** function above wherein the **rules** argument of **run** is a single number, by encoding the list of rules into that number, then inside **run** decomposing the number using numeric functions. Here you would use a **pairing function** as in problem 1 to encode the list. Use a **modified** version of the pairing function in question 2 that does not map any pair to 0 (by adding 1 to the result of pairing and subtracting one before recovering). In this way, you can use 0 to indicate the encoding of the empty list.

∎