Subarna Joshi
CS 354
Textbook assignment 2

3.2)
=>
The main thing that accounts for these differences is the fact that the language designers were different people and they had different ways of approaching the problem and saw that different types of variable allocations would be advantageous in certain situations i.e. FORTRAN as not being a recursive language figured that there would be no need to allocate more variables than what the program starts with so static allocation would be perfectly reasonable. Algol and its descendants' writers understood that since they were storing activation records on the stack so it made sense to them that the stack was the logical place for local variables to go.

Example Scheme program:
```
(define (sum seq)
 (if (null? seq)
 0
(+ (car seq) (sum (cdr seq)))))
(display (sum '(5 6 1 8 3 7)))
(display "\n")
```
Due to the recursion involved in this program, it would not work as expected.

Example Pascal program:
```
int x := something();
void subroutine()
{
 int x := somethingelse();
}
...stuff
```
This example would not work if the variables were statically allocated because there would be a naming clash.

3.4)

- The three examples in which a variable is live but not in scope is given below:

  Example 1:
  ```
  int main()
  {
  ...stuff
  someSubroutine()
  ...stuff
  }
  boolean someSubroutine()
  ```

```
{
struct *Node node = (struct * Node)malloc(sizeof(struct Node));
return node; //whether null or not
}
```

in this example, this struct has been allocated and is alive in memory but it is not in scope after the call to someSubroutine.

Example 2:

```
int main()
{
...stuff
someSubroutine()
...stuff}
boolean someSubroutine()
{
static int i = 0;
i++;
}
```

in this example, this static variable i is alive in memory but it is not in scope after the call to someSubroutine.

Example 3:

```
public class E3
{
int shadowed = 0;
public static void main (String[]args)
{
int shadowed = 5;
...stuff
}
}
```

in this example, the instance variable shadowed is alive in memory but it is not in scope during the main routine because it is shadowed.

3.5)

=> Firstly in C after invocation of main() are declared and initialized variables a and b. Then procedure middle() is declared and immediately called. Inside procedure middle() there is declared variable ‘b’ again, procedure inner() and variable ‘a’ also again in that order. After that procedure inner() is invoked and inside it is ‘a’ and ‘b’ printed. We know that C uses static scoping and that declaration must be before use. That means that name ‘b’ in procedure inner() is bound to declaration in procedure middle() and name ‘a’ is bound to declaration in main(), because ‘a’ in closest enclosing block (procedure middle()) is declared after declaration of inner(). So firstly 1, 1 is printed.

After that in procedure middle() are bound names of ‘a’ and ‘b’ to declaration in this procedure. When declared b : integer := a, ‘a’ is bound to the declaration in main() and then ‘a’ is declared again and initialized with 3. So 3, 1 is printed.
When printing ‘a’ and ‘b’ in main(), also here is declaration in this procedure is used so 1, 2 is printed.

In C# we can declare nested functions only using keyword delegate, but in this nested function is not possible to declare variable with same name as in any-level enclosing function so static semantic error occur.

In Modula-3 is situation much more complicated. When calling procedure inner(), there is a conflict with assignment of b, because in Procedure middle() is initialized with value of ‘a’, but ‘a’ has value from closest enclosing block and also from declaration in procedure middle(), because names can be declared in any order. So the final value of ‘b’ in procedure middle() is 0 because of this conflict Then variable ‘a’ is in middle() initialized on 3, which shadows declaration in main(), so then in procedure inner() 3 is printed as ‘a’ and ‘0’ as ‘b’, because of conflict in closest enclosing block middle(). Same it is in procedure middle(), ‘a’ is printed as 3, ‘b’ as 0 and in procedure main() have ‘a’ and ‘b’ values from local initialization, so a=1 and b=2.

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. In other words, 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. These refer to 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. Brad seems to have been lucky in that he isn’t corrupting the heap itself (maybe his widgets are the same size as list nodes), but without Janet’s help he may have a lot of trouble figuring out why widgets are changing value “spontaneously.”

3.14)
=> With static scoping it prints 1 1 2 2. With dynamic scoping it prints 1 1 2 1. The difference lies in whether set x sees the global x or the x declared in second when it is called from second.

3.18)
=> With shallow binding, set_x and print_x always access foo's local x. The program prints
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 program prints
1 0
5 2
0 0
4 4