Textbook Assignment #2

Sunday, March 27, 2022 12:31 PM

Edition 4: question 3.2, page 167.

Consider variable allocation. Give only Pascal/Java and Scheme examples. They don't need to compile/execute.

In Fortran 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 an example of a program in Scheme or Common Lisp that would not work correctly if local variables were allocated on the stack.

Prior to Fortran 90, Fortran did not support recursion. As a result, there could never be more than one invocation of a subroutine active at any given time, and a compiler may choose to use static allocation for local variables, effectively arranging for the **variables of different invocations to share the same locations**, and thereby avoiding any run-time overhead for creation and destruction.

Algol and its descendants typically permits recursion which makes static allocation of local variables an non-option here. The natural nesting of subroutine calls makes it easy to allocate space for local variables. By allocating In stack, variables are declared, stored and initialized during runtime and when the computing task is complete, the memory of the variable will be automatically erased.

Garbage collection is an essential part of any Lisp interpreter or compiler. It allows Lisp to manipulate lists and more complex data structures without requiring the programmer to be concerned about the allocation and deallocation of the heap memory they need.

Java

```
public class Ta2 {

        private static int i = 9;


        public static void main(String[] args) {

                System.out.println(i++);
        }
}
```

The output of this program is 9, even though the expected result was 10.

Scheme

```
 4 (define var 2)
 5
 6 (define f(x) (- var 1))
 7
 8 (display (+ var 1))
 9 (display "\n")
10 (display f(5))
```

11 (display "\n")

Edition 4: question 3.4, page 167.

Consider live, but invisible, variables. Your examples need not compile/execute. Code:

```
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          end
9          a:integer:=3
10         inner()
11         print a,b
12     end
13     middle()
14     print a,b
15 end
```

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

Example 1

```
public class Ta2 {

        private int x = 4;

        public static void main(String[] args) {

                System.out.println(x); //the private int x does not apply here.
        }
}
```

Example 2

```
public class Ta2 {

        public static void main(String[] args) {

                for(int i = 0; i < 8; i++) {
                        System.out.println(i);
                }
                i++; //  i is out of scope here. The declared variable, i, in the loop does not apply here
        }
}
```

```
        private int x = 4;

        public static void main(String[] args) {

                System.out.println(x);
        }
```

Example 3

```
public class Ta2 {

        private static int i = 8;

        private void printX() {
                int i = 9; //
        }

        public static void main(String[] args) {
                System.out.println(i);  //i  prints 8 here. "int i" in the printX() method call does not apply
                here.
        }
}
```

```
        private int x = 4;

        public static void main(String[] args) {

                System.out.println(x);
        }
```

Edition 4: question 3.5, page 167.

Consider declaration order. Consider only the C and Modula-3 sets of rules.

```
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           end
9           a:integer:=3
10          inner()
11          print a,b
12      end
13      middle()
14      print a,b
15  end
```

C:
a = 1
b = 1

Modula-3:
a = 3
b = 3

C:
a = 1
b = 2

Modula-3:
a = 3
b = 3

C:
a = 3
b = 1

Modula-3:
a = 1
b = 2

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).

C - 1 1 3 1 3 2
Modula-3: 3 3 3 3 1 2

Edition 4: question 3.7, page 169.

Analyze memory bugs. Figure 3.16 is within the exercise.

As 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:

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

Sadly, after running for a while, Brad's program always runs out of memory and crashes. Explain what's going wrong.

The problem here is that enough space was not allocated for L. The two different L expressions essentially for 2 different lists. C does not have garbage collection, so the first L expression remains even after L is assigned to a new expression.

(b) After Janet patiently explains the problem to him, Brad gives it another try:

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

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?

In deleting the L, we are attempting to free up memory space. We are deleting the original L in memory before assigning a new L. The problem here is that the contents of the memory space is what is deleted. The result of this is that we have a memory space pointing to nothing. This results in outputs that are corrupted.

Edition 4: question 3.14, page 171.

Consider static and dynamic scope. Code:

```
1   x:integer
2
3   procedure setx(n:integer)
4       x:=n
5   end
6
7   procedure printx()
8       print(x)
9   end
10
11  procedure first()
12      setx(1)
13      printx()
14  end
15
16  procedure second()
17      x:integer
18      setx(2)
19      printx()
20  end
21
22  setx(0)
23  first()  printx()
24  second() printx()
```

What does this program print if the language uses static scoping? What does it print with dynamic scoping? Why?

Static scoping, a variable always refers to its top-level environment.

Dynamic scope, a global identifier refers to the identifier associated with the most recent environment

Static scoping

setx(0) - sets global x to 0
first() - setx(1) sets global x to 1, and then printx() prints the x, 1.
printx() - print the global x, 1
second() - setx(2) sets the global x to 2 and the printx() prints the x, 2.
printx() - prints the global x, 2

Output: 1 1 2 2

Dynamic scoping

setx(0) - sets global x to 0
first() - setx(1) sets global x to 1, and then printx() prints the x, 1.
printx() print the global x, 1
second() - setx(2) sets the x to 2 and the printx() prints the x, 2
printx() - goes back to the main and prints the global x, 1

Output: 1 1 2 1

Edition 4: question 3.18, page 173.

Consider shallow and deep binding.

Consider the following pseudocode:

```
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()
```

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?

Shallow binding binds the environment when it is actually called.
where the deep binding binds the environment when it is passed as an argument.

Shallow binding
global x is set to 0.
considering n = 1. Since n is in {1,3}
x is set to 1.
n is in {1,2}, so we print 1
1 0

global x is set to 0.
considering n = 2. Since n is not in {1,3}
then S(n)
n = 2 is in {1,2}, so we print 2
2 0

global x is set to 0.
considering n = 3. n is in {1,3}
x is set to 3.
n = 3 is not in {1,2}, so P
3 0

global x is set to 0.
considering n = 4. n is not in {1,3}
set_x, n is not in {1,2}, so we print_x
4 0

1 0
2 0
3 0
4 0

Deep binding
global x is set to 0. x = 5.
considering n = 1. n is in {1,3}
x is set to 1. n is in {1,2}, so
print 1
1 0

global x is set to 0.
considering n = 2. Since n is not in {1,3}
then S(n), global x is set to 2. n = 2 is in
{1,2}, so we print local x, 5
5 2

global x is set to 0.
considering n = 3. n is in {1,3}, so global x
is set to 3. n = 3 is not in {1,2}, so P; print
global x
3 3

global x is set to 0.
considering n = 4. n is not in {1,3}, set global
x to 4. n = 4 is not in {1,2}, so print global x, 4
4 4

1 0
5 2
3 3
4 4