Homework 2: Functions, scope, and storage

CSE 130: Programming Languages

Early deadline: July 11 23:59, Hard deadline: July 14 23:59

Names & IDs:

1 Variable Bindings and Closures [9pts]

In this problem we will explore how variable capture differs in JavaScript and Haskell due to mutable/immutable bindings.

1. [3pts] Consider the following JavaScript code snippet:

```
let x = y => y + 1;
x = y => y + x(2);
x(2);
```

What happens when the function call `x(2)` is made? (Run the code.) Why does this happen instead of returning 2 + 2 + 1 (or 5)? (Hint: think about which function the binding `x` refers to – line 1 or 2 – in evaluating the function body on line 2.)

Answer:

2. [3pts] Now consider the following code snippet:

```
let x = y => y + 1;
x = (z => y => y + z(2))(x);
x(2);
```

What happens when this code is executed? In contrast to the previous code snippet, why does this code return the correct sum? Explain briefly.

Answer:

3. [3pts] Now consider the same code snippet but, written in Haskell:

```
let x = \y -> y + 1 in let x = (\z -> \y -> y + (z 2)) x in x 2
```

What happens when this expression is evaluated? (Run the code with GHCi.) Briefly explain why this behavior is different from its JavaScript counterpart.

Answer:

2 Closures and access links [14pts]

Consider the following JavaScript code.

```
1: let z = 2;
2: let f = function(x) {return f(x+1);}
3: let h = f;
4: f = function(x) {z++; return x+1;}
5: let y = h(4)*z;
```

1. [9pts] Fill in the missing parts in the following diagram of the run-time structures for execution of this code up to the point where the call inside `f(5)` is about to return. Note that `y` is still unassigned at the time.

   In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure or compiled code (▷). Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link". The first one is done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.

Activation Records				*Closures*	*Compiled Code*
(1)		access link	(0)		
		z			
		f	•	▷⟨ ( ), • ⟩	▷| code on line 2 |
		h	•		
		y	— —	▷⟨ ( ), • ⟩	▷| code on line 4 |
(2)	h(4)	access link			
		x			
(3)	f(5)	access link			
		x			

2. [1pts] If we run the code, what is the final value that is assigned to `y`?

Answer:

3. [4pts] Suppose we change the definition on line 2 to a named function as follows:

```
let f = function f(x) {return f(x + 1);}
```

If we run the code again, the evaluation of `h(4)` will not terminate with a return value. What do you suspect is the reason for this change in behavior? (Please keep your answer short, but specific.)

Answer:

3 Memory management and high-order functions [15pts]

This question asks about memory management in the evaluation of the following code that contains high-order functions.

```
let x = 5;
{
  function f(y) {
    return (x+y)-2;
  }
  {
    function g(h) {
      let x = 7;
      return h(x);
    }
    {
      let x = 10;
      g(f);
    }
  }
}
```

1. [13pts] Fill in the missing information in the following depiction of the run-time stack after the call to `h` inside the body of `g`. Remember that function values are represented by closures, and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code.

   In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure or compiled code (▷). Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link". The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.

Activation Records

(1)		access link	(0)
		`x`	
(2)		access link	(1)
		`f`	•
(3)		access link	( )
		`g`	•
(4)		access link	( )
		`x`	
(5)	`g(f)`	access link	( )
		`h`	•
		`x`	
(6)	`h(x)`	access link	( )
		`y`	

Closures

▷⟨ ( ), • ⟩

▷⟨ ( ), • ⟩

Compiled Code

▷| code for `f` |

▷| code for `g` |

2. [2pts] What is the value of the call `g(f)`? Why?

Answer:

4 More substitution and variable capture [12pts]

In this problem we're going to look at capture-avoiding substition again. Once you've mastered capture-avoiding substitution and β-reduction, you will be able to do more advance things (like encode numbers and booleans).

For each of the terms below, perform the capture-avoiding substitution showing intermediate steps.

1. [4pts] Perform this substitution $((\lambda x.((\lambda x.x)\ 2) + x)\ x)[x := 3]$

   Answer:

   Suppose you used non-capture-avoiding substitution instead. Would the result be incorrect in this case? Explain.

   Answer:

2. [4pts] Perform this substitution $(\lambda y.(\lambda xyz.z)\ y\ z\ y)[z := w]$

   Answer:

   Suppose you used non-capture-avoiding substitution instead. Would the result be incorrect in this case? Explain.

Answer:

3. [4pts] Perform this substitution $(\lambda p.(\lambda x.p\ (x\ x))\ (\lambda x.p))[x := p]$:

Answer:

Suppose you used non-capture-avoiding substitution instead. Would the result be incorrect in this case? Explain.

Answer:

5 [32pts] Type Inference

In this problem you will apply the Hindley-Milner type inference algorithm we discussed in class to figure out the type of two μHaskell declarations. For both, you must go through the five steps: creating the parse trees, assigning type variables, generating constraints, solving the constraints, and finally circling your final type answer (if no type error was encountered).

5.1 [14pts] Infer the type of `reverse`:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Answer:

5.2 [18pts] Infer the type of `foldl`:

```
foldl f acc []     = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
```

Answer:

Acknowledgements

Any acknowledgements, crediting external resources or people should be listed below.