COMP 333
Spring 2026
Midterm Practice Exam #2 (Solutions)

This is representative of the kinds of topics and kind of questions you may be asked on the midterm. This practice exam, along with the in-class handouts on types and higher-order functions in JavaScript, are intended to be comprehensive of everything on the exam. That is, I will not ask anything that's not somehow covered by those sources.

You are permitted to bring two 8.5 x 11 sheets of paper into the exam with you, containing either printed or handwritten notes. Both sides of both sheets can be used.

Types

1.) Name one reason why someone might prefer static typing over dynamic typing.

Possible answers:
- Can help eliminate programming errors at compile time
- Can help in structuring code (types can be a guide for program development)
- Compilers and runtimes can usually optimize code better with type information

2.) Name one reason why someone might prefer dynamic typing over static typing.

Possible answers:
- No need to annotate programs with types, which are a significant amount of code
- More programs are possible, and can fundamentally represent things which cannot be represented in statically-typed code.

3.) Name one reason why someone might prefer strong typing over weak typing.

Possible answers:
- Strongly-typed languages are more predictable, especially in the presence of bugs and errors
- Strongly-typed languages tend to emphasize code correctness

4.) Name one reason why someone might prefer weak typing over strong typing.

Possible answers:

- There is usually a performance benefit. For example, there is no need for runtime array bounds checking.
- Weakly-typed languages tend to be more expressive. (Beyond what I'd expect: C lets you do more than Java in terms of low-level manipulation, but it's easy to hit undefined behavior, meaning programs lose all meaning. This can happen even when the code appears to be working correctly.)

5.) Consider the following code, written in some unknown language:

```
define doSomething(x) {
  return x.foo(7);
}

obj = Foo("hello")
value = doSomething(obj)
print(value)
obj = Foo("goodbye")
```

Provide an argument why this language might be statically-typed, OR why it might be dynamically-typed. Both are possible; the explanation why is the only important part.

Reasoning for dynamic typing: no types are present in the code, consistent with any dynamically-typed programming language.

Reasoning for static typing: none of the variables ever appear to change type, and `doSomething` always returns something of the same type, as long as `foo` always returns something of the same type. This is consistent with static typing with type inference, where the types are determined at compile-time, but the programmer never needs to explicitly write the types.

6.) Consider the following code snippet in some unknown object-oriented language:

```
Object obj = new Foo();
Foo f = (Foo)obj;
```

The following checks are relevant:

1. Ensure that class `Object` and `Foo` exist
2. Ensure that `Foo`'s constructor takes no arguments
3. Ensure that `obj` is in scope on the second line
4. With respect to the cast on the second line, ensure that `obj`'s type is actually `Foo`

Identify which checks are most likely to be performed when for each kind of language.

	Statically-Typed	**Dynamically-Typed**
Strongly-Typed	Runtime: 4 Compile-time: 1, 2, 3	Runtime: 1, 2, 3, 4 Compile-time:
Weakly-Typed	Runtime: Compile-time: 1, 2, 3	

7.) The code below is written in some unknown language. The code attempts to access an array at some index, and then print the value that was at that position:

```
let element = arr[index];
print(element);
```

You run this code snippet with different values for `arr` and `index`, including for values of `index` which are out of bounds for `arr`. With out-of-bounds array accesses, you observe that the code prints out a variety of values, and even sometimes crashes.

From these observations, is this language most likely strongly-typed or weakly-typed? In 1-2 sentences, explain your reasoning.

Weakly-typed. The behavior is unpredictable, indicating that no bounds check appears to be happening. Weakly-typed languages avoid runtime checks like bounds checks.

Higher-Order Functions in JavaScript

8.) Write the output of the following JavaScript code:

```
function foo(fooParam) {
  return function (innerParam) {
    return fooParam - innerParam;
  }
}

let f1 = foo(7);      // fooParam = 7 for f1
let f2 = foo(10);     // fooParam = 10 for f2
console.log(f1(2));   // innerParam = 2 for f1; 7 - 2 = 5
console.log(f2(3));   // innerParam = 3 for f2; 10 - 3 = 7
console.log(f1(4));   // innerParam = 4 for f1; 7 - 4 = 3
console.log(f2(5));   // innerParam = 5 for f2; 10 - 5 = 5
```

```
5
7
3
5
```

9.) Write the output of the following JavaScript code:

```
// returns a function that will bound the output of the wrapped
// function, so the output is never less than min or greater
// than max
function cap(min, max, wrapped) {
  return function (param) {
    let temp = wrapped(param);
    if (temp < min) {
      return min;
    } else if (temp > max) {
      return max;
    } else {
      return temp;
    }
  };
}

function addTen(param) {
  return param + 10;
}

function subTen(param) {
  return param - 10;
}

let f1 = cap(0, 10, addTen);
let f2 = cap(0, 100, addTen);
let f3 = cap(0, 10, subTen);
let f4 = cap(0, 100, subTen);

console.log(f1(0)); // output: 10
console.log(f1(5)); // output: 10
console.log(); // prints an empty line

console.log(f2(0)); // output: 10
console.log(f2(5)); // output: 15
console.log(); // prints an empty line

console.log(f3(0)); // output: 0
console.log(f3(5)); // output: 0
console.log(); // prints an empty line

console.log(f4(0)); // output: 0
console.log(f4(5)); // output: 0
console.log(); // prints an empty line
```

10.) Write the output of the following JavaScript code:

```
// Representing lists as higher-order functions.
// The function returns true if the given element exists in
// the list, else false (e.g., contains).  The list _is_
// the function.

function nil() {
    // nil doesn't contain any elements, so it definitely
    // doesn't contain searchKey, either
    return function (searchKey) {
        return false;
    };
}

function cons(head, tail) {
    // cons contains the given element searchKey, if either the
    // head of the list is searchKey, or if the tail of the list
    // contains searchKey
    return function (searchKey) {
        if (searchKey === head) {
            return true;
        } else {
            return tail(searchKey);
        }
    }
}

let emptyList = nil();
let one = cons(1, nil());
let oneTwo = cons(1, cons(2, nil()));

console.log(emptyList(1)); // false
console.log(one(1));       // true
console.log(oneTwo(1));    // true

console.log();             // <newline>

console.log(emptyList(2)); // false
console.log(one(2));       // false
console.log(oneTwo(2));    // true
```

11.) Consider the following JavaScript code with corresponding output, which calls an unseen function called `duplicate`. `duplicate` takes a function, and returns a another function which will call the input function twice.

```
function output() {
    console.log("foo");
}

let f1 = duplicate(output);
f1();
console.log();

let f2 = duplicate(f1);
f2();
console.log();

let f3 = duplicate(f2);
f3();
console.log();
```

Output:

```
foo
foo

foo
foo
foo
foo

foo
foo
foo
foo
foo
foo
foo
foo
```

Define the `duplicate` function.

```
function duplicate(f) {
    return function() {
        f();
        f();
    };
}
```

12.) Consider the following incomplete JavaScript code and output, which calls an unprovided function named `invert`:

```
function greaterThanFive(input) {
  return input > 5;
}

let notGreaterThanFive = invert(greaterThanFive);
let notEqualsFoo = invert(function (e) { return e === "foo"; });

console.log(notGreaterThanFive(3));
console.log(notGreaterThanFive(7));
console.log(notEqualsFoo("foo"));
console.log(notEqualsFoo("bar"));


---OUTPUT---
true
false
false
true
```

`invert` should return a new function that effectively inverts the behavior of the provided function. `invert` should work for any function taking one parameter and returning a boolean, not just the calls shown here. Implement `invert` below.

```
function invert(f) {
  return function (param) {
    return !f(param);
  }
}
```

13.) Consider the following JavaScript code and output:

```
function add(x, y) {
  return x + y;
}

function subtract(x, y) {
  return x - y;
}

let debugAdd = debug(add);
let debugSubtract = debug(subtract);

let addResult = debugAdd(1, 2);
let subtractResult = debugSubtract(5, 1);

console.log(addResult);
console.log(subtractResult);
```

Output:

```
First param: 1
Second param: 2
Return value: 3
First param: 5
Second param: 1
Return value: 4
3
4
```

The `debug` function takes another function taking two parameters, and returns a new function taking two parameters. The returned function prints its parameters, calls the wrapped function, prints the wrapped function's return value, then returns the wrapped function's return value. Implement `debug` below.

```
function debug(f) {
    return function (x, y) {
        console.log("First param: " + x);
        console.log("Second param: " + y);
        let retval = f(x, y);
        console.log("Return value: " + retval);
        return retval;
    }
}
```

14.) Consider the following JavaScript code:

```
function printHello() {
    console.log("hello");
}

function dontCall() {
    return function (f) {}
}

function callOncePlusN(n) {
    return function (f) {
        f();
        n(f);
    }
}

let zeroCalls = dontCall();
let oneCall = callOncePlusN(zeroCalls);
let twoCalls = callOncePlusN(oneCall);
let threeCalls = callOncePlusN(twoCalls);

zeroCalls(printHello);
oneCall(printHello);
twoCalls(printHello);
threeCalls(printHello);
```

Write the output of this code below:

```
hello
hello
hello
hello
hello
hello
```

15.) Consider the following JavaScript code:

```
function extractMin(arr1, arr2) {
    for (let index = 0; index < arr1.length; index++) {
        if (arr1[index] < arr2[index]) {
            arr2[index] = arr1[index];
        }
    }
}

function extractMax(arr1, arr2) {
    for (let index = 0; index < arr1.length; index++) {
        if (arr1[index] > arr2[index]) {
            arr2[index] = arr1[index];
        }
    }
}
```

`extractMin` and `extractMax` are intended to take two arrays of the same length. For each index `i` in both arrays, either the smaller or the larger of `arr1[i]` and `arr2[i]` will assigned into `arr2[i]`.

A significant amount of code is duplicated between these two functions. Write a new function named `extract` that generalizes over `extractMin` and `extractMax`, where `extract` takes:

- The first array
- The second array
- A function which takes two numbers and returns a boolean

After implementing `extract`, rewrite `extractMin` and `extractMax` to call `extract`.

```
function extract(arr1, arr2, f) {
    for (let index = 0; index < arr1.length; index++) {
        if (f(arr1[index], arr2[index])) {
            arr2[index] = arr1[index];
        }
    }
}
function extractMin(arr1, arr2) {
    extract(arr1, arr2, (a, b) => a < b);
}
function extractMax(arr1, arr2) {
    extract(arr1, arr2, (a, b) => a > b);
}
```

16.) Write the output of the following JavaScript code:

```
function pair(a, b) {
    return function (f) {
        return f(a, b);
    };
}

let p1 = pair(1, 2);
let p2 = pair("foo", "bar");
let getFirst = (x, y) => x;
let getSecond = (x, y) => y;
let add = (x, y) => x + y;

console.log(p1(getFirst));
console.log(p1(getSecond));
console.log(p1(add));
console.log(p1((a, b) => a * b));

console.log(); // prints a newline

console.log(p2(getFirst));
console.log(p2(getSecond));
console.log(p2(add));
```

```
1
2
3
2

foo
bar
foobar
```