

Notes for Midterm 2

1 Array Operations

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

```
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:

```
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:

```
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:

```
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.

```
type List<T> =
  { kind: 'node', head: T,
    tail: List<T> } |
  { kind: 'empty' };
```

```

1 // map<S, T>(f: (x: S) => T, arr: S[]): T[]
2 function map(f, arr) {
3   let r = [];
4   for (let i = 0; i < arr.length; ++i) {
5     r.push(f(arr[i]));
6   }
7   return r;
8 }
9
10 // filter<T>(pred: (x: T) => boolean, arr: T[]): T[]
11 function filter(pred, arr) {
12   let r = [];
13   for (let i = 0; i < arr.length; ++i) {
14     if (pred(arr[i]) === true) {
15       r.push(arr[i]);
16     }
17   }
18   return r;
19 }
20 // reduce<S, T>(f: (acc: T, x: S) => T, init: T, arr: S[]): T
21 function reduce(f, init, arr) {
22   let acc = init;
23   for (let i = 0; i < arr.length; ++i) {
24     acc = f(acc, arr[i]);
25   }
26   return acc;
27 }

```

Figure 1: Implementations of higher-order functions over arrays.

These are the list constructors:

```

// node<T>(h: T, t: List<T>): List<T>
function node(h, t) {
  return { kind: 'node', head: h, tail: t };
}

// empty<T>(): List<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:

```

// listMap(f: (x: S) => T, alist: List<T>): List<T>
function listMap(f, alist) {
  if (alist.kind === 'empty') {
    return empty();
  } else {
    return node(f(alist.head), listMap(f, alist.tail));
  }
}

// listLength(alist: List<T>): number
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:

```
(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:

```
// 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`:

```
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:

```
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.

```
// 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`:

```
// 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`:

```
// F(g: T): T
function F(g) {
  return g;
}

let r = F(function() { console.log("Will not display"); });
r()
```

Information Hiding In the program below, there is no way to read or modify the value of the parameter x outside the function:

```
// F(x: number): (y: number) => number
function F(x) {
  function g(y) {
    return x + y;
  };
  return g;
}

let f = F(100);
f(10); // produces 110
f.x = 5; // signals an error
```

Similarly, in the program below, there is no way to access the value of the local variable z from outside the function:

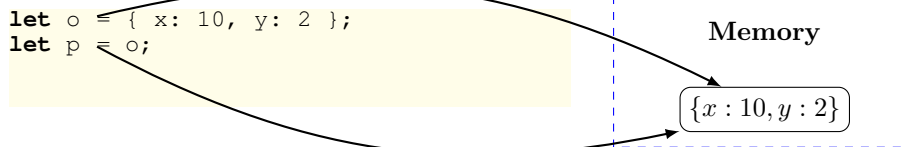
```
// F(x: number): (y: number) => number
function F(x) {
  let z = 9375739 * x;
  function g(y) {
    return y % z;
  };
  return g;
}

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 $g[x \rightarrow 2003, z \rightarrow 9375739 * 2003]$ *hides* the value of z .

4 Object References

The expression $\{x: v, y: w, \dots\}$ creates a new object with fields x, y, \dots 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 reference*). Therefore, if the variable o holds an object reference, then the statement $\text{let } o = p$ 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:

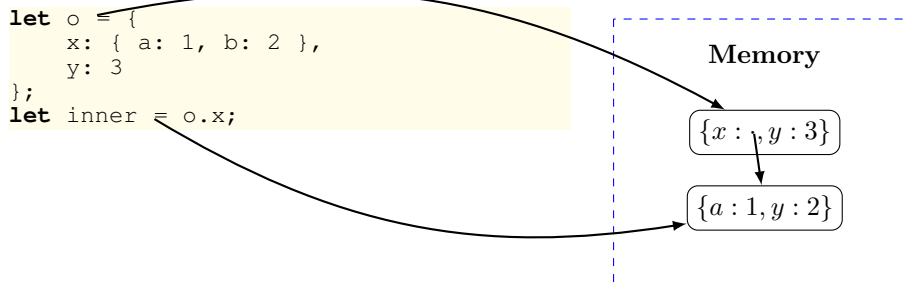


Therefore, updating $p.x$ also updates $o.x$, since both refer to the only object in memory:



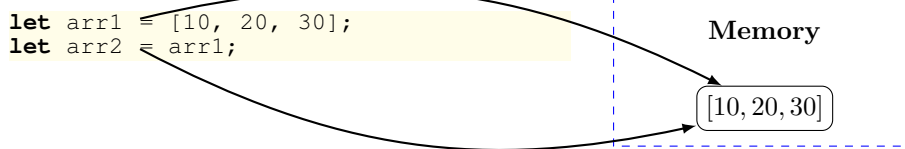
4.1 Nested Objects

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`.



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.



4.3 Arrays of Objects

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



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

```

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:

```
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:

```
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:

```
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:¹

```
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();
```

```
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 difference 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.

```

1 // type Memo<T> = { get: () => T }
2 // memo0<T>(f: () => T): Memo<T>
3 function memo0(f) {
4   let r = { kind: 'unknown' };
5   return {
6     get: function() {
7       if (r.kind === 'unknown') {
8         r = { kind: 'known', v: f() }
9       }
10      return r.v;
11    },
12    toString: function() {
13      if (r.kind === 'unknown') { return '<unevaluated>';
14      } else { return r.v.toString(); }
15    }
16  };
17 }

```

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(() => { return 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 $0, 1, 2, 3, \dots$:

```

// 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.

```

1 // empty: Stream<T>
2 let empty = {
3   kind: 'empty',
4   map: (f) => empty,
5   filter: (pred) => empty,
6   toString: () => 'empty'
7 };
8
9 // snode<T>(head: T, tail: Memo: Stream<T>): Stream<T>
10 function snode(head, tail) {
11   return {
12     kind: 'snode',
13     head: function() { return head; },
14     tail: function() { return tail.get(); },
15     map: function (f) {
16       return snode(f(head), memo0(() => tail.get().map(f)));
17     },
18     filter: function (pred) {
19       if (pred(head)) {
20         return snode(head, memo0(() => tail.get().filter(pred)));
21       } else {
22         return tail.get().filter(pred);
23       }
24     },
25     toString: function() {
26       return "snode(" + head.toString() + ", " + tail.toString() + ")";
27     }
28   }
29 }

```

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:

```

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:

```

F().then(function(x) { return G(x) })
  .then(function(y) {
    console.log("F and G succeeded");
    return new Success(true);
  });

```



```
1 class Success {
2   constructor(value) {
3     this.kind = 'success';
4     this.value = value;
5   }
6
7   then(f) {
8     return f(this.value);
9   }
10 }
11
12 class Failure {
13   constructor(reason) {
14     this.kind = 'failure';
15     this.reason = reason;
16   }
17
18   then(f) {
19     return this;
20   }
21 }
```

Figure 4: The Result abstraction.

This page intentionally left blank.

This page intentionally left blank.

This page intentionally left blank.