Topics

- Nested Subprograms
- Blocks
- Implementing Dynamic Scoping

Nested Subprograms

- Some non-C-based static-scoped languages (e.g., Fortran 95+, Ada, Python, JavaScript, Ruby, and Swift) use stack-dynamic local variables and allow subprograms to be nested
- All variables that can be non-locally accessed reside in some activation record instance in the stack
- The process of locating a non-local reference:
  1. Find the correct activation record instance
  2. Determine the correct offset within that activation record instance

Locating a Non-local Reference

- Finding the offset is easy
- Finding the correct activation record instance
  - Static semantic rules guarantee that all non-local variables that can be referenced have been allocated in some activation record instance that is on the stack when the reference is made

Static Scoping

- A *static chain* is a chain of static links that connects certain activation record instances
- The *static link* in an activation record instance for subprogram A points to the activation record instance of an activation of A's static parent
- The static chain from an activation record instance connects it to all of its static ancestors
- *Static_depth* is an integer associated with a static scope whose value is the depth of nesting of that scope

Static Scoping (continued)

- The *chain_offset* or *nesting_depth* of a nonlocal reference is the difference between the static_depth of the reference and that of the scope when it is declared

- A reference to a variable can be represented by the pair:
  (chain_offset, local_offset),
  where local_offset is the offset in the activation record of the variable being referenced

Example JavaScript Program

```
function main(){
  var x;
  function bigsub() {
    var a, b, c;
    function sub1 {
      var a, d;
      a = b + c; <-----1
      ...
    } // end of sub1
```

```
function sub2(x) {
  var b, e;
  function sub3() {
    var c, e;
    ...
    sub1();
    ...
    e = b + a; <-----------2
  } // end of sub3 ...
  sub3();
  ...
  a = d + e; <------------3
} // end of sub2
...
sub2(7);
...
} // end of bigsub
...
bigsub();
...
} // end of main
```

`main` calls `bigsub`

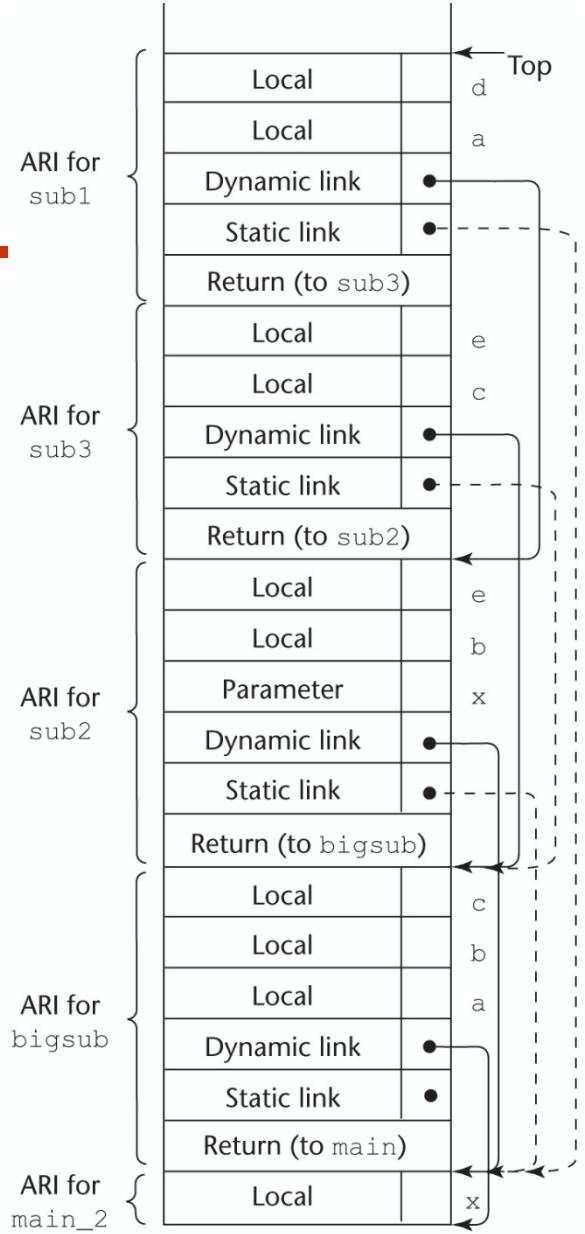
`bigsub` calls `sub2`

`sub2` calls `sub3`

`sub3` calls `sub1`

Stack Contents at Position 1

`main` **calls** `bigsub`
`bigsub` **calls** `sub2`
`sub2` **calls** `sub3`
`sub3` **calls** `sub1`



ARI = activation record instance

Static Chain Maintenance

- At the call,
  - The activation record instance must be built
  - The dynamic link is just the old stack top pointer
  - The static link must point to the most recent ari of the static parent
    - Two methods:
      1. Search the dynamic chain
      2. Treat subprogram calls and definitions like variable references and definitions

Evaluation of Static Chains

- Problems:
  1. A nonlocal areference is slow if the nesting depth is large
  2. Time-critical code is difficult:
     a. Costs of nonlocal references are difficult to determine
     b. Code changes can change the nesting depth, and therefore the cost

Blocks

- Blocks are user-specified local scopes for variables
- An example in C

```
{int temp;
 temp = list [upper];
 list [upper] = list [lower];
 list [lower] = temp
}
```

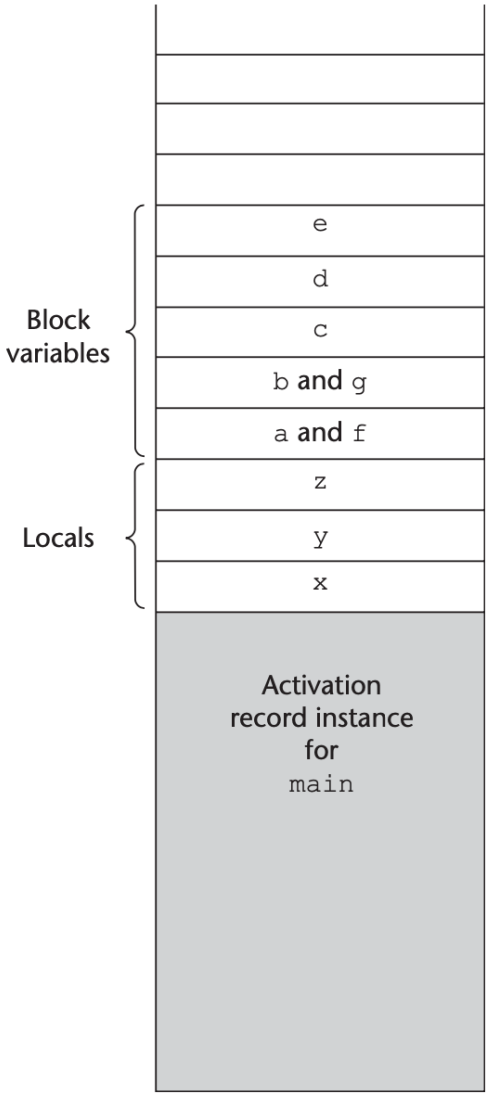
- The lifetime of `temp` in the above example begins when control enters the block
- An advantage of using a local variable like `temp` is that it cannot interfere with any other variable with the same name

Implementing Blocks

- Two Methods:
  1. Treat blocks as parameter-less subprograms that are always called from the same location
     - Every block has an activation record; an instance is created every time the block is executed
  2. Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record

Implementing Blocks Method 2

```
void main() {
  int x, y, z;
  while ( . .  ) {
    int a, b, c;
    .
    while ( . . . ) {
      int d, e;
      .
    }
  }
  while ( . . . ) {
    int f, g;
    . . .
  }
  . .
}
```



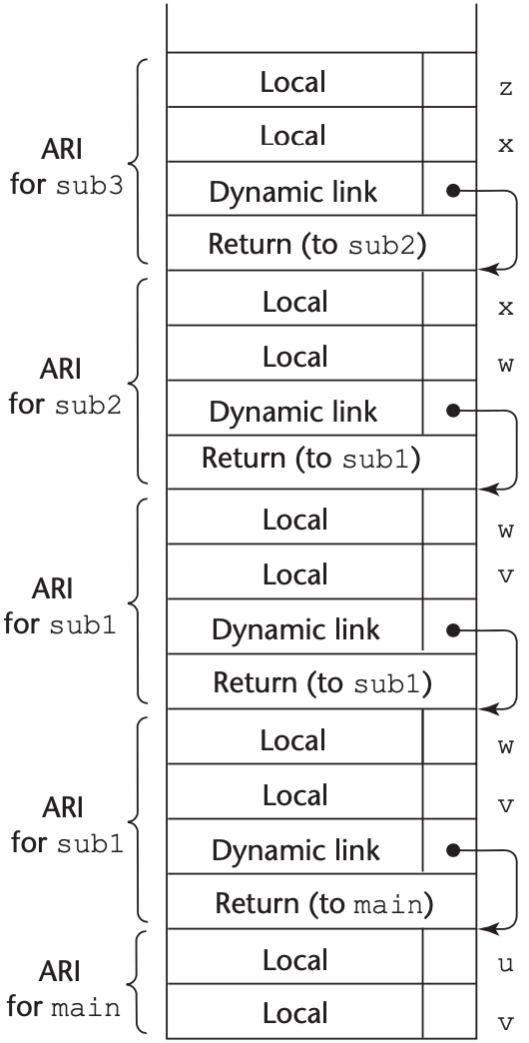
Implementing Dynamic Scoping

- *Deep Access*: non-local references are found by searching the activation record instances on the dynamic chain
  - Length of the chain cannot be statically determined
  - Every activation record instance must have variable names
- *Shallow Access*: put locals in a central place
  - One stack for each variable name
  - Central table with an entry for each variable name

Using Deep Access to Implement Dynamic Scoping

```
void sub3() {
  int x, z;
  x = u + v;
  …
}
void sub2() {
  int w, x;
  …
}
void sub1() {
  int v, w;
  …
}
void main() {
  int v, u;
  …
}
```

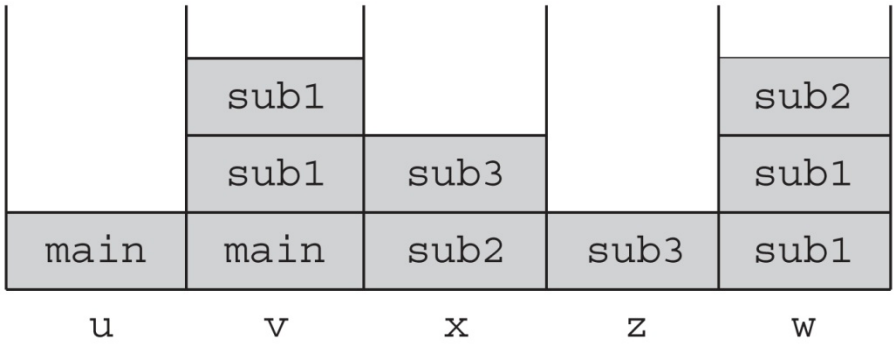
main->Sub1->sub1->sub2->sub3



ARI = activation record instance

Using Shallow Access to Implement Dynamic Scoping

```
void sub3() {
  int x, z;
  x = u + v;
  …
}
void sub2() {
  int w, x;
  …
}
void sub1() {
  int v, w;
  …
}
void main() {
  int v, u;
  …
}
```



(The names in the stack cells indicate the program units of the variable declaration.)

main->Sub1->sub1->sub2->sub3

Assignments

- Reading assignment: Chapter 10
- Written assignment: assignment four (4%), due on March 3