Homework 1: Javascript and λ calculus

CSE 130: Programminig Languages

Jack Shi - A92122910
Wyatt Guidry - A12994977
Austin Coleman - A12888539

Jan 23, 2018

Notation

For this problem set we may use (and you may find it helpful to use) a less verbose syntax for -calculus programs. Specifically, instead of writing x.(y.(z.e)), we may drop the parentheses and write this term as x.y. . . . z.e. In general, the body of a abstraction extends as far right as possible. To further shorten notation, we may also drop the λs in our example term and write it as xyz.e when the variable names are obvious and not ambiguous.

Similarly, instead of writing (e 1 e 2) e 3 well simply write e 1 e 2 e 3 . We can drop parentheses in function applications since applications in λ-calculus are always assumed to be left-associative.

1 JavaScript hoisting and block-scoping [12pts]

Recall that JavaScript variable declarations using var can have one of two scopes: function scope or global scope. Variables declared inside a function have function scope while variables declared outside of functions have global scope. Regardless of where variables are actually declared and initialized, their declarations (but not initializations) are hoisted to the top of the nearest (function or global) scope; vars are not block scoped. This is different from let and const variables which are block scoped, as in most languages you might have encountered.

In this problem, you will implement block scoping for var declarations using first-class functions. In each part, you will be asked what the result of running a snippet of code is, andto receive full credit

1. [1pts] What does f(x) return? What values of x and y are used in the computation and why?

```
1   var x = 5;
2   function f(y) {return x+y}
3   f(x);
```

Answer:

The function returns 10. x used in computation is resolved to the global variable x which is 5. x is passed in as parameter y, which points to the same global variable, resolves to 5.

2. [1pts] What does f(x) return? What values of x and y are used in the computation and why?

```
1   var x = 5;
2   function f(y) {return x+y}
3   if (true) {
4     var x = 10;
5   }
6   f(x);
```

Answer:

The function returns 20. x used in computation is resolved to 10 because at the time of calling f, x has already been reasigned to value 10. x is passed in as parameter y, which resolves to the global variable x that is reassigned to 10 before it is referenced as a parameter y. y therefore resolves to 10 during computation.

3. [1pts] What does f(z) return? What values of x and y are used in the computation and why?What does f(x) return? What values of x and y are used in the computation and why?

```
1       var x = 5;
2       function f(y) { return x + y; }
3       if (true) {
4         var z = 20;
5       }
6       f(z);
```

Answer:

The function f(z) returns 25. Since x is not captured by the fucntion, it evaluates to x in the global scope which is 5. z is passed in as parameter y. At the time of using z as a parameter y, z is declared and assigned value 20. y therefore resolves to 20 during computation.

4. [2pts] What does f(x) return? What values of x and y are used in the computation and why? Explain how this behavior differs from that of part (1) above.

```
1       (function () {
2         var x = 5;
3       })();
4       function f(y) { return x + y; }
5       f(x);
```

Answer:

f(x) causes an ReferenceError because x is not defined. The x in computation is undefined because there is no variable x within any scope that is visible to the function f. Since an undefined variable is passed in as parameter y, y is therefore also undefined.

In **part 1**, var x is global scope, therefore references to x correctly resolves to 5.

5. [2pts] What does f(x) return? What values of x and y are used in the computation and why? Explain how this behavior differs from that of part (2) above.

```
1       var x = 5;
2       function f(y) { return x + y; }
3       if(true) {
4         (function( {
5           var x = 10;
6         }) ();
7       }
8       f(x);
```

Answer:

f(x) returns 10. The variable x in computation resolves to 5 because the declaration that var x = 10 is in the a function scope that is not visible to f. The x that is passed in as y is also the global variable x. Therefore y in computation is resolved to global var x which is 5.

In **part 2**, var x = 10 is not wrapped in some function. It is a global scope expression therefore var x=10 is visible to funciton f.

6. [2pts] What does f(z) return? What values of x and y are used in the computation and why? Explain how this behavior differs from that of part (3) above.

```
1       var x = 5;
2       function f(y) { return x + y; }
3       if(true) {
4         (function () {
5           var z = 20;
6         }) ();
7       }
8       f(z);
```

Answer:

f(z) causes an ReferenceError because z is not defined in a scope visible to the caller. The value x is defined in global scope and is resolved to 5. z is passed into f as parameter y. Since var z is declared within a anonymous function where the variable is only visible within the function scope, y is therefore undefined because the caller is unable to resolve variable z.

This is different from **part 3** because in **part 3**, z is in global scope therefore f(z) works because z is visible from the scope where it is referenced.

7. [3pts] Rene thinks that Joes compiler may have been a bit too clever, leading it to behave incorrectly in some cases. Is she right? If so, explain where Joes compiler got it wrong above and describe a general wrapping algorithm that Joe should have used instead.

Answer:

Keep a list of tuples that represents the edges that connects expressions. Go through the code and do the following:

1. Whenever an expression that references a variable is encountered, create an edge between this expression and a declaration of the variable happened earlier within the same scope.
2. Whenever an variable declaration/assignment is encountered, create an edge between this declaration/assignment and any earlier declaration/assignment in the same scope if it exists.

Once the list of edges has been obtained, simply running DFS would give you all the disjointed components that need to be captured within the same scope. For each component, find the expressions that happened ealiest and lastest. Go through these boundaries for each component, merge them if they overlapse. Use these new boundaries as guidlines and add function scopping around each to mimic block scopping properly.

2 λ-calculus $\Leftrightarrow$ JavaScript [14pts]

To get you more comfortable with λ-calculus syntax, in this problem, well be converting -calculus expressions to JavaScript expressions and back. At first, you may find it helpful to think about expressions in terms of

their JavaScript counterparts; but once your comfortable with the -calculus syntax you may find the converse to be true!

First, well convert some terms to JavaScript. You may use JavaScript arrow functions, but dont have to.

1. Convert $\lambda xfgh.(f\ x)\ (g\ x)\ (h\ x)$ to the equivalent JavaScript expression.

   Answer: $\lambda xfgh.(f\ x)\ (g\ x)\ (h\ x) \equiv \lambda xfgh.(((f\ x)\ (g\ x))\ (h\ x))$

   ```
   1  (x, f, g, h) => ((f(x)) (g(x))) (h(x))
   ```

2. Convert the following term to the equivalent JavaScript expression.

   ```
   1  λa1a2c1c2      if a1 + a2 < 10
   2                 then c1, a2, a2
   3                 else c2, a2, a1
   ```

   Answer:

   ```
   1  (a1,a2,c1,c2) => (a1 + a2 > 10) ? [c1,a2,a2] : [c2,a2,a1];
   ```

3. Convert $(\lambda x.y)((\lambda x.x\ x)(\lambda y.y\ z))$ to the equivalent JavaScript expression.

   Answer:

   ```
   1  (x=>y)((x=>x(x))((y=>y)(z))
   ```

4. Write the JavaScript program f(g(3)) and helper function f and g as a single λ-calculus expression.

   ```
   1  const f = (x) => x*2;
   2  const g = function (y) { return y-1; }
   3  f(g(3))
   ```

   Answer: $\lambda x.(x * 2)(\lambda y.(y - 1)(3))$

3 λ-calculus and macro processors [10pts]

Macro processors, such as cpp and m4, do a form of program manipulation before the program is further compiled. You can think of this as symbolic program pre-evaluation. For example, if a program contains the macro

```
1  #define square(x) ((x)*(x))
```

macro expansion of a statement containing square(y+3) will replace square(y+3) with (y+3)*(y+3). One way to build a macro processor is to associate a lambda expression with each macro definition, and then do β-reduction wherever the macro is used. For example, we can represent square as the lambda expression $\lambda x.(x * x)$ and β-reduce:

$$\lambda x.(x * x))(y + 3) =_\beta (y + 3) * (y + 3)$$

f square(y+3) appears in the program. Some problems may arise, however, when variables that appear in macros also appear in the program.

1. Suppose a program contains three macros:

```
#define f(x) (x+x)
#define g(y) (y-2)
#define h(z) (f(g(z)))
```

Macro h can be written as the following lambda expression:

$$\lambda z.(\lambda x.x + x)((\lambda y.y - 2)z) \equiv \lambda z.f(g\ z)$$

Simplify the expression h(3) by using β-reduction. Do not simplify the arithmetic. Only reduce one step at a time. Use as many or as few lines as you need.

$$\begin{aligned}
&(\lambda z(\lambda x.x + x)((\lambda y.y - 2)z))3 \\
=&(\lambda x.x + x)((\lambda y.y - 2)3) \\
=&(\lambda x.x + x)(3 - 2) \\
=&(3 - 2) + (3 - 2)
\end{aligned}$$

2. A problem arises if a local variable inside a macro has the same name as an argument the macro is invoked with. Assuming that typeof is supported, the following macro works as long as neither of the parameters is named temp.

```
#define swap(a,b) { typeof(a) temp = a; a = b; b = temp; }
```

Show what happens if you macro-expand swap(x, temp) without doing anything special to recognize that temp is bound in the body of the macro. You do not need to convert this macro to lambda notation.

```
swap(x,temp) = { typeof(x) temp = x; x = temp; temp = temp; }
```

3. Explain what the problem is and how you can solve it.

Answer: The problem is that FV(swap)=temp and parameter temp $\in FV(swap)$. This means that the variable temp in function swap must be renamed to a name that is not a, b, or temp.

4. Show how your solution would expand swap(x, temp) properly.

Answer: Lets say variable temp in function swap is renamed to cat. The expanssion of swap(x, temp) would look like the following:

```
swap(x,temp) = { typeof(x) cat = x; x = temp; temp = cat; }
```

4 Free Variables [6pts]

The F V function we discussed in class is a function that operates on syntax. (If fact, you can think of F V as one of the simplest possible static analysis.) Since F V is defined recursively, it must be total, i.e., it must be able to take any program (term) written in -calculus as input. To this end, the function is defined for every kind of term:

$$\begin{aligned}
FV(x) &= x \\
FV(\lambda x.e) &= FV(e) \setminus \{x\} \\
FV(e_1\ e_2) &= FV(e_1) \cup FV(e_2)
\end{aligned}$$

Given this, find the free variables in the terms below:

1. $FV(\lambda x.(\lambda y.y)) = \emptyset$
2. $FV(\lambda x.(\lambda y.x)) = \emptyset$
3. $FV((\lambda x.x\ y)(\lambda y.y\ x)) = \{x, y\}$
4. $FV((\lambda p.\lambda q.\lambda r.p\ q\ r)(\lambda p.\lambda q.p\ q\ r)) = \{r\}$
5. $FV(x\ y) = \{x, y\}$
6. $FV(((\lambda f.\lambda y.\lambda x.f(y(x)))(\lambda x.y + z))(\lambda z.y - x)) = \{x, y, z\}$

5 λ-calculus reduction [6pts]

In class, we talked about β-reduction (reduction via capture-avoiding substitution) and α-conversion (variable renaming). These two rules can be used to fully describe the semantics of λ-calculus and evaluate (a la term rewriting) any λ program.
There is, however, a third rule called η-conversion that is often used in compiler optimizations and, in Haskell, for point-free programming (as we will see). -conversion says that you wrap any function term in a lambda expression: $\lambda x.e\ x =_\eta e$ if $x \notin FV(e)$. Since this is just another equation in our toolbox, you can also drop abstractions according to η-conversion.
In this problem you will reduce the following term $(\lambda x.\lambda y.x\ y)(\lambda x.x\ y)$ according to our equations theory. At every step note if you are doing a β-reduction, α-conversion, or η-conversion. You should do all possible reductions to get the shortest possible expression. There are multiple ways to reduce this expression; below you will consider two different ways, both starting with an α-conversion. You may not do multiple steps at once or use more lines than allocated.

First approach:

$$\begin{aligned} & (\lambda x.\lambda y.x\ y)(\lambda x.x\ y) \\ =_\alpha\ & (\lambda x.\lambda z.x\ z)(\lambda x.x\ y) \\ =_\beta\ & \lambda z.((\lambda x.x\ y)\ z) \\ =\ & \lambda z.(z\ y) \end{aligned}$$

Second approach:

$$\begin{aligned} & (\lambda x.\lambda y.x\ y)(\lambda x.x\ y) \\ =_\alpha\ & (\lambda x.\lambda z.x\ z)(\lambda x.x\ y) \\ =_\eta\ & (\lambda x.x)(\lambda x.x\ y) \\ =\ & \lambda x.(x\ y) \end{aligned}$$

Acknowledgements