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

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

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

6. Whats the maximum size of a stream?

   Solution: Infinity

7. 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). In the case of Scheme, this is called a promise.

8. When is the next element actually calculated?

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

3 What Would Scheme Print?

9. For each of the following lines of code, write what scheme would output.

```
scm> (define x 1)
```

Solution: `x`

```
scm> (if 2 3 4)
```

Solution: `3`

```
scm> (delay (+ x 1))
```

Solution:
`#[promise]`

```
scm> (define (foo x) (+ x 10))
```

Solution: `foo`

```
scm> (define bar (cons-stream (foo 1) (cons-stream (foo 2)
   bar)))
```

Solution: `bar`

```
scm> (car bar)
```

Solution: `11`

```
scm> (cdr bar)
```

Solution:
`#[promise]`

```
scm> (define (foo x) (+ x 1))
```

Solution: `foo`

```
scm> (cdr-stream bar)
```

Solution:
`(3 . #[promise])`

```
scm> (define (foo x) (+ x 5))
```

Solution: `foo`

```
scm> (car bar)
```

Solution: `11`

```
scm> (cdr-stream bar)
```

Solution:
`(3 . #[promise])`

4 Code Writing for Streams

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

```
(define (double_naturals)
    (double_naturals_helper 1 0)
)

(define (double_naturals_helper first flag)









)
```

Solution:

```
(define (double_naturals_helper first flag)
    (if (= 1 flag)
        (cons-stream first (double_naturals_helper (+ 1
           first) 0))
        (cons-stream first (double_naturals_helper first
           1))
    )
)

;Alternative Solutions
(define (double_naturals_helper first flag)
    (cons-stream first (double_naturals_helper (+ flag
       first) (- 1 flag)))
)
```

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

```
(define (interleave stream1 stream2)



)
```

Solution:

```
(define (interleave stream1 stream2)
(cons-stream (car stream1)
 (interleave stream2 (cdr-stream stream1)))

(cons-stream (car stream1)
 (cons-stream (car stream2)
  (interleave (cdr-stream stream1)
  (cdr-stream stream2))))
)
```