CSc 553
Principles of Compilation

02. Background: Symbol Tables

Saumya Debray

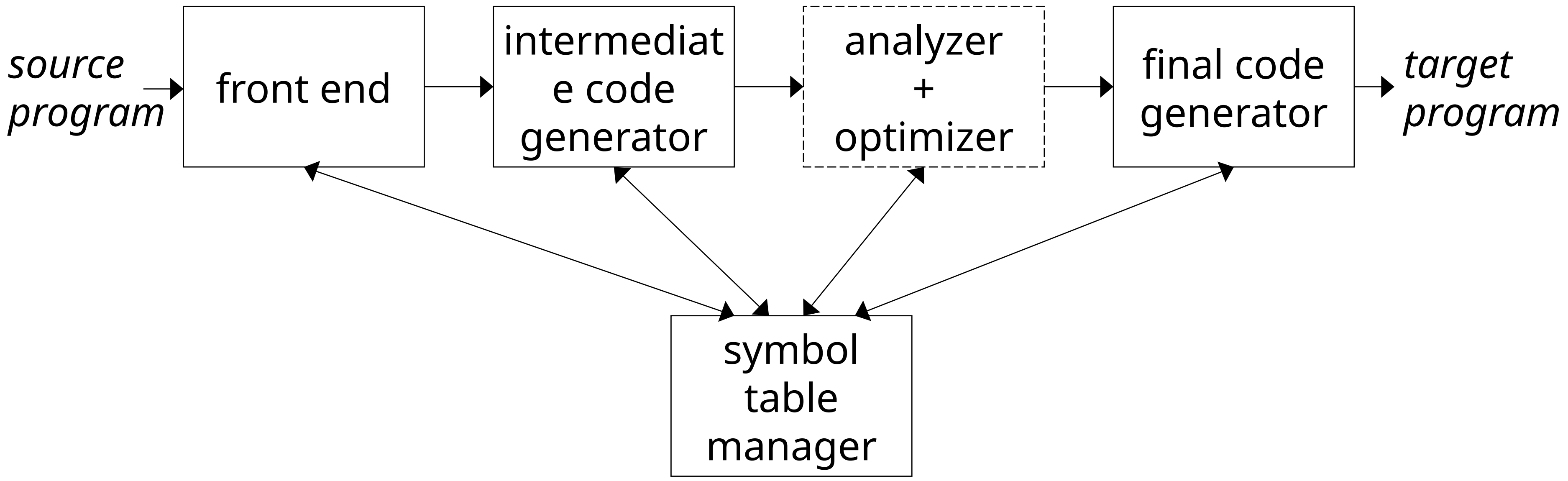
The University of Arizona

Tucson, AZ 85721

Symbol Tables

- A *symbol table* keeps track of information about *names* in the program
  - when a name is encountered during compilation, it is looked up in the symbol table
  - there is usually a different symbol table for each different scope (*e.g., global vs. local*)
- Information includes things like:
  - type
  - no. of elements (arrays); no. and types of arguments (functions);
  - …

Symbol Tables



Information needed about names

- Type checking:
  - Given the code `y = f(u,v,w)` :
    - is **`f`** a function?
    - no. of arguments OK?
    - argument types OK?

- Code generation:
  - Given the code `x = y + z` :
    - where in memory are **`x`**, **`y`**, **`z`**?
    - how much space do they occupy? (byte/word/…)

Symbol Tables

- *Purpose*: To hold information about identifiers that is computed at one point and used later.

  E.g.: type information:

  - computed during parsing;
  - used during type checking, code generation.

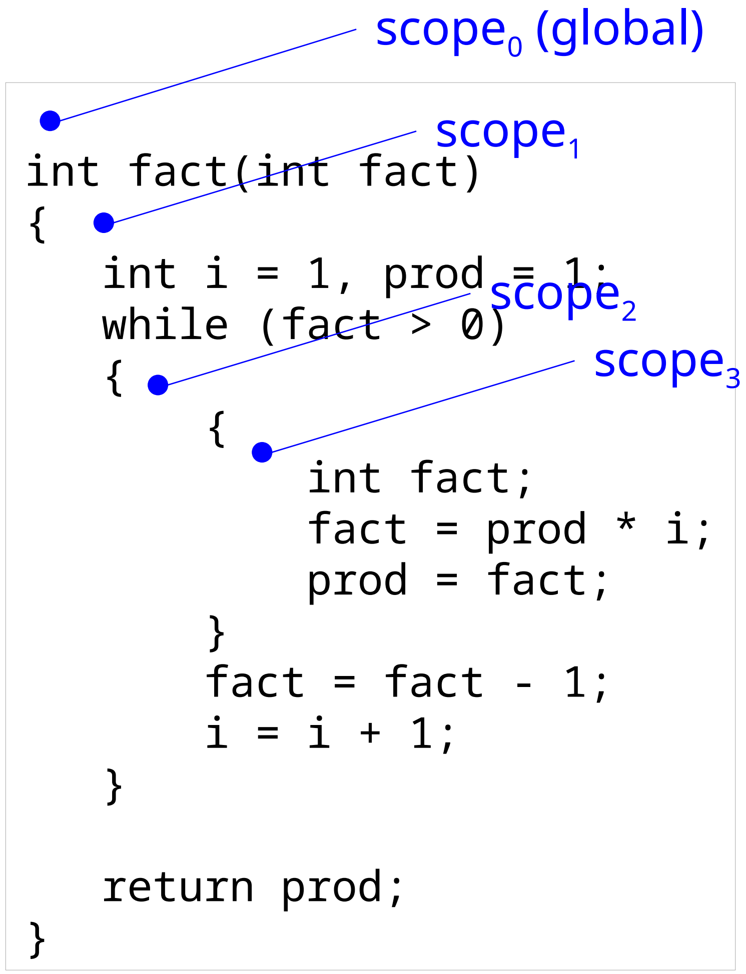
- *Operations*:

  - create, delete a symbol table;
  - insert, lookup an identifier

- *Typical implementations*: linked list, hash table.

Symbol Tables

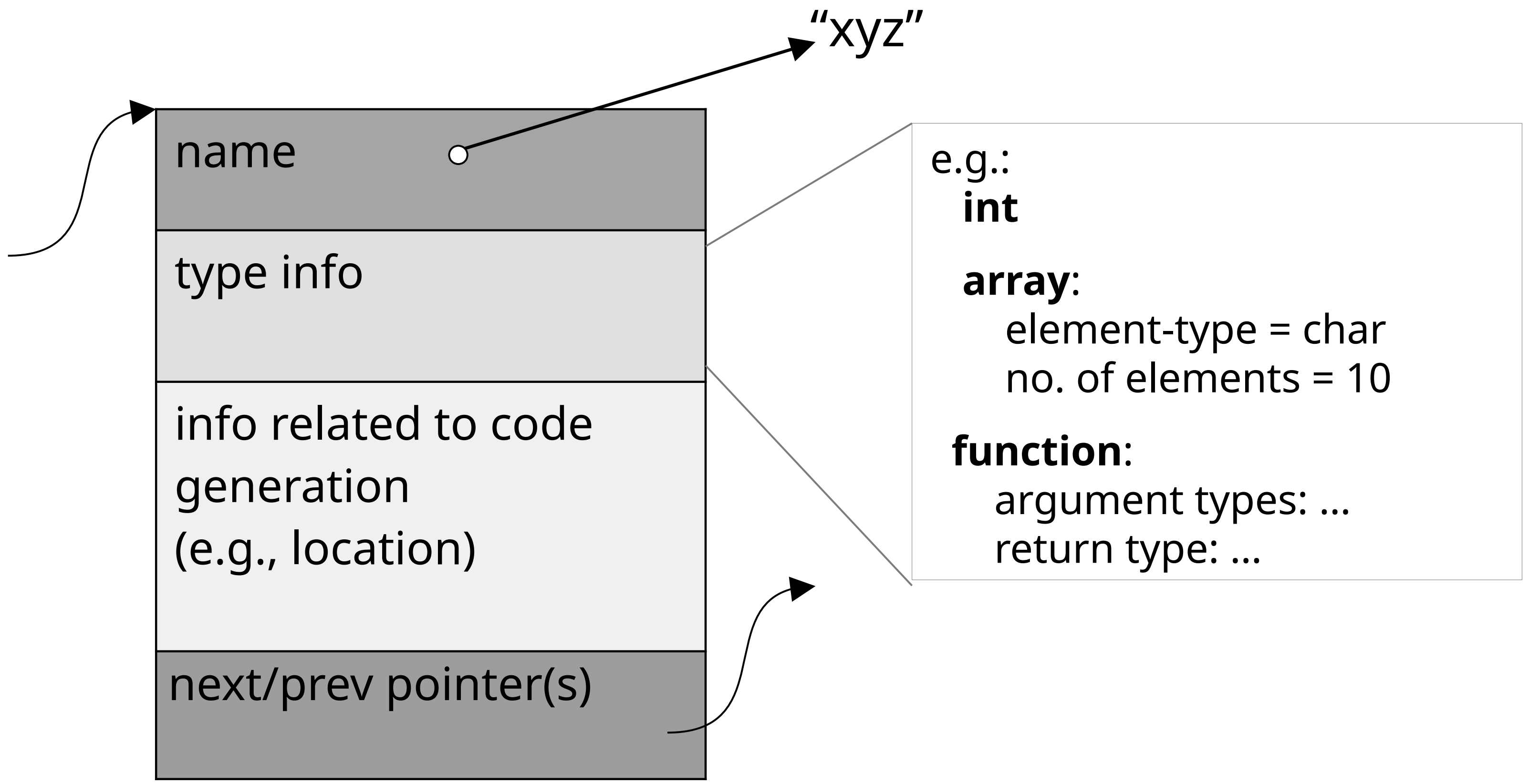
- Each distinct scope in the program has its own symbol table
  - for names local to that scope
- Symbol table entries are typically created when processing declarations
  - (also: some when generating code)



```
int fact(int fact)
{
   int i = 1, prod = 1;
   while (fact > 0)
   {
       {
           int fact;
           fact = prod * i;
           prod = fact;
       }
       fact = fact - 1;
       i = i + 1;
   }

   return prod;
}
```

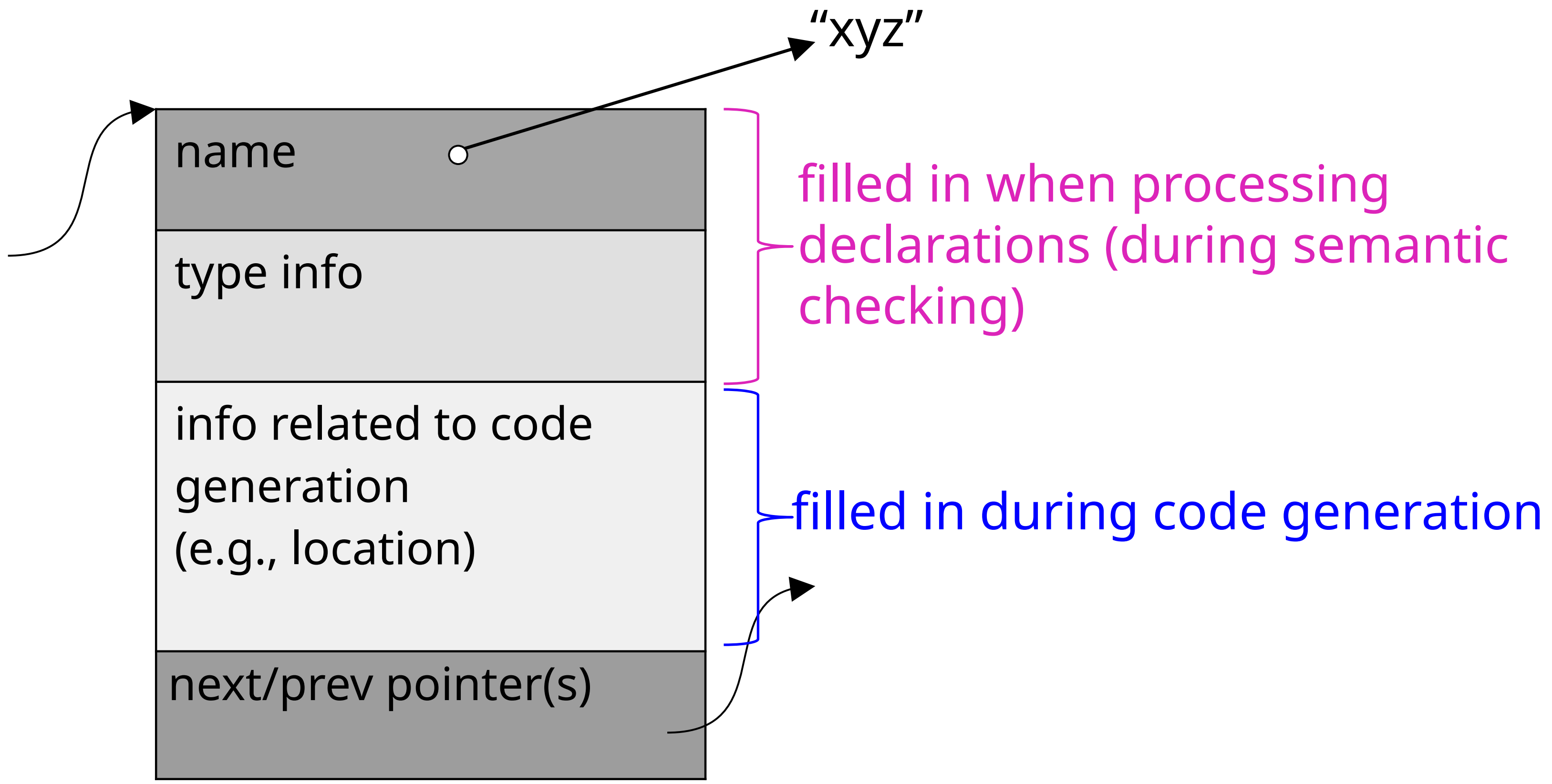
Symbol Tables

What does a symbol table entry look like?



Symbol Tables

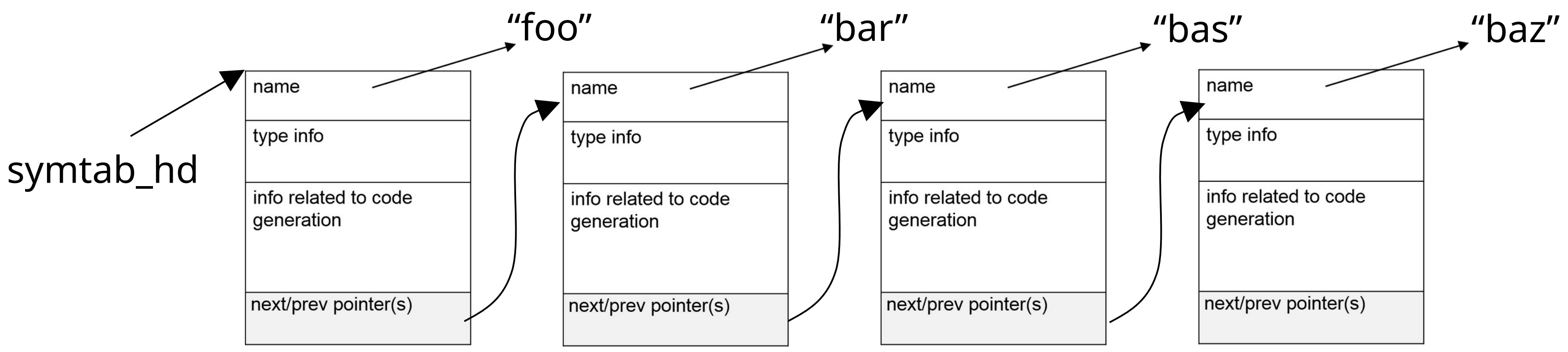
Information is filled in as it becomes available



"xyz"
name
type info
info related to code generation (e.g., location)
next/prev pointer(s)
filled in when processing declarations (during semantic checking)
filled in during code generation

Symbol Tables

What does a symbol table look like?



Note: Any data structure that allows insertion and lookup based on the symbol name will do.

E.g.: linked list, hash table, binary search tree, …

Managing Symbol Tables

- When looking up a name in a symbol table, we need to find the "appropriate" declaration.
  - *The scope rules of the language determine what is "appropriate."*
  - Often, we want the *most deeply nested* declaration for a name.
- *<u>Implementation</u>*: for each new scope: push a new symbol table on entry; pop on exit (*stack*).
  - implement symbol table stack as a linked list of symbol tables;

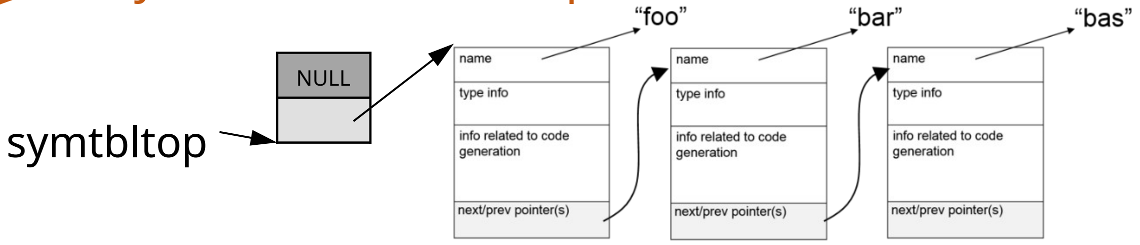
    newly declared identifiers go into the topmost symbol table.
  - lookup: search the symbol table stack from the top downwards.

Managing Symbol Tables

SCOPE 0

push symbol table for scope 0

```
int foo, bar, bas;
foo = …
bar = …
if (foo > bar)
{
  int foo, bar, bas;
  bar = …
  bas = …
  while (bar != bas)
  {
    int foo, bar, bas;
    …
  }
}
```



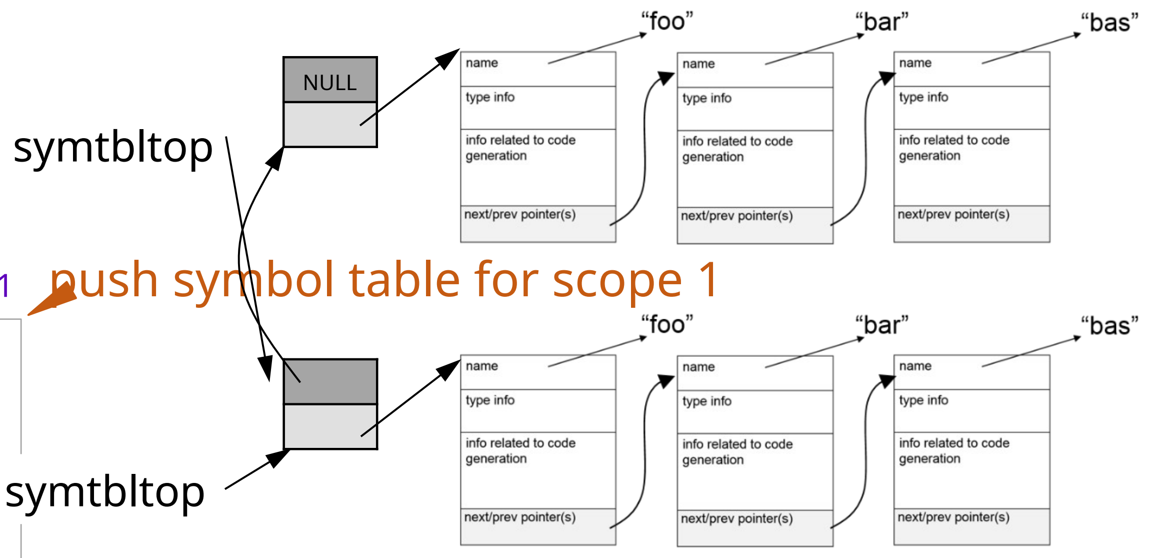
Managing Symbol Tables

SCOPE 0

```
int foo, bar, bas;

foo = …

bar = …

if (foo > bar)

{

  int foo, bar, bas;

  bar = …

  bas = …

  while (bar != bas)

  {

    int foo, bar, bas;

    …

  }

}
```

SCOPE 1

push symbol table for scope 1



Managing Symbol Tables

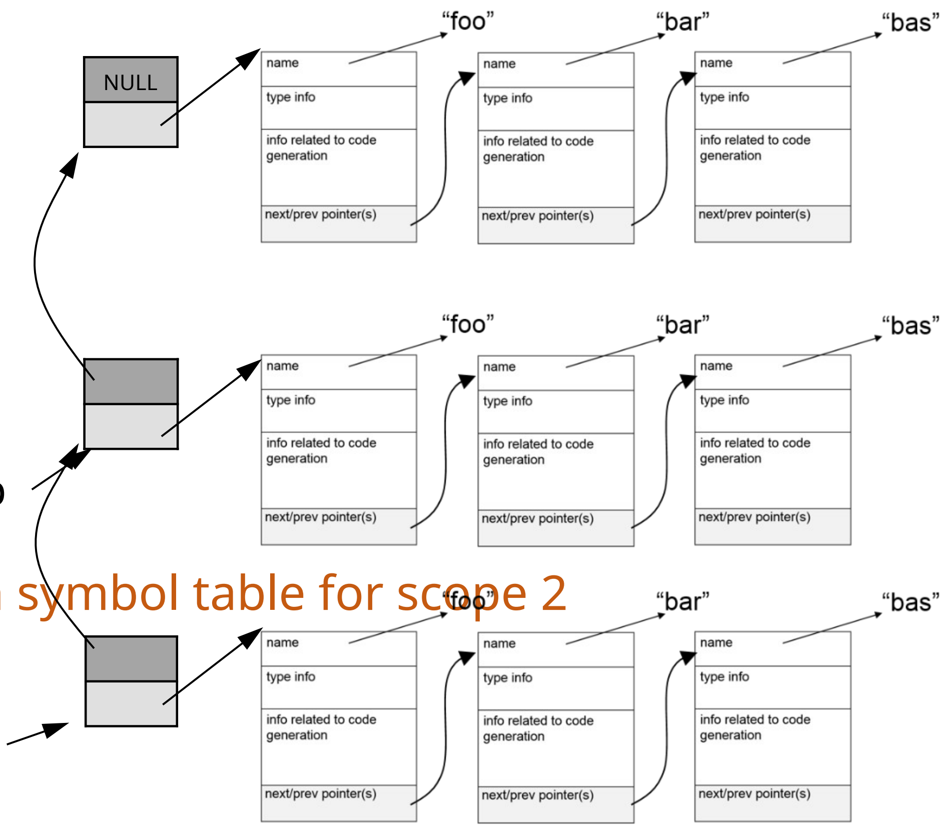
SCOPE 0

SCOPE 1

SCOPE 2

```
int foo, bar, bas;
foo = …
bar = …
if (foo > bar)
{
   int foo, bar, bas;
   bar = …
   bas = …
   while (bar != bas)
   {
      int foo, bar, bas;
      …
   }
}
```

push symbol table for scope 2



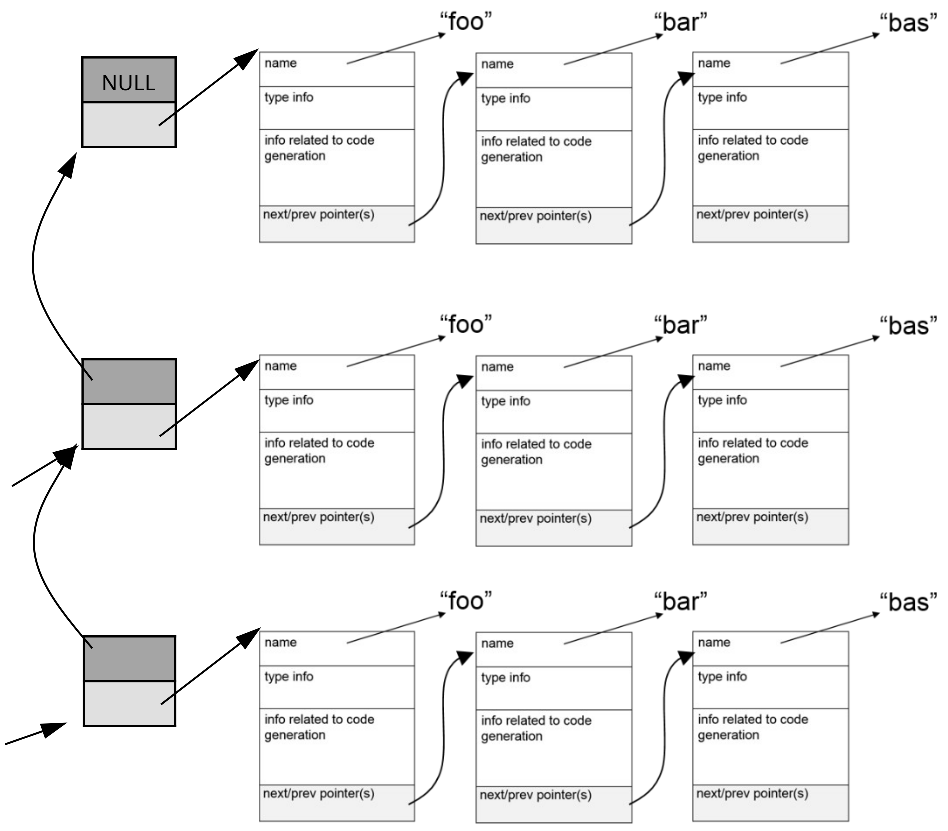
Managing Symbol Tables

SCOPE 0

```
int foo, bar, bas;
foo = …
bar = …
if (foo > bar)
{
   int foo, bar, bas;
   bar = …
   bas = …
   while (bar != bas)
   {
      int foo, bar, bas;
      …
   }
}
```

SCOPE 1

SCOPE 2



pop symbol table for scope 2

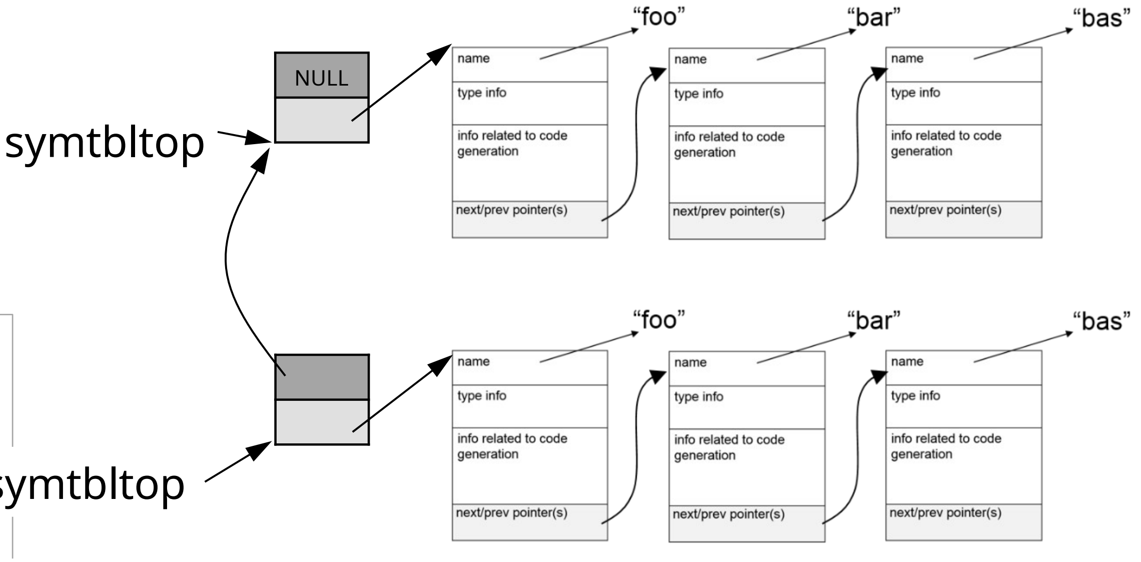
Managing Symbol Tables

SCOPE 0

```
int foo, bar, bas;
foo = …
bar = …
if (foo > bar)
{
  int foo, bar, bas;
  bar = …
  bas = …
  while (bar != bas)
  {
    int foo, bar, bas;
    …
  }
}
```

SCOPE 1

SCOPE 2

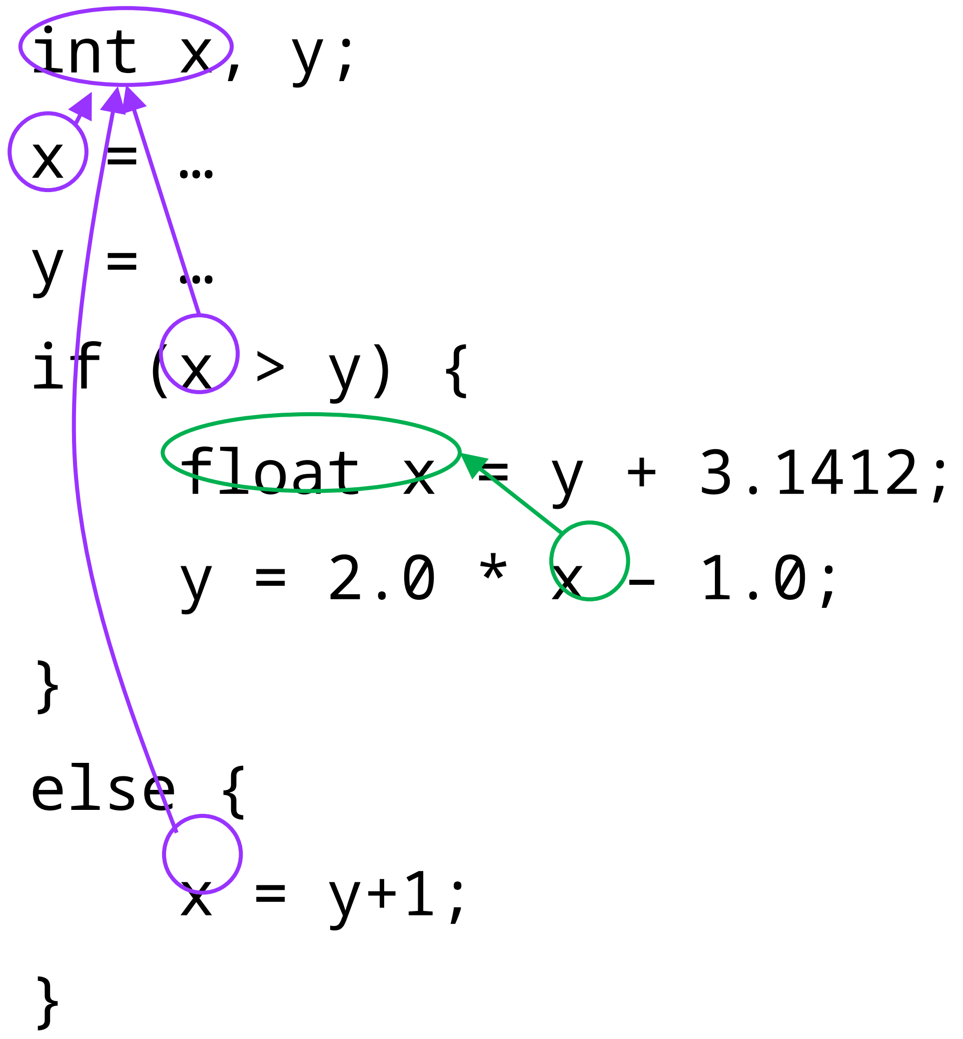


pop symbol table for scope 1

Symbol Table Lookups

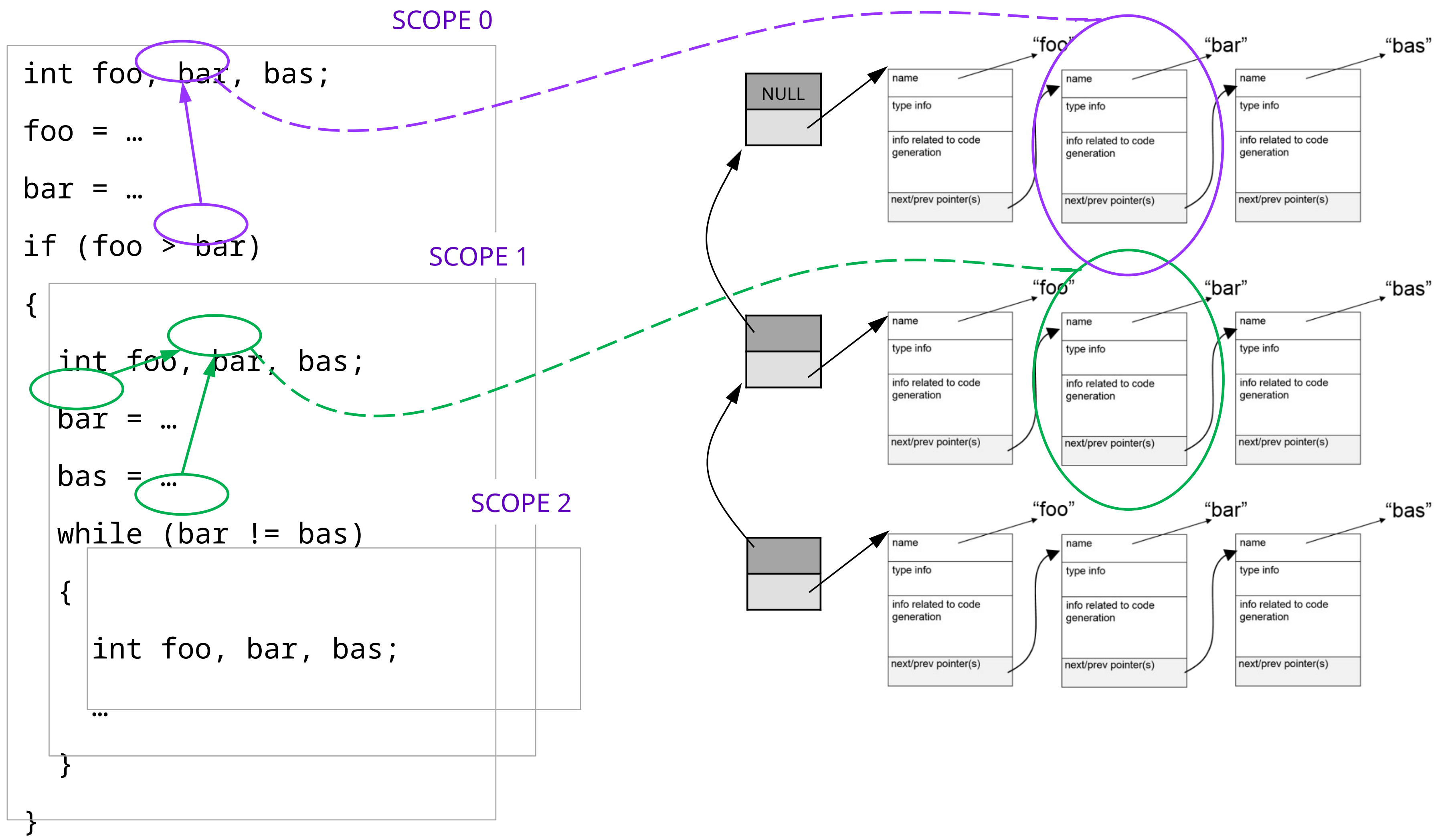
- In statically scoped languages (C, Java, …), each use of an identifier refers to the most deeply nested declaration enclosing that use
- At a use of an identifier, the symbol table is *looked up* to find its declaration:
    - start at the symbol table most deeply nested scope (i.e., at the top of the symbol table stack)
    - while not found: work down the symbol table stack, searching each symbol table in the stack

Symbol Table Lookups



```
int x, y;
x = …
y = …
if (x > y) {
    float x = y + 3.1412;
    y = 2.0 * x - 1.0;
}
else {
    x = y+1;
}
```

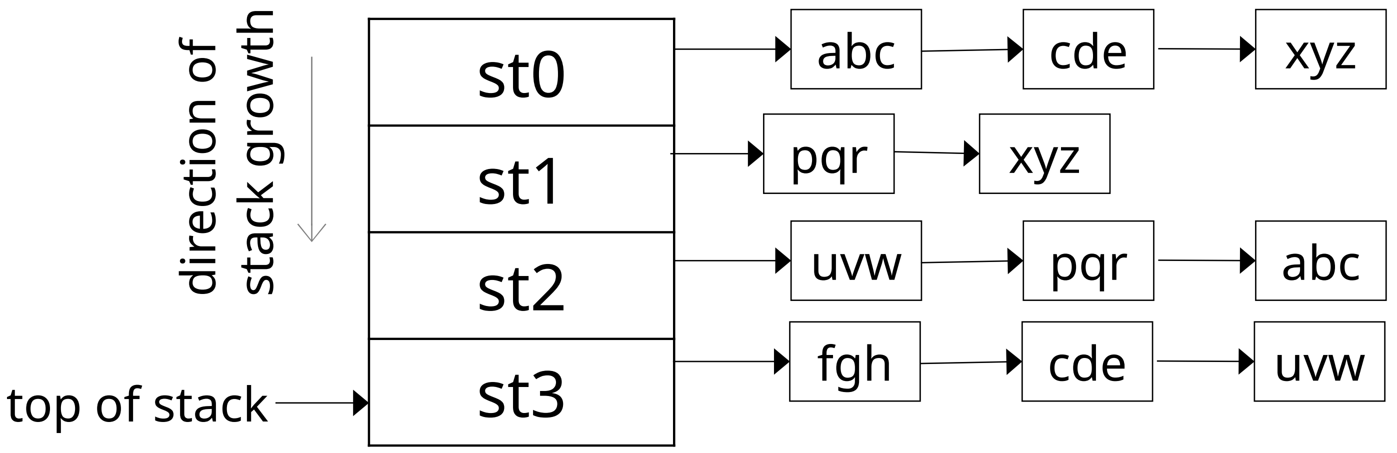
Symbol Table Lookups



```
int foo, bar, bas;
foo = …
bar = …
if (foo > bar)
{
  int foo, bar, bas;
  bar = …
  bas = …
  while (bar != bas)
  {
    int foo, bar, bas;
    …
  }
}
```

EXERCISE

Given the following stack of symbol tables:



in which symbol table will a lookup find:

- xyz
- pqr
- uvw