David Morris – Kyle Goodwill – Weronika Kowalczyk
Professor Eileen Head
Programming Languages – Homework 7
Due March 17, 2015

1)

Static Scoping:

1
1
2
2

Dynamic Scoping:

1
1
2
1

This change in the last digit printed occurs due to the procedure “second”. With Static scoping, any calls within that function will still reference the original declaration of x: integer –global, while with Dynamic Scoping the set_x(2) and print_x both reference the locally declared x:integer.

2)

Static Scoping: 54

Dynamic Scoping Deep Binding:

Deep Binding locks in the variables required at time of passing, so while Y was set to 3, the X value referred to by add is still 4, adding up to 7.

Dynamic Scoping Shallow Binding:

Shallow Binding locks in the variables at time of execution, so both assignments of Y := 3 and X := 1 are referenced, causing the result to be 4.

3)

Each nested subroutine has its own activation record that is kept throughout the entire execution of the program.

4)

The inconsistency of this program originates from line 7, “while (i++ <512) {“. Depending on the architecture, the Assembly could implement “i++” as Compare First then Increment, Increment first then Compare, or anything else under the sun. This could cause all the insecurities described in part B of the question.

5)

A)

A declaration introduces a name and indicates its scope, but may omit certain implementation details. A definition describes the object in sufficient detail for the compiler to determine its implementation.

B)

a. char* a; --> declaration
b. struct S; --> declaration
c. struct S1 {int x;} --> definition
d. typedef int* string; --> declaration
e. int foo(int i); --> definition
f. extern double x; --> declaration

g. void function(){} --> definition

6)

What does this program print?

```
using System;
public delegate int UnaryOp(int n);
        //type declaration: UnaryOp is a function from ints to ints

public class Foo{
        static int a = 2;
        static UnaryOp b(int c){
                int d = a+c;
                Console.WriteLine(d);
                return delegate(int n) {return c + n;};
        }
        public static void Main(string[] args){
                Console.WriteLine(b(3)(4));
        }
}
```

It'll print:

```
5
7
```

int a is statically allocated. The variables n, and a, are allocated in the heap, because they are static. Variables d and c are on the stack because they are local variables.

7)

These statements are not contradictory because, while the language specifications might stay the same, implementation across different architectures is bound to change.

8)

( - 16 ( 9 4 ) ) Will evaluate to 11 when executed, but when parens are removed the execution path becomes unclear as ( - 16 9 4 ).

Reworded:

Issues of precedence and associativity do not arise with prefix or postfix notation on binary operators.

9)

Is &(&i) ever valid in C? Explain.

It is not valid because &i would return the address of i, and addresses do not have addresses, so asking for the address of the address of i would give you an error. Even if i was declared as a double pointer (ex. int** i), the &i would give you the address of the pointer and asking for the address of the address would still give you an error.

10) (++/--) > (+) > (&&) > (=)
Left → Right

| | |
|---|---|
| int a[] = {0, 1, 2, 3, 4}; int n = 2; | **n=2** |
| **int x = (a[n++] = a[n--] + a[++n] && a[n]);** | x=? |
| A[2] = a[n--] + a[++n] && a[n] | N=3, X=? |
| A[2] = a[3] + a[++n] && a[n] | N=2, X=? |
| A[2] = a[3] + a[3] && a[n] | N=3, X=? |
| A[2] = a[3] + a[3] && a[3] | N=3, X=? |
| A[2] = 6 && a[3] | N=3, X=? |
| A[2] = 1 | N=3, X=? |
| 1 | N=3, X=1 |

Right → Left

| | |
|---|---|
| int a[] = {0, 1, 2, 3, 4}; int n = 2; | **n=2** |
| **int x = (a[n++] = a[n--] + a[++n] && a[n]);** | x=? |
| A[n++] = a[n--] + a[++n] && a[2] | N=2, X=? |
| A[n++] = a[n--] + a[3] && a[2] | N=3, X=? |
| A[n++] = a[3] + a[3] && a[2] | N=2, X=? |
| A[2] = a[3] + a[3] && a[2] | N=3, X=? |
| A[2] = 6 && a[2] | N=3, X=? |
| A[2] = 1 | N=3, X=? |
| 1 | N=3, X=1 |