Streams, Iterators, Generators

Functional programming in javascript

Streams/LazyLists/Infinite Lists

- A lazily evaluated data structure similar to a linked list.
- Being lazily evaluated, streams can have an *infinite* number of elements.
- Not all streams need be infinite, though all my examples will be

Data type

- Traditionally a pair of a value, and a thunk to produce the next value
- For our initial examples we’ll use a simple object with two properties, **val** and **next**

```
{
    val: /*any*/ ,
    next: function () {/* return any stream */ }
}
```

- Many implementations use more traditionally named **head/tail**

Infinite list of ones

```
function ones() {
    // as simple stream
    return {
      val: 1 /*value*/,
      next: ones/*thunk to generate next element in stream*/
   };
}
```

Natural Numbers Stream

```
function nats(x) {
    // generate the next element of this stream
    var next = function () { return nats(x +
1); };
    // the stream
    return { val: x, next: next };
}
```

Fibonacci Stream

```
function fibonacci(x, y) {
    // the continuation of our stream
    var next = function() {
        return fibonacci(y, x + y);
    };
    // the stream
    return { val: x, next: next };
}
```

Using Streams

- Common Helper function
- Take: to take the first x values of a stream
- Range: to take a range from a stream
- All the list/collection functions we’ve grown to love
  - filter
  - Map
  - Fold/reduce

take

```
// take (stream, x)
// stream:
//        source stream
// x:
//        number of elements to take
function take(stream, x) {
    /*initialize an array*/
    var vals = [];
    // push first x values onto array
    for (; x > 0; x--) {
        vals.push(stream.val);
        stream = stream.next();
    }

    return vals;
}
take(ones(), 10);
// [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
take(nat(1), 10);
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
take(fibonacci(0,1), 10);
// [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

range

```
// range(stream, x, y)
// stream:
//       source stream
// x:
//       starting index in range
// y:
//       ending index in range
function range(stream, x, y) {
    var values = take(stream, y);
    return values.slice(x-1); //zero based index
}

range(nats(1), 40, 50);
//[40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]

range(fibonacci(0, 1), 40, 50);
/*[63245986, 102334155, 165580141, 267914296, 433494437, 701408733,
1134903170, 1836311903, 2971215073, 4807526976, 7778742049]*/
```

filter

```
// filter (f, stream)
// f
//      filtering function to execute on each element of stream
// stream
//      source stream
// return stream
//      return a new stream, which will filter source stream as it is lazily evaluated
function filter(f, stream) {
    var val = stream.val;
    var next = stream.next();
    if (f(val)) {
        return {
            val: val,
            next: function() {
                return filter(f, next);
            }
        };
    }

    return filter(f, next);

}

take(filter(function (x) { return x % 2 == 0; }, nats(1)), 5);
// [2, 4, 6, 8, 10]
```

map

```
// map (f, stream)
// f:
// map function
// stream
//source stream
// return stream
function map(f, stream) {

    return {
        val: f(stream.val),
        next: function () { return map(f, stream.next()); }
    };
}
take(map(function (x) { return x * x; }, nats(1)), 5);
// [1, 4, 9, 16, 25]
```

Streams Implementations

- http://streamjs.org/
- Linq.js http://neue.cc/reference.htm
- Node-lazy https://github.com/pkrumins/node-lazy

Iterators

- An object that knows how to access items from a collection one at a time, while keeping track of its current position within that sequence
- Standard ways to iterate over objects
- In ES6, an iterator is an object with a next method that returns { done, value } tuples

Standard For . . in

```
var obj = { a: 1, b: 2, c: 3 };
// order not guaranteed
for (var key in obj) {
    console.log(key, obj[key]);
}

// a 1
// b 2
// c 3
```

Javascript 1.7, simple iterators

- Not alot of support, only available with 1.7

```
var obj = { a: 1, b: 2, c: 3 };
var it = Iterator(obj);

var pair = it.next(); // Pair is ["a", 1]
pair = it.next(); // Pair is ["b", 2]
it.next(); // [pair is "c", 3]
it.next(); // A StopIteration exception is thrown


// can be used with for... in and for each constructs
var it = Iterator(obj);
for (var pair in it) {
    console.log(pair); // prints each [key, value] pair in turn
}
```

Generators

- A special type of function, that works as a factory for iterators.
- A function becomes a generator if it contains one or more **yield** expressions (only 1.7+)
- A specific type of iterator whose *next* is lazily evaluated with a generator function
- Generator comprehensions are shorthand expressions for creating generators.

Javascript 1.7

```
function simpleGenerator(){
    yield "first";
    yield "second";
    yield "third";
    // throws StopIteration
}

var g = simpleGenerator();
console.log(g.next()); // prints "first"
console.log(g.next()); // prints "second"
```

Infinite Sequence Generator

```
• function fibonacci(){
    var fn1 = 1;
    var fn2 = 1;
    while (true){
        var current = fn2;
        fn2 = fn1;
        fn1 = fn1 + current;
        yield current;
    }
}
```

Generator Expressions

- Similar to array comprehensions (also not well supported)

```
var it = Iterator([1,2,3]);

var it2 = (i[1]*2 for (i in it)); // i is pair [index, value]

for(var i in it2){
    console.log(i);
}

// 2
// 4
// 6
```

Generators Simulated

- Generators, and iterators can be simulated using the lazy evaluation strategies shown earlier, or with closure scope.
- Generators and iterators can be made in object oriented manner similar to OOP conterparts

Java like iterator

```
function JavaLikeIterator(){
    this.x = 1;
    this.y = 1;

}

JavaLikeIterator.prototype.constructor = JavaLikeIterator;
JavaLikeIterator.prototype.next = function(){
    var current = this.y;
    this.y = this.x;
    this.x = this.x + current;

    return current;
};
JavaLikeIterator.prototype.hasNext = function(){
    return true;
}


var jFib = new JavaLikeIterator();
var count = 0;
while (jFib.hasNext() && count <= 50) {
    count++;
    console.log(jFib.next());
}
```

The End

SWIMMING
NOTICE
MINNESOTA
STATE LAW
STRICTLY
PROHIBITS
UNDERWATER
SMOKING

Interesting Streams

```
function sieve(s) {
    var val = s.val;
    return {
        val: val, next: function () {
            return sieve(filter(function (x) {
                return x % val != 0;
            }, s.next()))
        }
    }
}

take(sieve(nats(2)), 10);
// [2,3,5,7,11,13,17,19,23,29]
```