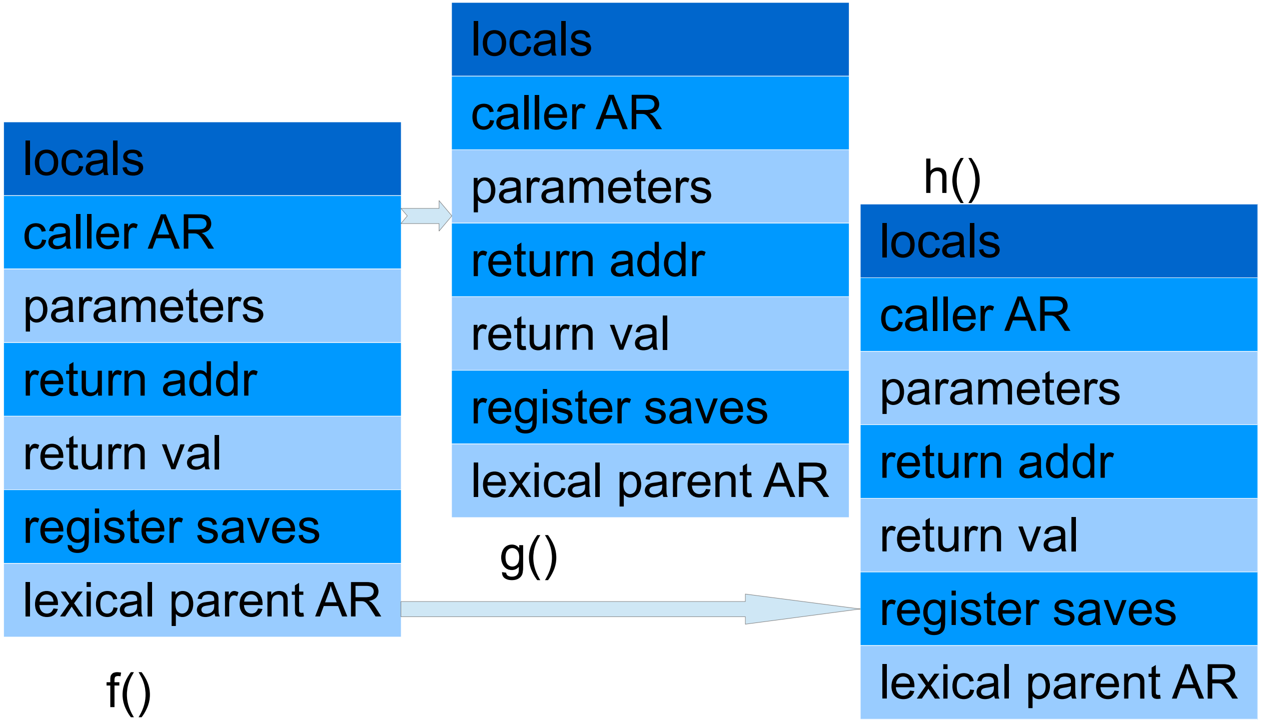
Activation records

- represent the runtime block of memory associated with a specific function invocation (i.e. each individual call)
- compiler likely needs to:
    - determine layout of the AR, size of each portion for the current call, and offsets to each element within
    - generate function code to set up, use, and clean up the ARs
    - allocate AR segments for return value, parameters, return address, saved registers, local variables
    - include mechanism to access scopes of lexical ancestors

AR chains

- callee AR likely includes a reference to that of the caller
- for nested declarations may include ref to AR of lexical parent



Example:
- f is defined inside h
- f is called from g

Accessing ancestor AR

- suppose function h includes definition of variable x, at offset of 8 from start of h's local variable space
- suppose function f is defined inside definition of function h (i.e. nested function definitions permitted)
- if function f refers to x, compiler inserts pair <n,offset> where n is the number of lexical ancestors to traverse, e.g.
  - <1,8> would mean look in the AR of f's lexical parent at offset 8
  - <0,4> would mean look in f's AR at offset 0

Optimizing ancestor access

- Maintain a global array of AR pointers
  - arr[0] is pointer to current AR
  - Arr[1] is pointer to lexical parent's AR
  - Arr[2] is pointer to their lexical parent's AR
  - etc
- Update the array contents on each function call/return
- On reference to <n,offset> simply look up arr[n] instead of following AR pointer chain

AR and the local code block

- abstraction might separate the allocation/initialization of local variables from the execution code statements

```
int f(int x) {
    int y = 3;
    y = y + x;
    return y;
}
```

- might initialize y to 3 as part of setting up the AR, in which case the first executable statement in the code block would become the “y = y + x;”

Storing the ARs

- stack-based approach:
  - caller's AR is on stack immediately below callee's AR
  - assumes callee exits before caller resumes, not possible for callee to outlive caller
- heap-based approach:
  - actually maintain linked list of AR's, allocated in heap
  - supports concurrency, where caller and callee can continue/end independently

Optimizations

- leaf subroutines: subroutines that don't call any others
  - don't actually need to be on stack
  - can keep a static AR someplace just for leaf subroutines
- Fixed call sequences: an invariant sequence of calls
  - e.g. X always calls Y which always calls Z, and Z is a leaf
  - Possible optimization by combining X,Y,Z into a single AR