Steven Meyers

Professor Buffenbarger

CS354

10/26/20

Textbook Assignment 2

1. The main difference in the allocation of local variables in these languages is how they are scoped. Algol and its descendants are often statically scoped. Being statically scoped allows the program to deallocate local variables at the end of their block because they are no longer accessible. Since you must leave a block to move up to the higher scope, it makes sense to use a stack and just take off the local variables off the top when leaving the block, making the higher scopes variables visible once again. Lisp and some of its descendants are dynamically scoped, so local variables being stored in the heap is a must since the program can not tell exactly what scope and reference environment will always be used. To prevent a lot of hanging references, the program usually hangs on to at least a little of the information of the local variable in the heap to point to if that variable is called. This is evident in the pairs in Scheme being stored in the heap with a pointer to the beginning of the pair. It is only deallocated once the program is sure that another reference to it does not exist.

   Java:

```
Public class javaClass {
        Public static void main(String[] args) {
                int x = 1;
                if(x==1) {
                        launchRocket();
                }
        }
        Public static void insideclass() {
                Int x = 0; //If x was static in the whole class,
        }

}
```

   Scheme:

This program is simply my super duper program, but it would not work on a stack allocation because of how it has to go into deep into the pairs and print the cars multiple times, which if implemented through a stack, would print the whole pair instead of car first then cdr second.

```
;;Dupes the atom from a pair and cons them together to form a list with count number dupes.

(define (multiplier source count remaining)

        (if(zero? remaining)

                (super-duper (cdr source) count) ;;moves to the next pair in the list
```

```
                (cons (super-duper (car source) count) (multiplier source count (- remaining 1
))))) ;;loops and cons the atom

;;Filters out pairs and nonpairs

(define (super-duper source count)

        (if(pair? source)

                (multiplier source count count) ;;looks down

                source

        )

)
```

2. Java:

```
Public static void main(String[] args){
        Int x = 20;
        diffScope();
        System.out.println(x);
}
private void diffScope() {
        print(x); //this will return an error despite the variable still being live in main.
}
```

Java 2:

```
Class className {
        Private int x = 20;
        Public void xMethod() {
                X = 40; //this hides the main private int x, cause if this method is not called, the
original value must be remembered.
        }
}
```

C:

```
Int main() {
        Int x = 20;
                {
                        Int x = 30; //outer block x hidden here
                }
}
```

3. At the inner() print, it is referencing middle b, main a. At the middle() print, it is referencing middle b, inner a. At the main() print, it is referencing middle b, inner a. For the C language with nested subroutines, it would print out: 1,1 3,1 3,1. For Modula 3, it would print out: 3,1 3,1 3,1. This is if all references to a and b have access to the scope above them and reference the same variable. If not, C would print out: 1,1 1,1 1,2 and Modula 3 would print out: 3,1 1,1 1,2.

4.
    a. Brad is not deallocating the previous memory from L when he reassigns a new list. This means there is a lot of allocated memory that is never being deallocated leading to the eventual memory crash.
    b. He needs to create a new list object T rather than a pointer, as the memory that the pointer points could be rewritten between the delete list call and the L = T call. This means that sometimes the program will assign correct data to L, but it can also assign the corrupted over written data to L.
5. Static scoping prints out: 1121. This because the only x that gets called with a value of two is within the second() function. This is immediately put out of scope when it calls print_x() again, which prints the global x as the second() x has been destroyed already.
   Dynamic scoping prints out 1122 because the second x is the most recent x put on the stack and so it prints out that value rather than the first x.
6. I'm going to be honest, I have no clue really and a lack of time to work on this project. I'll take very in depth notes when we cover this on Monday.