# Notes for Midterm 2

## 1 Array Operations

The `.pop()` and `.push(x)` methods remove and add an element from the end of an array:

```javascript
let arr = [10, 20, 30];
arr.push(40);
arr // produces [10, 20, 30, 40]
arr.pop(); // produces 40
arr; // produces [10, 20, 30]
```

The `.shift(x)` and `.unshift()` methods remove and add an element to the beginning of an array:

```javascript
let arr = [10, 20, 30];
arr.unshift(0);
arr // produces [0, 10, 20, 30]
arr.shift(); // produces 0
arr; // produces [10, 20, 30]
```

The `.slice(lo, hi)` method produces a new array with elements starting from `lo` (inclusive) up to `hi` (exclusive): specified range:

```javascript
let arr = [10, 20, 30, 40];
arr.slice(1, 2); // produces [20]
arr.slice(1, 3); // produces [20, 30]
```

You can create a copy of an array using `.slice` as follows:

```javascript
let arr = [10, 20, 30, 40];
let arr2 = arr.slice(0, arr.length); // [10, 20, 30, 40];
arr === arr2; // produces false
```

The `.map(f)` method produces a new array, where the elements are the result of applying `f` to each element of the original array.

```
[10, 20, 30].map(x => x + 1) // [11, 21, 31]
["compsci", "220"].map(x => x.length) // [7, 3]
```

The `.filter(pred)` method produces a new array, which only contains the elements of the original array on which `pred` returns true.

```
[1, 2, 3, 4].filter(x => x % 2 === 0) // [2, 4]
[1, 2, 3, 4].filter(x => x % 2 === 1) // [1, 3]
```

These are examples of `reduce`:

```
[1, 2, 3].reduce((acc, x) => acc + x, 0) // 6
["compsci", "220"].reduce((acc, x) => acc + x.length, 0) // 10
```

It is straightforward to implement these array methods ourselves. Figure 1 implements the higher-order methods as higher-order functions.

## 2 Singly Linked Lists

A list is either a node (with a head and tail) or the empty list.

```typescript
type List<T> =
  { kind: 'node', head: T,
    tail: List<T> } |
  { kind: 'empty' };
```
function map(f, arr) {
  let r = [];
  for (let i = 0; i < arr.length; ++i) {
    r.push(f(arr[i]));
  }
  return r;
}

function filter(pred, arr) {
  let r = [];
  for (let i = 0; i < arr.length; ++i) {
    if (pred(arr[i]) === true) {
      r.push(arr[i]);
    }
  }
  return r;
}

function reduce(f, init, arr) {
  let acc = init;
  for (let i = 0; i < arr.length; ++i) {
    acc = f(acc, arr[i]);
  }
  return acc;
}

These are the list constructors:

function node(h, t) {
  return { kind: node, head: h, tail: t };}

function empty() {
  return { kind: 'empty' };}

An example of a single-linked list:

let alist = node(10, node(20, node(30, empty())));

Some example functions:

function listMap(f, alist) {
  if (alist.kind === 'empty') {
    return empty();
  } else {
    return node(f(alist.head), listMap(f, alist.tail));
  }
}

function listLength(alist) {
  if (alist.kind === 'empty') {
    return 0;
  } else {
    return 1 + listLength(alist.tail);
  }
}
3 Properties of Functions

JavaScript supports first-class functions, which have several important properties.

Anonymous Functions First-class functions do not have to be named. A function without a name is known as an anonymous function. For example, the following program immediately applies an anonymous function to an argument:

```javascript
(function(x) { return x + 1; })(10) // produces 11
```

Nested Functions First-class functions can be arbitrarily nested within each other. Moreover, the nested function can read and write to the variables of the enclosing function. For example:

```javascript
// F(x: number): (y: number) => number
function makeAdder(x) {
    function add(y) {
        return x + y; // note that y is not an argument of add
    };
    return add;
}
```

Closures are Values In JavaScript, closures are values. When a function \( F \) returns another function \( G \), it is really returning a closure of \( G \), which maps the variables outside of \( G \) to their values. For example, we can call the `makeAdder` function defined above twice, which produces two closures of `add`:

```javascript
let f1 = makeAdder(100); // the value of f1 is add[x -> 100]
let f2 = makeAdder(200); // the value of f2 is add[x -> 200]
```

If we invoke \( f1 \) or \( f2 \), we run the body of the `add` function, which is `return x + y`. The `add` function receives \( y \) as an argument, and the value of \( x \) is taken from the closure. Therefore, we get the following results:

```javascript
f1(10); // produces 110
f2(10); // produces 210
f1(5); // produces 105
```

Delayed Evaluation Functions can be used to delay evaluation. For example, the program below does not display anything.

```javascript
// F(g: T): T
function F(g) {
    return g;
}
F(function() { console.log("Will not display"); })
```

This occurs because \( F \) does not call its argument (i.e., it delays the evaluation of `console.log`), but merely returns it.

The following program actually displays the string, since it calls \( g \):

```javascript
// F(g: () => T): T
function F(g) {
    return g();
}
F(function() { console.log("Will display"); })
```

The following program also displays the string, since it calls the result of \( F \):
Information Hiding  
In the program below, there is no way to read or modify the value of the parameter $x$ outside the function:

```javascript
// F(x: number): (y: number) => number
function F(x) {
    let z = 9375739 * x;
    function g(y) {
        return y % z;
    }
    return g;
}

let f = F(2003);
console.log(f.z); // signals an error
console.log(f.x); // signals an error, as before
```

Therefore, we say that the closure $g[x \to 2003, z \to 9375739 \times 2003]$ hides the value of $z$.

4 Object References

The expression $(x: v, y: w, \cdots)$ creates a new object with fields $x, y, \cdots$ in memory and returns a reference to that object. Therefore, variables do not directly store objects. Instead, objects are stored in memory and variables store references to object (i.e., object references). Therefore, if the variable $o$ holds an object reference, then the statement `let o = p` creates a copy of that reference. It does not create a copy of the object. For example, in the program below, both variables store references to the same object:

```javascript
let o = { x: 10, y: 2 };  
let p = o;
```

Therefore, updating $p.x$ also updates $o.x$, since both refer to the only object in memory:
4.1 Nested Objects

The same line of reasoning we used above holds for objects’ fields. The following example creates two nested objects. However, the field o.x does not hold the object {a: 1, b: 2} itself. Instead, it holds a reference to the object. Therefore, the expression inner = o.x stores a copy of the reference in o.x.

```javascript
let o = { x: { a: 1, b: 2 }, y: 3};
let inner = o.x;
```

4.2 Arrays

Arrays are similar to objects: the expression [a, b, · · ·] creates an array with elements a, b, · · · in memory and returns a reference to the array. If arr1 is a variable that holds a reference to an array, then arr2 = arr1 creates a copy of the reference, and not a copy of the array, as shown below.

```javascript
let arr1 = [10, 20, 30];
let arr2 = arr1;
```

4.3 Arrays of Objects

The following example creates an array with two references to the same object:

```javascript
let arr = [{ x: 10 }];
arr.push(arr[0]);
```

Therefore, updating the field x in via one reference, updates the both references, since they both refer to the same object in memory:

```javascript
arr[0].x = 100;
arr[1].x // produces 100
```
5 Call by Value

When a program calls a function (or method), the method receives a new copy of its argument. This is known as call by value. For example, consider the following program:

```javascript
function F(x) {
    x = 10;
}
let y = 20;
F(y);
console.log(y); // displays 20
```

`F` receives a copy of the value in `y`. Therefore, when it updates its argument to 10, the update does not affect the value in `y`.

Call by value is more subtle when working with objects:

```javascript
function G(x) {
    x.m = 20;
}
let y = { m: 10 };
G(y);
console.log(y.m); // displays 20
```

Recall that objects are not values, but object references are values. Therefore, `G` receives a copy of the reference to the object `{ m: 10 }` and updates its field `m`. In contrast, the following example updates the value stored in `x` to a reference to a new object:

```javascript
function G(x) {
    x = { m: 20 }
}
let y = { m: 10 };
G(y);
console.log(y.m); // displays 10
```

6 Class Syntax

The two programs below are essentially equivalent:

```javascript
class Point {
    constructor(initX, initY) {
        this.x = initX;
        this.y = initY;
    }
    magnitude() {
        return Math.sqrt(this.x + this.x * this.y + this.y);
    }
}
let p = new Point(3, 4);
p.magnitude()
```

```javascript
function Point(initX, initY) {
    let o = {
        x: initX,
        y: initY,
        magnitude: function() {
            return Math.sqrt(o.x + o.x * o.y + o.y);
        }
    }
    return o;
}
let p = Point(3, 4);
p.magnitude()
```

7 Memoizer

A memoizer is an abstraction that wraps a function and “memorizes” its result. Figure 2 shows an implementation of memoizers for a zero-argument function.

In the following example, we use the memoizer to “memorize” the result of `factorial(10)`.

---

1The most significant different is that methods that use `this` can exhibit unusual behavior when used as a function. This is a JavaScript-specific detail that is beyond the scope of this class.
// type Memo<T> = { get: () => T }
// memo0<T>(f: () => T): Memo<T>

function memo0(f) {
  let r = { kind: 'unknown' };
  return {
    get: function() {
      if (r.kind === 'unknown') {
        r = { kind: 'known', v: f() }
      }
      return r.v;
    },
    toString: function() {
      if (r.kind === 'unknown') { return '<unevaluated>'; }
      else { return r.v.toString(); }
    }
  };
}

Figure 2: A memoizer for 0-argument functions.

function factorial(n) {
  let r = 1;
  while (n > 0) {
    r = r * n;
    console.log("Thinking ... ");
  }
  console.log("Done!");
  return r;
}
let f = memo0(() => factorial(10));
f.get();
f.get();

Therefore, the program only calls factorial once, even though it uses the result twice. If we did not call f.get(), then factorial would not be called at all.

Memoizers can be used independently, and are a building block for streams.

8 Streams

A stream is an abstraction that represents a conceptually unbounded sequence of values. Figure 3 shows an implementation of streams, along with some canonical higher-order functions on streams.

The following stream is the stream of natural numbers 0, 1, 2, 3, ...:

// grow(n: number): Stream<number>
function grow(n) {
  return snode(n, memo0(() => grow(n + 1)));
}
let nats = grow(0);

Using this stream, we can create streams of even numbers, odd numbers, etc.:

let evens = nats.map((n) => n * 2);
let odds = evens.map((n) => n + 1);

We can also use .filter to create streams of even and odd numbers:

let evens = nats.filter((n) => n % 2 === 0);
let odds = nats.filter((n) => n % 2 === 1);

9 Results

The Result abstraction is a uniform way to indicate successful and failing computations, without resorting to exceptions or special values (e.g., NaN or null). Figure 4 shows an implementation of results.
// sempty: Stream<T>
let sempty = {
  kind: 'sempty',
  map: (f) => sempty,
  filter: (pred) => sempty,
  toString: () => 'sempty'
};

function snode(head, tail) {
  return {
    kind: 'snode',
    head: function() { return head; },
    tail: function() { return tail.get(); },
    map: function (f) {
      return snode(f(head), memo0(() => tail.get().map(f)));
    },
    filter: function (pred) {
      if (pred(head)) {
        return snode(head, memo0(() => tail.get().filter(pred)));
      } else {
        return tail.get().filter(pred);
      }
    },
    toString: function() {
      return "snode(" + head.toString() + ", " + tail.toString() + ")";
    }
  }
}

Figure 3: Streams, with higher-order functions.

There are two constructors for results:

1. `new Success(value)` represents a successful computation that produces the value `value`.
2. `new Failure(reason)` represents a failed computation that failed for the given `reason` (usually a string).

Both kinds of results have a `.then`.

In the example below, suppose that `F` produces a result, then `G` gets called only if `F` produces a `Success`:

```javascript
F().then(function(x) {
  console.log("F succeeded and produced " + x.toString());
  return G();
});
```

Results can be "chained together". In the example below, if both `F` and `G` produce results, then the final result is `Success` only if they both produce `Success`:

```javascript
F().then(function(x) { return G(x) })
  .then(function(y) {
    console.log("F and G succeeded");
    return new Success(true);
  });
```
```javascript
class Success {
    constructor(value) {
        this.kind = 'success';
        this.value = value;
    }

    then(f) {
        return f(this.value);
    }
}

class Failure {
    constructor(reason) {
        this.kind = 'failure';
        this.reason = reason;
    }

    then(f) {
        return this;
    }
}
```

Figure 4: The Result abstraction.
This page intentionally left blank.