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 Singleton pattern revolves around the idea of creating a single object of a single class that we want to use our Singleton pattern around and ensures that the single object is the only instance of the class that will be made. (I will refer to this single object as the Singleton object). This Singleton object that is created becomes a single source of information that is considered a share resource that many other components of a program can use without recreating that object. All the methods, states, and fields are available to all objects that are a reference to the Singleton object. Since there is only ever one instance of the object created, all the information that is manipulated or added by the references to the Singleton object is retained in the single object. In essence, the Singleton pattern uses a single object of a class to hold and use all the information of that class.

In order to implement the Singleton pattern in Scheme, one would have to define a Singleton function that acts like a Singleton object in object-oriented programming. In Object-oriented programming, the single object is created within the class and declared static so that it belongs to the class, instead of instances of that class. Furthermore, its constructor is made private so that no other instances of the class could be made. The way to retrieve the single object is through the get method which returns the single object.

PSUEDOCODE:

class SingletonClass

// creating static SingletonClass obj so that it belongs to the class

public static SingletonClass obj = new SingletonClass()

// creating private constructor so that no other instances could be made

private SingletonClass ()

// getter method to get the Singleton object

public static SingletonClass getObj()

// return obj

return obj;

endClass

In order to make a Singleton function act similar to a Singleton object, we need to first realize some things about Scheme. Since Scheme is a functional programming language, we cannot create new instances of an object, let alone create one. Therefore, the function itself is the only one that exists. The next thing we have to notice is that Scheme is statically scoped, meaning all of the variables are defined and binded within the block of code that they were created in. This also means that when we are trying to rebind the value of a variable within a nested function, the nested name masks the name that appeared in the enclosing scope. Therefore, the value is not stored. This is because the value is binded to the nested scope name, not the enclosing scope name. If we want to manipulate the information, we have to fix this issue. We must find a way to change the field values of the Singleton function while using the nested functions. One way to do this is with the function ‘set!’. ‘set!’ allows us to bind a value to a variable in the enclosing scope.

Another thing to note is that Scheme keeps track of the bindings allocated in the scope of a function, which don’t die until the program terminates. Therefore, when we call a function, we will have a connection to all of the bindings that were previously in that function. We will therefore define our Singleton function in the same scope as the scope that we will be referencing it in; this allows us to define functions that can reference our Singleton function to manipulate and access its fields and methods which makes it function like global variables and global functions. Below is the pseudocode for an example Singleton function with comments that describe their scopes and functionality.

PSUEDOCODE:

// SingleP is a function that anything in the global here can access.

(define SingleP

// create fields at the very top so each nested function can access it.

(let ((field1 0) (field2 0))

// create methods to get field1 and field2 as well as any functions that want to rebind the values

// get value of field1

(define (getField1) field1)

// get value of field2

(define (getField1) field2)

// method3 can be anything u want to alter the value. Example method I have here adds amt to field1 and field2.

// the set! function sets the sums to the field1 and field2 names in the enclosing scope

// because the field1 and field2 in the current add function scope are masks.

(define (add amt) (set! field1 (+ field1 amt)) (set! field2 (+ field2 amt)))

// need a function to know what function to apply

(define (method meth . amt)

(case meth

((getField1) (apply getField1 amt))

((getField2) (apply getField2 amt))

((add) (apply add amt)

)

)

(lambda () method)

)  
)

// define functions and add value and both functions that refer to SingleP will have access to the same information.

// Since all these functions are in the same global scope, they can access SingleP function

// function 1 that references singleton (SingleP) function

(define a (SingleP))

// function 2 that references singleton (SingleP) function

(define b (SingleP))

(display (a 'getField1)) ; 0

(display (b 'getField2)) ; 0

(display "\n")

// adding 20 to field1 and 1 to field2

(b 'add 20)

(display (b 'getField1)) ; 20

(display (a 'getField2)) ; 1

**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.)

Semantic analysis that falls into the “check” category prevents errors that pass through the syntactic analysis phase to ensure our program works logically. One example of semantic analysis that falls into the “check” category, is the analysis of performing checks after parsing on things such as visibility modifiers. This is also a static check because it checks for errors during compile time by analyzing the program. For example, in Java there are visibility modifiers such as “public”, “static”, “private”, “protected”, “final”, and “synchronized”. These visibility modifiers cannot repeat, and some with one modifier cannot also be another modifier. For example, a private method cannot also be a public method, and there cannot be a repetition like private private method. The normal parsing tree may not exactly detect these issues because it is not part of its grammar. To detect these issues, a separate grammar must be made to check after parsing to ensure a properly built data structure. The separate grammar will prevent these visibility modifier issues by checking whether or not after a visibility modifier, something that is valid follows. If it is not something valid, it would throw an error. Otherwise, it would function normally. There are many more semantic analysis examples that are like the visibility modifiers that falls into the “check” category. One similar example is detecting types. For example, in Java, declaring a variable “c” with type char and giving it an “int” could still pass through the syntactic analysis phase, but it is the semantic grammar that spots the error at compile time. There are many more examples, but each example of a semantic analysis that falls into the “check’ category is trying to ensure our program works logically.

A semantic analysis that falls into the “transform “category is usually used to implement a sequence of optimizing transformations on a program to produce a more efficient semantically equivalent program. An example of semantic analysis that falls into the “transform” category is the analysis of optimizing parsing trees by transforming them into AST trees. Given a parsing tree, the parsing tree can be condensed into smaller and simpler parsing trees which are much more efficient to run while having the same semantically equivalent output. To do this, the AST abstracts out certain details and has just enough of the original parsing tree’s information to build a proper data structure with equivalent semantics. It does this by being designed to have certain properties and following certain methods. One method is recognizing redundant non-terminals that lead to a constant. For example, E ->T and T-> 7 which can just be E->7. Another thing about AST is that it groups inessential delimiters and punctuation, such as parenthesis, that is part of our syntactic grammar which condenses the number of nodes it took up in the original parsing tree. This property can be very helpful as it condenses a very large parsing tree with a bunch of delimiters and punctuation to something much smaller and more efficient that already knows the delimiters and punctuation. Another AST property is containing extra information about the program. This may be helpful for associating information with nodes to provide more efficient information. One example would be storing the position of elements which could help detect where errors occur. Knowing where errors occur could help the compiler print error messages associated with where it occurred. There are many more properties that the AST has and methods that it follows, but it is important to understand how ASTs are designed. Their design is closely linked with the design of a compiler to notice the small details that can be condensed. Though across almost all of them, they follow a set of important requirements. One of the most important ones is properly storing variable types, location of declaration, order of executable statements. This requirement allows the AST to be a much more condensed version of the original parsing tree and perform operations efficiently because it has access to additional information when performing its operations. AST’s are a very useful semantic analysis that falls into the “transform” category of its properties because of its ability to condense parsing trees into smaller and more efficient semantically equivalent trees.