Ahram Kim

Jim Buffenbarger

CS354 - 001

October 24, 2017

Textbook Assignment 2

1. Question 3.2. In Fortan 77, local variables were typically allocated statically. In Algol and its descendants (e.g., Ada and C), they are typically allocated in the stack. In Lisp they are typically allocated at least partially in the heap. What accounts for these differences? Give an example of a program in Ada or C that would not work correctly if local variables were allocated statically. Give and example of a program in Scheme or Common Lisp that would not work correctly if local variables were allocated on the stack.

: As Fortran 77 has the scarcity of recursion so more than one live instance of local variables of a given subroutine can never be places. To accommodate multiple copies Algol and its descendants require a stack. The following will not work perfectly with statically allocated loval variables:

<function sum(f, low, high) if low = high return f(low) else return f(low) +(f,low + 1,high)>

During the recursive call, it is needed to remember the value for function sum.

For the local variables, Algol and its descendants have limited extent. The values of those variables are destroyed when the control leaves the scope in which they were declared. To accommodate unlimited extent Lisp and its descendants require allocation in the heap. With the stack-located variables, the following would not work properly,

<function add n(n) return {function(k) return n+k}>

Here the intention is to return a function which and when called, will add to its argument k the value n basically passed to add n accessible as long as the function returned by add n remains accessible the value (n) must need to remain accessible.

2. Question 3.4. Give three concrete examples drawn from programming languages with which you are familiar in which a variable is live but not in scope.

: A global variable named x will be live but not in scope when executing P whether procedure P declares a local variable named x in Ada. A global variable which is declared in a module is live but not in scope when the execution is not inside the function in Modula-2. In C, a static variable which is declared inside a function is live but in scope when execution is not inside the function. In C++, non-public fields of an object of class C are live but in the time of execution, it is, not inside a method of C.

3. Question 3.5. Consider the following pseudocode: Suppose this was code for a language with the declaration-order rules of C(but with nested subroutines) - that is, names must be declared before use, and the scope of a name extends from its declaration through the end of the block. At each print statement, indicate which declarations of a and b are in the referencing environment. What does the program print (or will the compiler identify static semantic errors)? Repeat the exercise for the declaration-order rules of C# (names must be declared before use, but the scope of a

name is the entire block in which it is declared) and of Modula-3 (names can be declared in any order, and their scope is the entire block in which they are declared).

```
1.  procedure main()
2.      a : integer := 1
3.      b : integer := 2
4.      procedure middle()
5.          b : integer := a
6.          procedure inner()
7.              print a, b
8.      a : integer := 3
9.      -- body of middle
10.     inner()
11.     print a, b
12. -- body of main
13. middle()
14. print a, b
```

: C - integer a declared in main as 1.

integer b declared in main as 2.

function middle is executed.

a new local variable b is set to global variable a as 1.

a new local variable a is set to 3.

function inner is executed and returns a as 3 and b as 1.

prints a as 3 and b as 1.

lastly, main prints a as 1 and b as 2.

C# - integer a declared in main as 1.

integer b declared in main as 2.

function middle is executed.

a global variable b is set to global variable a as 1.

a global variable a is set to 3.

Function inner is executed and returns a as 3 and b as 1.

prints a as 3 and b as 1.

lastly, main prints a as 3 and b as 1.

Modula-3 - integer a declared in main as 1.

integer b declared in main as 2.

Function middle is executed.

A new local variable b is set to global variable a as 1.

A new local variable a is set to 3.

Function inner is executed and returns a as 1 and b as 2.

Prints a as 1 and b as 2.

Lastly, main prints a as 1 and b as 2.

4. Question 3.7. A part of the development team at MumbleTech.com, Janet has written a list manipulation library for C that contains, among other things, the code in Figure 3.16.

    a. Accustomed to Java, new team member Brad includes the following code in the main loop of his program: Sadly, after running for a while, Brad's program always runs out of memory and crashes. Explain what's going wrong.

```
list_node* L = 0;
while (more_widgets()) {
    L = insert(next_widget(), L);
}
L = reverse(L);
```

: A new list, composed of new list nodes are produced by the reverse_list routine. Bard loses track of the old list nodes when he assigns the return value back into L and never reclaims them. In other words, his program has a memory leak. After few of iterations of his main loop, Brad has tired the heap and his program cannot endure.

b. After Janet patiently explains the problem to him, Brad gives it another try: This seems to solve the insufficient memory problem, but where the program used to produce correct results (before running out of memory), now its output is strangely corrupted, and Brad goes back to Janet for advice. What will she tell him this time?

```
list_node* L = 0;
while (more_widgets()) {
    L = insert(next_widget(), L);
}
list_node* T = reverse(L);
delete_list(L);
L = T;
```

: It also reclaims the widgets while the call to delete_list successfully reclaims the old list nodes. The new, reversed list contains dangling references. It mention to the locations that may be used in the heap for newly allocated data that might be damaged by the uses of the elements in reversed list. Brad is lucky that he is not damaging the heap itself but he might face a lot of troubles without Janet's help to figure out the reason that why widgets are changing value spontaneously.

5. Question 3.14. Consider the following pseudocode: What does this program print if the language uses static scoping? What does it print with dynamic scoping? Why?

```
x : integer        -- global

procedure set_x(n : integer)
    x := n


procedure print_x()
    write_integer(x)

procedure first()
    set_x(1)
    print_x()

procedure second()
    x : integer
    set_x(2)
    print_x()

set_x(0)
first()
print_x()
second()
print_x()
```

: It prints 1 1 2 2 with dynamic scoping it prints 1 1 2 1 with static scoping. The difference lies in whether the x declared in second when it is called from second or set x sees the global x.

6. Question 3.18. Consider the following pseudocode: Assume that the language uses dynamic scoping. What does the program print if the language uses shallow binding? What does it print with deep binding? Why?

```
x : integer      -- global

procedure set_x(n : integer)
    x := n

procedure print_x()
    write_integer(x)

procedure foo(S, P : function; n : integer)
    x : integer := 5
    if n in {1, 3}
        set_x(n)
    else
        S(n)


    if n in {1, 2}
        print_x()
    else
        P

set_x(0); foo(set_x, print_x, 1); print_x()
set_x(0); foo(set_x, print_x, 2); print_x()
set_x(0); foo(set_x, print_x, 3); print_x()
set_x(0); foo(set_x, print_x, 4); print_x()
```

: Set_x and print_x always access foo's local x with shallow binding. The program will print 1 0 2 0 3 0 4 0. Set_x accesses the global x when n is even and foo's local x when n is odd with deep binding. Correspondingly, print_x accesses the global x when n is 3 or 4 and foo's local x when n is 1 or 2. The program will print 1 0 5 2 0 0 4 4.