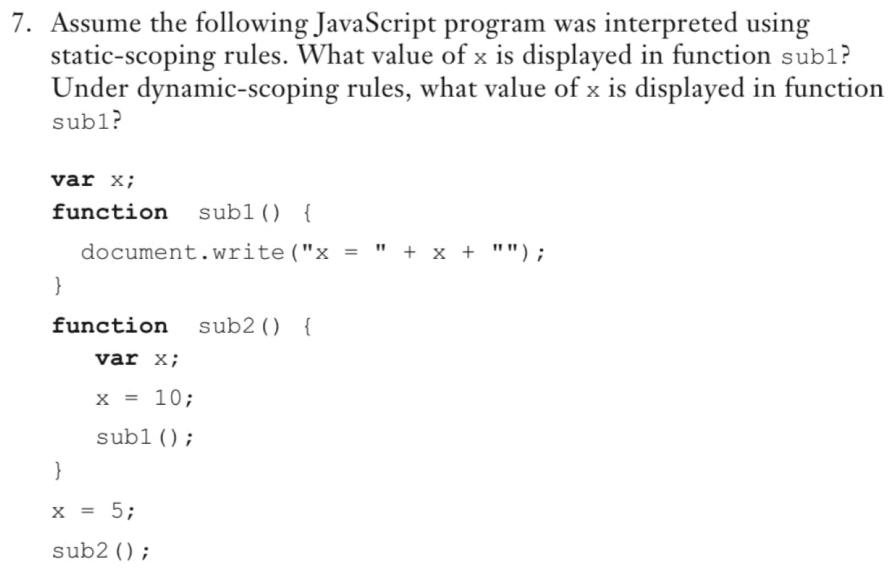
1) Problem 7 of chapter 5 on page 229

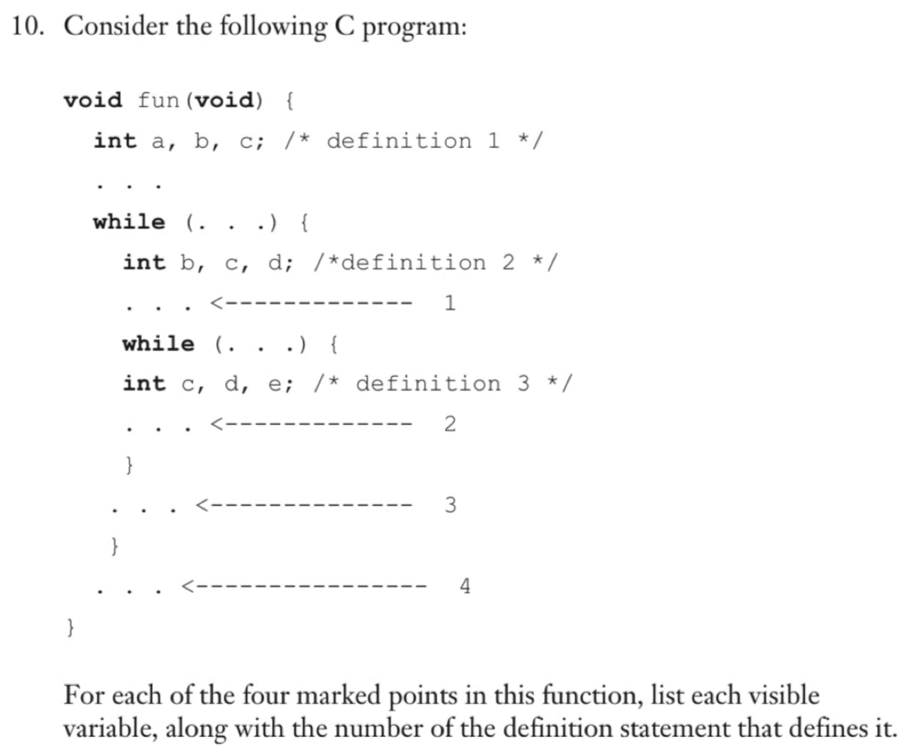


7. Assume the following JavaScript program was interpreted using static-scoping rules. What value of x is displayed in function sub1? Under dynamic-scoping rules, what value of x is displayed in function sub1?

```
var x;
function  sub1() {
  document.write("x = " + x + "");
}
function  sub2() {
   var x;
   x = 10;
   sub1();
}
x = 5;
sub2();
```

Static scope: x = 5
Dynamic: x = 10

2) Problem 10 of Chapter 5 on page 231



10. Consider the following C program:

```
void fun(void) {
  int a, b, c; /* definition 1 */
  . . .
  while (. . .) {
    int b, c, d; /*definition 2 */
    . . . <------------- 1
    while (. . .) {
    int c, d, e; /* definition 3 */
    . . . <------------- 2
    }
   . . . <-------------- 3
   }
  . . . <---------------- 4
}
```

For each of the four marked points in this function, list each visible variable, along with the number of the definition statement that defines it.

1) A,b,c, d (definition 1 & 2)

2) A, b, c, d, e (definition 1 & 2 & 3)

3) A, b, c, d (definition 1 & 2)

4) A, b, c (definition 1 & 2)

1) b, c, d (definition 2)
   a (definition 1)
2) c, d, e (definition 3)
   b (definition 2)
   a (definition 1)
3) b, c, d (definition 2)
   a (definition 1)
4) a, b, c (definition 1)

3) Problem 11 (a-e) of Chapter 5 on Page 231

11. Consider the following skeletal C program:

```
void fun1(void);  /* prototype */
void fun2(void);  /* prototype */
void fun3(void);  /* prototype */
void main() {
  int a, b, c;
  . . .
 }
void  fun1(void) {
  int  b, c, d;
  . . .
 }
void  fun2(void) {
  int  c, d, e;
  . . .
 }
void  fun3(void) {
  int  d, e, f;
  . . .
 }
```

Given the following calling sequences and assuming that dynamic scoping is used, what variables are visible during execution of the last function called? Include with each visible variable the name of the function in which it was defined.

a. `main` calls `fun1`; `fun1` calls `fun2`; `fun2` calls `fun3`.

b. `main` calls `fun1`; `fun1` calls `fun3`.

c. `main` calls `fun2`; `fun2` calls `fun3`; `fun3` calls `fun1`.

d. `main` calls `fun3`; `fun3` calls `fun1`.

e. `main` calls `fun1`; `fun1` calls `fun3`; `fun3` calls `fun2`.

a. Main: a
   Fun1: b
   Fun2: c
   Fun3: d, e, f

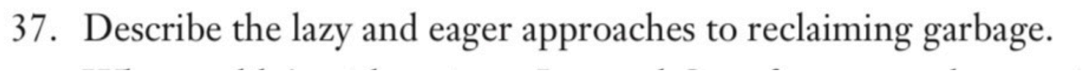
b. Main: a
   Fun1: b, c
   Fun3: d, e, f

c. Main: a
   Fun3: e, f
   Fun1: b, c, d

d. Main: a
   fun3: e, f
   fun1: b, c, d

e. Main: a
   fun1: b
   fun3: f
   fun2: c, d, e

4) Review Question 37 of Chapter 6 on page 296


37. Describe the lazy and eager approaches to reclaiming garbage.

Lazy approach / mark sweep: reclamation occurs when the list of variable space becomes empty.The run time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark sweep then begins.
Eager / reference counters: maintain a counter in every cell that stores the number of pointers currently pointing at the cell. Reclamation is gradual. Reclamation occurs only when the list of available space becomes empty.

5) Describe the tombstone and lock-and-key methods of avoiding dangling pointers

Tombstone:

-> Every heap-dynamic variable includes a special cell (the tombstone). The actual pointer variable points only at the tombstones and never to heap-dynamic variables when the h-d variable is deallocated. The tombstone remains but is set to nil, the h-d variable no longer exists. This approach prevents a pointer from pointing to a deallocated variable.

Locks-and-Keys:
-> Pointer values are represented as (key, address) ordered pairs, where the key is an integer. Heap-dynamic variables are represented as the storage for the variable plus a header cell that stores an integer lock value. When a h-d variable is allocated, a lock is created/placed in the lock cell of the h-d variable and in the key cell of the pointer that is specified in the call to new. Each access to the dereferenced pointer compares the key value of the pointer to the lock value in the h-d variable. If there exists a match, then the access is legal. Otherwise, treat it as a runtime error. Any copies of the pointer to other pointers must copy the key value. When h-d value is deallocated, its lock value is cleared to an illegal lock value. If a pointer other than specified in dispose is dereferenced, its address value will be intact but its key value will no longer match the lock -> the access will not be allowed.