CS61A Final Review Problems

August 14, 2013

Interpreters
Reading

How many times is `scheme_read` and `read_tail` called for the following function calls?

```
(+ 1 2 3)
scheme_read: __      read_tail: __

(+ (- 2 1) 6 (* 3 5))
scheme_read: __      read_tail: __

(append '(1) (cons 2 '(3 . ())))
scheme_read: __      read_tail: __
```

Interpreters
Reading

How many times is `scheme_read` and `read_tail` called for the following function calls?

```
(+ 1 2 3)
scheme_read: 5       read_tail: 5

(+ (- 2 1) 6 (* 3 5))
scheme_read: 11      read_tail: 13

(append '(1) (cons 2 '(3 . ())))
scheme_read: 12      read_tail: 13
```

Interpreters
Evaluating

How many times is `scheme_eval` and `scheme_apply` called for the following function calls?

```
(+ 1 2 3)
scheme_eval: __      scheme_apply: __

(+ (- 2 1) 6 (* 3 5))
scheme_eval: __      scheme_apply: __

(append '(1) (cons 2 '(3 . ())))
scheme_eval: __      scheme_apply: __
```

Interpreters
Evaluating

How many times is `scheme_eval` and `scheme_apply` called for the following function calls?

```
(+ 1 2 3)
scheme_eval: 5       scheme_apply: 1

(+ (- 2 1) 6 (* 3 5))
scheme_eval: 11      scheme_apply: 3

(append '(1) (cons 2 '(3 . ())))
scheme_eval: 7       scheme_apply: 2
```

Streams

Define `make_list_cycle_stream`, which takes in a non-empty list, and returns a stream that continuously cycles through all the elements in the list.

```
def make_list_cycle_stream(lst):
    '***YOUR CODE HERE***'
```

Define `make_fib_stream`, which returns a stream containing the fibonacci sequence.

```
def make_fib_stream():
    '***YOUR CODE HERE***'
```

Streams

```
def make_list_cycle_stream(lst):
    def cycle_stream(n):
        def compute_rest():
            return cycle_stream(n + 1)
        return Stream(lst[n % len(lst)], compute_rest)
  return cycle_stream(0)
```

Streams

```
def make_fib_stream(lst):
    def fib_stream(first, second):
        def compute_rest():
            return fib_stream(second, first + second)
        return Stream(first, compute_rest)
  return fib_stream(0, 1)
```

Iterators and Generators

Define a generator that outputs the following:

```
>>> l = list_gen()
>>> next(l)
[[1]]
>>> next(l)
[[[2]], [1]]
>>> next(l)
[[[[3]]], [[2]], [1]]
```

First, you might want to make two helper functions. Define *nest_each_item* which takes in a list and makes each item in the list a nested item. This function can either be mutating or not mutating.

Iterators and Generators

Define a generator that outputs the following:

```
>>> l = list_gen()
>>> next(l)
  [[1]]
>>> next(l)
[[[2]], [1]]
>>> next(l)
[[[[3]]], [[2]], [1]]

def nest_each_item(lst):
  """
  >>> lst = [[1], [2], [3]]
  >>> nest_each_item(lst)
  >>> lst
  [[[1]], [[2]], [[3]]]
```

Iterators and Generators

Define a generator that outputs the following:

```
>>> l = list_gen()
>>> next(l)
  [[1]]
>>> next(l)
[[[2]], [1]]
>>> next(l)
[[[[3]]], [[2]], [1]]

def nest_each_item(lst):
  for index, item in enumerate(lst):
      lst[index] = [item]
```

Iterators and Generators

Define a generator that outputs the following:

```
>>> l = list_gen()
>>> next(l)
  [[1]]
>>> next(l)
[[[2]], [1]]
>>> next(l)
[[[[3]]], [[2]], [1]]
```

Now, write another helper called *increment_each_item* which takes in a list and increments the number that is nested within each of the lists. Remember that the number can be nested in any number of levels.

Iterators and Generators

Define a generator that outputs the following:

```
>>> l = list_gen()
>>> next(l)
   [[1]]
>>> next(l)
[[[2]], [1]]
>>> next(l)
[[[[3]]], [[2]], [1]]

def increment_each_item(lst):
   """
   >>> lst = [[1], [[2]], [[[3]]]]
   >>> increment_each_item(lst)
   >>> lst
   [[2], [[3]], [[[4]]]]
```

Iterators and Generators

Define a generator that outputs the following:

```
>>> l = list_gen()
>>> next(l)
   [[1]]
>>> next(l)
[[[2]], [1]]
>>> next(l)
[[[[3]]], [[2]], [1]]

def increment_each_item(lst):
   for orig_item in lst:
        item = orig_item
        while isinstance(item[0], list):
               item = item[0]
        item[0] += 1
```

Iterators and Generators

Define a generator that outputs the following:

```
>>> l = list_gen()
>>> next(l)
   [[1]]
>>> next(l)
[[[2]], [1]]
>>> next(l)
[[[[3]]], [[2]], [1]]
```

Finally, put those together to make the generator that outputs the above output. The idea is to take the existing list and nest each of the items, then add one to all of the items, then append a [1] onto the end of the list.

Iterators and Generators

Define a generator that outputs the following:

```
>>> l = list_gen()
>>> next(l)
   [[1]]
>>> next(l)
[[[2]], [1]]
>>> next(l)
[[[[3]]], [[2]], [1]]

def list_gen():
```

Iterators and Generators

Define a generator that outputs the following:

```
>>> l = list_gen()
>>> next(l)
   [[1]]
>>> next(l)
[[[2]], [1]]
>>> next(l)
[[[[3]]], [[2]], [1]]

def list_gen():
   lst = [[1]]
   while True:
        yield lst
        nest_each_item(lst)
        increment_each_item(lst)
```

Logic
What Would Logic Print?

```
(fact (> 5 3))
(fact (> 14 5))
(fact (> 21 14))
(fact (> 43 21))
(fact (> inf ?num))

(fact (gt? ?a ?b)
      (> ?a ?b))
(fact (gt? ?a ?b)
      (> ?a ?c)
      (> ?c ?b))

(fact (lt? ?a ?b)
      (gt? ?b ?a))
```

```
logic> (query (> 5 3))
```

Logic
What Would Logic Print?

```
(fact (> 5 3))
(fact (> 14 5))
(fact (> 21 14))
(fact (> 43 21))
(fact (> inf ?num))

(fact (gt? ?a ?b)
      (> ?a ?b))
(fact (gt? ?a ?b)
      (> ?a ?c)
      (> ?c ?b))

(fact (lt? ?a ?b)
      (gt? ?b ?a))
```

```
logic> (query (> 5 3))
Success!
```

Logic
What Would Logic Print?

```
(fact (> 5 3))
(fact (> 14 5))
(fact (> 21 14))
(fact (> 43 21))
(fact (> inf ?num))

(fact (gt? ?a ?b)
      (> ?a ?b))
(fact (gt? ?a ?b)
      (> ?a ?c)
      (> ?c ?b))

(fact (lt? ?a ?b)
      (gt? ?b ?a))
```

```
logic> (query (gt? ?num 3))
```

Logic
What Would Logic Print?

```
(fact (> 5 3))
(fact (> 14 5))
(fact (> 21 14))
(fact (> 43 21))
(fact (> inf ?num))

(fact (gt? ?a ?b)
      (> ?a ?b))
(fact (gt? ?a ?b)
      (> ?a ?c)
      (> ?c ?b))

(fact (lt? ?a ?b)
      (gt? ?b ?a))
```

```
logic> (query (gt? ?num 3))
Success!
num: 5
num: 14
num: inf
```

Logic
What Would Logic Print?

```
(fact (> 5 3))
(fact (> 14 5))
(fact (> 21 14))
(fact (> 43 21))
(fact (> inf ?num))

(fact (gt? ?a ?b)
      (> ?a ?b))
(fact (gt? ?a ?b)
      (> ?a ?c)
      (> ?c ?b))

(fact (lt? ?a ?b)
      (gt? ?b ?a))
```

```
logic> (query (lt? ?num
   43))
```

Logic
What Would Logic Print?

```
(fact (> 5 3))
(fact (> 14 5))
(fact (> 21 14))
(fact (> 43 21))
(fact (> inf ?num))

(fact (gt? ?a ?b)
      (> ?a ?b))
(fact (gt? ?a ?b)
      (> ?a ?c)
      (> ?c ?b))

(fact (lt? ?a ?b)
      (gt? ?b ?a))
```

```
logic> (query (lt? ?num
   43))
Success!
num: 21
num: 14
```

Logic
Coding Practice

```
logic> (fact (even? ...)
logic> (query (even (1 1 1 1 1)))
Failed.
logic> (query (even (1 1 1 1))
Success!

logic> (fact (interleave ...)
logic> (query (interleave (1 3) (2 4 6 8) ?what))
Success!
what: (1 2 3 4 6 8)
```

Logic
Coding Practice

```
(fact (even? ()))
(fact (even? (1 1 . ?rest))
      (even? ?rest))


(fact (interleave () ?b ?b))
(fact (interleave (?first . ?rest) ?b (?first . ?result))
      (interleave ?b ?rest ?result))
```

Parallelism
Parallel Threads

Assume that initially, `x = 10`. The following two lines are then executed in parallel:

Thread 1	Thread 2
`x = x + x`	`x = x * x`

What are all the possible values of `x` after both threads are finished being executed?

What are the correct values of `x`?

Parallelism
Parallel Threads

Assume that initially, `x = 10`. The following two lines are then executed in parallel:

Thread 1	Thread 2
`x = x + x`	`x = x * x`

What are all the possible values of `x` after both threads are finished being executed?

20, 100, 110, 200, 400

What are the correct values of `x`?

200, 400

Parallelism
Locks

Assume the following lines are executed before entering separate threads:

```
x, y = 5, 3
xlock, ylock = Lock(), Lock()
```

Thread 1	Thread 2
`ylock.acquire()`	`xlock.acquire()`
`y = y - 1`	`x = x * y`
`ylock.release()`	`xlock.release()`
`xlock.acquire()`	`ylock.acquire()`
`x = x + y`	`y = y * 2`
`xlock.release()`	`ylock.release()`

What are all the possible paired values of `x` and `y` after both threads are finished being executed?

Parallelism
Locks

Assume the following lines are executed before entering separate threads:

```
x, y = 5, 3
xlock, ylock = Lock(), Lock()
```

Thread 1	Thread 2
`ylock.acquire()`	`xlock.acquire()`
`y = y - 1`	`x = x * y`
`ylock.release()`	`xlock.release()`
`xlock.acquire()`	`ylock.acquire()`
`x = x + y`	`y = y * 2`
`xlock.release()`	`ylock.release()`

What are all the possible paired values of `x` and `y` after both threads are finished being executed?

~~(x, y): (19, 8), (24, 8), (36, 8), (24, 9)~~

MapReduce

Write the mapper and reducer to solve the following problem: Given a file of input, you want to count the number of times that a word appeared on a line with x number of words.

Example:

the quick brown fox jumped over the lazy river
the other fox went to the mall
the river was brown

WORD [(number of words in that line, number of times it appeared)]

the [(9, 2), (7, 2), (4, 1)]
quick [(9, 1)]
brown [(9, 1), (4, 1)]

MapReduce

Example:
the quick brown fox jumped over the lazy river
the other fox went to the mall
the river was brown

WORD [(number of words in that line, number of times it appeared)]

the [(9, 2), (7, 1), (4, 1)]
quick [(9, 1)]
brown [(9, 1), (4, 1)]

```
def mapper(line):
   "***YOUR CODE HERE***"
```

MapReduce

the [(9, 2), (7, 1), (4, 1)]
quick [(9, 1)]
brown [(9, 1), (4, 1)]

```
from mapreduce import emit
def mapper(line):
   "***YOUR CODE HERE***"
   word_lst= line.split()
   for word in word_lst:
       emit(word, (len(word_lst), 1))

for line in sys.stdin:
   mapper(line)
```

MapReduce

the [(9, 2), (7, 1), (4, 1)]
quick [(9, 1)]
brown [(9, 1), (4, 1)]

```
def reducer(input):
   "***YOUR CODE HERE***"
```

MapReduce

the [(9, 2), (7, 1), (4, 1)]
quick [(9, 1)]
brown [(9, 1), (4, 1)]

```
from mapreduce import emit, group_values_by_key
def reducer(input):
   for key, value_iterator in group_values_by_key(input):
       items = {}
       for length, count in value_iterator:
           if length not in items:
               items[length] = 0
           items[length] += count
       emit(key, [(key, value) for key, value in
   items.items()])
reducer(sys.stdin)
```