CSE 262: Quiz #2  
Due September 30th, 2022 at 11:59 PM

The quiz has TWO questions. Please submit your answer by updating this file in the quizzes folder of your Bitbucket account, and then committing and pushing. You should use as much space as you want for each answer. Please be detailed in your answers. Remember: this quiz is worth 9% of your grade, and you will not receive very many points if you do not give detailed answers.

**Question 1:** In software engineering classes, a popular topic is “patterns,” one of which is the “Singleton” pattern. What is the Singleton pattern? Please be detailed in your answer. Then describe (in text, possibly with some accompanying pseudocode) how you would implement the Singleton pattern in Scheme. (Hint: a function is not able to define new globals, so if you’re going in that direction, you’re probably not thinking about it correctly).

The general idea of Singleton pattern is that a class should only have one instance, and this instance should be globally accessible by others. Traditional class has traditional constructor, and each time when we create a new object by the constructor we will get a new object. However, in the singleton pattern, where we make a private constructor which cannot be called to create a new object outside of the class. Then, create a static instance to call the private constructor and cache it to ensure that the this instance can be accessed by static method. Finally, one static method to return the static instance when it is called. To access the instance, outer member can call static method, and each of them will access the same instance of the class.`

My general idea is to use a closure to lock the instance, and this closure should keep its value unchanged every time when we define a new object to it. Firstly, we assign the instance of class to the result of make-instance by passing the argument ‘singleton, then it will get into the lambda to determine if the instance we stored in the closure is equal to the argument ‘singleton, if so we will return the instance we stored in the closure, if not we can build new one. Then a function which will capture the instance, and return it.

define make-singleton

let ((instance (make-instance 'singleton))

lambda (arg)

if (eq? arg instance)

instance

make-instance 'singleton

define singleton (make-singleton)

(singleton 'oldinstance)

(singleton 'newinstance)

**Question 2:** In our discussion of semantic analysis, we talked about how it can be used to “check” a program (to find semantic errors or produce warnings for programs that are able to be parsed) and also to “transform” a program (typically to make it faster). There are quite a few examples of semantic analysis online and in the book. Study one analysis that falls into the “check” category, and one that falls into the “transform” category. For each, describe it in detail. (Note: if possible, please describe analyses that we did not discuss in depth in class; if that’s not possible, please be sure to go into more detail than what we discussed in class.)

The example for the “check” category I picked is the example in Figure 4.14 in textbook. In the figure 4.14, there is an example of attribute grammar used to decorate the abstract syntax tree. Besides the normal attributes like name, type, and symtab(inherit attribute), there are error\_in, error\_out, errors attributes in the grammar. For each node, nodes will inherit the error\_in attributes from its parent if there is an error in parents, and then pass error\_in all the way down to childs until the null node. Then the accumulated error\_in attributes for all the nodes will flow to the error\_out of null node. To pass the error\_out, each node will pass its error\_out back to its parent node. For example, in the ‘read’ node, if there is an undefined error, grammar will put this error with the error\_in error that “read” node inherit from parent together into its error\_in attribute, and then pass error\_in to its child node. If there is no other errors, this error message will flow all the way down to null node and flow back to root through error\_out. Eventually the accumulated error\_out attributes of the whole tree will be passed back to the error attribute of the root. By doing so, all the error message will be capture without stopping going deeper, which can identify all the errors.

The example I picked for the “transform” category is the data flow analysis which could help to find the flow of data in the program, specifically, the propagation of data in the program. In data flow analysis, we can determine where identifiers are declared, when they are initialized, when they are updated, and who reads (refers to) them. One of the data flow property that can perform the optimization is the live variable which suggests that a variable x is live at some point p if from p to the end the variable x is used before it is redefined, otherwise, it is dead.

For example

X = 3

A = 3 (x is live)

B = x + a (x is live)

A = 80 (x is redefined)

This property is helpful for the dead code elimination where we never use a defined variable.

Another property that can be used to optimize program is the reaching definition, which suggests that a definition D is reaches a point p if there is path from D to p in which D is not redefined.

For example

D1: x = 60

D2: x = x + 3

D3: y = x

In D2, the D1 definition x = 60 is the reaching definition for this node of program. However, in D3, D1 is not since D2 redefined x.

This property is helpful for the constant and variable propagation where we can hold the value of constant and variable until it is redefined