3.7.1 Efficiency

- Provide a template for a program that illustrates the non-linearity of the stack-based cache implementation.

```
(with ((x 4))
      (with (( y 5))
            (with ((x 6))
                  (+ x y)))
```

- Explain briefly why its execution time is non-linear in its size.

  When it gets to the plus operator, it will have a list of binding size N (in this case, 2). We can assume that every binding is used at least once, otherwise there would be no point in using a wit function. Therefore, for each of the N identifiers corresponding to the N bindings, the list of N bindings must be searched, resulting in an order N^2 operation.

- Describe a data structure for a substitution cache that a FWAE interpreter can use to improve its complexity,
  - and show how the interpreter should use it (if the interpreter must change to accommodate your data structure, describe these changes by providing a pseudocode version of the new interpreter).

  A hash map would be a much better data structure to use for substitution. This because a hash map can have constant time data retrieval. However, this would only improve the time complexity in some cases (when there is only one binding / variable name). When there are multiple bindings per variable, if implemented incorrectly, you would have to search through a list to find the latest binding. To avoid this problem, a good alternative would be to use a hash map where variable names map to a stack of bindings. Then whenever a variable found deeper in scope is found that matches the name of an assigned variable, the binding is pushed onto the corresponding stack in the hash map. When a variable is found, the hash map finds the corresponding variable name and uses the first element found in that stack.

Code to insert:

```
if(hashmap.get(bindingName) == null){
      Stack st = new Stack();
      st.push(bin);
      hashmap.put(bindingName, st);
}else{
      hashmap.get(bin.hashCode).push(bin);
}
```

Code to retrieve:

```
if(hashmap.get(bindingName) != null){
      if(hashmap.get(bindingName).size() != 0){
            return hashmap.get(bindingName).pop();
      }
}else{
      throw (error, "no bindings in scope for current
Variable");
}
```

- State the new complexity of the interpreter, and (informally but rigorously) prove it. You don't need to restrict yourself to the subset of Racket we are using in this course; you may employ all your knowledge of, say, Java. However, the responsibility for providing a clear enough description lies on you. Remember that simple code is often the clearest description.

  Order N. For N bindings, (again, assuming they are used a minimum of once) The interpreter would look up the binding from the variable name in constant time. Then it would pop off the first binding value in constant time and substitute it in. Because this has to be repeated for each binding variable, it is order N.

3.7.2 The Bitdiddle Algorithm

- Is Ben right in general? No

- If so, justify. If not, provide a counterexample program and explain

```
{with {x 5}
      {with {x 4}
            {with {f {fun {y} {+ x y}}}
                  {with {x 5}
                        {f 10}}}}}
```

The earliest made binding is not necessarily the one that is still in scope when the function is declared. Like in the example above, x is bound to 5 but then later to 4 before the function is declared.

- A bonus point

  He is wrong because the environment is all about what is in scope at a given time and has nothing to do with the order in which things are declared. As I'll discuss later, I've used first order functions to access a variable in a class in which it was never even declared. Because of this, it's important to evaluate bindings by whether or not they were active at the time a function was declared (if using static scoping) and not simply by the order it was declared.

3.7.3 Application

One time at work, we needed to access a variable inside of a different class and do some manipulations using it. I can't remember the details but we were unable to make it public or use normal getters and setters. Instead, we had that class call a function on our class, passing in a lambda expression that held that variable inside and performed the needed manipulation. Then later, when the original instance had gone out of scope, we were still able to access the variable we needed.
It is also extremely useful for making easy to read code when you only need a simple function as in the case of a onClickHandler or something like that. You can just put a function as the property directly (as in Javascript) rather than putting a name to a short function that would just have to be looked up anyway, Someone who needs to modify the code can see the small function right where it is needed.