1 First-Order Methods over Arrays

The .pop and .push 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 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 method produces a new array with elements in the specified range:

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

Note that the first argument to .slice (the lower bound) is inclusive, and the second argument (the upper bound) is exclusive. 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); // produces [10, 20, 30, 40];
arr === arr2; // produces false
```

2 Higher-Order Functions over Arrays

2.1 Map

This is the implementation of map:

```javascript
function map(f, arr) {
  let r = [];
  for (let i = 0; i < arr.length; ++i) {
    r.push(f(arr[i]));
  }
  return r;
}
```

These are examples of map:

```javascript
map(function(x) { return x + 1; }, [10, 20, 30]) // produces [11, 21, 31]
map(function(x) { return x.length; }, ["compsci", "220"]) // produces [7, 3]
```

Note that map is also a method on arrays:

```javascript
arr.map(f)
```
2.2 Filter

This is the implementation of filter:

```javascript
// filter<T>(pred: (x: T) => boolean, arr: T[]): T[]
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;
}
```

These are examples of filter:

```javascript
filter(function(x) { return x % 2 === 0; }, [1, 2, 3, 4]) // produces [2, 4]
filter(function(x) { return x % 2 === 1; }, [1, 2, 3, 4]) // produces [1, 3]
```

Note that filter is also a method on arrays:

```javascript
arr.filter(pred)
```

2.3 Reduce

This is the implementation of reduce

```javascript
// reduce<S, T>(f: (acc: T, x: S) => T, init: T, arr: S[]): T
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 examples of reduce:

```javascript
reduce(function(acc, x) { return acc + x; }, 0, [1, 2, 3]) // produces 6
reduce(function(acc, x) { return acc + x.length; }, 0, ["compsci", "220"])) // produces 10
```

Note that reduce is also a method on arrays:

```javascript
arr.reduce(f, init)
```

3 Singly Linked Lists

These are the list constructors:

```javascript
// node<T>(head: T, tail: List<T>): List<T>
function node(head, tail) {
    return { kind: node, head: head, tail: tail };
}

// empty<T>(): empty<T>
function empty() {
    return { kind: 'empty' };
}
```

This function tests a list to check if it is empty:

```javascript
// function isEmpty<T>(list: List<T>): boolean
function isEmpty(list) {
    return list.kind === 'empty';
}
```
These are the list accessors:

```javascript
function head(list) {
    return list.head;
}

function tail(list) {
    return list.tail;
}
```

An example of a single-linked list:

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

Some example functions:

```javascript
// listMap(f: (x: S) => T, alist: List<T>): List<T>
function listMap(f, alist) {
    if (isEmpty(alist)) {
        return empty();
    } else {
        return node(f(head(alist)), listMap(f, tail(alist)));
    }
}

// listLength(alist: List<T>): number
function listLength(alist) {
    if (isEmpty(alist)) {
        return 0;
    } else {
        return 1 + listLength(tail(alist));
    }
}
```

## 4 Key Properties of First-Class Functions

### 4.1 Anonymous Functions

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

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

### 4.2 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 a parameter of this function
    }
    return add;
}
```

### 4.3 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
(function(x) { x + 1; })(10) // produces 11
```
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

4.4 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:

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

4.5 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) {
  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:

```javascript
// F(x: number): (y: number) => number
function F(x) {
  let z = 9375739 * x;
  function g(y) {
    return z + y;
  };
  return g;
}
```
```javascript
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 \( g[x \rightarrow 2003, z \rightarrow 9375739 \times 2003] \) hides the value of \( z \).

## 5 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 \( \text{let } o = p \) 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:

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

Therefore, updating \( p.x \) also updates \( o.x \), since both refer to the same object in memory:

```javascript
let o = { x: 10, y: 2 };
let p = o;
p.x = 200;
console.log(o.x); // displays 200
```

### 5.1 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 \( \text{inner} = o.x \) stores a copy of the reference in \( o.x \).

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

### 5.2 Arrays

Arrays are similar to objects: the expression \( [a, b, \cdots] \) creates an array with elements \( a, b, \cdots \) 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 = [1, 2, 3];
let arr2 = arr1;
arr2[0] = 5;
console.log(arr1[0]); // displays 5
```
5.3 Arrays of Objects

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

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

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
```
This page intentionally left blank.
This page intentionally left blank.