CSCI 3155 Spring 2018 Lab 3 Writeup
Gordon Gu, Arthi Kumar (02/24)

1.
```
val x = 1

def funct1(): Int {
 val temp1 = x + 1
    temp1
}

def funct2(): Int {
 val x = 2
    val temp2 = funct1()
    temp2
}
```

With the given test case, both `funct1` and `funct2` will return a value of 2 when executed with static scoping. When executed with dynamic scoping, `funct1` will return a value of 2 while `funct2` will return a value of 3. This is because under dynamic scoping, variables ignore local scoping when nested in functions, and variable definitions are dependent on variables of the same name under local scoping within functions, while under static scoping, variables contained within their scopes and are unable to affect variable definitions outside of their respective scope.

2. The evaluation order is deterministic as specified by the judgment form $e \to e'$ because according to the search rules, the order of evaluation is explicit. In viewing the `SearchBinary` rules, it can be seen that the left operand must be stepped to a value before the expression can step to the right operand. Afterwards, the Do Rules also provide deterministic evaluation once the expressions have stepped to values. An example from the code would be the `If`, `Const`, and `Call` functions which use the first expression e_1 and step it to a value before interaction with e_2 is done and before and Do Rules are called.

3. The evaluation order for e_1+e_2 is stepping e_1 to a value v_1 first using `SearchBinary1`, then using the new expression $v_1 + e_2$ to step e_2 any number of steps until it is a value v_2 using `SearchBinaryArith2`, leaving the new expression $v_1 + v_2$. Afterwards, either `DoPlusNumber`, `DoPlusString1`, or `DoPlusString2` is applied depending on the variable types of v_1 and v_2. To obtain the opposite evaluation order, the rules of `SearchBinary1` and `SearchBinaryArith2` can be changed to the following respectively: $\dfrac{e_2 \to e_2'}{e_1(bop)e_2 \to e_1(bop)e_2'}$ and $\dfrac{e_1 \to e_1'(bop) \in \{+, -, *, /, <, <=, >, >=\}}{e_1(bop)v_2 \to e_1'(bop)v_2}$

4. (a) Short-circuit evaluation is useful because a given expression can stop evaluation early, allowing it to not perform any additional computation, which would be required if the given expression was unable to short-circuit. An example of short-circuit evaluation would occur with an efficient implementation of interpreting e_1 && e_2. If e_1 evaluates to a value v_1 which is $true$, then the only remaining step is to simply return e_2, because if e_2 is either $true$ or $false$ then the returned value would be correct without ever comparing e_1 and e_2 with an AND operation. This would allow the expression to save computation time in certain cases.

   (b) e_1 && e_2 does short circuit because the inference rules state that the expression must take the necessary steps to step e_1 to v_1 while only stepping e_2 if necessary, doing the minimal amount of evaluations and specifying that v_1 && e_2 cannot be determined from knowing v_1 alone. The Search Rules specify that `SearchBinary` steps e_1 to a value v_1 before interacting with e_2, and once the expression is stepped to v_1 && e_2 either `DoAndFalse` or `DoAndTrue` is applied depending on the boolean returned. If v_1 is false `DoAndFalse` is applied and the expression short-circuits and steps directly to v_2 without ever stepping any part of e_2. If v_2 is true `DoAndTrue` is applied and the expression is required to step e_2 further.