CSCI 3155: Lab 3 (due February 23)

Ian Smith & Katie Gray

2. (a) *Give one test case that behaves differently under dynamic scoping versus static scoping (and does not crash). Explain the test case and how they behave differently in your write-up.*

   To write a test that behaves differently under dynamic scoping versus static scoping, we use the fact that the dynamic scope extends its environment map and passes it into recursive calls of eval but that these extended maps are never passed back up. On the other hand, step (the static scoping method) replaces variables at the place in the code they are written, so there is no worrying about the replaced variables not being passed back up. Here is a test case we can use to look at the difference:

   ```
   const x = 3;
   const g = function (x) { return function (y) {return x}};
   g(7)(1)
   ```

   For this test case, dynamic scoping will return 3 and static scoping will return 7. Let us see why.

   For the dynamic scoping case of our `eval` function, our environment map is extended at `const x = 3`. We then call `eval` with env = {x:3} on the rest of the program. In the second line, the anonymous function (which returns a function) is bound to g with another extension of the environment. `eval` is then called again on the remainder of the program, `g(7)(1)` with `env = {x:3, g: function (x) { return function (y) {return x}}}`. Since `g(7)(1)` is equal to AST of `Call(Call(Var(‘‘g"), N(7.0), N(1.0))`, this is evaluated by looking up `g` and the mapping {x :7} is passed along with the inner function `function (y) {return x}` to `eval`. Because this function is just a value, it is returned (as is) by `eval` and the mapping {x:7} is lost. This inner function is returned up the stack and `eval` is called on the inner function but the original mapping of `{x:3}` is the environment that is used. The entire `eval` thus evaluates to 3 when `eval` is called on `function (y) {return x}` with argument $y = 1$ and map `env = {x:3}` since x is only defined in the global scope for this dynamic scoping evaluation.

   On the other hand, static scoping implemented via the step function actually replaces the `x` inside the function bound to `g` with value `N(7.0)`, so `g(7)` returns `function (y) {return 7}` and when this returned function is called with argument 1, it simply returns 7 because body of the function does not depend on the parameter `y = 1` that is passed into the function. Thus, we have two different answers.

3. (d) *Explain whether the evaluation order is deterministic as specified by the judgment form* $e \to e'$.

   The inference rules given for the small-step operational semantics do indeed specify a deterministic evaluation order. We can see that this is the case by observing the Search Rules. In the search rules, the order of evaluation is explicit. For `Unary` ops the evaluation order is trivial as the single operand is simply stepped until it reaches a value. However looking at `SearchBinary` rules, we can see that the left operand must be stepped to a value before we begin stepping the right operand. This is always the case and thus the evaluation order of step on `Binary` operation operands (step left to value then step right to value) is always deterministic. The Do Rules then provide deterministic evaluation rules once the expressions are stepped to values. We can even see that in the search rules for `If, Const, Call` that the first expression e_1 is stepped to a value before anything is done with e_2 or before any Do Rule is called. Thus, these inference rules specify a deterministic evaluation order for our small-step semantics of JAVASCRIPTY

4. **Evaluation Order** *Consider the small-step operational semantics for JAVASCRIPTY shown in Figures 7, 8, and 9. What is the evaluation order for* $e_1 + e_2$*? Explain. How do we change the rules obtain the opposite evaluation order?*

   Looking at the small-step operational semantics for `JAVASCRIPTY` shown in Figures 7, 8, and 9, we can see a clear evaluation order for $e_1 + e_2$. Based on the rule SearchBinary1, we can see that for $e_1 \text{bop} e_2$, we step e_1 to a value before anything else. Thus, for $e_1 + e_2$ we step e_1 to a value v_1 first. Then we have the expression $v_1 + e_2$. According the the search rule, SearchBinaryArith2 for which our expression satisfies the premises (v_1 is a value and the binary op is in $\{+, -, *, /, < . <=, >, >=\}$), we then step e_2 until it is a value. We step e_2 to a value v_2 in any number of steps according to the rule SearchBinaryArith2 and we are left with $v_1 + v_2$. With both `expr`'s now stepped to values, one of three DoRules applies.

   Case 1: Neither v_1 nor v_2 is a string, so we apply rule DoPlusNumber and call toNumber on both v_1 and v_2 and add the resulting floating point numbers to some value n'. This is the last step in this evaluation then because we have reached a value.

   Case 2: v_1 is a string, so we apply rule DoPlusString1 and convert v_2 to a string with `toStr` and then concatenate the two strings to some `str'` and return this as the value of the fully evaluated expression $e_1 + e_2$.

   Case 3: v_2 is a string (there is no real difference in Case 2 and 3, if one of the values is a string, we convert both to strings), so we apply rule DoPlusString2 and convert v_1 to a string with `toStr` and then concatenate the two strings to some `str'` and return this as the value of the fully evaluated expression $e_1 + e_2$.

   We can change the rules to obtain the opposite evaluation order by changing SearchBinary1 to

$$\frac{e_2 \to e_2'}{e_1 \; bop \; e_2 \to e_1 \; bop \; e_2'}$$

and also changing SearchBinaryArith2 to

$$\frac{e_1 \to e_1' \;\; bop \in \{+, -, *, /, <, <=, >, >=\}}{e_1\, bop\, v_2 \to e_1'\, bop\, v_2}$$

.

5. **Short-Circuit Evaluation**

   (a) ***Concept.*** *Give an example that illustrates the usefulness of short-circuit evaluation. Explain your example.*
   The usefulness of short-circuit evaluation is that we can stop evaluation early and not have to perform the additional computation from if had there not been a short-circuit.
   An example of this is:

```
true ||
  (const g = function (x) { return function (y) {return x*x*y}};
    g(5)(8))
```

   In this case, we can short-circuit before having to evaluate anything to the right of the &&. While in this case, we don't save too much computational time, if the right side were more complex or we were performing this evaluation many times it would add up. The short-circuiting in the example above comes from the do rule DoOrTrue, because v_1 in $v_1 \&\& e_2$ is true, we can "ignore" e_2 and simply evaluate to v_1.

   (b) ***JAVASCRIPTY.*** *Consider the small-step operational semantics for JAVASCRIPTY shown in Figures 7, 8, and 9. Does e_1 && e_2 short circuit? Explain.*
   Yes, e_1 && e_2 does short-circuit when evaluating. In the small-step operational semantics, we can see that the inference rules specify that e_1 && e_2 does the minimal amount of evaluations. We see that SearchBinary steps e_1 to a value v_1 before doing anything with e_2. Once we are stepped to v_1 && e_2, we apply one of two DoRules. If v_1 is false-y (toBoolean(v_1) returns false) then DoAndFalse applies and we short-circuit and step to v_2 (without ever stepping any part of e_2). However, if v_1 is truthy then DoAndTrue applies and we **must** step e_2 further. (e_2 will then be stepped to a value by the inference rules that apply to it). This is an example of a short circuit because we make take the necessary steps of e_1 to v_1 and then only step e_2 if it is necessary and v_1 && e_2 can't be determined from knowing v_1 alone.