TAIL RECURSION AND STREAMS

COMPUTER SCIENCE MENTORS 61A

April 10 to April 14, 2017

1 Tail Recursion

1. What is a tail context/tail call? What is a tail recursive function?

Solution: A tail call is a call expression in a tail context. A tail context is usually the final action of a procedure/function.

A tail recursive function is where all the recursive calls of the function are in tail contexts.

An ordinary recursive function is like building up a long chain of domino pieces, then knocking down the last one. A tail recursive function is like putting a domino piece up, knocking it down, putting a domino piece up again, knocking it down again, and so on. This metaphor helps explain why tail calls can be done in constant space, whereas ordinary recursive calls need space linear to the number of frames (in the metaphor, domino pieces are equivalent to frames).

2. Why are tail calls useful for recursive functions?

Solution: When a function is tail recursive, it can effectively discard all the past recursive frames and only keep the current frame in memory. This means we can use a constant amount of memory with recursion, and that we can deal with an unbounded number of tail calls with our Scheme interpreter.

Answer the following questions with respect to the following function:

```
(define (sum-list lst)
  (if (null? lst)
    0
    (+ (car lst) (sum-list (cdr lst)))
  )
)
```

3. Why is sum-list not a tail call? Optional: draw out the environment diagram of this sum-list with list: (1 2 3). When do you add 2 and 3?

   Solution: Sum list is not the last call we make, its actually the other addition which we do after we evaluate sum-list. Sum list is not the last expression we evaluate.

4. Rewrite sum-list in a tail recursive context.

   Solution:
   ```
   (define (sum-list-tail lst)
     (define (sum-list-helper lst sofar)
       (if (null? lst)
         sofar
         (sum-list-helper (cdr lst) (+ sofar (car lst)))
       )
     )
     (sum-list-helper lst 0)
   )
   ```

2 Streams

A `Stream` is a linked list where the first element is calculated, but the `rest` isnt until it is needed. Here is the definiton of `Stream`:

```
class Stream:
    class empty:
        def __repr__(self):
            return 'Stream.empty'

    empty = empty()

    def __init__(self, first, compute_rest=lambda: Stream.empty):
       assert callable(compute_rest), 'compute_rest must  be
          callable.'
       self.first = first
```

```
        self._compute_rest = compute_rest

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

    def __repr__(self):
        return 'Stream({0}, <...>)'.format(repr(self.first))

empty_stream = Stream.empty
```

Here is an example of how to construct a `Stream` of integers:

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

2.1 General Streams

1. Whats the advantage of using a stream over a linked list?

   Solution: Lazy evaluation. We only evaluate up to what we need.

2. Whats the maximum size of a stream?

   Solution: Infinity

3. Whats stored in first and rest? What are their types?

   Solution: First is a value, rest is another stream (either a method to calculate it, or an already calculated stream).

4. When is the next element actually calculated?

   Solution: Only when it's requested (and hasn't already been calculated)

2.2 What Would Python Print?

1. For each of the following lines of code, write what Python would output.

```
>>> a = make_integer_stream()
>>> a
```

Solution: `Stream(1, <...>)`

```
>>> a.first
```

Solution: `1`

```
>>> a.rest
```

Solution:
```
computing rest
Stream(2, <...>)
```

```
>>> a.rest
```

Solution: `Stream(2, <...>)`

```
>>> a.rest.rest.rest
```

Solution:
```
computing rest
computing rest
Stream(4, <...>)
```

```
>>> a.rest.rest
```

Solution: `Stream(3, <...>)`

```
>>> a.rest.rest.rest.rest.first
```

Solution:
```
computing rest
5
```

2.3 Code Writing for Streams

1. Write out `double_naturals`, which is a stream that evaluates to the sequence 1, 1, 2, 2, 3, 3, etc.

```
def double_natural(first=1, go_next=False):
  """
    >>> a = double_natural()
  >>> a.first
  1
  >>> a.rest.rest.first
  computing rest
  computing rest
  2
  """
  def compute_rest():
    print('computing rest')
    #Your code here
    return Stream(first, compute_rest)
```

Solution:

```
if go_next:
  return double_natural(first+1, go_next=False)
return double_natural(first, go_next=True)
```

2. Write out `interleave`, which returns a stream that alternates between the values in `stream1` and `stream2`. Assume that the streams are infinitely long.

```
def interleave(stream1, stream2):
  """
  Note: ignore "compute rest" prints from make_integer_stream.
  >>> s1, s2 = make_integer_stream(1), make_integer_stream(10)
  >>> mixed = interleave(s1, s2)
  >>> mixed.first
  1
  >>> mixed.rest.first
  10
  >>> very_mixed = interleave(mixed, mixed)
  >>> very_mixed.first
  1
  >>> very_mixed.rest.first
  1
  >>> very_mixed.rest.rest.first
  10
  >>> very_mixed.rest.rest.rest.first
  10
  """
```

Solution:

```
  def compute_rest():
    return interleave(stream2, stream1.rest)
  return Stream(stream1.first, compute_rest)
```