# Symbols for Functions

# why do functions need to be in symbol tables?

- Functions are symbols, so tracking them is important!
- Avoid name conflicts (different functions with the same name)
  - This interacts in a funny way with function overloading
- Keep track of the arguments and return information about a function
  - To make sure that functions are called properly
  - This *also* interacts in a funny way with function overloading
- Keep track of the names of parameters to a function
  - To make sure they are accessed correctly during code generation

# functions are symbols *and* scopes

- Functions also have their own scope!
- Local variables in functions are in a different scope than local variables in other functions or global variables
  - Variable names can be reused
- In global scope: need to track memory address of variables
- In local scope: need to track *stack offset* of variables
  - Remember, local variables are stored on the stack, accessed relative to stack/frame pointers

```
void foo(int x, int y)
{
  int a;
  int b;
  ...
}
```

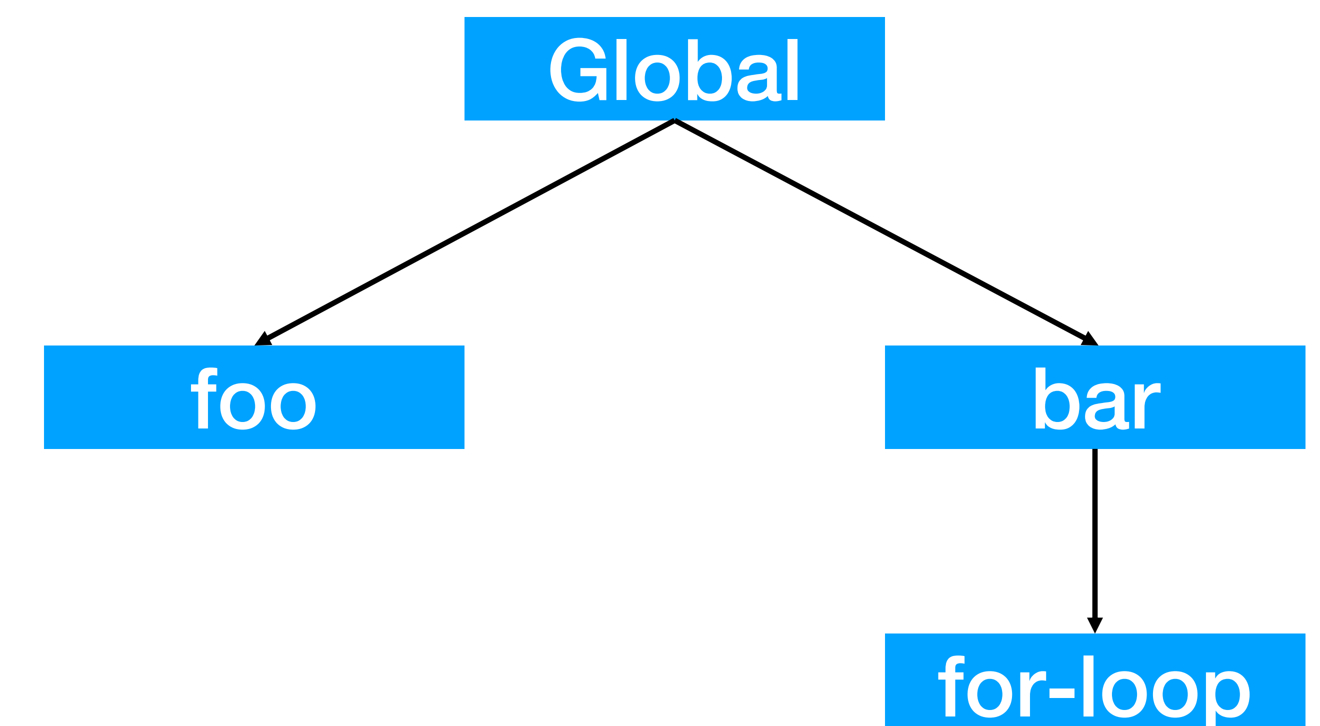
| Name | Type | Location |
|---|---|---|
| x | int | reg: a1 |
| y | int | fp: +8 |
| a | int | fp: -4 |
| b | int | fp: -8 |

# symbol tables are trees

- Scopes are nested within one another
  - Global scope
    - Function scope nested within global scope
      - Local blocks nested within functions (not in uC)
- Variables can be accessed if they are in scope: if they exist in the current scope or any scope this scope is nested inside
- Store pointers from parent scopes to children scopes (e.g., global scope has a child scope for each function), and from children scope to parent scope

```
int foo( ... ) {
  ...
}
int bar( ... ) {
  ...
  for ( ... ) {
    ...

}
```

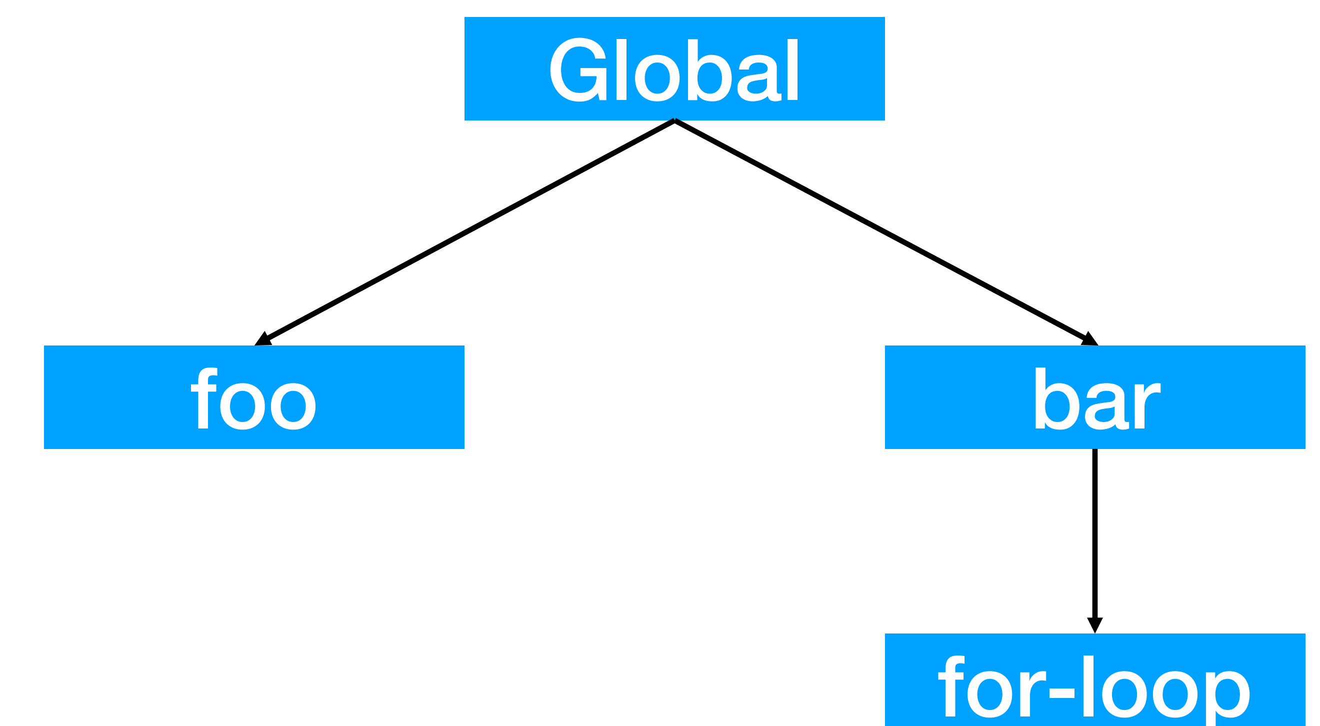


# looking for symbols

- When you access a variable in code, you want to check the *current* scope for the variable, as well as all parent scopes
  - Bind the variable to the entry in the "closest" scope
- When generating code for that variable, generate address based on entry
  - Global scope: absolute address
  - Local scope: address offset from frame pointer

```
int foo( ... ) {
  ...
}
int bar( ... ) {
  ...
  for ( ... ) {
    ...

}
```



# dealing with overloading

- Some language support *function overloading*
  - Multiple functions with the same name, but different numbers/types of arguments
- How do we deal with repeated names for functions?
  - Use *name mangling*: encode additional information into each function to incorporate information about argument types
  - Creates a different name for each distinct function

```
void foo(int x, float y)

becomes

void foo3_int_float(int x, float y) //why put “3” at the end of foo?
```

next: code generation for functions