CS224 – Activation Records

Purpose: Activation records are created dynamically as a function or method is called, and contain information that the function needs such as parameters, local variables, and the return address. We will examine the different ways that activation records can be maintained.

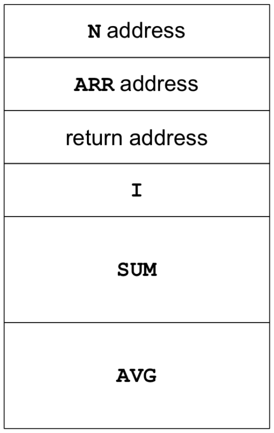
Knowledge: This activity will help you become familiar with the following content knowledge:

- The difference between static and stack-based and fully dynamic activation records.
- How activation records allow scoping

Activity: With your group perform the following tasks and answer the questions. You will be reporting your answers back to the class in 30 minutes.

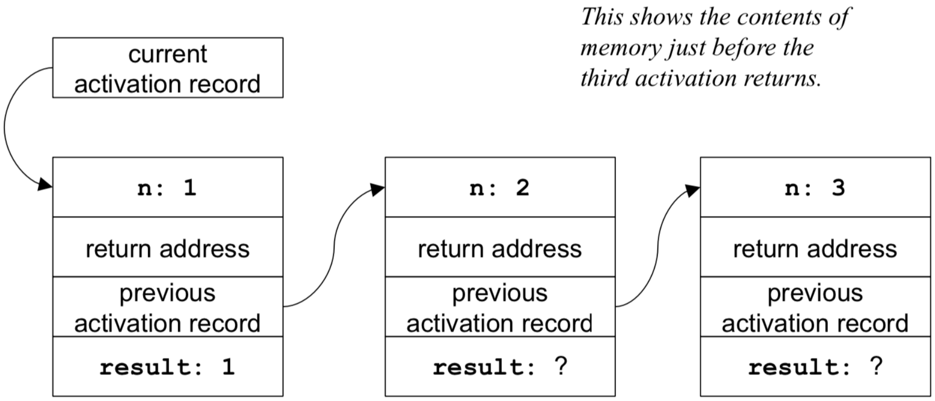
1. Back in the dark ages when programmers were chiseling their programs onto stone tablets, the programming language FORTRAN used static activation records. Each activation record was allocated before the start of the execution of the program. Here is an example FORTRAN program and its activation record. What are the drawbacks of static activation records?

```
    FUNCTION AVG (ARR, N)
    DIMENSION ARR(N)
    SUM = 0.0
    DO 100 I = 1, N
          SUM = SUM + ARR(I)
100 CONTINUE
    AVG = SUM / FLOAT(N)
    RETURN
    END
```



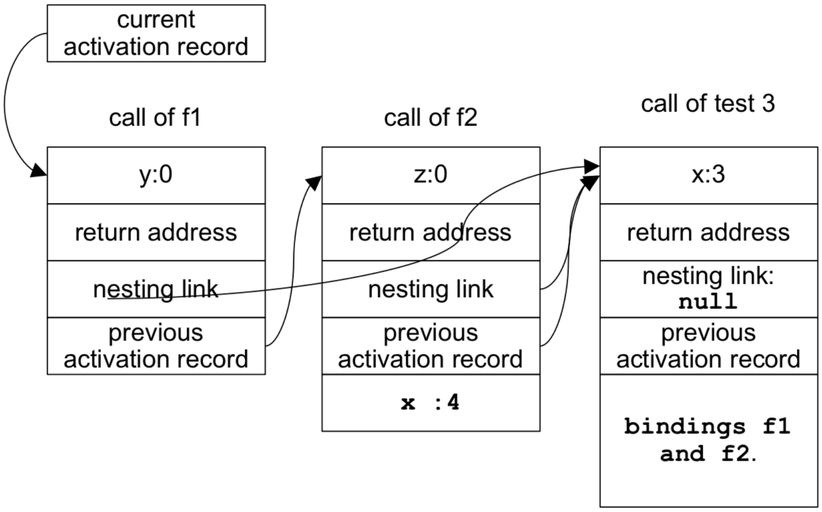
2. If we want recursion we need to create an activation record on each recursive call. Consider the function `fact` and its activation records for the call `fact 3`. What type of data structure would we use to store these activation records?

```
int fact(int n) {
     int result;
     if (n<2) result = 1;
     else result = n * fact(n-1);
     return result;
}
```


This shows the contents of memory just before the third activation returns.
current activation record
n: 1
return address
previous activation record
result: 1
n: 2
return address
previous activation record
result: ?
n: 3
return address
previous activation record
result: ?

3. We can use activation records to perform scoping as long as we add one more piece of information to each record to include a pointer to the record in which the function is defined. This pointer is called the static link (or the nesting link). Consider the Haskell example:

```
test x =
   let f1 y = x
       f2 z = let x = 4 in f1 0
   in f2 0
```



Why is the nesting link (static link) for `f1` pointing to the activation record for `test`? Why is the previous activation record (dynamic link) for `f1` pointing to the activation record for `f2`?
Since Haskell uses static scoping, how does `f1` determine the value of `x` to return?

4. There are some wrinkles that need to be ironed out with languages like Haskell that can have functions as both return values and as parameters. Consider the following Haskell function which adds a value x to every item in a list.

   ```
   addXToAll x theList =
             let
                 addX y = y + x;
             in
                 map addX theList
   ```

   Draw the activation record for the call addXToAll 2 [1,2,3]. It will contain locations for the parameters x and theList

   In the execution of addXToAll the function addX is created. This function does not reside in the activation record but in other memory that our program can allocate. For now draw a circle off to the side to represent the function addX

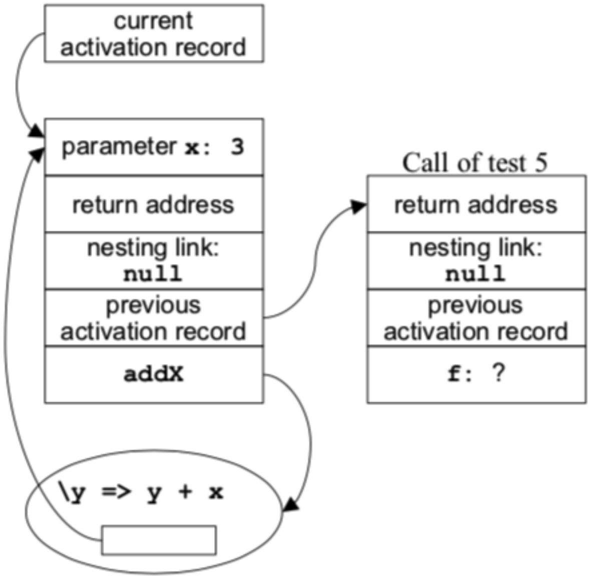
   Now the function addXToAll calls the function map. Draw the activation record for map with its two parameters. We don't really know (or care) about the nesting link for this activation record. What will be the dynamic link?

   The function map is going to call the function addX on the first value of the list which in this case is 1. Here is where things get interesting. Try drawing the activation record for addX with it's parameter y and it's static and dynamic links. How can we know what the static link is for addX? Perhaps this information needs to be stored along with the function itself. How does addX determine the proper value for x?

5. Here is another interesting Haskell example in which a function returns another function and the activations records for the call `test 5` at the point where it has called `funToAddX 3`:

```
test x =
  let
       f = funToAddX 3
  in
       f x

 funToAddX x =
   let
        addX y = y + x
   in
        addX
```



If we are using a stack of activation records, at the end of the execution the activation record for `funToAddX` would get popped off and go ”poof”. Why would that be a big problem here?

If we don’t use a stack, what would we have to do with these activation records? Try to come up with some general strategies.