CIS 425: Principles of Programming Languages
Lecture 11: Scope, functions, and storage management

Zahra Alsuliman

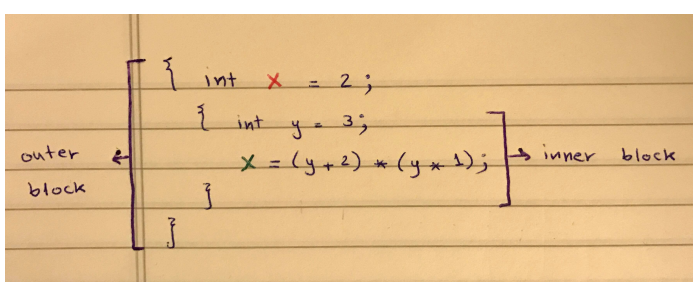
6 May 2019

Today's lecture focused on memory, in particular where to allocate variable and how to access them. We will analyze languages with different expressive power, we will see how the storage management becomes more complicated once we enhance the language. Here are the languages under consideration:

- Block structured languages (see Section 1)
- First-order functions (see Section 2)
- Functions in input (see Section 3)
- Functions in input and output (see Section 5)

1 Block structured languages

Most of the modern languages provide some form of blocks. What is a block? It's a region of program text identified by the `begin` and `end` markers that contains declarations local to that region. In SML, a block starts and ends with the key words (let-in-end). Consider this code in C:



As you see here in the code, the outer block starts with the first left brace and ends with the last right brace. The inner block is nested inside the outer

block. Variables can be local or global to their blocks.

Local variables are the variables declared within a block. For example, x with the red color is local to the outer block, y is local to the inner block.
Global variables are the variables declared in an enclosing block. For example, x with the green color is global in the inner block.

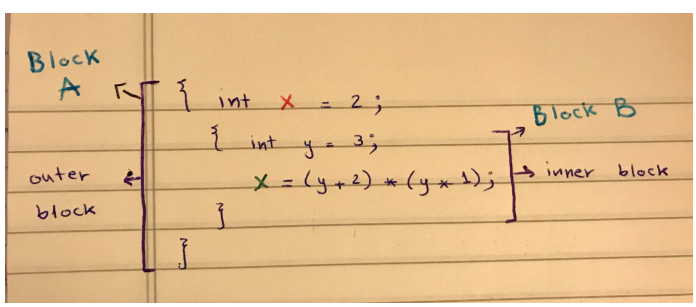
How do we access variables: first you look for the definition of a variable in the local block. If you do not find the definition, you look at the block which encloses it. So the questions to answers are: 1) where do we allocate the variables, such as x, y? 2) how do we implement the above idea.

Variables declared in a block are allocated when we enter a block, when we exit the block the memory allocated for those variables is deallocated. We collect all the information associated to a block in the so called *activation record* or stack frame. We deal with the activation records following a stack discipline (also called LIFO): we push an activation record on entry to the block, we pop the activation record on exit from a block. A register, called *environment pointer*, points to the current block.

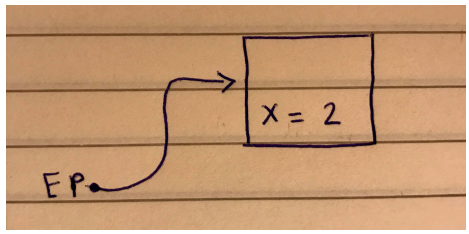
An activation record contains the following:

1. space for variables. Example: the variable x in the previous C code.
2. space for temporary variables. These are values that are not given names in the code. Example: (y+2) and (y*1) in the previous C code will be stored in temporary variables t1, t2 respectively.
3. pointer to the activation record of the calling or enclosing block, called *control link* or *dynamic link*.

Example 1 Let's draw the activation record for our example of C code. We label the outer block A, and the inner block B block as follows:

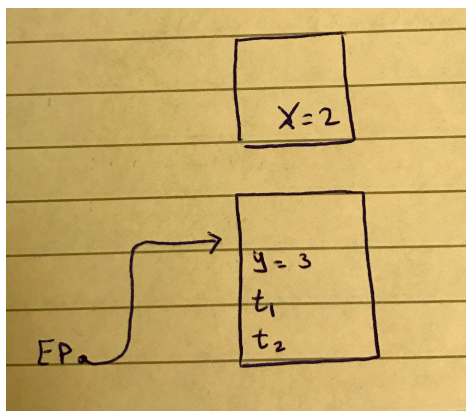


Drawing step 1:
We enter block A and we create a new activation record for block A. We allocate a space for x in the activation record and we set the environment pointer to the current activation record as follows:



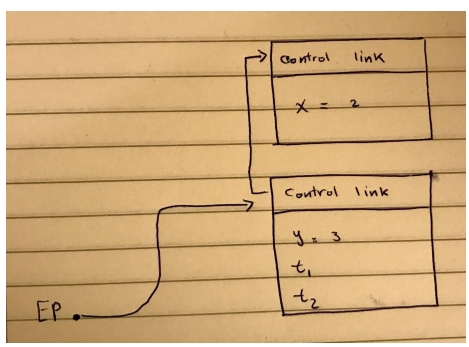
Drawing step 2:

We enter a new block which is B and we create a new activation record for this block. We allocate space for y, and space for 2 temporary variables for (y+2) and (y*1). We reset the environment pointer to point to the new current activation record as follows:



However, we face a problem here, which is how do we access variable x which is defined in block A?

This problem can be solved by storing more information in the activation record. What we need to store is a **control link/ dynamic link** that points to the block you came from. So, control link in block B points to A, as displayed below:



To access variable x from block B, the compiler will generate the address

```
(*EP) + 4
```

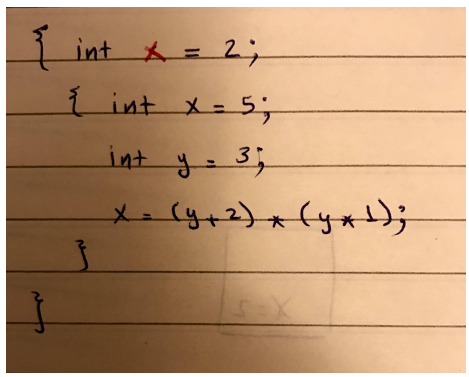
which says: take the content of `EP` and add 4, assuming that the control link is stored in 4 bytes.

Scope vs Life time:

Scope: region of text in which a declaration is visible.

Life time: the duration during a run of a program, during which a location is allocated as the result of a specific declaration.

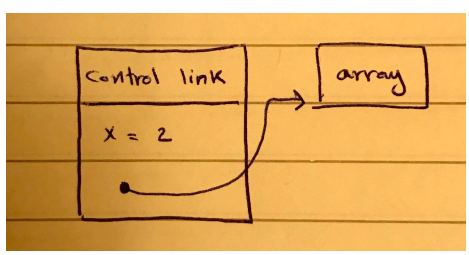
Variable x with the red color in the code is accessible to the entire block unless we have another definition that creates a hole in the scope of x. Creating a hole means that the inner declaration of x hides the outer one. For example, if we declare another x in the inner block like this:



Then x with the red color is not in the scope of the inner block but it's still alive, that is, the memory allocated for variable x is still there. This shows that scope is different from lifetime: the x in red is not in scope in the inner block but it is still alive.

Arrays and activation record:

If in a block you define a variable Z bound to an array, say Z = array[n], where the bound n is not known at compile time, how much memory do you reserve for Z in the activation record? Since the size of the activation record needs to be know to the compiler, the array will be allocated in the heap and the only thing to be stored in the activation record is the pointer to it, as follows:



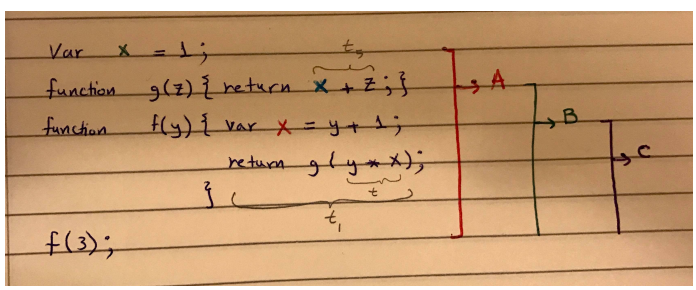
Same principle holds for any variable that has no known size at compile time.

2 Extend a block structured language with first-order functions

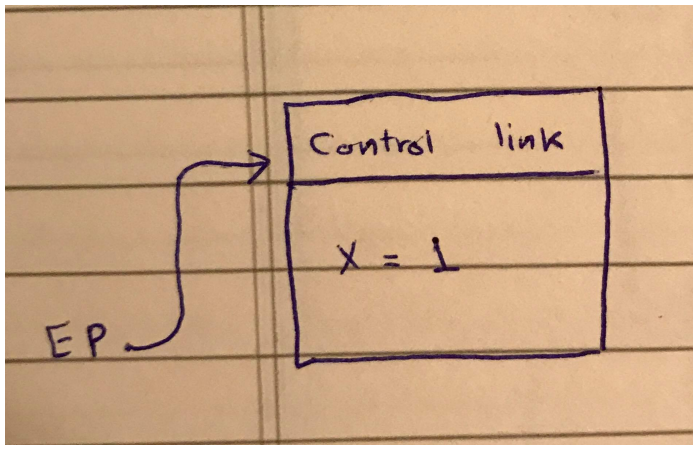
An activation record contains the following:

1. space for variables. Example: the variable x in the previous C code.
2. space for temporary variables. These are values that are not given names in the code. Example: (y+2) and (y*1) in the previous C code will be stored in temporary variables t1, t2 respectively.
3. pointer to the activation record of the calling or enclosing block, called *control link* or *dynamic link*.
4. space for parameters.
5. return address. This the pointer to the instruction you need to execute after the function call (Pointer to code)
6. return result address (RRA). This is the pointer to a location in the calling activation record to where you need to save the result of the function call
7. static link, the is the pointer to the activation frame which contains the definition of the function

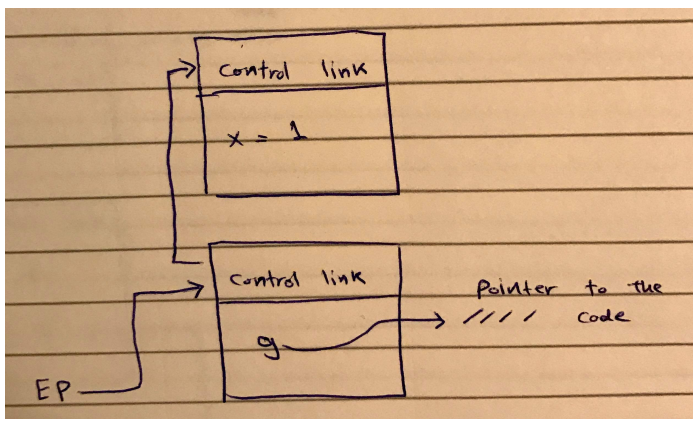
Consider this code with the labeled blocks:



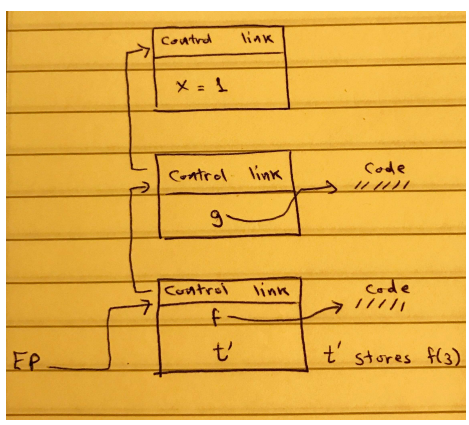
Let's start drawing the activation records
Step 1: Enter block A



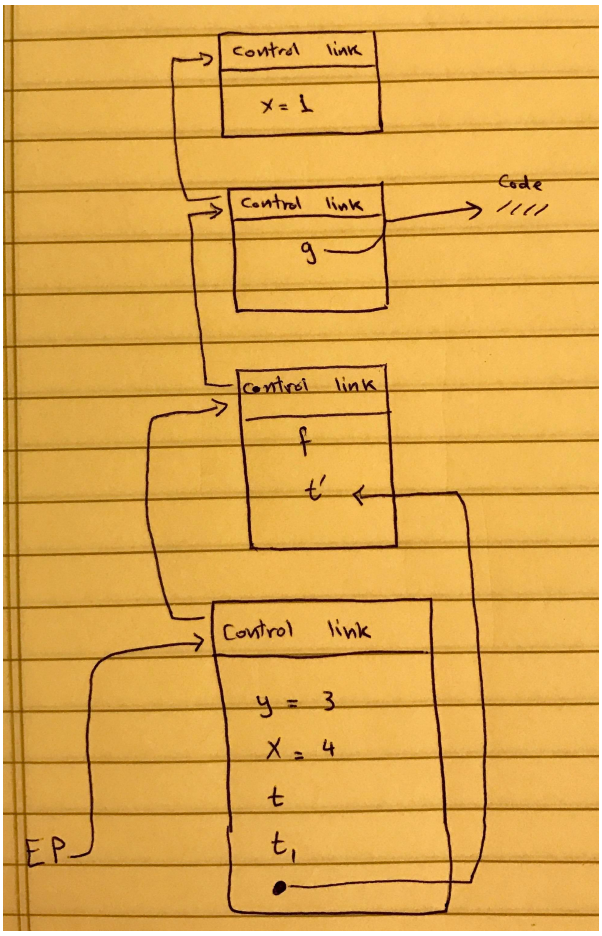
Step 2: Enter block B



Step 3: Enter block C

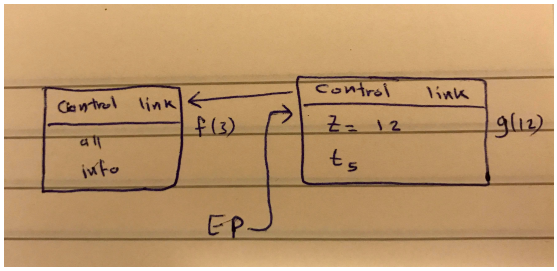


Step 4: Enter f(3)



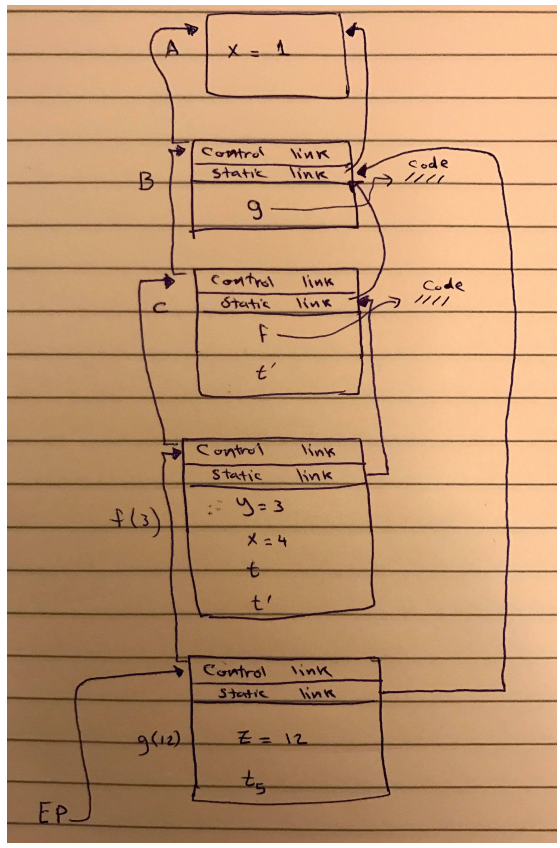
Note that we drew the RRA, which points to here I need to store the result of f(3), which is t'.

Step 5: Execute g(y*x)



We now need to execute x+z, but which x am I referring to? So far I only have the control link which points to the caller which contains a definition of x which is bound to 4. This is the x which Lisp would access. In order to implement

ML or javascript we need to access the outer x which is bound to 1. To solve this problem, we use a **static link/ access link**. Here is how the drawing of the activation record after adding a static link:



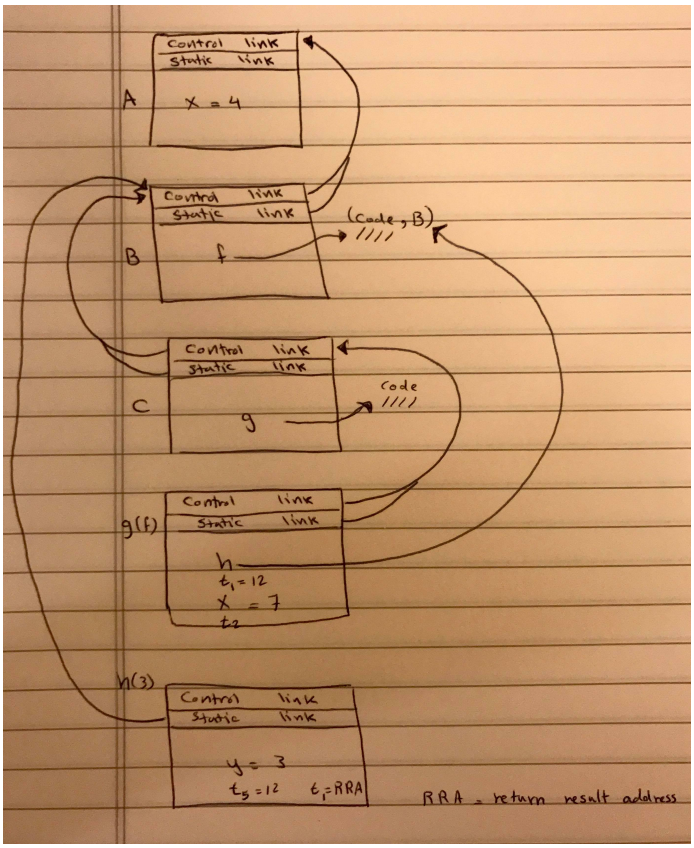
You follow the static link in g(12) to go to the static link of block B and then from the static link in B you go to the block A to access x = 1 The compiler will do this step: (*((*EP)+4))+8 to get x.

3 Extend a block structured language with functions in input

Consider this code:

```
let
val x = 4
in
  let
    fun f y = x * y;
  in
    let
      fun g h = let val x = 7;
                  in
                    h 3 + x
                  end
    in
      gf
```

To initialize the static link properly in passing functions as a parameter, we use the concept of a *closure*. A closure is a pair consisting of a pointer to function code and a pointer to the activation record containing the definition of the function. The activation record of the code above is as follows:



4 What we have so far

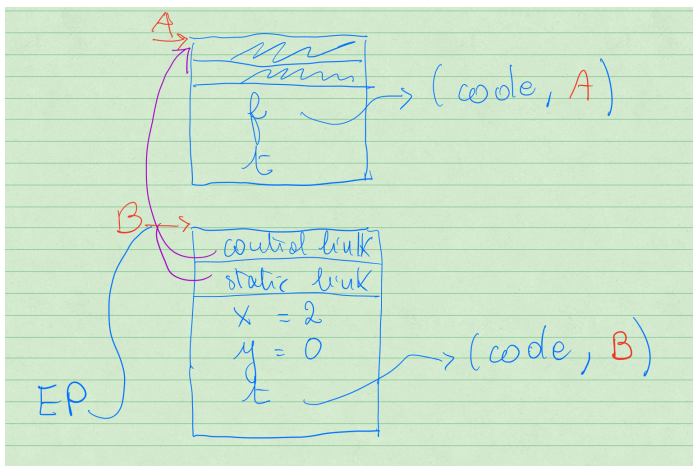
1. To implement the notion of a *block* we use an activation record which contains in addition to variables a control link. We follow a stack discipline
2. We added first-order function and in order to implement static scope we introduce the static link
3. We added functions in input and we introduced the concept of a closure.

5 Extend a block structured language with functions in input and output (higher-order functions)

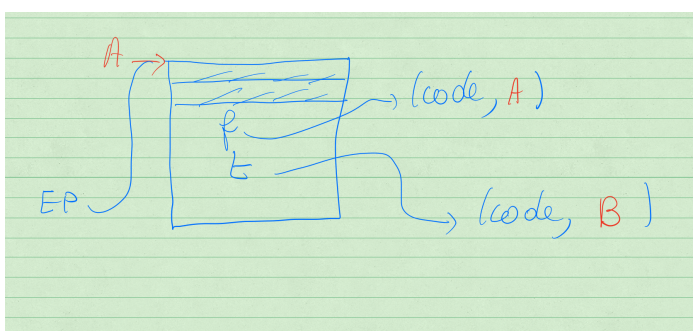
Consider this code:

```
let
    fun f x = let val y = 0
                in
                    fn z => x + y + z
                end
    val t = f 2
  in
    t 3
end
```

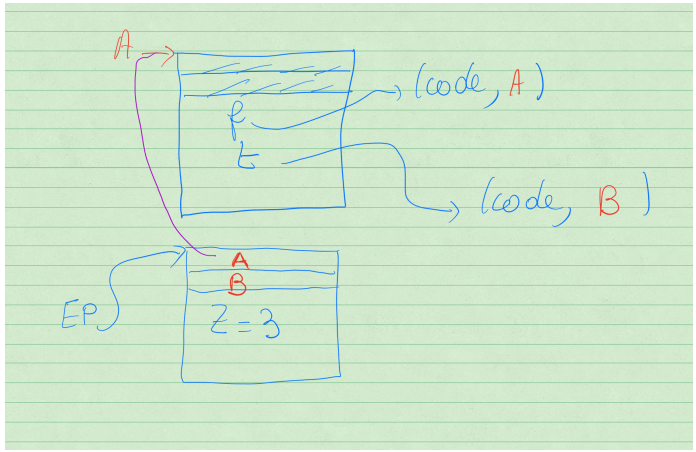
Let us draw the activation records as we learned so far:



When we are ready to return from function f we store the result in t of Block A, and then we update the Environment Pointer, obtaining:



Now we are ready to invoke the function stored in t with actual parameter 3, obtaining:



and we ready to execute x + y + z. Variable z is local and is bound to 3, what about variables x and y? To access them I follow the static link which points to B, *but this is a problem!*. The memory pointed to by B has been deallocated. Since I am accessing a location which does not belong to my address space the system raises an error (segmentation violation or bus error). *How do we solve the problem?* When we return from a function we update the Environment Pointer, but we do not deallocate the activation record associated to the call. Deallocation is performed by the **Garbage Collector** which will traverse the memory and free all memory which is not accessible by the program.