CS224 – Scoping and Evaluation

Purpose: Scope is the area of a program in which an identifier is known. We will examine two ways in which scope can be handled by programming languages. Additionally, we will examine ways in which expressions can be evaluated.

Knowledge: This activity will help you become familiar with the following content knowledge:

- The difference between static and dynamic scoping
- The difference between eager and lazy evaluation

Activity: With your group perform the following tasks and answer the questions. You will be reporting your answers back to the class in 30 minutes.

1. Here is a weird (but legal!) Haskell program:

   ```
   square square = square * square
   ```

   Can you figure out what the result would be for the expression `square 3` ?
   There are two different scopes that both have defined the identifier `square` and there is no confusion between them. Explain what is happening with this example.

2. Here is another example that makes the scoping more explicit.

   ```
   let
       n=1
   in
      let
         n=2
      in
         3*n
   ```

   What do you think is the value of this expression? It all depends on which of the two `n`'s we are referring to when we compute `3*n`. You can always try this out in Haskell and see. This is called block scoping since we are explicitly defining a block of code with the `let` in which we are defining a variable. In Java, a block is created with curly brackets.

3. Here is another look at scoping:

```
scopeTest =
   let x = 3 in
       let f1 = x
            f2 =
                 let x = 4 in f1
       in
            f2
```

   Oh this one is evil! We are letting `f1` be the value of `x` and returning this value inside of `f2`. But which `x` is being used in `f1=x` ?!! Is it the `x` that has the value of 4 where `f1` is being called or is it the `x` with the value of 3 where `f1` is being defined? Try this out in Haskell. If we are getting the value from where `f1` is called, this is defined as **dynamic scoping**. It we are getting the value from where `f1` is being defined, this is **static scoping**. Which kind of scoping is used in Haskell? Why is this kind of scoping preferred?

4. Changing gears a bit, we will now examine expression evaluation. Consider the two possible ways that we could evaluate the expression `square(3+4)`:

   Innermost evaluation:
   `square (3+4) => square 7 => 7*7 => 49`

   Outermost evaluation:
   `square (3 + 4) => (3+4) * (3+4) => 7 * (3+4) => 7 * 7 => 49`

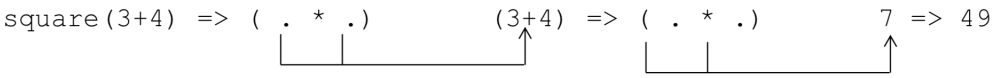
   Which do you prefer, innermost or outermost?

   What about when we evaluate `fst (square 4, square 2)` ?

   Innermost evaluation:
   `fst (square 4, square 2) => fst (4 * 4, square 2) => fst (16, square 2) =>`
   `fst (16, 2 * 2) => fst (16, 4) => 16`

   Outermost evaluation:
   `fst (square 4, square 2) => square 4 => 4 * 4 => 16`

   Outermost didn't work well in the first example `square(3+4)` but what if we represent the reduction with a graph so we can share subexpressions as illustrated below:



   Which do you prefer, innermost (eager evaluation), or outermost graph reduction (lazy evaluation)?