CSCI 334: Principles of Programming Languages

Lecture 12: Control Structures III

Instructor: Dan Barowy

Williams

Announcements

Midterm exam next class.

Thursday, March 15 in TCL 206

during class meeting time.

Announcements

Study session tonight from 4-5pm.

Will be audio recorded if you cannot make it.

You will drive; bring questions, please.

Announcements

HW3 grades nearly done.

I will do my best to get you HW4 grades but no promises!

A note about stack diagrams

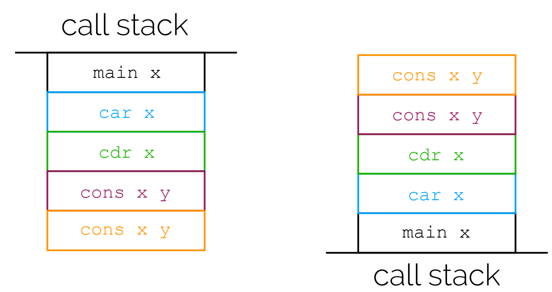
Mitchell draws them upside down for historical reasons.

Pedagogically bad.
We push values "on the top" of a stack.

Also, for functional languages, not technically correct.

Anyway, draw them whichever way you want.
I will draw them right-side up.

A note about stack diagrams



Mitchell   me

Blocks

- What is a "block"?
- Not the same "block" as in "block structured language"!
- A block denotes scope.
- You've seen them before.

```
public static void main(String[] args) {
  // code inside block
}
```

Blocks

```
public static void main(String[] args) {
  return x;
}
```

- What kind of variable is `x`?
- `x` "is global to" the *scope* of `main`.

Scope

- A variable is a binding of a *value* to a *name*.
- Scope is the region of a computer program where a variable binding is valid.

```
class Program {
  static int x = 5;
  public static void main(String[] args) {
    return x;
  }
}
```

Block Scope

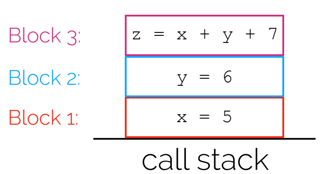
- A block is therefore a region associated with variable bindings.
- This is valid (although not very useful) C:

```
{ int x = 5;
  { int y = 6;
    { int z = x + y + 7; }
  }
}
```

- Scopes are tracked on the runtime call stack.

Block Scope Evaluation

```
Start block 1:   { int x = 5;
Start block 2:     { int y = 6;
Start block 3:       { int z = x + y + 7; }   End block 3:
End block 2:       }
End block 1:     }
```



- `z` is local to block 3; `x` and `y` are global to block 3.

Scoping Rules

```
val x = 5
fun f y = x + y
fun g () =
  let val x = 6 in
    f 7
  end
g ()
```

- Show of hands:
  - Option 1: result is 13 — dynamic scope
  - Option 2: result is 12 — lexical (static) scope

Dynamic Scope

```
val x = 5
fun f y = x + y
fun g () =
  let val x = 6 in
    f 7
  end
g ()
```

Scope of f's variable x is where f is *used*.

Lexical (Static) Scope

```
val x = 5
fun f y = x + y
fun g () =
  let val x = 6 in
    f 7
  end
g ()
```

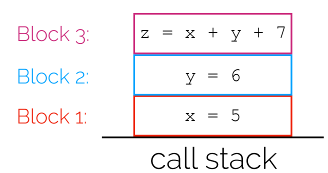
Scope of f' variable x is where f is *defined*.

Dynamic vs Lexical Scope

- Dynamic scope is very confusing for programmers.
- LISP originally had dynamic scope.
- Scheme introduced lexical scope into LISP; Common LISP did the same.
- Some modern languages still make this mistake! (e.g., R; demo)

Lexical Scope Rules Are Simple

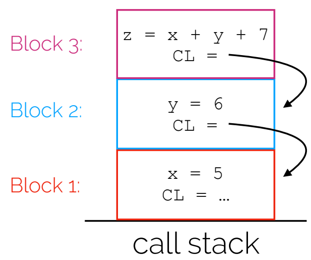
```
Start block 1:   { int x = 5;
Start block 2:     { int y = 6;
Start block 3:       { int z = x + y + 7; }   End block 3:
End block 2:       }
End block 1:     }
```



- When resolving the value of a variable, start search locally, then traverse up the call stack.

Lexical Scope Rules Are Simple

- Some languages (mostly functional ones) maintain explicit "control link" pointers to previous stack frames.



- (You'll see why a bit later)

First Class Functions

- A language with *first-class functions* treats functions no differently than any other value:
- You can assign functions to variables:

```
val f = fn x => x + 1
```

- You can pass functions as arguments:

```
fun g h = h 3
g f
```

- You can return functions:

```
fun k x = fn () => x + 3
```

- First-class function support complicates *implementation* of lexical scope.

First Class Functions

- To implement support for first class functions, we need two additional data structures:
  - Access links
  - Closures
- The implementation difficulty of maintaining lexical scope for first class function is called the *funarg problem*.
- Scheme was the first language to fix it.
- This difficulty was why LISP had dynamic scope!

Access link

- An *access link* is a pointer from the current activation record to the activation record of the closest lexical scope.

- In other words, the access link in the activation frame for a function f points to where f was defined.

Closure

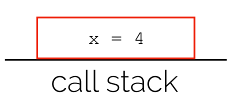
- A *closure* is a tuple that represents a function value. One tuple value points to a function's code and the other value points to the activation record of the point of definition of the function (i.e., closest lexical scope).

Example

```
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
g f
```

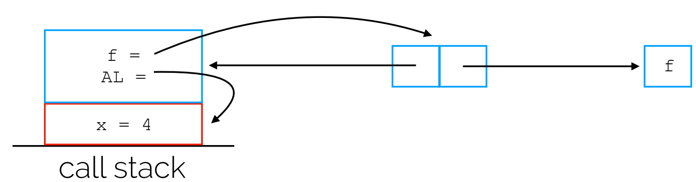
Blocks Define Activation Records

```
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
g f
```



Blocks Define Activation Records

```
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
g f
```

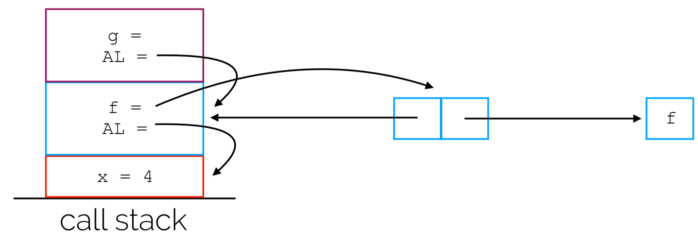


Desugared `fun` Bindings

```
let val x = 4 in
  let f = fn y => x * 4 in
    let g =
      fn h => let val x = 7 in (h 3) + x in
        g f
    end
  end
end
```

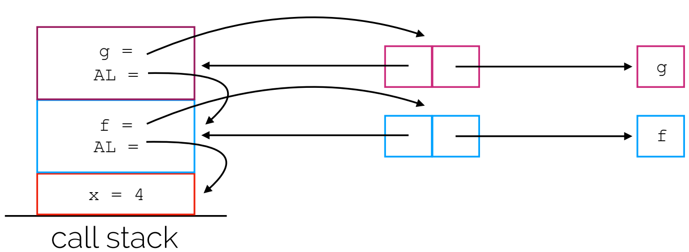
Blocks Define Activation Records

```
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
g f
```



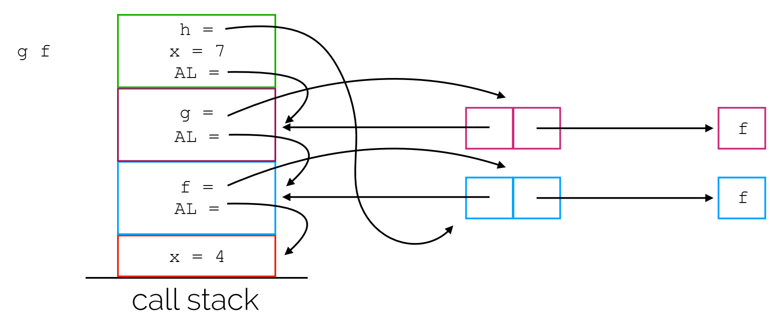
Blocks Define Activation Records

```
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
g f
```



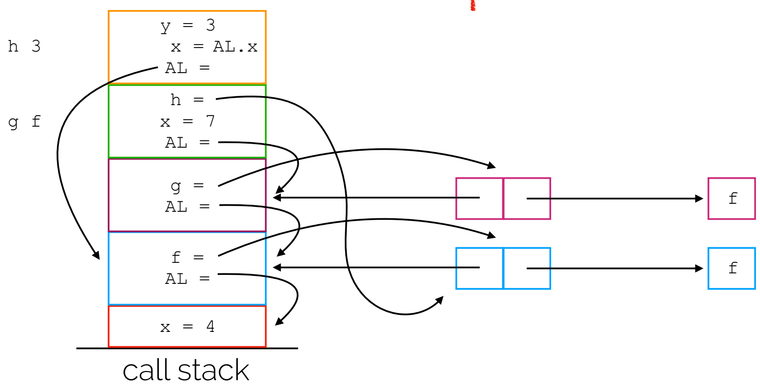
Blocks Define Activation Records

```
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
g f
```



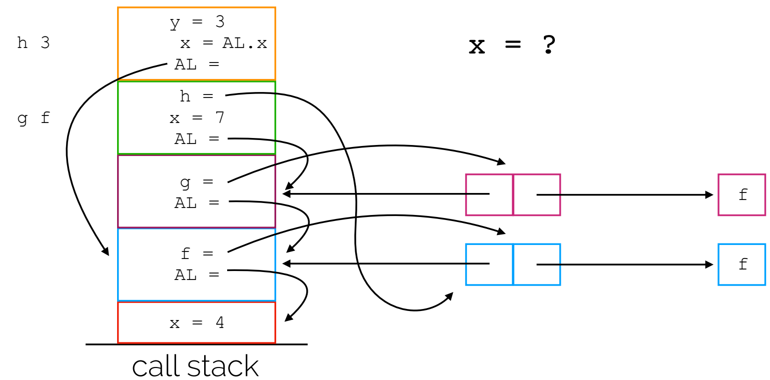
Blocks Define Activation Records

```
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
g f
```



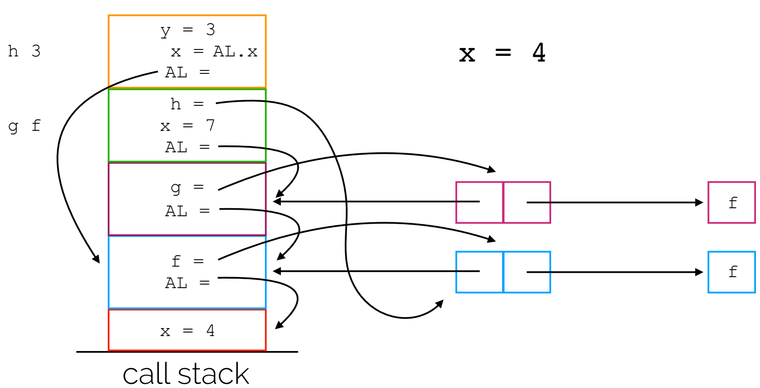
Blocks Define Activation Records

```
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
g f
```



Blocks Define Activation Records

```
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
g f
```



Activation Records in Functional Langs

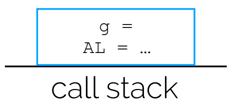
```
let val g =
  let
    val x = 1
    fun f () = x + 1
  in
    f
  end
in
  g()
end
```

How is this function evaluated? Do we have a problem when we call `g()`?

Upward funargs

```
let val g =
  let val x = 1
      fun f () = x + 1
  in f end
in g() end
```

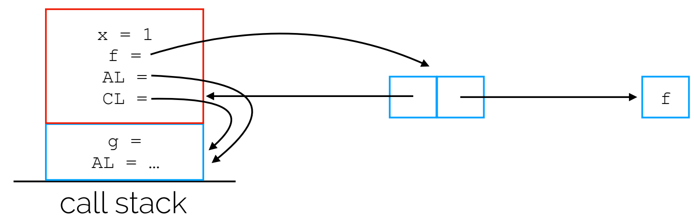
1. Push `let` block for g onto call stack. We don't yet know g's value.



Upward funargs

```
let val g =
  let val x = 1
      fun f () = x + 1
  in f end
in g() end
```

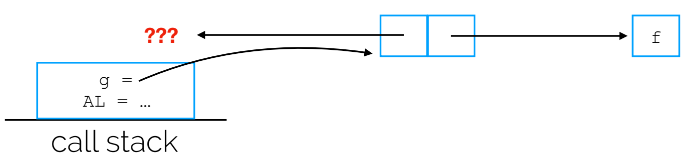
1. Push `let` block for g onto call stack. We don't yet know g's value.
2. Push let block for x and f.



Upward funargs

```
let val g =
  let val x = 1
      fun f () = x + 1
  in f end
in g() end
```

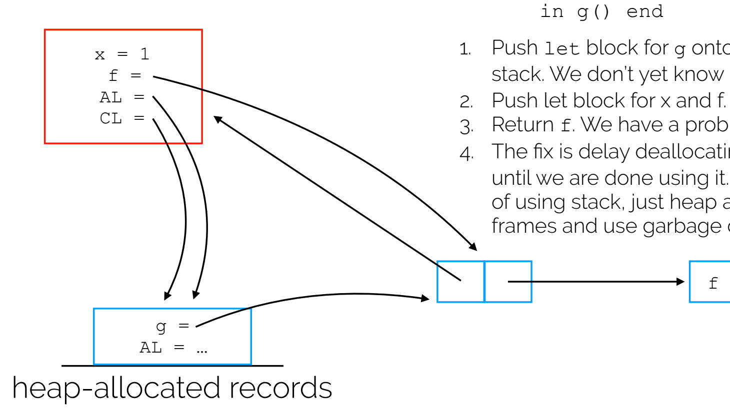
1. Push `let` block for g onto call stack. We don't yet know g's value.
2. Push let block for x and f.
3. Return `f`. We have a problem!



Upward funargs

```
let val g =
  let val x = 1
      fun f () = x + 1
  in f end
in g() end
```

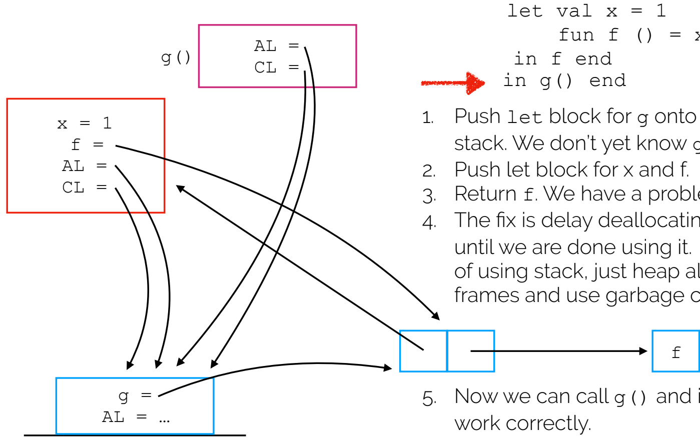
1. Push `let` block for g onto call stack. We don't yet know g's value.
2. Push let block for x and f.
3. Return `f`. We have a problem!
4. The fix is delay deallocating record until we are done using it. Instead of using stack, just heap allocate frames and use garbage collector!



Upward funargs

```
let val g =
 let val x = 1
     fun f () = x + 1
  in f end
 in g() end
```

1. Push `let` block for `g` onto call stack. We don't yet know `g`'s value.
2. Push let block for x and f.
3. Return `f`. We have a problem!
4. The fix is delay deallocating record until we are done using it. Instead of using stack, just heap allocate frames and use garbage collector!



heap-allocated records

5. Now we can call `g()` and it will work correctly.