Name: Milson Munakami

Student ID: 114012811

Course: CS 354 Programming Languages

Instructor: Dr. Jim Buffenbarger

Assignment #2: Names, Scopes, and Bindings

TA2:

Exercise 3.2

In Fortran 77 there is no recursion, so the local variables of a given method can never have more than one instance. Whereas Algol and its descendants (e.g., Pascal and Ada) require a stack to accommodate multiple copies of variables. The following example code would not work correctly if we have used statically allocated local variables:

```
function Summation (num : integer) : integer;
    begin
        if num = 1 then
            Summation : = 1
        else
            Summation : = Summation(num-1) + num
    end;
```

Here, function Summation needs to remember the value of *num* during the recursive call.

As we can see Pascal has limited extent for local variables i.e. that the values of those variables are lost when control leaves the scope in which they were declared. Whereas, Lisp and Scheme require allocation in the heap to accommodate unlimited extent of variables. The following code would not work correctly if it use stack-allocated variables:

```
function sum n(n) return { function(k) return n + k }
```

Here the function *sum* whenever called, it will add to its argument k the value n originally passed to *sum* n. That value (n) must remain accessible as long as the function returned by *sum* n remains accessible.

Exercise 3.4

In C, a static variable declared inside a function is live but not in scope when execution is not inside the function.

```
void foo()
{
   static int x = 5;
   x++;
   printf("%d", x);
}

int main()
{
   foo();
   foo();
   return 0;
}
```

(b) In C++, non-public fields of an object of class Rectangle are live but not in scope when execution

is not inside a method of Rectangle.

```
class Rectangle {
        int width, height;
        public:
                void set_values (int,int);
                int area() {return width*height;}
};
```

Exercise 3.5

The scope of the variable is based on its declaration order. I think the value of a and b will be as follows:

In C,

1. Print from inner() → 3,1
2. Print from middle() → 3,1
3. Print from main → 1,2

In c#, the names must be declared before use, but the scope of a name is the entire block in which it is declared so output will be:

1. Print from inner() → 3,1
2. Print from middle() → 1,1
3. Print from main → 3,1

Whereas, in Modula-3, it will print:

1. Print from inner() → 1,2
2. Print from middle() → 1,2
3. Print from main → 1,2

Exercise 3.7

(a) The *reverse_list* routine produces a new list, composed of new list nodes. When Brad assigns the return value back into L he loses track of the old list nodes, and never reclaims them that means, his program has a **memory leak**. After a number of iterations of his main loop, Brad has exhausted the heap and his program can't continue.

(b) While the call to *delete_list* successfully reclaims the old list nodes, it also reclaims the widgets. The new, reversed list thus contains **dangling references** which are unused locations in the heap that may be used for newly allocated data, which may be corrupted by uses of the elements in the *reversed_list*.

Exercise 3.14

With **static scoping** it will print **1 1 2 2**. While with **dynamic scoping** it will print **1 1 2 1**. The reason for this difference in output is because of *set_x* sees the global x or the local x declared inside the *second* procedure when it is called.

Exercise 3.18

With **shallow binding**, *set_x* and *print_x* always access foo's local x.

The output will be **1 0 2 0 3 0 4 0**

With **deep binding**, *set_x* accesses the global x when n is even and foo's local x when n is odd. Similarly, *print_x* accesses the global x when n is 3 or 4 and foo's local x when n is 1 or 2.

The output will be **1 0 5 2 0 0 4 4**