Memory Locations for Variables

Modern programming languages have many different classes of Variables, e.g.

Chap 12

(1) Global variables
(2) Parameters
(3) (function) local variables (also called automatic or activation-specific)
(4) (object-oriented) member variables
(5) Etc.

It is the job of the language system to keep track of the values of these variables during the runtime of a program.

⇒ The language system accomplishes this by binding a variable to a memory location and then uses that memory location to store the value of the variable.

Memory Locations for Variables

In imperative programs this is a fairly transparent process - the assignment operator mimics what happens at the hardware level - namely, the updating of memory cells.

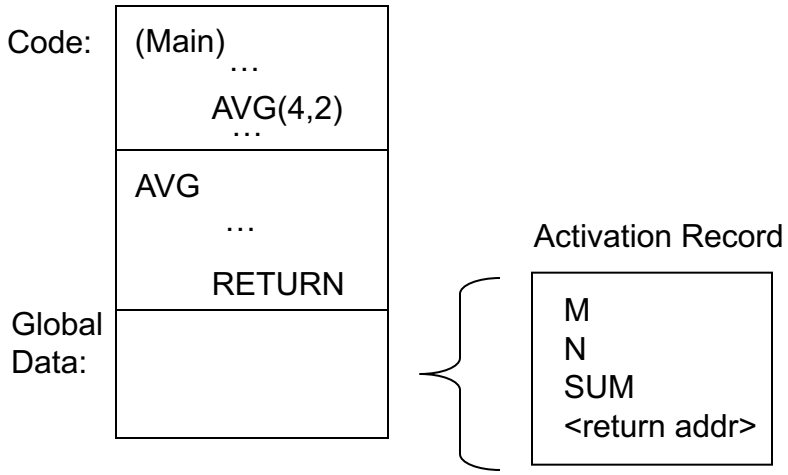
In functional languages this is often not so obvious, since there is no global State, but still, variables are bound to memory locations.

Activation Records

In order to track variables for functions, compilers use a data structure called activation record - collects all the variables belonging to one function into this structure.

Example: FORTRAN

```
FUNCTION AVG (M,N)
SUM = M + N
AVG = SUM/2.0
RETURN
END
```

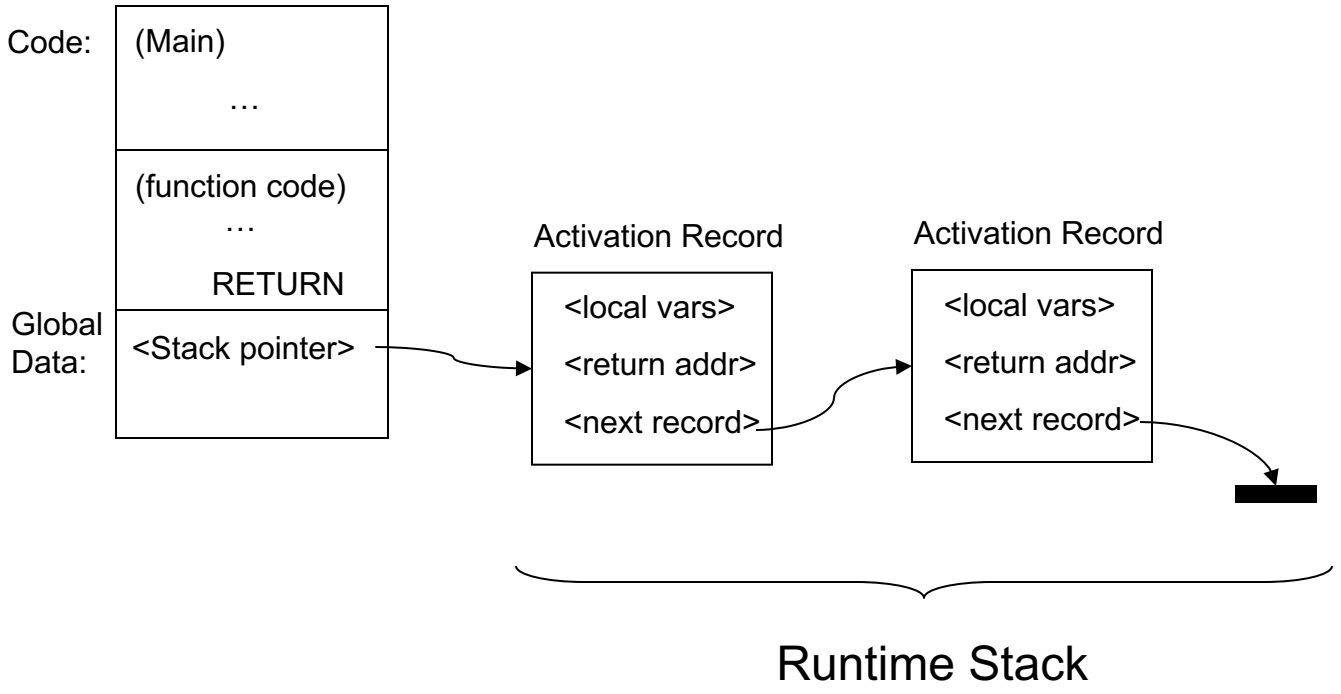


Activation Records

Note: Non-recursive languages such as FORTRAN keep a single activation record per function in the program.

Recursive languages (ML, Java, C, C++, etc) keep a stack of activation records; one per function call.

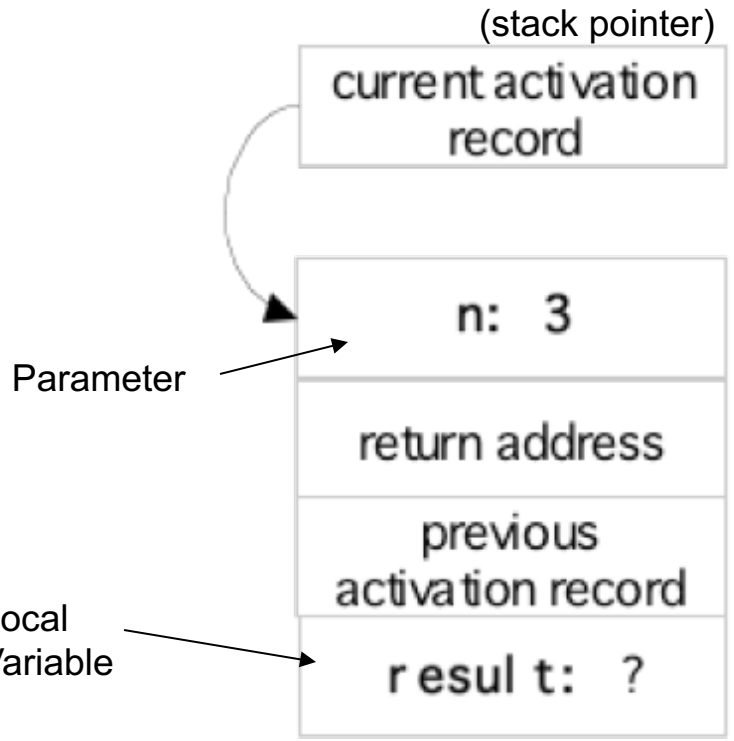
The Runtime Stack



Java Example

We are evaluating **fact(3)**. This shows the contents of memory just before the recursive call that creates a second activation.

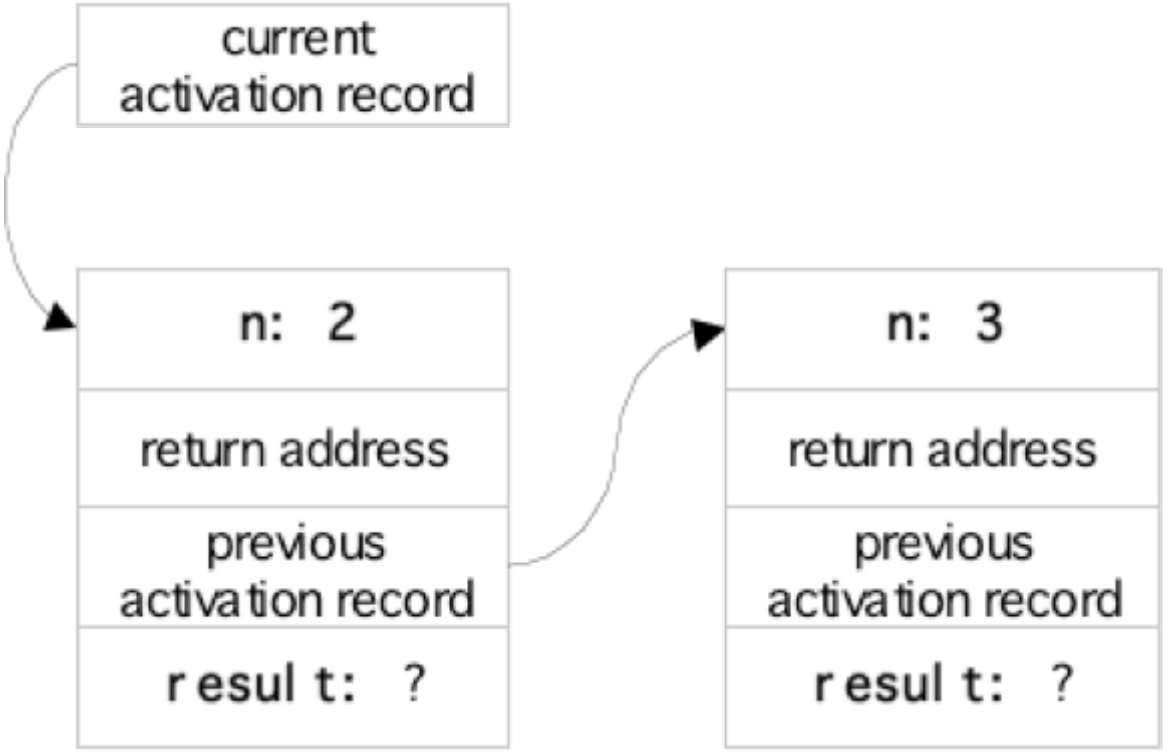
```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



Java Example

This shows the contents of memory just before the third activation.

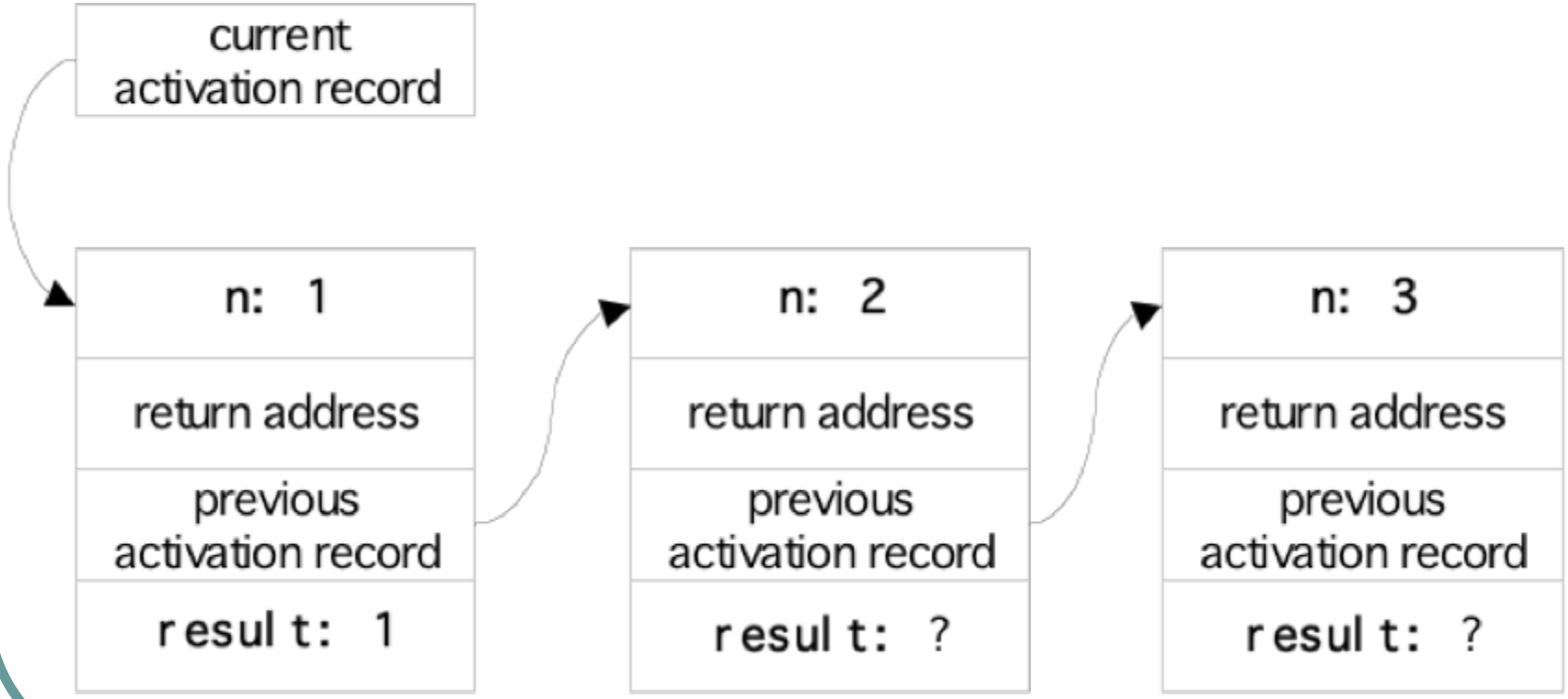
```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



Java Example

This shows the contents of memory just before the third activation returns.

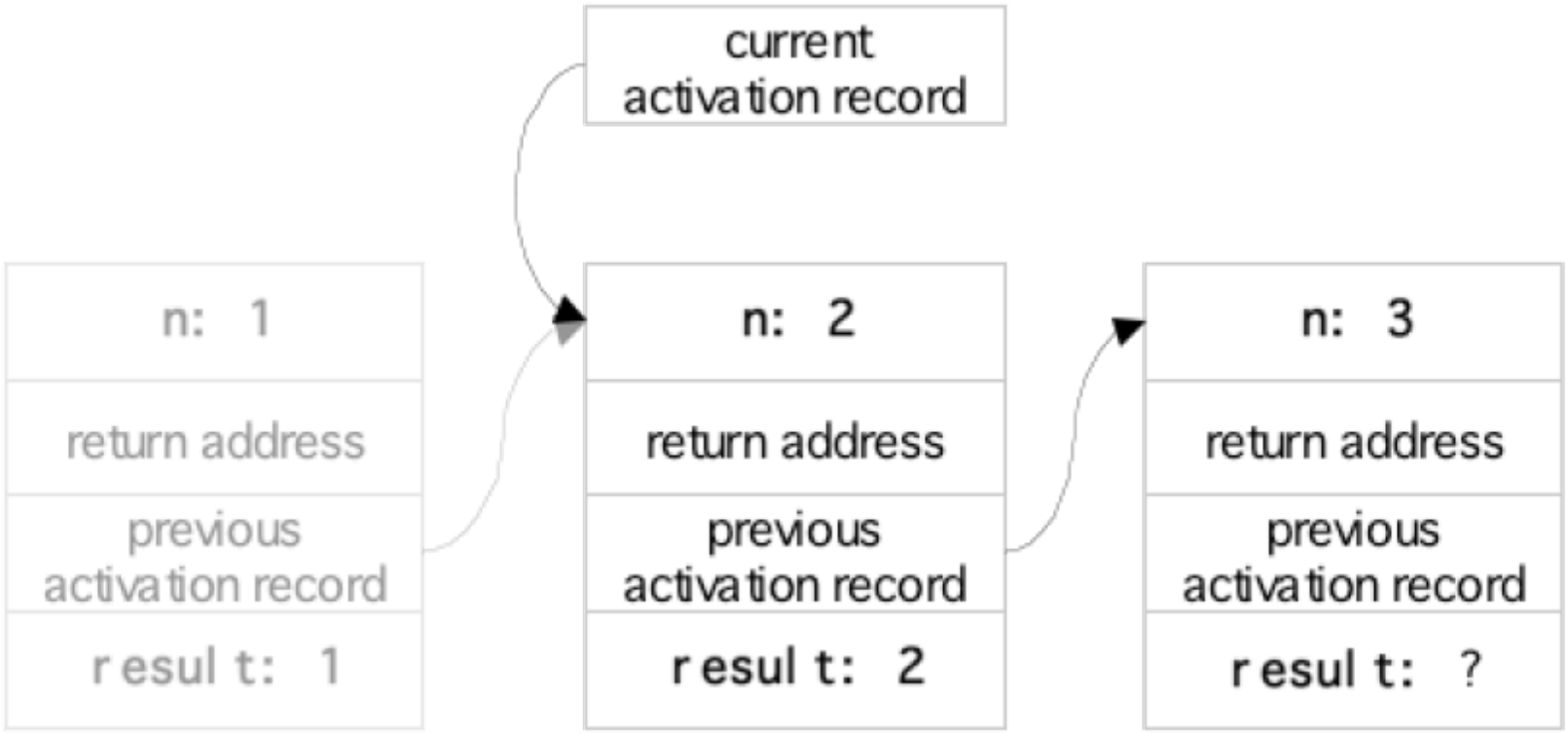
```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



Java Example

The second activation is about to return.

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



Java Example

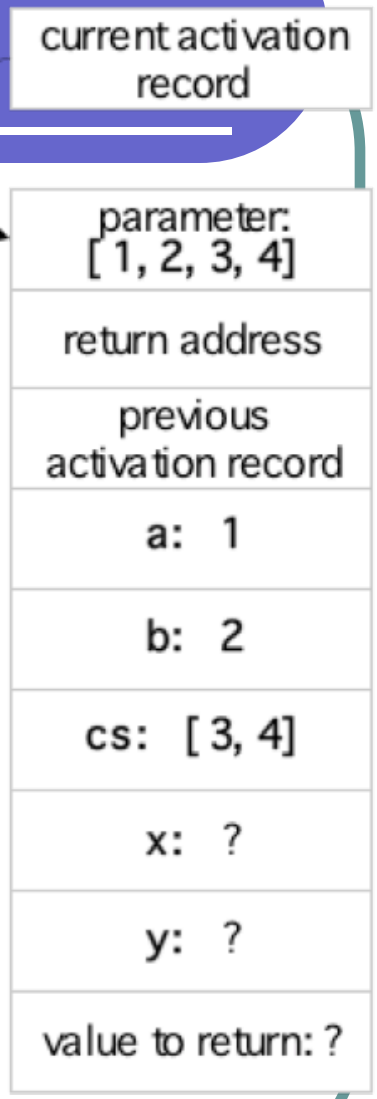
The first activation is about to return with the result **fact(3) = 6**.

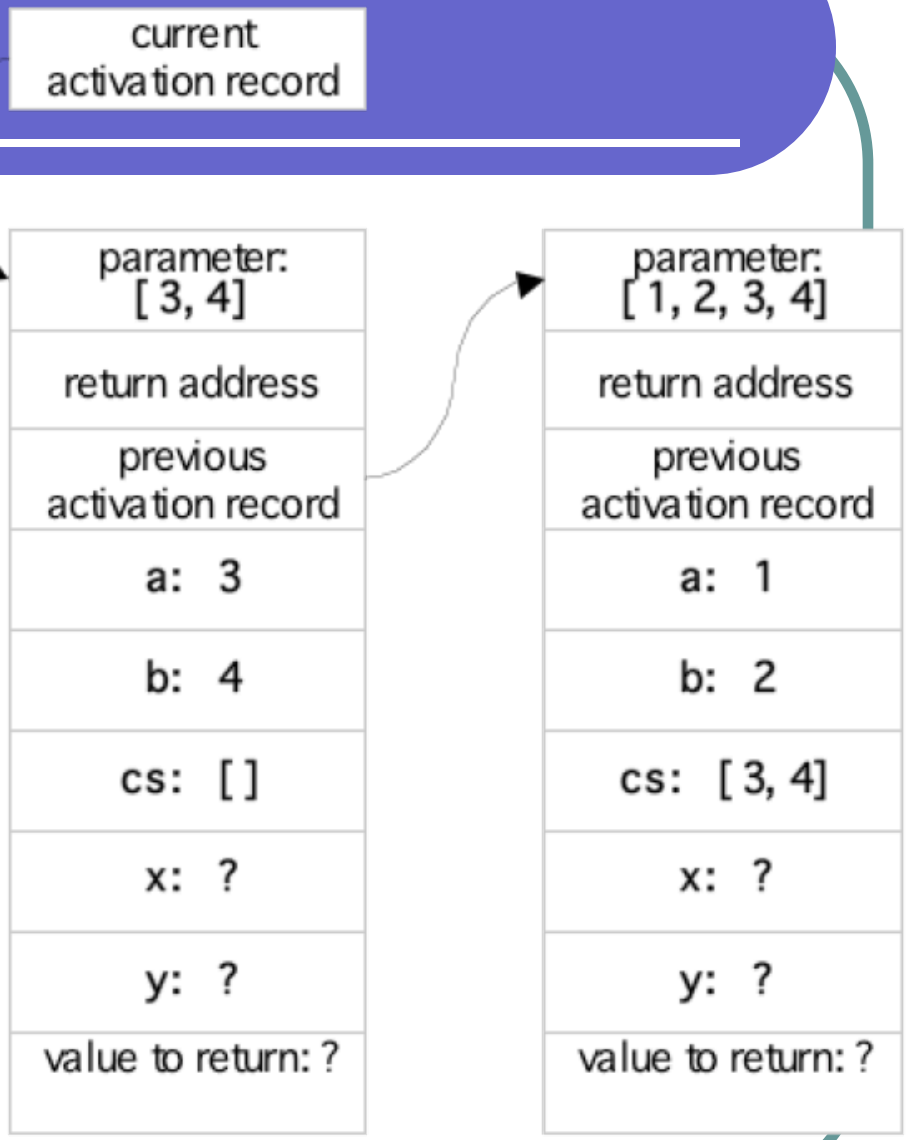
```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```

ML Example

We are evaluating **halve [1,2,3,4]**. This shows the contents of memory just before the recursive call that creates a second activation.

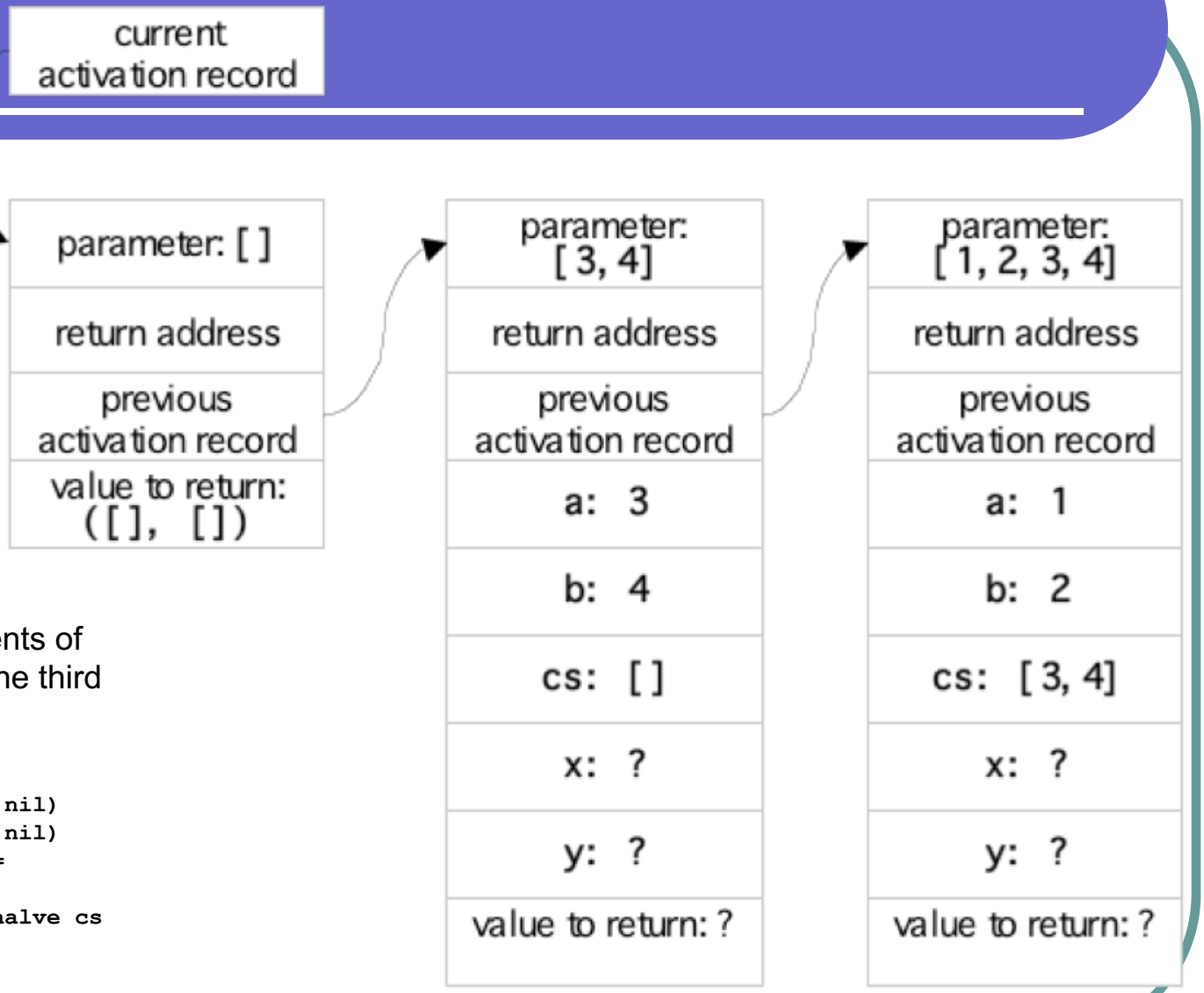
```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
      let
        val (x, y) = halve cs
      in
        (a::x, b::y)
      end;
```





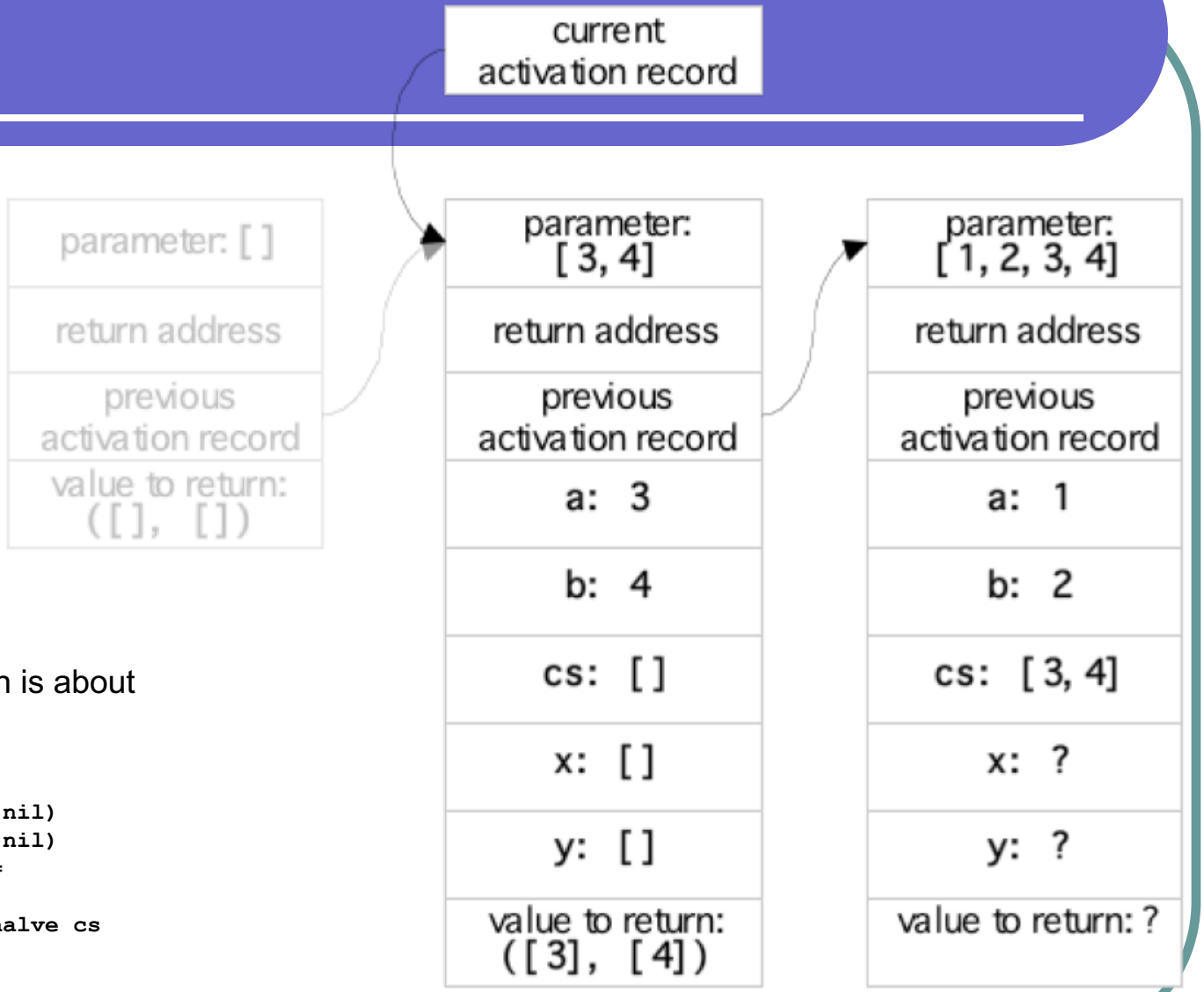
This shows the contents of memory just before the third activation.

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
      let
        val (x, y) = halve cs
      in
        (a::x, b::y)
      end;
```



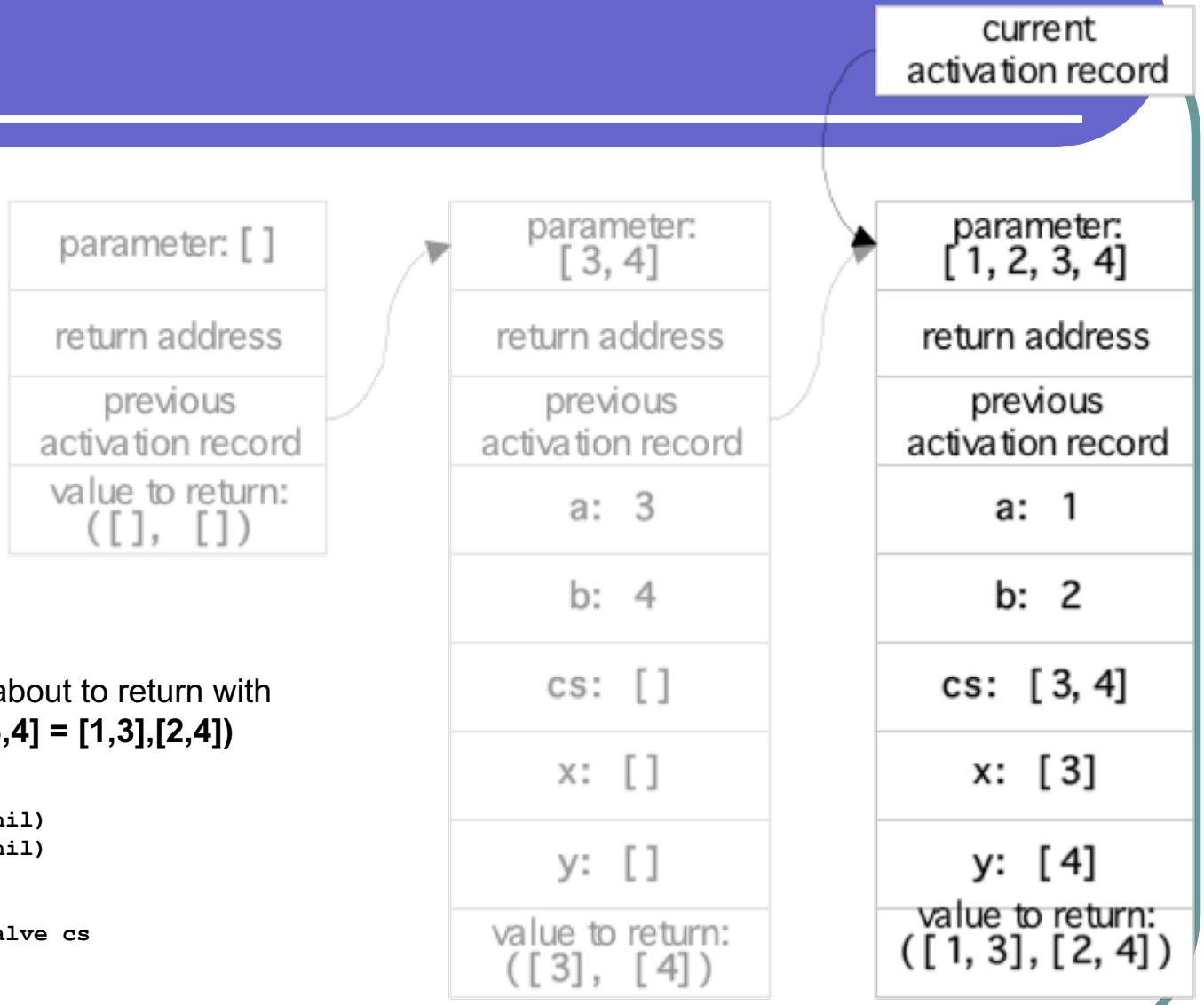
This shows the contents of memory just before the third activation returns.

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
      let
        val (x, y) = halve cs
      in
        (a::x, b::y)
      end;
```



The second activation is about to return.

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
      let
        val (x, y) = halve cs
      in
        (a::x, b::y)
      end;
```



The first activation is about to return with the result **halve [1,2,3,4] = [1,3],[2,4])**

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
      let
        val (x, y) = halve cs
      in
        (a::x, b::y)
      end;
```