CS342: Organization of Prog. Languages

Topic 7: Bindings and Scopes

- Bound and free names.
- Scope = Visibility.
- Extent = Lifetime.
- Lexical and dynamic scoping.
- Implementation methods for lexical and dynamic scoping.

Bound and Free Names

- Names (variables, parameters, etc) are introduced by various language constructs, e.g. `lambda`, `let`, `do`, etc in Scheme, or `auto`, `static`, `extern`, etc in C.
- Constructs which introduce a new variable and make it local to a particular region of code are called *binding* constructs.
- *Statically*, these give *declarations* for the names. *Dynamically*, thes constructs create *locations* in which to store the values.
- The association of a name to a location is called a *binding*.
- A name which is local to a region of code (e.g. a function, statement or expression) is said to be *bound*.
- A name wich is not local is said to be *free*.
- Sometimes we talk about bound and free variables, even though the concept is more general and applies to names of any sort, not just variables.

Scope = Visibility

- A given name will be visible in some parts of a program, but not others.
- The parts of the program where a name is visible is called the name's "scope."
- E.g. consider the following program:

  ```
  (define (my-apply fn arg) (fn arg))
  (define (f a)
     (define (g) (cons 2 a))
     (my-apply g a) )

  (f (list 3 4 5))
  ```

  In the Scheme programming language, the scope of the parameter "a" of f is the entire body of the function f, including the nested functions and expressions.

  I.e. the "a" inside g refers to the parameter "a" of f.
- This is different in different Lisps.

Extent = Lifetime

- A given name will retain its value for the execution of a certain part of a program.
- The part of the program's execution during which a name binding is defined is called the name binding's "extent."
- E.g. consider the following program:

```
(define (my-apply fn arg) (fn arg))
(define (f a)
   (define (g) (cons 2 a))
   (my-apply g a) )

(f (list 3 4 5))
```

  In the Scheme programming language, the extent of the binding for parameter "`a`" of f, is the entire evaluation of the call to f.

- The extent of a name binding should not be confused with the lifetime of the object to which it refers.

Scope $\neq$ Extent

- In many programming languages, a variable may be out of scope but within the extent of its binding.
- Consider our Scheme example:

```
(define (my-apply fn arg) (fn arg))
(define (f a)
   (define (g) (cons 2 a))
   (my-apply g a) )

(f (list 3 4 5))
```

  During the call to `my-apply` we are in the *extent* of the binding of f's `a` to `'(3 4 5)`, but we are outside its `scope`.

- Extent is a dynamic concept related to lifetimes of variable bindings or objects.
- Scope can be either a dynamic concept or a static concept, depending on the programming language.

Static Scoping

- Scheme, C, Pascal, and many other languages have what is known as *static scoping*.

  This is sometimes callde *lexical scoping*.

- With static scoping you can tell precisely which variable bindings will be visible to a particular program just by reading it.

  The execution environment cannot change what is in scope.

- In C, the static scoping is trivial because there cannot be nested functions. There can, however, be local bindings in a compound statement.

- In Pascal, the static scoping is applied to the nested procedure and function blocks.

  The fact that functions and procedures can only be passed as arguments (and cannot be returned as values), guarantees that extent of the variables of functions and procedures behaves according to a stack discipline.

- In Scheme, static scoping is applied to nested lambda expressions (functions).

  Since functions can be returned as values, it is possible (and indeed quite useful) that theresult functions may refer to variable bindings of functions which are no longer active.

  ```
  (define (make-counter)
     (let ((n 0))
          (lambda () (set! n (+ n 1)) n)))

  (define click-counter (make-counter))
  (define error-counter (make-counter))
  ```

  Now `click-counter` and `error-counter` each refer to their own captured binding of n, even though `make-counter` is no longer active.

  Thus using static scope and closures we can produce a more powerful version of C's "static" (or Algol's "own") variables.

- With lexical scoping, it is possible to create “holes” in the scope of a name, by binding variable with the same name in a nested scope.

  These holes can always be recognized by a simple examination of the program.

Dynamic Scoping

- Dynamic scoping of names defers the determination of which bindings are visible until execution time.

  Dynamically scoped names are sometimes called "*fluid variables*" or variables with "*fluid scoping.*"

- The usual rule for dynamic scoping is that a free name is bound by the closest binding for that name as we ascend the call chain.

- Dynamic scope is error prone when it is the sole non-local scoping mechanism.

- Dynamic scope is very useful when it is used judiciously to modify the environment in which a computation will take place.

  E.g. rebinding standard input and output streams, rebinding error handlers, ...

Example of Dynamic Binding

Pseudo-code example:

```
#include "axllib"

fluid s: String := "Outermost";

h(): () == {
   print << "h Beg s = " << s << newline;
   f();
   print << "h Mid s = " << s << newline;
   g();
   print << "h End s = " << s << newline;
}

g(): () == {
   fluid s: String := "g’s very own string";
   f();
}

f(): () == {
   print << "f     s = " << s << newline;
}
h();
```

This yields

```
h Beg s = Outermost
f     s = Outermost
h Mid s = Outermost
f     s = g’s very own string
h End s = Outermost
```

Implementation of Bindings

Psuedo-code example:

```
function hout() {
    local x = "hox", y = "hoy";

    function hin() {
        local x = "hix", a = "hia";
        print x, y, a;
    }
    function hbro() {
        local y = "hby";
        call hin();
    }
    function hcall(f) {
        local y = "hcy";
        call f();
    }

   call hbro();
   call hcall(hin);
}

call hout();
```

Static scoping gives:

```
hix hoy hia
hix hoy hia
```

Dynamic scoping gives:

```
hix hby hia
hix hcy hia
```

Static Scoping

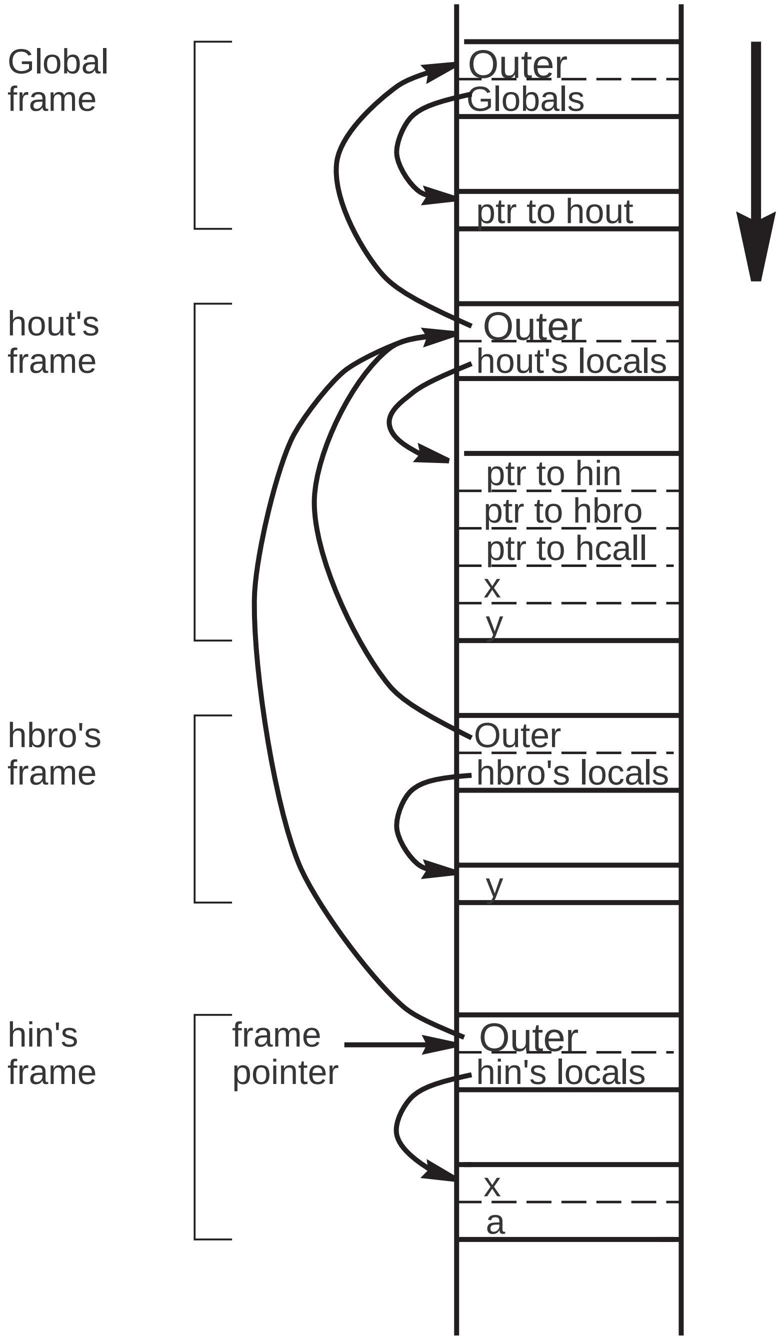
This diagram shows the layout of the stack after hout calls hbro which in turn calls hin (simplified view).

Each frame has a pointer to the closest active frame of the function which lexically enclosed it in the program source.

Static variables are found by looking up along this chain for the closest cell corresponding to the desired name.

In this example, x is found in hin's locals and y is found in hout's locals.

Note that the number of links to follow to find the correct frame, and the offset of the desired variable within that frame are completely determined statically.

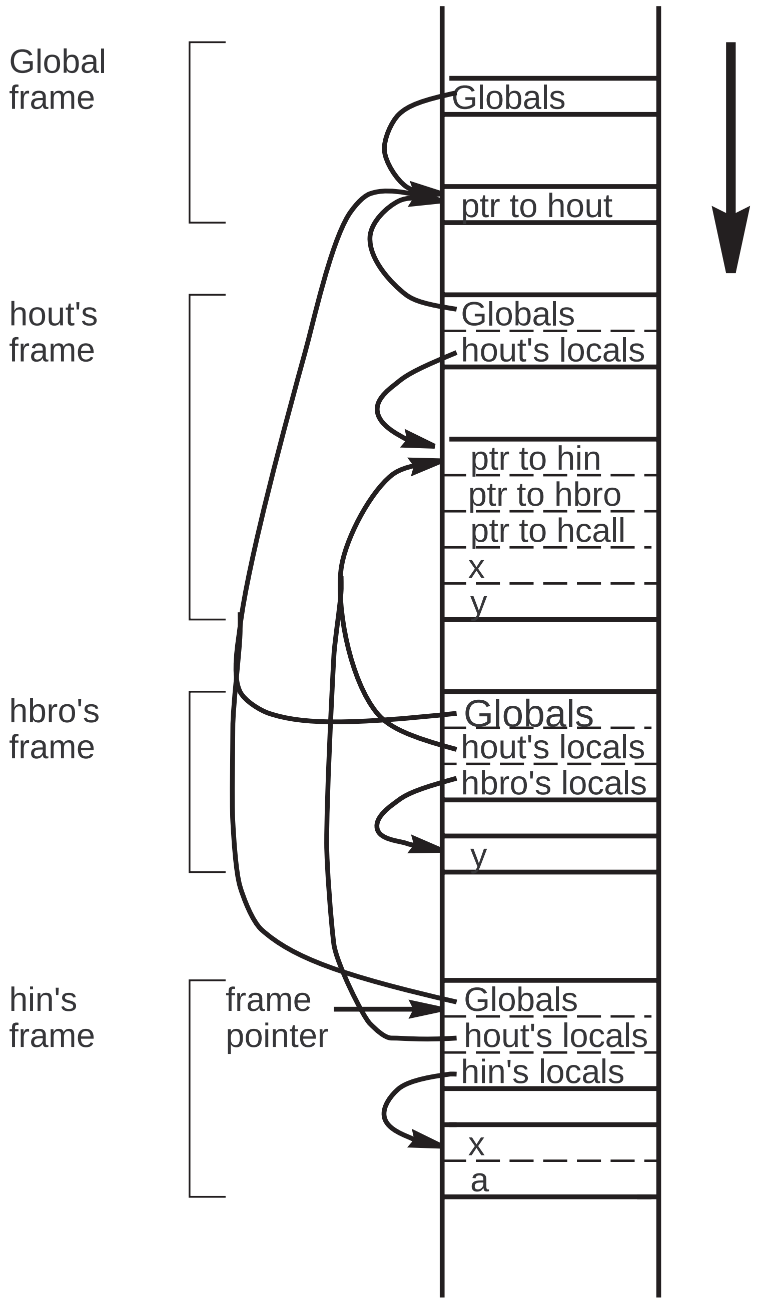


A faster implementation of Static Scoping

Rather than daisy chain our way up the stack to find the n-th outer level, it is faster to maintain a short array of pointers to all of the outer levels in each frame.

This array of pointers is called a "display."

We can order the elements of the display so that to access the k-th variable in the i-th level out, we essentially access FP[i][k].



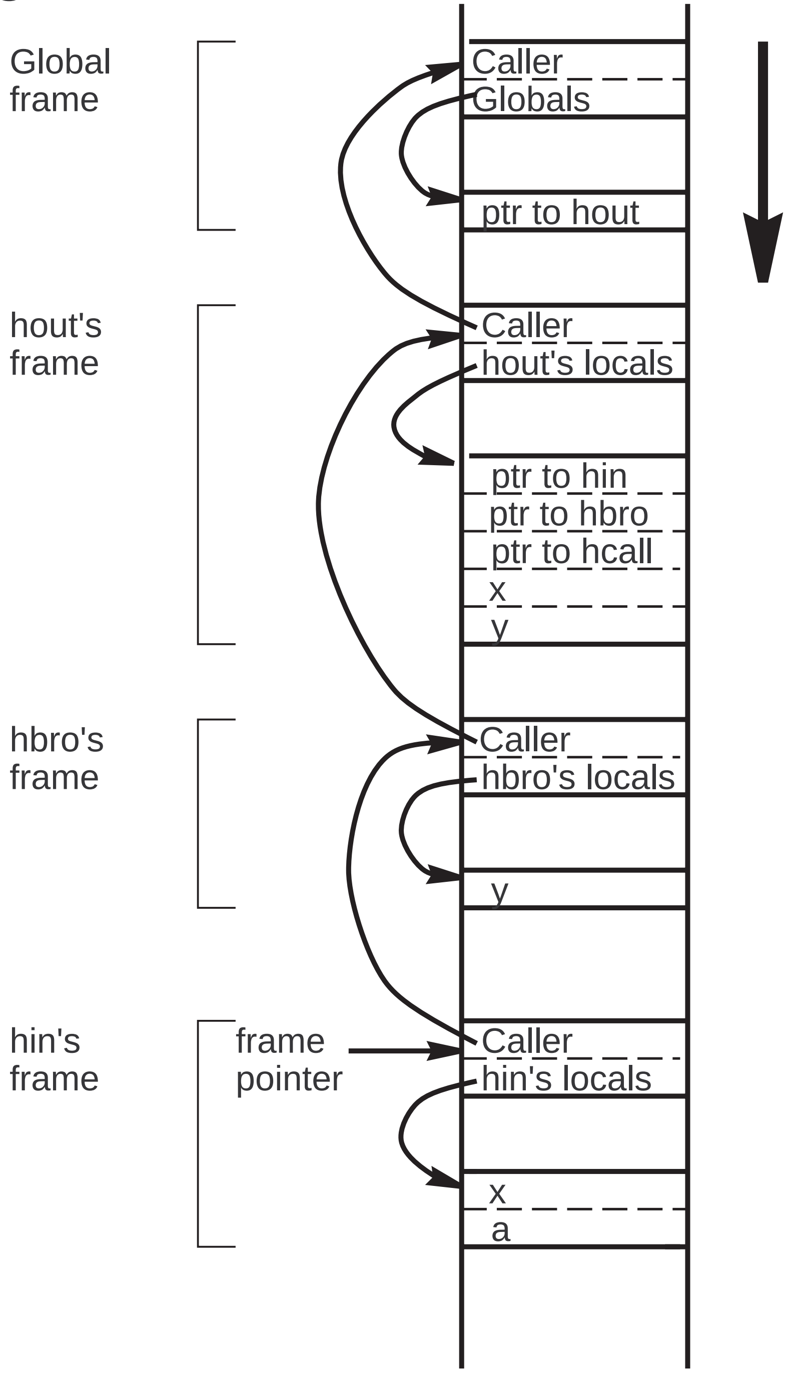
Dynamic Scoping

Here hout has called hbro which has called hin.

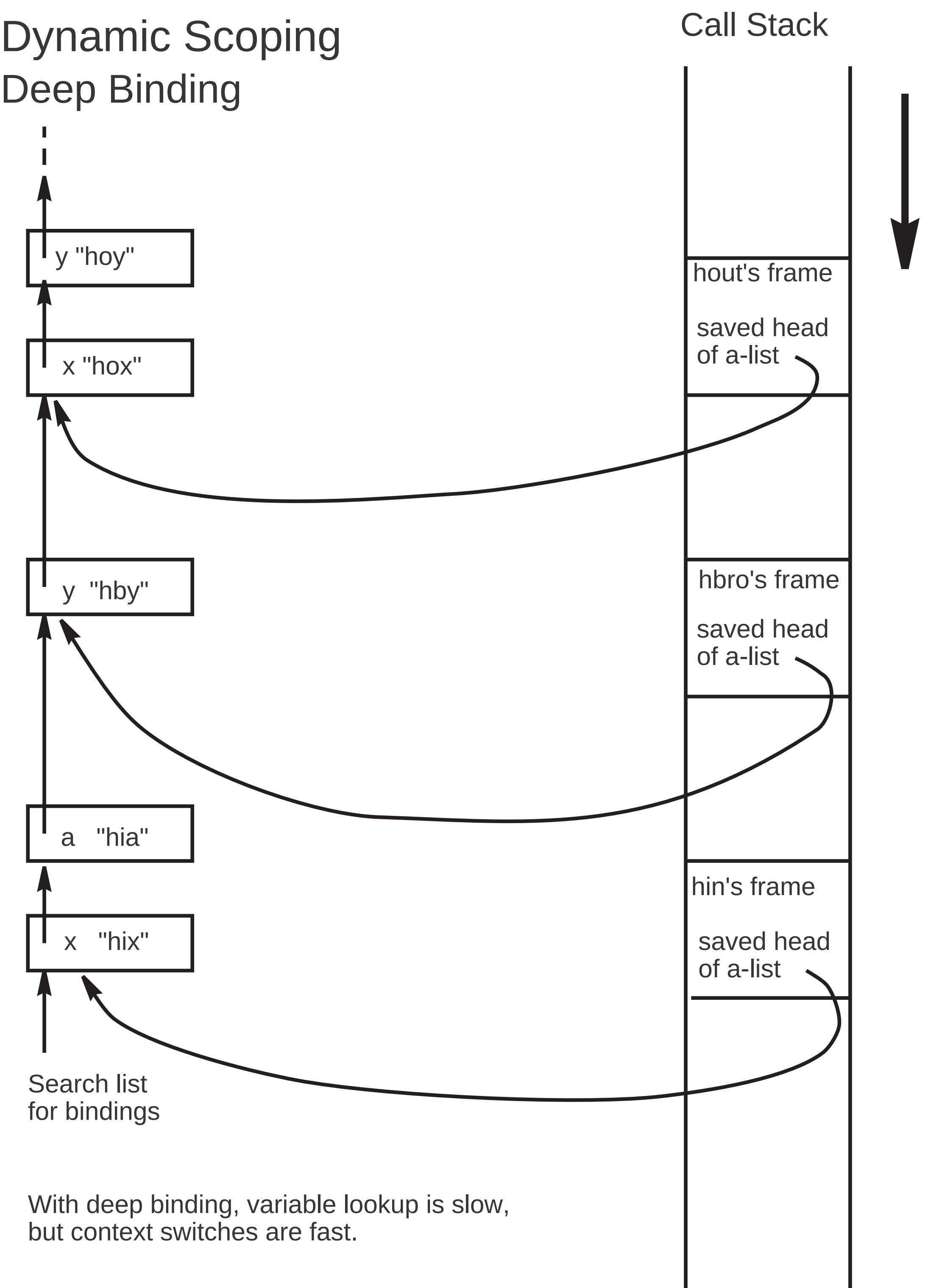
We can find the binding cell for any name by starting with the current frame pointer and looking up the chain of callers to find the first instance of a variable with the given name.

E.g. Here x would be found in hin's locals, and y would be found in hbro's locals.

In practice, the binding cells need not be integrated in the call stack.

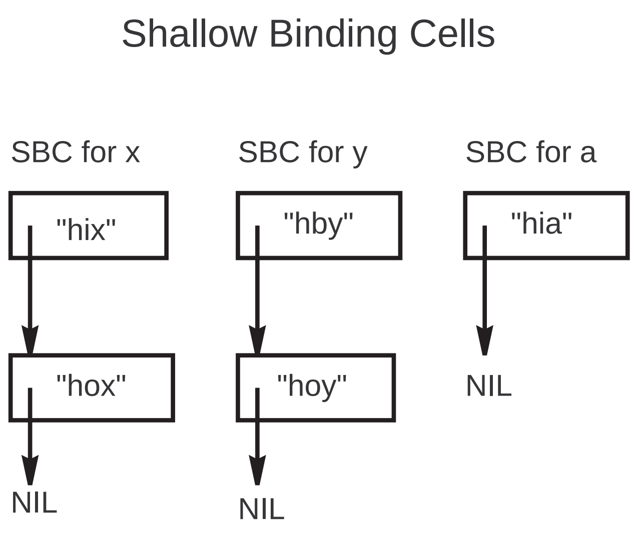


Dynamic Scoping
Deep Binding



With deep binding, variable lookup is slow,
but context switches are fast.

Dynamic Scoping
Shallow Binding

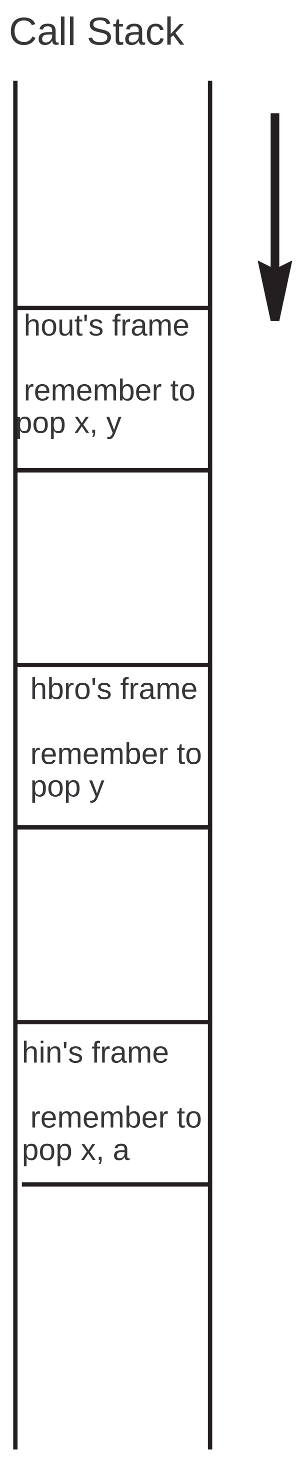


This diagram sketches the "shallow binding" implementation of dynamically scoped variables.

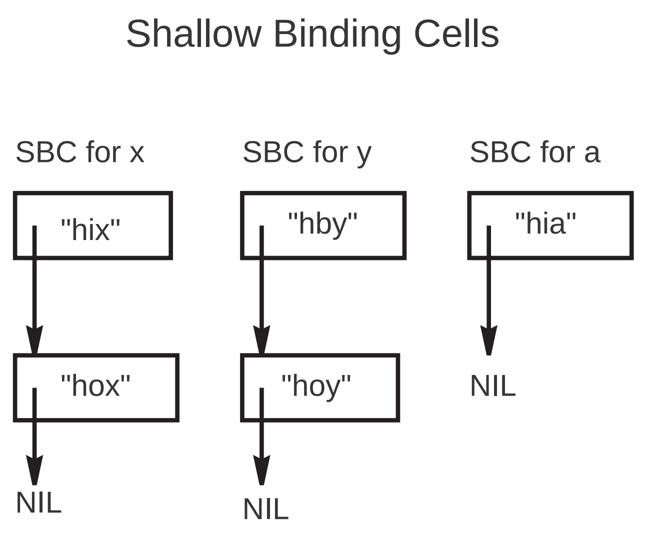
Each variable has its own "shallow binding cell" which can be found in a hash table or pointed to directly from a symbol object.

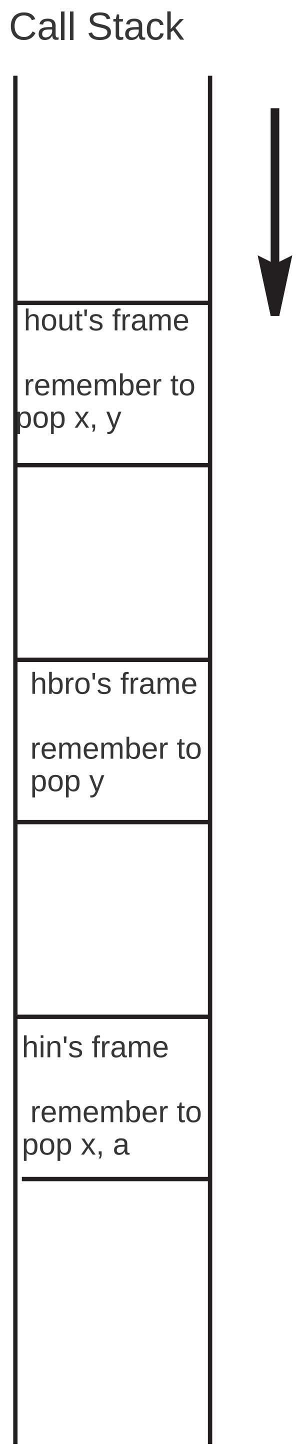
Function entry pushes a new element onto the sbc for each bound variable.

With this implementation, variable lookup is fast, but context switching is slow because heads of many sbcs need to be adjusted.



Dynamic Scoping
Shallow Binding





This diagram sketches the "shallow binding" implementation of dynamically scoped variables.

Each variable has its own "shallow binding cell" which can be found in a hash table or pointed to directly from a symbol object.

Function entry pushes a new element onto the sbc for each bound variable.

With this implementation, variable lookup is fast, but context switching is slow because heads of many sbcs need to be adjusted.