Chapter 12
Memory Locations for Variables

变量的内存位置.

A Binding Question

- Variables are bound (dynamically) to values
- Those values must be stored somewhere
- Therefore, variables must somehow be bound to memory locations
- How?

命令式语言.

Functional Meets Imperative

- Imperative languages expose the concept of memory locations: **`a := 0`**
  - Store a zero in **`a`**’s memory location
- Functional languages hide it: **`val a = 0`**
  - Bind **`a`** to the value zero
- But both need to connect variables to values represented in memory
- So both face the same binding question

3

Function Activations

生命的周期 称为 函数的激活.

- The lifetime of one execution of a function, from call to corresponding return, is called an *activation* of the function
- When each activation has its own binding of a variable to a memory locations, it is an *activation-specific* variable

4

Activation-Specific Variables

- In most modern languages, activation-specific variables are the most common kind:

```
fun days2ms days =
  let
    val hours = days * 24.0
    val minutes = hours * 60.0
    val seconds = minutes * 60.0
  in
    seconds * 1000.0
  end;
```

5

Block Activations

- For block constructs that contain code, we can speak of an activation of the *block*
- The lifetime of one execution of the block
- A variable might be specific to an activation of a particular block within a function:

```
fun fact n =
  if (n=0) then 1
  else let val b = fact (n-1) in n*b
end;
```

6

Other Lifetimes For Variables

- Most imperative languages have a way to declare a variable that is bound to a single memory location for the entire runtime
- Obvious binding solution: static allocation (classically, the loader allocates these)

```
int count = 0;
int nextcount() {
  count = count + 1;
  return count;
}
```

long lifetime static variable

7

Scope and Lifetime Differ

- In most modern languages, variables with local *scope* have activation-specific *lifetimes*, at least by default
- However, these two aspects can be separated, as in C:

```
int nextcount() {
  static int count = 0;
  count = count + 1;
  return count;
}
```

static variable within a scope

8

Other Lifetimes For Variables

- Object-oriented languages use variables whose lifetimes are associated with object lifetimes
- Some languages have variables whose values are persistent: they last across multiple executions of the program
- Today, we will focus on activation-specific variables

9

Activation Records

- Language implementations usually allocate all the activation-specific variables of a function together as an *activation record*
- The activation record also contains other activation-specific data, such as
  - Return address: where to go in the program when this activation returns
  - Link to caller’s activation record: more about this in a moment

10

Block Activation Records

静态分布

- When a block is entered, space must be found for the local variables of that block
- Various possibilities:
  - Preallocate in the containing function's activation record
  - Extend the function's activation record when the block is entered (and revert when exited)
  - Allocate separate block activation records
- Our illustrations will show the first option

11

发生在编译时间

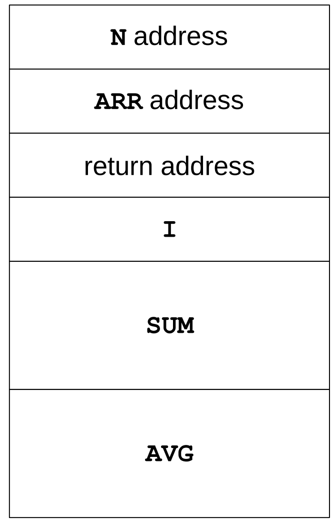
Static Allocation

- Static allocation of activation records is the simplest approach: allocate one activation record for every function, statically
- Older dialects of Fortran and Cobol used this system
- Simple and fast

12

Example

```
        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
```



13

Drawback

- Each function has one activation record
- There can be only one activation alive at a time
- Modern languages (including modern dialects of Cobol and Fortran) do not obey this restriction:
  - Recursion

14

Stacks of Activation Records

- To support recursion, we need to allocate a new activation record for each activation
- Dynamic allocation: activation record allocated when function is called
- For many languages, like C, it can be deallocated when the function returns
- A stack of activation records: *stack frames* pushed on call, popped on return

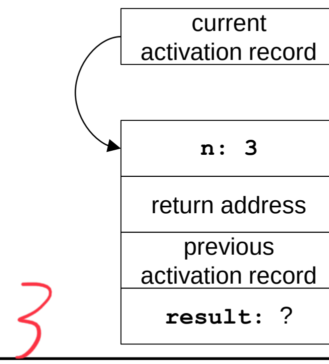
Current Activation Record

- Before, static: location of activation record was determined before runtime
- Now, dynamic: location of the *current* activation record is not known until runtime
- A function must know how to find the address of its current activation record
- Often, a machine register is reserved to hold this

C 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;
}
```

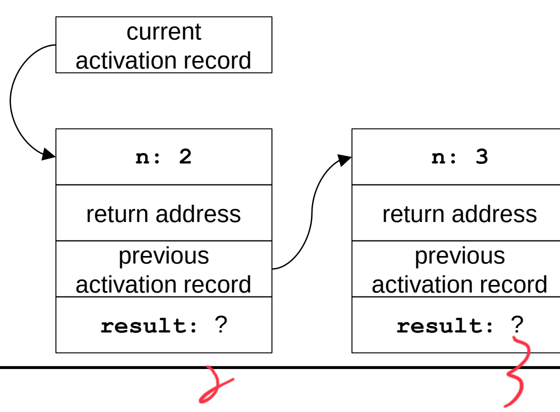


17

C 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;
}
```

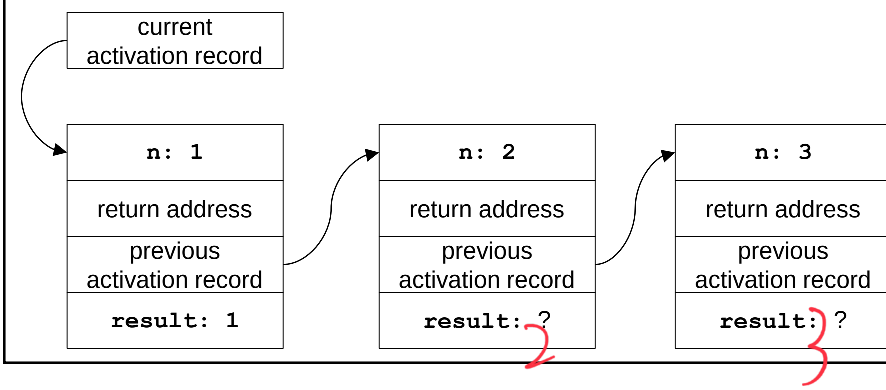


18

C 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;
}
```

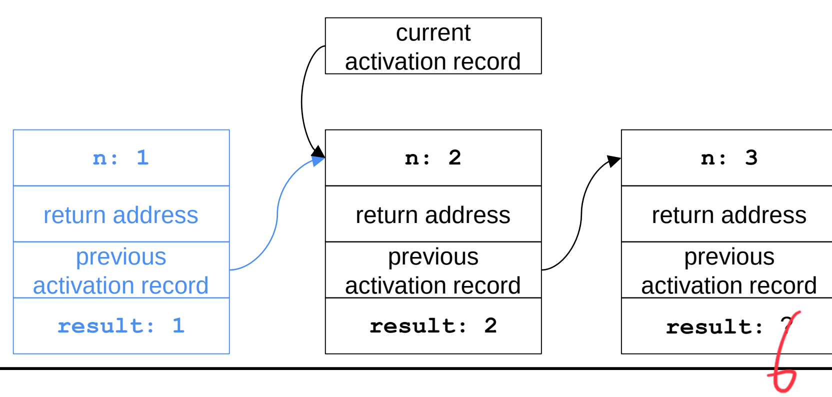


19

C 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;
}
```

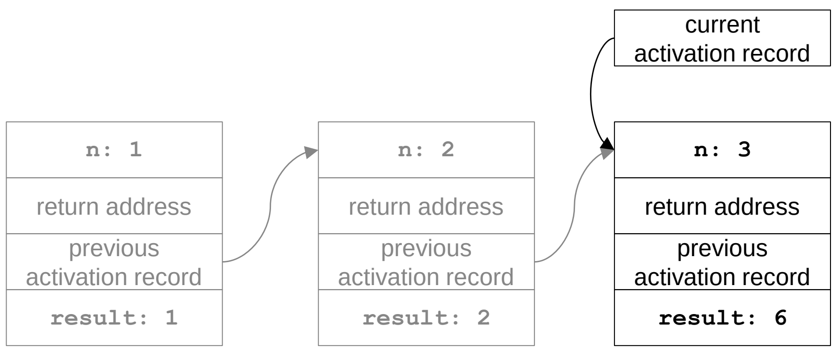


20

C Example

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

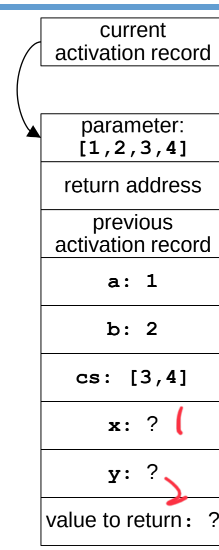
```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



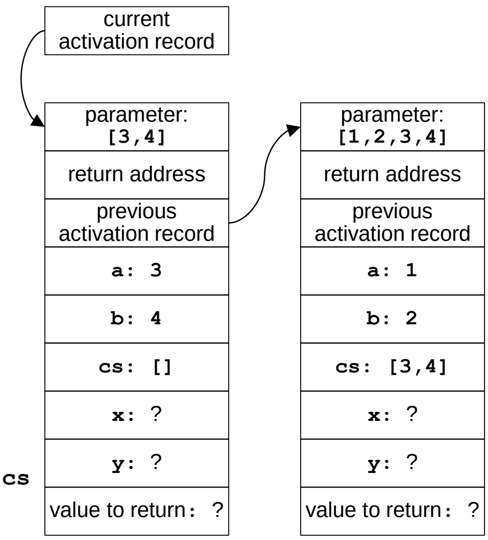
ML Example

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;
```



ML Example



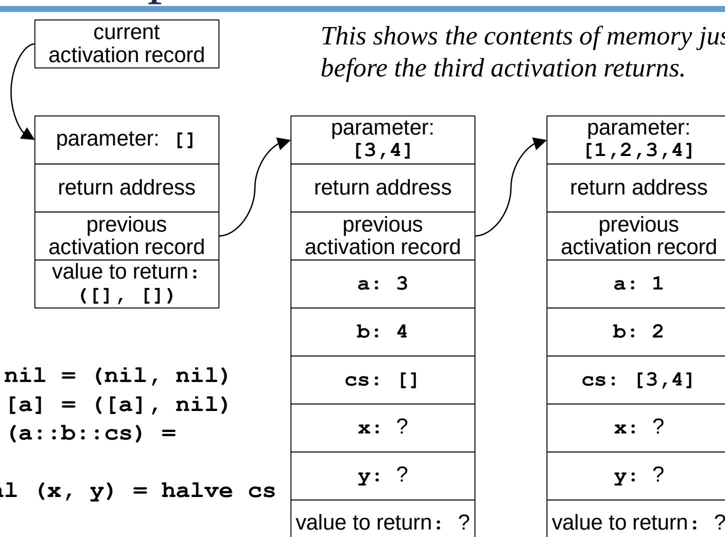
This shows the contents of memory just before the third activation.

```
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;
```

23

ML Example

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

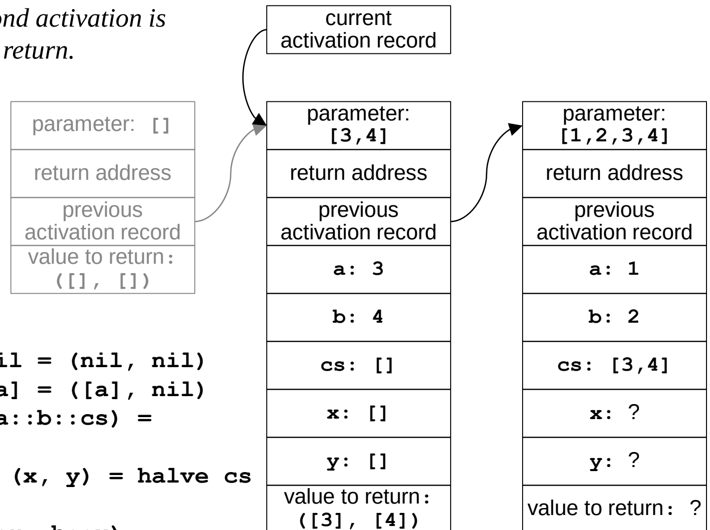



```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
      let
        val (x, y) = halve cs
      in
        (a::x, b::y)
      end;
```

24

ML Example

The second activation is about to return.

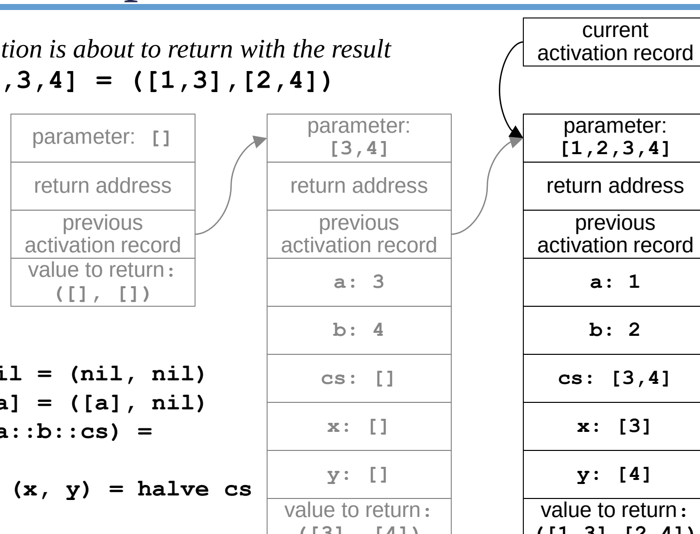



```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
      let
        val (x, y) = halve cs
      in
        (a::x, b::y)
      end;
```

25

ML Example

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;
```

26

Nesting Functions

- What we just saw is adequate for many languages, including C
- But not for languages that allow this trick:
  - Function definitions can be nested inside other function definitions
  - Inner functions can refer to local variables of the outer functions (under the usual block scoping rule)
- Like ML, Ada, Pascal, etc.

27

Example

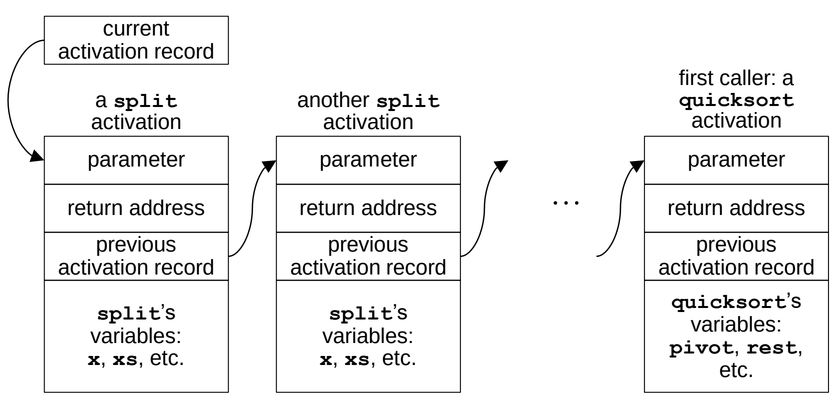
```
fun quicksort nil = nil
|   quicksort (pivot::rest) =
      let
        fun split(nil) = (nil,nil)
        |   split(x::xs) =
              let
                val (below, above) = split(xs)
              in
                if x<pivot then (x::below, above)
                else (below, x::above)
              end;
        val (below, above) = split(rest)
      in
        quicksort below @ [pivot] @ quicksort above
      end;
```

28

The Problem

- How can an activation of the inner function (**`split`**) find the activation record of the outer function (**`quicksort`**)?
- It isn't necessarily the previous activation record, since the caller of the inner function may be another inner function
- Or it may call itself recursively, as **`split`** does…

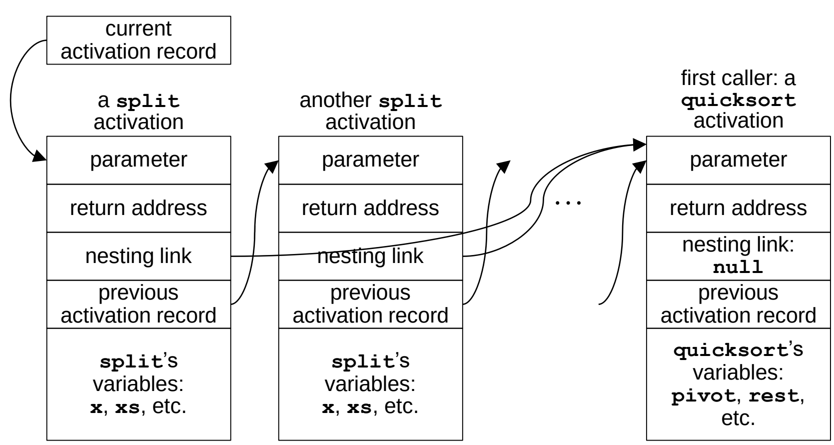
The Problem



Nesting Link

- An inner function needs to be able to find the address of the most recent activation for the outer function
- We can keep this *nesting link* in the activation record…

Nesting Link



Functions as Parameters

- When you pass a function as a parameter, what really gets passed?
- Code must be part of it: source code, compiled code, pointer to code, or implementation in some other form
- For some languages, something more is required…

Example

```
fun addXToAll (x,theList) =
  let
    fun addX y =
      y + x;
  in
    map addX theList
  end;
```

- This function adds **`x`** to each element of **`theList`**
- Notice: **`addXToAll`** calls **`map`**, **`map`** calls **`addX`**, and **`addX`** refers to a variable **`x`** in **`addXToAll`**'s activation record

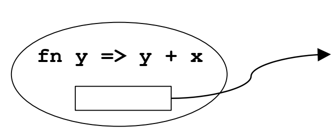
Nesting Links Again

- When **`map`** calls **`addX`**, what nesting link will **`addX`** be given?
  - Not **`map`**'s activation record: **`addX`** is not nested inside **`map`**
  - Not **`map`**'s nesting link: **`map`** is not nested inside anything
- To make this work, the parameter **`addX`** passed to **`map`** must include the nesting link to use when **`addX`** is called

35

Not Just For Parameters

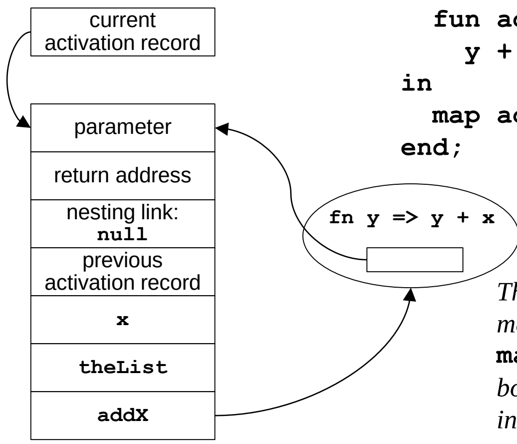
- Many languages allow functions to be passed as parameters
- Functional languages allow many more kinds of operations on function-values:
  - passed as parameters, returned from functions, constructed by expressions, etc.
- Function-values include both parts: code to call, and nesting link to use when calling it



36

Example

```
fun addXToAll (x,theList) =
  let
    fun addX y =
      y + x;
  in
    map addX theList
  end;
```



This shows the contents of memory just before the call to **`map`**. *The variable* **`addX`** *is bound to a function-value including code and nesting link.*

37

Questions?

38