Name:______________________________________

EECS 368 sp’18 Questions on Scoping, Closure & Hoisting

Due: Friday 9th March, 10am (start of class)

1. For each of the following JavaScript snippets, list what will be printed to the console and explain why.

a.
```
let foo = (x) =>  x + x;
console.log(foo("44"));
foo = (x) =>  x + x;
function foo(t){
   return t;
}
//-------------------------------
/*answer:
This prints an “uncaught SyntaxError” Identifier 'foo' has already been declared”. This
is because the ‘function foo()...’ declaration is hoisted above all else. THEN the ‘let
foo...’ statement is hit which tries to create a variable called ‘foo’, but that
binding already exists! You could redefine foo by leaving off the ‘let’.
*/
```

b.
```
function bar(x){
   if(x){
       let x = 1;
       if(x){
           let x = "";
           if(x){
               let x = "0"
               console.log(x);
           }
           console.log(x);
       }
       console.log(x);
   }
   console.log(x);
}
bar(-1);
//----------------------------------
/* Answer:

1
-1


This is because bar is passed the value -1 which gets coerced into true in the first if
statement. (only 0 & NaN get coerced into false. Remember NaN is never equal to
anything else! NaN == _ //always false.). Within this if statement, we create a “hole
in the scope” of the x that was passed in by declaring “let x =1;” which has its own
place in memory. Now whenever we use ‘x’ within this if branch, we find this one and
never have to climb the static chain any farther. Therefore the x in the next check of
“if(x)...” the value bound to x is 1 which is coerced to true. Inside this branch, we
once again we put a “hole in the scope” of the outer x, and within this block, x is
```

Name:____

```
bound to “”. Now the next “if(x)...” coerces “” into false and the body of this if
branch ( let x = "0"; console.log(x); ) is not executed.
Then when we console.log(x), static scoping says “” (the empty line in the output) and
we exit that if branch. Now the console.log(x) refers to the one bound to 1, then we
exit that if branch. The final console.log(x), the ‘x’ in scope is the one passed to
the function: -1.
*/
```

c.
```
function compose(f,g,x){return f(g(x))};
function h(f,x){return compose(f,f,x)};
h(console.log, "message");
//-----------------------------------
/* Answer:
message
undefined

Compose takes 2 functions and an argument and returns to you the result of calling the
second function with the argument then passing the result to the first function and
return that result to you. For example compose(x=>x*2, x => x + 1,3); returns 8. This
is because it first calls the second function (named g and in this case: x => x + 1)
with the value 3 which is 4. The result (4) is then passed to the first function (named
f and in this case: x => x*2) which returns 8.
Function h simply takes 1 function and a value and composes it with itself. For
example, h(x => x + 1, 9) would mathematically speaking become: f(9) = (9 + 1) + 1.
Therefore, when we call h(console.log, “message”), this is defined as:
compose(console.log, console.log, “message”). Which is defined as
(console.log(console.log(“message”))).  The second function is then called with the
value given, i.e. (console.log(“message”)). This prints “message” to the console. The
function console.log(/*ANYTHING*/) returns the undefined value. This is what is passed
to the first function (console.log). Therefore, we have console.log(undefined) which
prints undefined.
*/
```

d.
```
let f = x => x + 2;
let g = x => x + 6;
function higher(f){
   return x => f(g(x));
}
function foo(){
   console.log(higher(x=> x - 3)(9) );
}
foo();
//------------------------------------
/*Answer:
Higher and foo are both hoisted to the top, of course, but this doesn’t really make a
difference in this example. We call foo, which prints the result of calling (higher(x=>
x - 3)(9). In ‘higher’ we call our parameter f. Therefore, we have no access to the
outer f defined as x => x + 2. Remember our scoping rules: Whenever ‘f’ is referred to
in this function, the runtime finds it within the function. There is no need to search
outwardly anymore (therefore, we can’t access the outer f anymore because we re-used
that name!).
```

Name:____

```
In our call to higher, f is bound to the function value ‘x => x - 3’. Calling ‘higher’
with this argument returns a function which is the composition of first calling g
(which is currently bound to x => x + 6) and passing the result to f (which is
currently bound to x => x - 3). Therefore, the result of calling higher(x => x - 3) is
mathematically similar to this function f’: f’(x) = f ○ g = (x + 6) - 3. Calling this
with the value 9 is then simply (9 + 6) - 3 = 12.
*/
```

e.
```
let f = x => x + x ;
let g = y => y - 2;
function foo(){
   let f = x => x + 2;
   console.log(f(g(3)));
   f = x => 1;
}
foo();
console.log(f(2));
/* Answer:
We first call foo. Inside foo, we bind the name ‘f’ to the function x => x + 2. We then
print the result of calling g(3) and passing that value to f. When the runtime looks up
‘g’, it does not find it in the scope of foo. It then pops up one level
(statically/lexically) to look for it and finds it defined as y => y -2. Calling g(3)
therefore returns to us the value 1. This value is then immediately passed to f which
gives us 3-- the printed value.  We reassign our f to be x => 1. Foo then returns. We
then ask to see the result of calling f(2). However this time when the runtime looks
for the value of f, we see that it is x => x + x. (when foo ended with f = x => 1, the
‘f’ that it modified was the ‘f’ defined within foo, not the outer ‘f’ to which we are
now referring). The result of 2 + 2 is 4 which is then printed.
*/
```

2. The following code *looks* like JavaScript but is an entirely different made up language called DynaScript that has dynamic scoping rules rather than the traditional static. Assuming the console.log function outputs a string, what is printed in the following code snippet?

```
let x = 1;
let y = 2;
function f(a){
   return a + x;
}
function g(){
   return f(x) + y;
}
function foo(){
   let x = 3;
   console.log(f(x));
   console.log(g());
```

Name:____

```
        }
        foo();
        console.log(g());
```

If the above code were executed in a JavaScript environment, what would the output be?

/* **Answer:**

NOTE: When reading through this answer, I would strongly recommend having the code snippet on a piece of paper to follow along with! It is easy to get lost.

Part 1 DynaScript: It is helpful to keep a record of the call stack for making sense of dynamic scoping. Throughout this answer you will see a ‘pseudo-stack’ which only keeps function names for brevity.

The first interesting bit is we call foo from our global environment.

```
[TOP OF STACK]
foo
global
[BOTTOM OF STACK]
```

Inside foo we define a variable x to be 3. We then print the result of calling f(x) so we must call f with the value x. Before we actually put f onto the call stack, we have to figure out what the value is we are passing to it.

So far, whether we are statically or dynamically scoped has not made a difference because in both designs, we first check our immediate scope and find x. Assuming we are a ‘pass by value’ language, a is then assigned its copy of 3.

```
[TOP OF STACK]
f
foo
global
[BOTTOM OF STACK]
```

F then returns the result of adding ‘a’ (which is 3) to ‘x’. Now our scoping rules come into play in determining what value retrieved for calling ‘x’ should be. We first check in our immediate scope (dynamic + static scoping both do this) and don’t find ‘x’. We then follow the *dynamic* chain (the *“who called me?”* chain) to look for ‘x’. Foo is who called me and it is here that I find the value of ‘x’ (defined as 3). Therefore, we return 3 + 3 which is 6. ‘f’ is now done and pops off the stack:

```
[TOP OF STACK]
foo
global
[BOTTOM OF STACK]
```

foo then prints this value of 6

Foo moves on to execute the ‘console.log(g());’ line. So we have called the function g.

```
[TOP OF STACK]
g
```

Name:____

foo
global
[BOTTOM OF STACK]
g does nothing more than return the result of f(x) + y. None of these three things (f, x, & y) are defined locally, so we must use our dynamic scoping rules to retrieve their values. The first thing we do is call f with x, so let's find out what 'x' is. Since 'x' is not defined in g, we move up one in the dynamic chain looking for it. We are now investigating the scope of foo (since foo called g). Here we find an 'x' and its value is 3. So so far in g we know we must return f(3) + y. Now what 'f' are we talking about? Again, we don't find an 'f' defined in the scope of g so we move up one in the dynamic chain looking for it (foo once again). We don't see it inside foo, so we move up one more in the dynamic chain: who called foo? The global scope called foo. Therefore we know the 'f' we are talking about is the one defined there: function f(a){ return a + x}.

So my return statement can be simplified from 'return f(3) + y' to actually calling this function f(a){return a + x}.
[TOP OF STACK]
f
g
foo
global
[BOTTOM OF STACK]
Well we know the value of 'a' in this case will be '3' since that is what is passed to f, but what the heck is x inside f? Remember our call stack looks like this right now: we are currently executing f FROM g:
[TOP OF STACK]
f
g
foo
global
[BOTTOM OF STACK]

We don't see an 'x' defined in the scope of f, so we follow the dynamic chain up one ('down' one with the above representation of the stack) to g. We see no 'x' defined in g, so we pop to foo. Here, we find an x whose value is 3. Therefore our call to 'f' returns 3 + 3 or '6'. 'f' can then be popped off the stack:
[TOP OF STACK]
g
foo
global
[BOTTOM OF STACK]
And the body of g is simplified to return 6 + y; We don't have a 'y' in our scope so we go check foo. foo doesn't have a y, so we check our global environment (because it called foo) and find 'y' to be 2. G return 6 + 2 or '8'.
[TOP OF STACK]

Name:____

foo
global
[BOTTOM OF STACK]
Now foo continues execution and prints 8. Foo now returns and can be popped off the stack:
[TOP OF STACK]
global
[BOTTOM OF STACK]
Global environment then continues with the next statement which is: ‘console.log(g());’
So we call g:
[TOP OF STACK]
g
global
[BOTTOM OF STACK]
g calls f with x so we first look up the value of x. This time, we look in global scope because that is where the call came from and see that x is 1 so g becomes ‘return f(1) + y. We then push a call to f onto the stack after we dynamically find which f we are talking about.
[TOP OF STACK]
f
g
global
[BOTTOM OF STACK]
f returns 1 + x. We don’t find ‘x’ in f, nor g, but we eventually find it in the global scope with value 1. So f returns 1 + 1 or ‘2’ to g and pops off the stack:
[TOP OF STACK]
g
global
[BOTTOM OF STACK]
g’s body is now: return 2 + y. We look up y and find it in the global scope as 2. g returns 2 + 2 or 4 and pops off the stack. global then continues execution
and prints this result 4.

We just witnessed that calling g() had 2 different results depending on *where* it was called!
You an see how confusing dynamic scoping can be.
IF IN JAVASCRIPT ANSWER:
In short, we first call foo which first prints the result of f(3) when f’s body asks for the value of x, we go up one statically in scope and see it in global scope with value 1. F returns 3 + 1 or 4 and **foo prints 4.** Then foo moves on to the console.log(g()) line and calls g. When g looks for x, it finds the global version and the return value becomes:
f(1) + 1 -->   (1 + x) + y -->   (1 + 1) + y -->   (1 + 1) + 2 -->   4
4 is printed for the result of console.log(g()).
foo pops off the stack. When global scope calls console.log(g()), **it will print 4** just like the previous call to g(), because its definition in code is static-- you can’t change where g was defined. Therefore, all non-local variable lookups will have the same result (no variables have been altered either).
*/

Name:____

3. Thanks to hoisting, what is the “hoisted” equivalence of the following JavaScript snippet?

```
foo = () => console.log("la la la");
foo();
function foo(){
   innards();
   function innards(){
       //work work work;
   }
   let x = "value";
   return x;
}
/* Answer:
function foo(){
  let x;
  function innards(){
       //work work work;
  }
  innards();
  x = "value";
  return x;
}

foo = () => console.log("la la la");
foo();
foo is moved to the top of its scope. Innards is moved to the top of its scope. Also the let
declaration + assignment is split into declaration & assignment. Inside foo, whether you put
the let x before the innards function or after is not terribly important. Both ways are okay
so long as the function declaration and the let x are the first 2 things in the function foo.
In reality these are hoisted to a magical equivalent level and both can see each other. This
is important if you declare two functions and and they call each other. f(){g()}; g(){f()}
They both must be declared before the other. JavaScript makes sure they know about each other.
Thanks JavaScript!
 */
```