Lecture 23 : Streams

Kaleel Mahmood

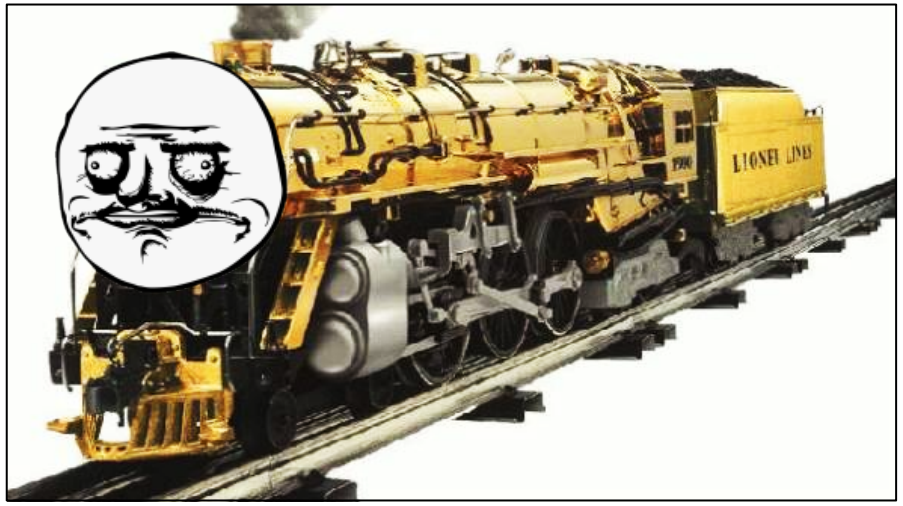
Department of Computer Science and Engineering

University of Connecticut

Today's Problem: The Infinite Treasure Train

- You are given a train with infinite carriages attached.
- Each carriage is attached sequentially and contains treasure inside.
- Whenever you need money, you can simply open a non-empty carriage and sell off the treasure inside the carriage.

What is the catch?



- You need to keep track of the carriage number every time you remove a treasure (so you know that the carriage is empty and so that you don't revisit it again).
- The carriage numbering is NOT straight forward (some complicated function).

2 ➡ -1/2 ➡ 1.25 ➡ -0.875

Also the train is cursed.

If you open a carriage that is empty (meaning you already looked for the treasure in this carriage once before):

The train will come to life and run you over. Uh oh.

Question: How can you keep track of the indexing of the carriages in the train?

This question becomes a matter of which data structure to implement:

Possible answers:

1. Use a stack and push all the carriage numbers onto the stack. Then use pop to remove elements from the stack.
2. Use a queue and enqueue all the carriage numbers onto the queue. Use dequeue to remove elements from the queue.
3. Give up and let the train run you over.

Question: How can you keep track of the indexing of the carriages in the train?

This question becomes a matter of which data structure to implement:

Possible answers:

1. ~~Use a stack and push all the carriage numbers onto the stack. Then use pop to remove elements from the stack.~~
2. ~~Use a queue and enqueue all the carriage numbers onto the queue. Use dequeue to remove elements from the queue.~~
3. ~~Give up and let the train run you over.~~

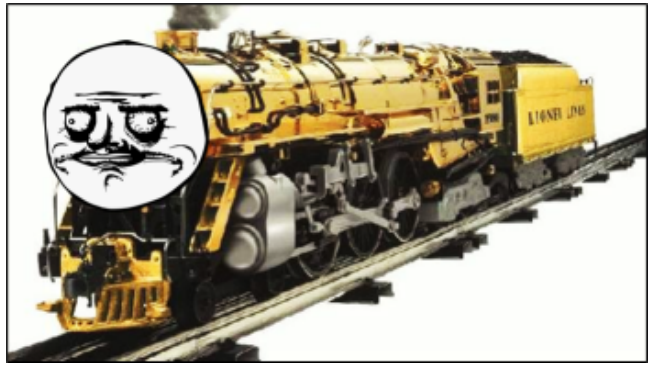
None of the above answers will work. *Why?*
=Computers have finite memory and you cannot store an infinite number of objects in a queue or stack.
=So we need a new data structure. The stream!

STREAMS

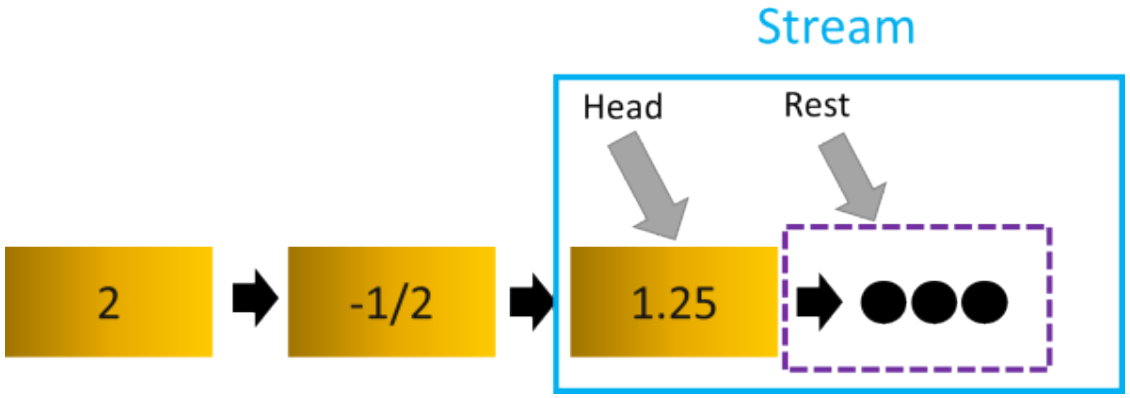
- A stream is an abstract data type that supports: empty?, head, and rest.
- You may imagine a stream as representing a sequence of elements so that:
  - empty? determines if the sequence is empty,
  - head returns the next element of the stream (leaving it at the front of the stream),
  - rest returns the remainder of the stream (after removing the first element).

- What's the point making this an abstract data type?
  - Are there other implementations that might be valuable?

What would a stream look like in the cursed treasure train problem?



What is the advantage of the stream?



- Stacks and queues can store a finite amount of arbitrary elements based on the amount of RAM allocated to them.
- What if you didn't need to store arbitrary elements but need to keep track of where you were in an infinite sequence?
- You can't store the entire sequence because it is infinite, but a stream is a good way to keep track of where you are in the sequence and also allows you to progress further when needed.

An Example Of Streams in Practice

- The Hailstone function:

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \text{ even.} \\ 3n+1 & \text{if } n \text{ odd.} \end{cases}$$

- **Collatz conjecture**. Start with any positive integer x. Consider the sequence:

$$x, f(x), f(f(x)), \ldots$$

- Eventually it reaches 1.

An Example Of Streams in Practice

- Recall our "objectified" version of the hailstone sequence:
  - this yields a infinite "stream" of numbers;
  - *you can't do that with a list.*
- Can we perfect it, so that it really adheres to the stream notion?

```
(define (hailstone x)
  (define (next)
    (if (eq? (modulo x 2) 0)
        (begin (set! x (/ x 2)) x)
        (begin (set! x (+ (* 3 x) 1)) x)))
  next)
```

TAKE 1: WELL, WE CAN ADAPT THIS SO THAT IT PROVIDES HEAD & ADVANCE

- We can adapt to return the head and (destructively) advance,
- But this doesn't yield the stream semantics:
  - in a stream, rest should return a new stream object!

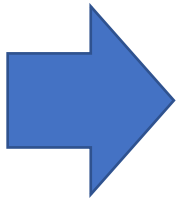
```
(define (hailstone x)
  (define (advance)
    (if (eq? (modulo x 2) 0)
        (set! x (/ x 2))
        (set! x (+ (* 3 x) 1))))
  (define (head) x)
  (define (dispatcher method)
    (cond ((eq? method 'advance) advance)
          ((eq? method 'head) head)))
  dispatcher)
```

Essentially we are still missing the "rest" function that returns a new stream object

Hailstone Sequence Take 2

In a stream the rest method should return a new stream object:

```
(define (hailstone x)
  (define (rest)
    (if (eq? (modulo x 2) 0)
        (hailstone (/ x 2))
        (hailstone (+ (* 3 x) 1)))))
```



```
(define (hailstone x)
  (define (rest)
    (if (eq? (modulo x 2) 0)
        (hailstone (/ x 2))
        (hailstone (+ (* 3 x) 1))))
  (define (head) x)
  (define (dispatcher method)
    (cond ((eq? method 'rest) rest)
          ((eq? method 'head) head)))
  dispatcher)
```

THIS STREAM OBJECT, IN ACTION

```
(define h-stream (hailstone 100))
((h-stream 'head))
> 100
```

```
(define rest-of-stream ((h-stream 'rest)))
((rest-of-stream 'head))
> 50
```

```
(define and-more-of-it ((rest-of-stream 'rest)))
((and-more-of-it 'head))
> 25
```

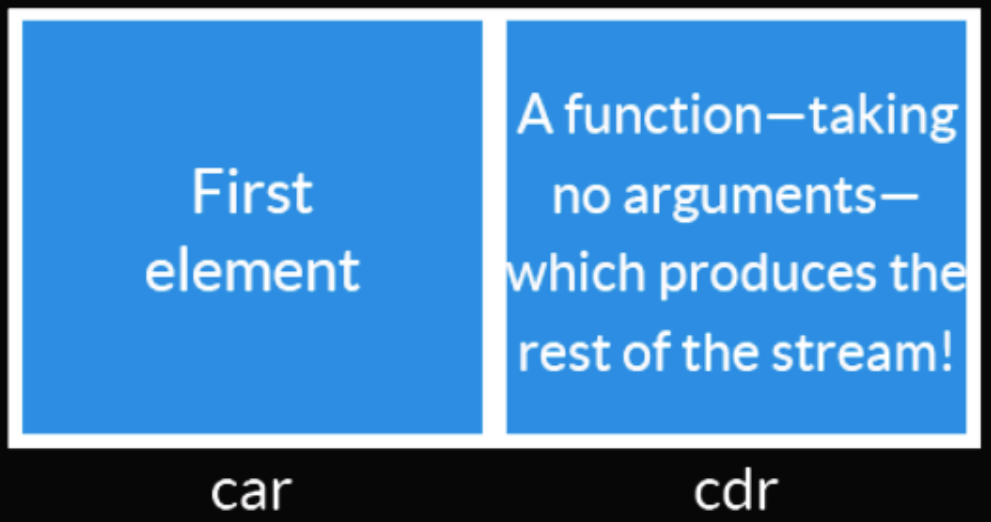
STREAMS BY EXPLICIT DELAYED EVALUATION

- We can strip away some of the complexity here, and recapture this idea in a more abbreviated and general fashion.
- The idea: We will implement a stream as either the empty list (for the empty steam), or a *pair*,
  - The car of the pair is the head of the stream.
  - The cdr of the pair is a function which, when called, returns the pair representing the rest of the stream.
- Then...

```
(define (head s) (car s))
(define (rest s) ((cdr s)))
```

THEN A STREAM IS A SINGLE PAIR...

- So a stream is the empty list...or...a pair:



Warning: Scheme is about to get strange…

THE STREAM OF NATURALS 0, 1, 2, 3, ...

- We can naturally define the stream of integers starting at k:

```
(define (integer-stream x)
  (cons x
        (lambda () (integer-stream (+ 1 x)))))
```

- Then:

```
(head (integer-stream 0))
> 0
(head (rest (integer-stream 0)))
> 1
(head (rest (rest (integer-stream 0))))
> 2
```

WHY DOES THIS NEED TO BE SO COMPLICATED?

- What would happen if we just defined

```
(define (integer-stream x)
  (cons x
        (integer-stream (+ 1 x))))
```

Hint:

What do we need “lambda ()” ?

```
(define (integer-stream x)
  (cons x
        (integer-stream (+ 1 x))))
```

- Then the stream constructor would never terminate. The lambda is a way to delay evaluation of the rest of the stream until we need it.
- To remind us why the lambda is there, Scheme has a special macro called delay:

`(delay something)` stands for `(lambda () something)`

DELAY PUTS OFF EVALUATION UNTIL...IT IS FORCED.

- Then

```
> (delay 1)
#<procedure:...ream-objects.rkt:47:26>
> ((delay 1))
1
```

- If we define `(define (force c) (c))`, then

```
> (delay 1)
#<procedure:...ream-objects.rkt:47:26>
> (force (delay 1))
1
```

REMEMBER THE CONVENTION FOR A STREAM...

- In our convention, a stream is a pair.
- Either it is the empty list, or
  - The `car` is the first element of the stream;
  - The `cdr` is a function which, when evaluated with no arguments, returns the rest of the stream (a new pair).

THE STREAM OF NATURALS FROM k

- With this new terminology:

```
(define (head s) (car s)
(define (rest s) (force (cdr s)))
(define (integer-stream x)
  (cons x (delay (integer-stream (+ 1 x)))))
```

just means (lambda () (integer-stream (+ 1 x)))

- Then:

```
(head (integer-stream 21))
> 21
(head (rest (integer-stream 21)))
> 22
(head (rest (rest (integer-stream 21))))
> 23
```

EXTRACTION

- Or how to get the first k elements (as a list) out of a stream s

```
(define (take s k)
  (cond ((= k 0) '())
        (else (cons (head s) (take (rest s) (- k 1))))))
```

- Simple idea
  - Count as you extract
  - Add extracted element in front of the list.

THEN, TO GENERATE THE PRIMES...

- If `prime?` tests for primality, we can obtain the list of the first k primes by this natural process:

```
(define (primes-upto k)
  (take (filter prime? (integer-stream 2) k)
```

- Benefits?
  - Lazy approach: only generate what you need.
  - Totally done on demand. Works well with streaming data

STREAMING A FINITE STRUCTURE

- What if….
  - You wish to “convert” a list to a stream ?
  - i.e., you want to stream a list…
- Easy
  - Write a stream conversion function!

```
(define (stream-list l)
  (cond ((null? l) '())
        (else (cons (car l)
                    (delay (stream-list (cdr l)))))))
```

Administrative Stuff

- Final Exam Schedule and Room available here: https://registrar.uconn.edu/exams/
- The final exam is cumulative.
- We will have a review session for the final on Wednesday.
- We will HAVE Lab 11 on Tuesday like normal.
- We will NOT have Problem Set 11.
- Do you want to have 2 homework grades dropped? This will happen IF 70% of the class fills out the Set Survey: https://blueapp.grove.ad.uconn.edu/Blue/

End of the new material before the final!

Figure Sources

- http://s3.amazonaws.com/lionel-initial-assets/Products/ProductNavigator/_ProductImages_Orig/6-28062_5908.jpg?v=2
- https://www.kindpng.com/picc/m/125-1254800_funny-face-drawing-meme-hd-png-download.png
- https://www.worldatlas.com/r/w1200/upload/e5/bf/05/shutterstock-75885847.jpg
- https://magicspecialevents.com/event-rentals/wp-content/uploads/Treasure-Chest-Lock-And-Key-Game-3.jpg
- https://m.media-amazon.com/images/M/MV5BNWM0ZGJlMzMtZmYwMi00NzI3LTgzMzMtNjMzNjliNDRmZmFlXkEyXkFqcGdeQXVyMTM1MTE1NDMx._V1_.jpg
- https://whereisrusnivek.files.wordpress.com/2015/09/6007-590x444.jpg
- https://addicted2success.com/wp-content/uploads/2016/10/6-Reasons-to-Celebrate-Other-Peoples-Success.jpg