STREAMS, ITERATORS, AND BINARY TREES 10

COMPUTER SCIENCE 61A

April 10, 2017

1 Streams

A *stream* is a lazily-evaluated linked list. A stream's elements (except for the first element) are only computed when those values are needed.

```
class Stream:
    class empty:
        """An empty stream"""
    empty = empty()

    def __init__(self, first, compute_rest=empty):
        self.first = first
        if compute_rest is Stream.empty or isinstance(
           compute_rest, Stream):
            self._rest, self._compute_rest = compute_rest, None
        else:
            assert callable(compute_rest)
            self._compute_rest = compute_rest

    @property
    def rest(self):
        """Return the rest, computing it if necessary."""
        if self._compute_rest is not None:
            self._rest = self._compute_rest()
            self._compute_rest = None
        return self._rest
```

A `Stream` instance is similar to a `Link` instance. Both have `first` and `rest` attributes. The rest of a `Link` is either a `Link` or `Link.empty`. Likewise, the rest of a `Stream` is either a `Stream` or `Stream.empty`.

However, instead of specifying all of the elements in `__init__`, we provide a function, `compute_rest`, that will be called to compute the remaining elements of the stream. Remember that the code in the function body is not evaluated until it is called, which lets us implement the desired evaluation behavior.

This implementation of streams also uses *memoization*. The first time a program asks a `Stream` for its `rest` field, the `Stream` code computes the required value using `compute_rest`, saves the resulting value, and then returns it. After that, every time the `rest` field is referenced, the stored value is simply returned.

Here is an example:

```
def make_integer_stream(first=1):
    def compute_rest():
        return make_integer_stream(first+1)
    return Stream(first, compute_rest)
```

Here, we start out with a stream whose first element is 1, and whose `compute_rest` function creates another stream. So when we do compute the `rest`, we get another stream whose first element is one greater than the previous element, and whose `compute_rest` creates another stream. Hence, we effectively get an infinite stream of integers, computed one at a time. This is almost like an infinite recursion, but one which can be viewed one step at a time, and so does not crash.

1.1 Higher Order Functions

Stream processing functions can be higher-order, abstracting a general computational process over streams. Take a look at `filter_stream`:

```
def filter_stream(filter_func, s):
    def compute_rest():
        return filter_stream(filter_func, s.rest)
    if s is Stream.empty:
        return s
    elif filter_func(s.first):
        return Stream(s.first, compute_rest)
    else:
        return compute_rest()
```

The Stream we create has as its `compute_rest` a function that "promises" to filter the rest of the Stream when called. So at any one point, the entire stream has not been filtered. Instead, only the part that has been referenced has been filtered.

1.2 Questions

1. In a similar model to `filter_stream`, let's recreate the function `map_stream` from lecture, that given a stream `s` and a one-argument function `func`, returns a new stream that is the result of applying `func` on every element in `s`.

```
def map_stream(func, s):
```

2. Write a function `every_other`, which takes in an infinite stream and returns a stream containing its even indexed elements.

```
def every_other(s):
```

3. Suppose one wants to define a random infinite stream of numbers via the recursive definition: "a random infinite stream consists of a first random number, followed by a remaining random infinite stream." Consider an attempt to implement this via the code. Are there any problems with this? How can we fix this?

```
from random import random
random_stream = Stream(random(), lambda: random_stream)
```

4. What does the following Stream output? Try writing out the first few values of the stream to see the pattern.

```
def my_stream():
    def compute_rest():
        return add_streams(map_stream(double, my_stream()),
                           my_stream())
    return Stream(1, compute_rest)
```

5. Write a function fib_stream that creates an infinite stream of Fibonacci Numbers, using the add_streams function that was introduced in lab.

```
def fib_stream():
```

6. Write a function seventh that creates an infinite stream of the decimal expansion of dividing n by 7.

```
def seventh(n):
    """The decimal expansion of n divided by 7.

    >>> first_k(seventh(1), 10)
    [1, 4, 2, 8, 5, 7, 1, 4, 2, 8]
    """
```

2 Iterator Review

We have already seen something very similar to the lazy evaluation of Streams in another topic covered: iterators! When an iterator is created, no elements are actually calculated. It is not until a `next` call triggers the evaluation of the next element in the sequence. This property allows us to work with infinite iterators, just like the infinite streams we've worked on in this discussion. Recall the naturals generator function:

```
def naturals():
    i = 1
    while True:
        yield i
        i += 1
```

One way to think about iterators is that they are to lists as streams are to linked lists. They are lazy versions of each data structure.

2.1 Questions

1. Implement a generator function called `filter(iterable, fn)` that only yields elements of iterable for which fn returns True.

```
def filter(iterable, fn):
    """
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter(range(5), is_even))
    [0 , 2 , 4]
    >>> all_odd = (2*y-1 for y in range (5))
    >>> list(filter(all_odd, is_even))
    []
    >>> s = filter(naturals(), is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
```

2. Implement an iterator class called Filter. The `__init__` method for Filter takes an iterable and a one-argument function that either returns True or False. The Filter iterator represents a sequence that only contains elements of the iterable for which the predicate function returns True. Do not use a generator in your solution.

```
class Filter :
    """
    >>> is_even = lambda x: x % 2 == 0
    >>> for elem in Filter(range(5) , is_even):
    ...     print(elem)
    0
    2
    4
    >>> all_odd = (2*y-1 for y in range (5))
    >>> for elem in Filter(all_odd, is_even):
    ...     print(elem) # No elements are even !
    >>> s = Filter(naturals(), is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
    def __init__(self, iterable, fn):




    def __iter__( self ):



    def __next__( self ):
```

3 Binary Trees

A Binary Search Tree (BST) is a special kind of tree that satisfies the following properties:

- Every node of a BST has at most two children called `left` and `right`. The children are also BST's.
- For every node, all labels in the left child are less than or equal to the parent's label.
- For every node, all labels in the right child are greater than or equal to the parent's label.

```
# Binary Search Tree (BST) Class
class BinTree:
    empty = ()
    def __init__(self, label, left=empty, right=empty):
        self.label = label
        self.left = left
        self.right = right
```

Note that there is an empty tree (`BinTree.empty`) defined. If a node does not have a left or right child, that attribute will be an empty tree.

3.1 Questions

1. Given that the input `t` is a non-empty binary search tree, write `tree_max`. `tree_max` must run in $O(H)$, where H is the height of the tree.

```
def tree_max(t):
    """Returns the max label in a binary search tree t."""
```

2. Define a function `insert` that takes in a `BinTree`, `bst`, and a number, `n`, and returns a new `BinTree` that has is a copy of `bst` with a new node inserted. `insert` should place the new node as a leaf in the correct position. If `t` is the `BinTree` on the left, then calling `insert(t, 3)` will return the `BinTree` on the right.

```
          4                                  4
        /   \                              /   \
       2     5             ->             2     5
      /                                 /   \
     1                                 1     3
```

```
def insert(bst, n):
    """
    >>> bst = BinTree(4, BinTree(2, BinTree(1)), BinTree(5))
    >>> new_bst = insert(bst, 3)
    >>> new_bst.left.right.label
    3
    """
```