Memory Locations For Variables
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?
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: \( \text{val } a = 0 \)
  - Bind \( a \) to the value zero

- But both need to connect variables to values represented in memory

- So both face the same binding question
Outline

- Activation records
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records
Function Activations

- 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.

- (Also called *dynamic* or *automatic*)
Activation-Specific Variables

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

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

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

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

```c
int nextcount() {
    static int count = 0;
    count = count + 1;
    return count;
}
```
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
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
Block Activation Records

- 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
Outline

- Activation-specific variables
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records
Static Allocation

- The simplest approach: allocate one activation record for every function, statically
- Older dialects of Fortran and Cobol used this system
- Simple and fast
**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
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
  - Multithreading
Outline

- Activation-specific variables
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records
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.

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

![Diagram of activation records]

- Current activation record
- Previous activation record
  - `n: 3`
  - Return address
  - Result: ?
This shows the contents of memory just before the third activation.

```c
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.

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

<table>
<thead>
<tr>
<th>n: 1</th>
<th>n: 2</th>
<th>n: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>return address</td>
<td>return address</td>
<td>return address</td>
</tr>
<tr>
<td>previous activation record</td>
<td>previous activation record</td>
<td>previous activation record</td>
</tr>
<tr>
<td>result: 1</td>
<td>result: ?</td>
<td>result: ?</td>
</tr>
</tbody>
</table>
The second activation is about to return.

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

```
<table>
<thead>
<tr>
<th>n: 1</th>
<th>n: 2</th>
<th>n: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return address</td>
<td>return address</td>
<td>return address</td>
</tr>
<tr>
<td>previous activation record</td>
<td>previous activation record</td>
<td>previous activation record</td>
</tr>
<tr>
<td>result: 1</td>
<td>result: 2</td>
<td>result: ?</td>
</tr>
</tbody>
</table>
```
The first activation is about to return with the result \( \text{fact}(3) = 6 \).

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

<table>
<thead>
<tr>
<th>n: 1</th>
<th>n: 2</th>
<th>n: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>return address</td>
<td>return address</td>
<td>return address</td>
</tr>
<tr>
<td>previous activation record</td>
<td>previous activation record</td>
<td>previous activation record</td>
</tr>
<tr>
<td>result: 1</td>
<td>result: 2</td>
<td>result: 6</td>
</tr>
</tbody>
</table>
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.

```plaintext
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;
```

<table>
<thead>
<tr>
<th>current activation record</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter: [3, 4]</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>previous activation record</td>
</tr>
<tr>
<td>a: 3</td>
</tr>
<tr>
<td>b: 4</td>
</tr>
<tr>
<td>cs: []</td>
</tr>
<tr>
<td>x: ?</td>
</tr>
<tr>
<td>y: ?</td>
</tr>
<tr>
<td>value to return: ?</td>
</tr>
</tbody>
</table>

| previous activation record |
| a: 1                       |
| b: 2                       |
| cs: [3, 4]                 |
| x: ?                       |
| y: ?                       |
| value 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;

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

<table>
<thead>
<tr>
<th>parameter:</th>
<th>[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>return address</td>
<td></td>
</tr>
<tr>
<td>previous activation record</td>
<td></td>
</tr>
<tr>
<td>value to return:</td>
<td>([], [])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parameter:</th>
<th>[3, 4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>return address</td>
<td></td>
</tr>
<tr>
<td>previous activation record</td>
<td></td>
</tr>
<tr>
<td>a:</td>
<td>3</td>
</tr>
<tr>
<td>b:</td>
<td>4</td>
</tr>
<tr>
<td>cs:</td>
<td>[]</td>
</tr>
<tr>
<td>x:</td>
<td>?</td>
</tr>
<tr>
<td>y:</td>
<td>?</td>
</tr>
<tr>
<td>value to return:</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parameter:</th>
<th>[1, 2, 3, 4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>return address</td>
<td></td>
</tr>
<tr>
<td>previous activation record</td>
<td></td>
</tr>
<tr>
<td>a:</td>
<td>1</td>
</tr>
<tr>
<td>b:</td>
<td>2</td>
</tr>
<tr>
<td>cs:</td>
<td>[3, 4]</td>
</tr>
<tr>
<td>x:</td>
<td>?</td>
</tr>
<tr>
<td>y:</td>
<td>?</td>
</tr>
<tr>
<td>value to return:</td>
<td>?</td>
</tr>
</tbody>
</table>
The second activation is about to return.

```plaintext
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 \( \text{halve} \ [1,2,3,4] = ([1,3],[2,4]) \)

```
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;
```
Outline

- Activation-specific variables
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records
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.
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 \\
|                               | end;
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...
current activation record

a split activation

parameter

return address

previous activation record

split’s variables: x, xs, etc.

another split activation

parameter

return address

previous activation record

split’s variables: x, xs, etc.

... first caller: a quicksort activation

parameter

return address

previous activation record

quicksort’s variables: pivot, rest, etc.
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.
current activation record

a split activation

parameter

return address

nesting link

previous activation record

split’s variables: x, xs, etc.

another split activation

parameter

return address

nesting link

previous activation record

split’s variables: x, xs, etc.

... first caller: a quicksort activation

parameter

return address

nesting link: null

previous activation record

quicksort’s variables: pivot, rest, etc.
Setting The Nesting Link

■ Easy if there is only one level of nesting:
  – Calling outer function: set to null
  – Calling from outer to inner: set nesting link same as caller’s activation record
  – Calling from inner to inner: set nesting link same as caller’s nesting link

■ More complicated if there are multiple levels of nesting...
Multiple Levels Of Nesting

function f1
  variable v1

function f2
  variable v2

  function f3
    variable v3

- References at the same level ($f_1$ to $v_1$, $f_2$ to $v_2$, $f_3$ to $v_3$) use current activation record
- References $n$ nesting levels away chain back through $n$ nesting links
Other Solutions

- The problem: references from inner functions to variables in outer ones
  - Nesting links in activation records: as shown
  - Displays: nesting links not in the activation records, but collected in a single static array
  - Lambda lifting: problem references replaced by references to new, hidden parameters
Outline

■ Activation-specific variables
■ Static allocation of activation records
■ Stacks of activation records
■ Handling nested function definitions
■ Functions as parameters
■ Long-lived activation records
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

```plaintext
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 \texttt{map} calls \texttt{addx}, what nesting link will \texttt{addx} be given?
  - Not \texttt{map}’s activation record: \texttt{addx} is not nested inside \texttt{map}
  - Not \texttt{map}’s nesting link: \texttt{map} is not nested inside anything

■ To make this work, the parameter \texttt{addx} passed to \texttt{map} must include the nesting link to use when \texttt{addx} is called
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 code to call, and nesting link to use when calling it
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.
Outline

- Activation-specific variables
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records
One More Complication

What happens if a function value is used after the function that created it has returned?

```plaintext
fun test = let
  val f = funToAddX 3;
  in
  f 5
end;

fun funToAddX x = let
  fun addX y = y + x;
  in
  addX
end;
```
fun test = 
  let
    val f = funToAddX 3;
  in
    f 5
  end;

fun funToAddX x = 
  let
    fun addX y = 
      y + x;
  in
    addX
  end;

This shows the contents of memory just before
funToAddX returns.
fun test = 
  let 
    val f = funToAddX 3; 
    in 
      f 5 
      end; 
  end; 
fun funToAddX x = 
  let 
    fun addX y = 
      y + x; 
    in 
      addX 
      end; 
  end;

After \texttt{funToAddX} returns, \texttt{f} is the bound to the new function-value.
The Problem

- When `test` calls `f`, the function will use its nesting link to access `x`
- That is a link to an activation record for an activation that is finished
- This will fail if the language system deallocated that activation record when the function returned
The Solution

- For ML, and other languages that have this problem, activation records cannot always be allocated and deallocated in stack order.

- Even when a function returns, there may be links to its activation record that will be used; it can’t be deallocated it is unreachable.

- Garbage collection: chapter 14, coming soon!
Conclusion

The more sophisticated the language, the harder it is to bind activation-specific variables to memory locations

- Static allocation: works for languages that permit only one activation at a time (like early dialects of Fortran and Cobol)
- Simple stack allocation: works for languages that do not allow nested functions (like C)
Conclusion, Continued

- Nesting links (or some such trick): required for languages that allow nested functions (like ML, Ada and Pascal); function values must include both code and nesting link
- Some languages (like ML) permit references to activation records for activations that are finished; so activation records cannot be deallocated on return