Memory Management: Exercises and Solutions

1 A Code Fragment with a Recursion Depth That Is Not Statically Determinable

In many languages, the "depth" of recursive calls may depend on runtime input. For example, in the pseudo-language below, the recursion depth depends on an input value read at runtime:

```
function f():
    int x;
    read(x);         // runtime input, not known at compile time
    if x  0 then
        f();         // recursive call without a compile-time bound
    else
        return;

// Start the recursion.
f();
```

Because the number of recursive calls depends on the (unknown) value of x read from input, no static analysis can determine the maximum number of activation records that will ever be on the stack.

2 A Recursive Function with a Statically Determinable Maximum Stack Depth

If the recursion is "bounded" by a constant value, then the maximum depth is known at compile time. For example, consider the following:

```
function countdown(n):
    if n = 0 then
        return;
    else
        countdown(n - 1);

// Call with a constant argument.
countdown(10);
```

In this code, the recursion always unfolds exactly 11 times (including the call when $n = 0$). In this case, the maximum number of activation records is statically determinable.

Generalisation: Any recursion where the recursive depth is bounded by a constant (or an expression whose value is statically known) is statically determinable. In contrast, if the recursion depends on runtime values, then the maximum stack depth cannot be determined in advance.

3 Why the Immediate Predecessor AR Is Not Sufficient to Resolve a Reference

Consider the fragment (with "..." to be filled in):

```
A: { int X = 1;
     ...          // (Gap 1)
     B: { X = 3;
          ...    // (Gap 2)
     }
     ...
}
```

A counter-example is to have an intervening procedure call that does not bind X. For example, let A define a procedure F (which does not declare X) that, when called, in turn calls B. Then in B, the reference to X should be resolved to the X declared in A—not to any X (or "no binding") in F, which is the activation record immediately preceding B.

A correct filled-in counter-example is:

```
A: {
    int X = 1;
    // Gap 1: new procedure defined in A (F does not declare X)
    void F() {
         // B is nested within F (dynamically, F's AR is immediately before B's)
         B: {
              // Reference to X: F's activation record does not bind X,
                 so resolution must search further.
              X = 3;
              write(X); // (for instance, to demonstrate that the binding is from A)
         }
    }
    F();              // Call F so that its activation record is on the stack.
}
```

In this example, the AR immediately preceding B is that of F—but F does not bind X. The correct resolution must skip F's AR and "climb" further up the static (lexical) chain to A's activation record, where X is declared. This

shows that merely examining the immediately preceding activation record is not enough for proper static-scope resolution.

4 Activation-Record Stack Between the Static and Dynamic Chain Pointers

Consider the following pseudo-language code using static scope:

```
void P1 {
    void P2 {
         /* body-of-P2 */
    }
    void P3 {
         void P4 {
              /* body-of-P4 */
         }
         /* body-of-P3 */
    }
    /* body-of-P1 */
}
```

Now suppose the following sequence of calls is made (none has yet returned):

$$\text{P1} \rightarrow \text{P2} \rightarrow \text{P3} \rightarrow \text{P4} \rightarrow \text{P2}$$

and assume that all these procedures remain active.

We can diagram the stack as follows (showing only the relevant fields for static-scope handling):

[Top of Stack]

AR for P2 (second call) Static link: points to AR for P2 (first call)

AR for P4 Static link: points to AR for P3

AR for P3 Static link: points to AR for P1

AR for P2 (first call) Static link: points to AR for P1

AR for P1 No static link needed.

[Bottom of Stack]

For each activation record (AR) at a given lexical level, a field is saved that holds the previous value of the display (or static-link pointer) for that level.

5 Display and Stack for a Pseudo-Language with Goto and Nested Blocks

Consider the code fragment:

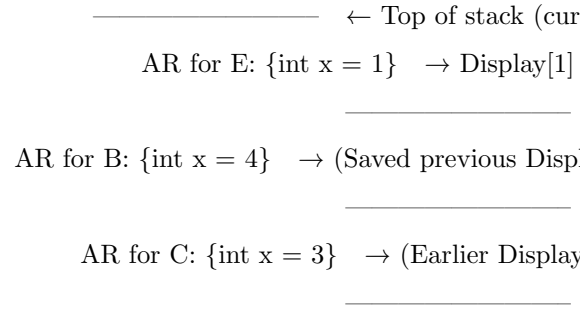
```
A: { int x = 5;
      goto C;
      B: { int x = 4;
           goto E;
      }
      C: { int x = 3;
           D: { int x = 2;
           }
           goto B;
      }
      E: { int x = 1; // (**)
      }
}
```

Assume that static scope is implemented using a display. When execution reaches the point marked (**), the active nested blocks (by static nesting) are as follows:

- Block A (level 1): declares int x = 5
- Block C (level 1): declares int x = 3
- Block B (level 1): was activated by a goto from C and is still active.
- Block E (level 1): is now active; it declares int x = 1.

Because all these blocks are lexically nested within A, the display (an array indexed by lexical level) is used to quickly locate the binding for x at level 1. In this case, the display entry for level 1 must point to the activation record for the currently active block at that level—in this case, block E.

A simplified diagram might be:



AR for A: {int x = 5}

For display handling, the only information stored in each activation record is the "saved" value of the display entry for the corresponding lexical level (i.e., the base address of the activation record for that level). This value is needed to restore the display when the procedure returns.

6 Implementing Static versus Dynamic Scope

It is generally easier to implement dynamic scope because resolving a variable simply requires scanning the chain of activation records (i.e., the dynamic chain) at runtime. In contrast, static scope requires extra bookkeeping (for example, maintaining static links or a display) to locate the correct binding based on the program's lexical structure. Although many modern compilers use displays to make static-scope resolution efficient, the underlying mechanism is more complex than the simple "search the call stack" approach used for dynamic scope.

7 Display and Activation-Record Diagram for the Second Call to A

Consider the code fragment (using static scope and call by reference):

```
{
  int x = 0;

  int A(reference int y) {
      int x = 2;
      y = y + 1;
      return B(y) + x;
  }

  int B(reference int y) {
      int C(reference int y) {
          int x = 3;
          return A(y) + x + y;
      }
      if (y == 1)
          return C(x) + y;
      else
          return x + y;
  }

  write(A(x));
}
```

A possible execution scenario is as follows:

- First call to A:
- Called from the global block with global variable $x = 0$.
- In A, a local variable x is declared (set to 2).
- The parameter y is a reference to the global x.
- A updates y (thus global x) from 0 to 1 and then calls B(y).
- Call to B: B is called with y referencing global x (now 1).
- Since $y = 1$, B calls C with argument x. Here, x refers to the binding in B's environment (which in this case is the global x, value 1).
- Call to C: C is defined inside B (lexical level 2).
- In C, a local x is declared (set to 3).
- C then calls A(y) with y still referencing global x (value 1).
- This is the second call to A.

At the moment when control enters the second call to A (inside C), the activation-record (AR) stack and display (used for static scope) look roughly as follows (we show only the "saved display" field for each AR):

Activation-Record Stack (from bottom to top):

AR for A (first call) – Level 1

AR for B – Level 1

AR for C – Level 2

AR for A (second call) – Level 1

(Top of stack)

Display Array at the Moment Immediately After Entering the Second A:

Display[0]: Points to the global activation record.

Display[1]: Now updated to point to the second call to A's AR.

Display[2]: Still points to AR for C.