ANTON LESLIE Cs354

Ta2

3.2) In Pascal and Java, local variables are allocated on the stack. program pascal Example;

var

x, y : integer;

begin

x := 5;

y := x + 3;

writeln(y);

end.

For example, in Java:

public class Example {

public static void main(String[] args) {

int x = 5;

int y = x + 3;

System.out.println(y);

}

}

In Scheme, local variables are typically allocated on the heap, since Scheme is a dynamically-typed language with garbage collection. For example:

(define (factorial n)

(if (= n 0)

1

(\* n (factorial (- n 1)))))

The differences in variable allocation are largely due to the different design goals and philosophies of the languages. Pascal and Java were designed to be statically-typed languages with predictable memory usage, so stack allocation of variables was a natural choice. Scheme, on the other hand, was designed to be a flexible language with dynamic memory management, which led to heap allocation of variables.

3.4)  
 example in Java that demonstrates the concept of live, but invisible, variables: Code:

1 public class

Example {

2 private int a = 1;

3 private int b = 2;

4

5 public static void main(String[] args) {

6 Example ex = new Example();

7 ex.middle();

8 ex.outer();

9 }

10

11 public void middle() {

12 int b = a;

13 inner();

14 }

15

16 public void inner() {

17 System.out.println("a: " + a + ", b: " + b);

18 }

19

20 public void outer() {

21 a = 3;

22 inner();

23 System.out.println("a: " + a + ", b: " + b);   
24 }

25 } In this example, we have three methods: main(), middle(), and inner().

The Example class has two private instance variables: a and b, both initialized to 1 and 2, respectively. In the middle () method, we create a local variable b and assign it the value of the instance variable a. We then call the inner () method, which prints out the values of a and b. In the outer () method, we change the value of the instance variable a to 3 and call the inner() method again, which now prints out the updated value of an as well as the original value of b. The important thing to note here is that although we have two variables named b in the code (one instance variable and one local variable), we can access both by using different scopes. The instance variable b is accessible throughout the entire class, while the local variable b is only accessible within the middle () method.

3.5) The first print (located in inner()) will print out the b from middle() and the a from main(), the second print will print out its own b (same as the previous b) and main’s a (not inner()’s because middle() can only look up in scope to main() and not down into inner()), the last print will print out both of its own a and b values(i.e. print #1: 1,1; print #2: 1,1; print #3: 1,2). This is because as previously mentioned, in a language like C, children can inherit parent variables but parents can’t access child variables.

However, if we were to apply this to C#, while I do not have any personal experience with it, just looking at the scope rules for it I’d assume that it’d throw a syntactical error before the first print since the lifetime and scope of C# variables extend until it hits the end. Therefore declaring another variable of the same name (i.e. “b”) will conflict with the first declaration of b in main().

Looking at Modula-3, and the information on it, a program written in it should have the same structure as that of a C-program. That is to say, it’d work roughly the same as the C, or at least print the same things as C in the same order. Even with its specific declaration rule, Modula-3 would still need at least a value or a type to be declared, so even if we didn’t tell the middle() b and inner() a what type to be, just by passing the values we want to it should allow Module-3 to interpret what type these variables ought to be.

3.7) a) Unfortunately, the way that C allocates and frees memory does not align with the way Java goes around doing it (manual vs auto-garbage-collected). Therefore, whenever the size of the list\_node is reached Brad’s code will breach the memory limit if that memory block isn’t freed and will crash the program.

b) She’ll tell him that when the program reaches the last line of his new code, it’ll pass the value of T. But this value is not what Brad expects, because T is a pointer its value is not the reversed list structure but the address to that list.

3.14) With a statically scoped program, such as C, the pseudocode would print out: 1,1,2,2.

With a dynamic program given that a variable is considered active whenever the function it’s located in is in execution. My guess would be that if we were to do a direct assignment, like in C, it’d probably have a hard time distinguishing which ‘x’ value to use in setx() as it wouldn’t consider them to be the same and the initial ‘x’ wouldn’t be live during the execution of setx()

3.18) An instance in which overloading may not be advantageous, over just setting a variable to be a float (and coercing integers) would be whenever you’re dealing with square roots in calculations. For example, doing a quadratic formula calculation could take parameters *a, b,* and *c* and these parameters could very well be integers, that we’ll coerce into being doubles/floats.