SI 413, Unit 6: Names and Scopes

Daniel S. Roche (roche@usna.edu)

Fall 2018

1 Overview

Readings for this section: PLP, Chapter 3 intro

A *name* in a computer program is anything that the programmer is allowed to decide what it's called: variables, functions, classes, (depending on the language) even things like types and namespaces. In all these instances, what the programmer is asking to do is to associate (*bind*) some name to some kind of meaning or *value*.

There are two big questions we have to ask as compiler/interpreter writers (and as programmers) when we come across a name:

1) What/which *binding* does this name refer to? Observe that this is not the same question as what *value* does that binding have. The value probably can't be known until the program is running, but the binding is often known at compile-time, just by looking at the program.
2) Where is the value stored? When is that storage *allocated* and de-allocated? Again, we usually can't know what the value is at compile-time, but we might be able to say where it can be found.

These two questions have to do with the *scope* and *allocation* of a name, and there will be some important choices to make here. This continues a common theme for the course: **trade-offs**. In designing programming languages and the compilers/interpreters to implement them, we are frequently faced with a compromise between:

- Program *expressiveness* (how easy it is to write a program in the language)
- Compiler efficiency
- Program efficiency (Not the same thing as compiler efficiency! A very slow compiler might produce really fast machine code.)

We will see that the designers of different programming languages have made different choices with respect to these aspects, depending on the goals that that language is supposed to fulfill. For example, older programming languages such as Fortran were designed to be easy to compile and run on what we would today consider to be very limited hardware. So they sacrificed expressiveness for efficiency, especially compiler efficiency. Modern scripting languages like Ruby, on the other hand, are designed with the programmer in mind. Ruby compromises somewhat on speed in order to allow the most expressiveness and ease of programming.

Here, we will look at three options for scoping rules. These are defined by the language, and so this is a decision for the language designer:

- **Single Global Scope**: There's just one big symbol table, and every time the same name appears in a program, it means the same thing.
- **Dynamic Scope**: Whenever we enter a function, we push its local variable bindings onto a stack for that name. When we leave some scope, those bindings are popped off. This is a little bit unusual, but has its purposes, and it's usually easier to implement than…
- **Lexical Scope**: The binding of every name is determined based on the actual nested structure of the code, and can be known at compile-time. This is what you get in most modern languages like C, Java,

and Scheme. But it's also the trickiest to implement, although it seems oh-so simple.

2 Allocation

Readings for this section: PLP, Section 3.2 through 3.2.3.

As possible implementations of these scoping rules, we will also look at three choices for allocation. This usually wouldn't be defined by the language, but would be up to the compiler or interpreter writer to decide:

- **Static Allocation**: The memory locations for all variables are pre-determined at compile time. This means that, in the compiled code, those variables can just look into that spot in memory directly
  - there is no run-time symbol table! Unfortunately, the gains in speed are more that offset by losses in program flexibility. For example, with this allocation method it will be impossible to have any kind of recursion! Can you see why that is?
- **Stack Allocation**: A fixed amount of contiguous memory is given to each program when it starts, called the "stack". Generally, this space is used *contiguously* (without gaps), according to function calls. Whenever a function is called, we push the local variables for that function onto the stack. When the function returns, its variables get popped off the stack. This is quite efficient because there aren't any real decisions about *where* to allocate the memory, and the operating system doesn't need to get involved. As we will see, stack allocation is also good enough to cover many kinds of simple programs where the *lifetime* of every local variable matches the extent of the function call.
- **Heap Allocation**: The "heap" is the other part of memory that a running program can grab pieces from. The difference is, heap-allocated memory doesn't need to be contiguous like the stack does. What this means in practice is that heap memory can be allocated and de-allocated at any time during execution, and doesn't need to be tied to function calls. The downside is that the operating system must be used to find a new hole in the heap memory space for every allocation, and this makes heap allocation slower. Besides being slower, heap allocation is also more work because we have to worry about de-allocating (freeing) memory as well as allocating it. But the upside is that this allocation method is the most powerful, and can be used to implement any of the scoping rules above.

In a nutshell, static allocation is the fastest but the most limited in terms of the language features it supports. Heap allocation can support any kind of scoping rules or language features, but it will also be the slowest at run-time, and it's harder to implement, especially if memory leaks bother you (as they should).

3 Scoping Intro

Readings for this section: PLP, Section 3.3 intro and 3.3.3

A *scope* is simply the part of a program where some binding is "active", meaning the thing that that name refers to is in memory, and (unless it's been hidden by another name), the name is available to be referenced. This definition is a bit confusing, but the concept is actually pretty simple. For example, the scope of a global variable is the entire program. The scope of a local variable to a function, or an argument to a function, is just that function itself.

Different languages have different rules about where scopes get created. In some languages such as BASIC, there is only one global scope, which makes things pretty simple! Usually, though, major control structures such as function bodies and class definitions always make new scopes. In C++, other control function bodies also make new scopes (like the body of a while or for loop). In fact, C, C++ and Java all follow the rule that every opening curly brace { enters a new scope. (Our rules in SPL for the labs will be similar.)

In Scheme, things like define statements, lambdas, and let expressions are mostly what make scopes. Since it's Scheme, the scope goes until the matching closing paren).

Scoping rules are different in different programming languages, and they are mostly about determining the meaning of *non-local references*. That is, references to names which are *not* local variables, since local variables are always pretty straightforward. For example, check out this Perl program:

```
my $x=1;

sub foo() {
    $x = 5;
}

sub bar() {
    local $x = 2;
    foo();
    print $x,"\n";
}

bar();
```

The question is, when foo() gets called from within the function bar(), what does the name x refer to? It's not a local variable, so we definitely have a nonlocal reference.

In most languages you have used, *lexical scoping* is the norm, which means we can figure out what x should refer to just by looking at the source code, working our way out in the scopes until we find a declaration for that name. In this case, the x in foo() would always refer to the global x (the one that's initialized to 1), no matter where foo() is called from.

There is another option, however, called *dynamic scoping*, in which the binding for each variable is determined by the most recent declaration in the function call stack, not necessarily the closest in terms of nested scopes. This is actually what Perl uses when you declare a variable local, so in the example above, the x in foo() refers to the one declared from within bar() (initialized to 2), since foo() was called from within bar().

We'll see more carefully how these two scoping schemes work below. But you should already be noticing that the difference between a variable *declaration* and variable *reference* is pretty important. A *declaration* actually creates a new binding in the current scope, and the place where a variable is declared determines its scope and usually its lifetime (how long it stays in memory) as well. A variable *reference* is just looking up an existing binding; it doesn't create anything new. Note that we'll use the term *reference* whether we mean getting or setting the value of the binding.

3.1 Declaration Order

Some interesting stuff happens when we get into what order names are declared within a certain scope. Some languages like plain C and Ada enforce a rule that *all names must be declared at the beginning of the scope*. This sort of keeps things simple, so that every scope must start with the declarations for its local variables, before actually doing anything with them. Many languages (C++, Java, Scheme, . . .) allow you to declare a variable anywhere within a scope, but usually you can only use that variable *after* its declaration.

This is really important in languages like C++ and Java because they support what's called *compile-time name resolution*. What this means is that every name in your program is "resolved" (meaning things like its type are known) at compile-time. This is definitely not the case in Scheme, which is why a Scheme program that, say, multiplies a number times a string will only be a *run-time error*, while in C++ or Java such nonsense would be detected at compile-time. More to the point, a Scheme program such as

```
(define (f x) (* x y))
```

is perfectly valid, even if no y is declared anywhere. It's only when we *call* the function f that we'll get a run-time error saying y is undefined.

Well, compile-time name resolution gets a little bit tricky with mutually recursive structures. Consider the case where two functions both call each other. Obviously one of these functions is going to have to be declared first. But then the other one won't have been declared - so how can we call it already? The solution is what's called *forward declarations*, where we declare something (like a function) without actually defining it. In C++, this is accomplished by writing function prototypes, which you should be familiar with. That allows the C++ compiler to read a program in a single pass, do compile-time name resolution, and still allow awesome features like mutual recursion. Many other languages, such as Java, just solve this issue by reading in all the relevant source code first, then going back to do the type checking and name resolution. This is a bit slower to compile, and requires more memory, but makes the programmer's job easier.

4 Dynamic Scope

Readings for this section: PLP, Section 3.3.6

As we mentioned above, dynamic scoping is based on the following general rule:

The current value of a name is determined by the *most recent binding* on the function call stack.

As each new scope is entered (usually by function calls), we *push* any variable declarations in that scope onto a stack. When that scope exits, its bindings are popped off the stack. And when a variable reference is made, we just look at the top of the stack for that variable.

Dynamic scope has historically been used in "immature" languages, including many scripting languages, for two reasons. First, it makes it easier to pass information from on part of the code to another part, in a function call. The reason is that every function call essentially inherits the local scope of wherever the function was called from. Especially when some function takes a lot of arguments, but you don't really want to have to specify all of them on every call, this can be rather convenient.

The second reason to use dynamic scoping is that it's pretty easy to implement, compared with the alternative. You'll discover this when you get to implement lexical scoping in a later lab!

The most common mechanism to implement dynamic scoping is through a data structure called a *Central Reference Table*, or CRT. This data structure consists of two parts:

- The "names in scope" is a stack of sets of names. Make sure you understand what that means: there's a stack, each thing in the stack is a set (i.e., a collection), and each thing in each set is a name (i.e., an identifier). This keeps track of all the names that have been declared in each scope, so that when that scope exits, we know which bindings should go out of scope.
- For each variable name that is used anywhere in the program, there is a stack of bindings for that name. Values are pushed onto this stack when a new binding is created (i.e., declarations), and popped off when the scope of that binding exits. All variable references are to whatever binding is currently on the top of the stack for that name.

Let's see how this works with a small example. Here's a program in our SPL language from labs, which we'll assume for the moment is dynamically scoped:

```
{
  new x := 0;
  new i := -1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    ifelse i > 0
    { ret := p@0; }
    {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0;});
}
```

Here is what the CRT looks like when we reach the write statement at the bottom. Notice all that's in there is really just the global names and their bindings.

(In class, and in these notes, I'm using the shorthand notation like y −> 0 to mean "a function that takes an argument y and always returns 0". When the function is too long to write like that, I'll just write ... and assume you know which function we're referring to.)

```
{x,i,g,f}     0    -1     z->i     p->...
_________    ___   ___   ______   _______

  NAMES       x     i      g         f
 IN SCOPE
```

Now to evaluate the write statement, the interpreter must first evaluate the function call to f. When this happens, we immediately get a few things: First, a new set is pushed onto the "names in scope" stack, since we're entering a new scope. The names in that set will be p (the argument name for f) and ret (to store the return value). Actual bindings for those two names are also created, so the CRT becomes:

```
 {p,ret}
{x,i,g,f}     0    -1     z->i     p->...     y->0     (unset)
_________    ___   ___   ______   _______   ______   _______

  NAMES       x     i      g         f          p        ret
 IN SCOPE
```

Now the interpreter begins evaluating the function call. Two interesting things happen in the very first statement

```
new i := x;
```

First, the right-hand side is evaluated, which means looking up the current binding (top of the stack) for x; that's 0. Then a new binding is created for i, which means adding i to the top set in "names in scope", and pushing the value onto the stack for i, so we get:

```
{p,ret,i}           0
{x,i,g,f}     0    -1     z->i     p->...     y->0     (unset)
_________    ___   ___   ______   _______   ______   _______

  NAMES       x     i      g         f          p        ret
 IN SCOPE
```

Now i evaluates to 0, so the next condition is false, and the else branch is taken. The first line in this branch is

```
x := x + 1
```

which is *not a declaration*, but a reassignment, so it only changes the top of the stack of bindings for x. Similarly with the next line for i. Then we have a recursive function call f(g), which causes a new set in the "names in scope" stack, and two new bindings, like before. So at the beginning of this recursive call, the CRT looks like:

```
 {p,ret}
{p,ret,i}           3                         z->i     (unset)
{x,i,g,f}     1    -1     z->i     p->...     y->0     (unset)
_________    ___   ___   ______   _______   ______   _______

  NAMES       x     i      g         f          p        ret
 IN SCOPE
```

Make sure you notice and understand all those changes! Now this function call starts by making a new binding for i in the current scope, equal to the current value of x, which is 1. That makes the if condition evaluate to true, so we take the "then" branch of the if, which means another function call. This time the function that gets called is p(0), and in this context the value of the name p is the function that takes an argument z and returns i (see that on the top of the stack for p?). So when *this* function call begins, the CRT gets another set for "names in scope" and two more bindings, so it looks like:

```
 {z, ret}
{p, ret, i}          1                                (unset)
{p, ret, i}          3                      z->i      (unset)
{x, i, g, f}    1   -1     z->i    p->...   y->0      (unset)    0
-----------    ---  ---   ------  -------  ------    --------   ---
   NAMES        x    i      g        f        p         ret      z
 IN SCOPE
```

Okay, now we have a series of return statements to process, from the three functions that have been called inside one another. First, the current function (the function currently known as p) returns i, and the current value for i (top of the stack) is 1. Here's the situation just before the function finishes:

```
 {z, ret}
{p, ret, i}          1                                   1
{p, ret, i}          3                      z->i      (unset)
{x, i, g, f}    1   -1     z->i    p->...   y->0      (unset)    0
-----------    ---  ---   ------  -------  ------    --------   ---
   NAMES        x    i      g        f        p         ret      z
 IN SCOPE
```

A few things happen when this function actually finishes. For one, the return value is the top value of ret in the stack, which is 1. In addition, since the function is over, we have to clean up after it. This means popping off the top set in "names in scope", and *for each name in that set*, popping off the most recent binding. So you'll notice the stacks for z and ret each go down by one. This is what the situation is just before the second call to f returns:

```
{p, ret, i}          1
{p, ret, i}          3                      z->i         1
{x, i, g, f}    1   -1     z->i    p->...   y->0      (unset)
-----------    ---  ---   ------  -------  ------    --------   ---
   NAMES        x    i      g        f        p         ret      z
 IN SCOPE
```

Then that call to f returns its computed value (1), to the first call to f, which in turn returns its computed value to the global scope. Each time a function finishes, the bindings are popped off as indicated in the top set of "names in scope". Here's what the CRT looks like just before the original call to f returns:

```
{p, ret, i}          3
{x, i, g, f}    1   -1     z->i    p->...   y->0         1
-----------    ---  ---   ------  -------  ------    --------   ---
   NAMES        x    i      g        f        p         ret      z
 IN SCOPE
```

So you can see that ultimately all this program does is print out the number 1. This was a relatively complicated example to demonstrate the CRT for you, but you should be able to follow and reconstruct examples like this on your own. This is how our interpreter that handles dynamic scoping will work as well.

5 Lexical Scope

Readings for this section: SICP Section 3.2 and PLP Section 3.6

We have already seen a few examples of the differences between lexical and dynamic scoping. In both cases, local variables and *local references* to variables declared in the same scope will work the same. The differences only come with *non-local* references within function calls. So it should be clear that function calls are pretty important in implementing either set of scoping rules!

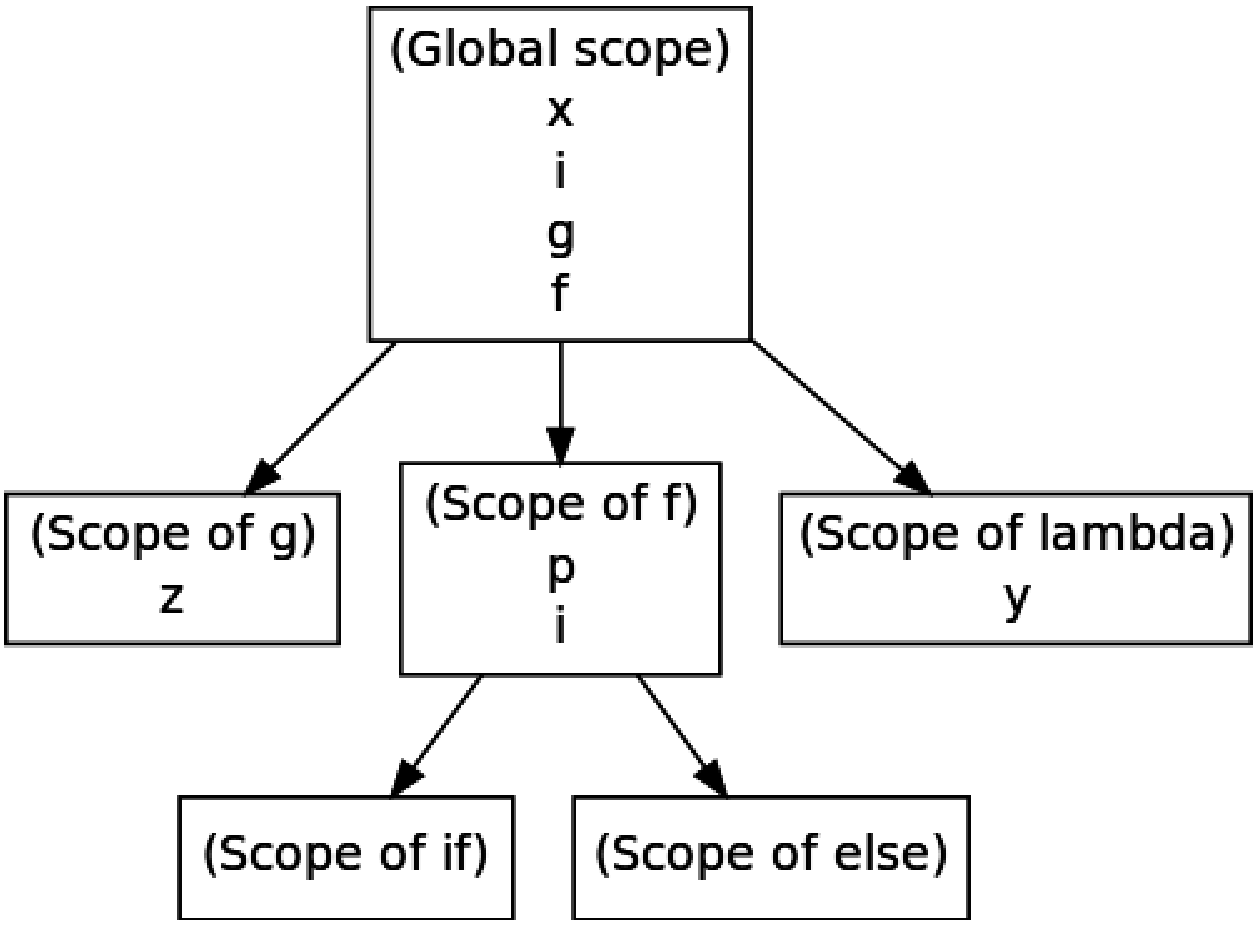
5.1 Scope trees

Lexical scoping is somewhat easier to understand than dynamic scoping, although it is a bit more difficult to implement. It's also what you're used to from C++, Java, and Scheme - they all use lexical scoping. The important thing here is the structure of the code itself (the "lexical structure"), and in particular the nesting of scopes in the code. One tool to visualize this nesting is a *scope tree*.

A scope tree is a very simple structure that just has a box for each scope in the code, containing the local variables (including arguments to functions) defined in that scope. Every box *except for the global scope* also has a link to the outer scope that contains it. So for example, here is the scope tree for the SPL program in the CRT example above:

```
{
  new x := 0;
  new i := -1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    ifelse i > 0
    { ret := p@0; }
    {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0;});
}
```

Here is the scope tree:



The names of the scopes (in parentheses) are just for us to remember and don't actually matter. You will see that every pair of curly braces defines a scope, and every scope in the program is in this tree. Notice that:

- The scope of f is only in the tree *once*, even though *f* will get called twice in an actual run. The scope tree has nothing to do with the run-time of the program, just with its lexical structure.
- There is a scope for the body of the lambda on the last line, even though this function is never called! (Same reason as above).
- The if and else blocks inside f each get their own node in the scope tree. These are empty because no new variables are declared inside those blocks.

The reason a lexical scope tree like this is useful is that it allows us to say what every variable reference in the program refers to - without having to rung the actual program! (This is the advantage of static, or lexical, scoping.) The way to figure out what any variable reference refers to is to start in the scope where the reference is made, then work your way up to the root until the *first* time you find that variable name in scope - that's the one it refers to!

More formally stated, the binding for any name in the program comes from searching the path in the scope tree from the root node (global scope) down to the current scope where the reference occurs, and taking the bottom-most node where that name was declared.

So for example, the else block within f makes two references to x. We start in that block - no x there. Then move up to its parent, the scope of x. No x there either. So it's up to the global scope, which does have x. Therefore the x inside that else block is referring to the global x defined on the first line.

Contrast this with the reference to i in that same block. Again, there is no mention of i in the else block's scope, so we move up to the parent, the scope of f. But now i appears here! So the i in the else block is referring to the one defined within f, not to the global i.

5.2 Frames and Closures

Scope trees are great for reasoning about programs, but how can we actually implement those rules in a running program? Take this example, in SPL:

```
new f := lambda x {
  ret := lambda y { ret := x + y; };
};

new g := f@5;
write g@6;
```

What does this program do? Well, the tricky part is the *non-local* reference to x inside the inner lambda. From a scope tree, we would see that this x is referring to the argument to f. Now here's the issue: by the time that lambda gets called (on the last line, after it's been renamed as g), the function call to f has already returned! So the x that f was called on (namely, 5) is out of scope, and yet g must still refer to that 5 so it can return 11!

This tricky business is what makes implementing lexical scope somewhat of a difficult prospect, at least when we are allowed to write programs like this. The actual issue is when *functions are first-class*, which you might recall from our Scheme days means that they can be given as arguments to functions and can also be returned from functions. As it turns out, when functions are first-class, we have to allocate memory for them on the heap instead of the stack. And the specific way this is accomplished is by using a data structure called *Frames*. A frame contains two things:

- A link to its parent frame (unless its the *global frame*, which is very special and has no parent).
- A simple *symbol table* mapping the names of local variables to their values.

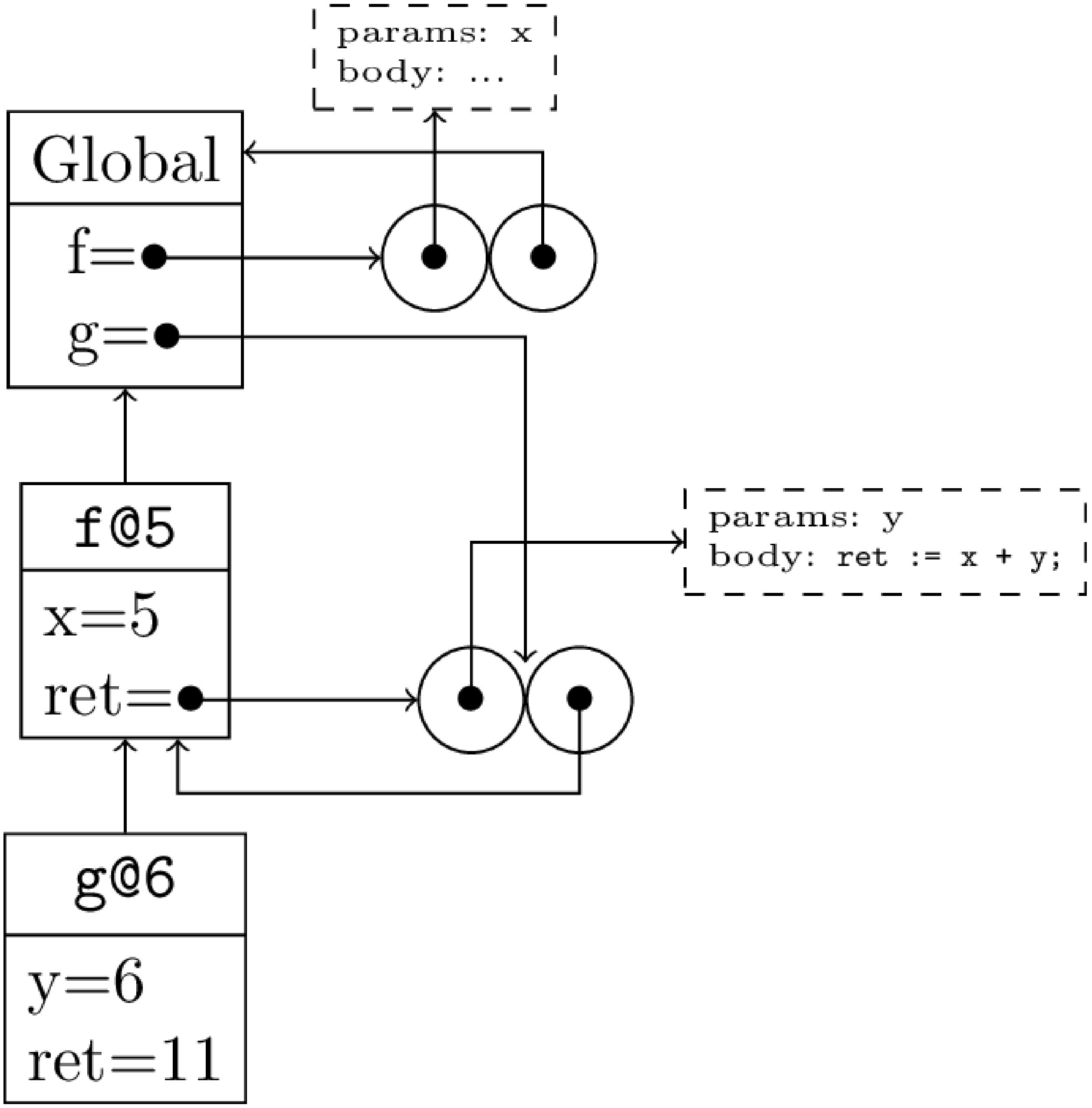
A frame is exactly the same as a node in the Scope Tree, except that the names are actually given values. In a running program, a new frame is created *whenever a new scope is entered*. In particular, we will get a new frame every time there is a function call.

Now if you look back in the example above, you will notice that, if there are many calls to f, each one of them will create a new frame and return a new function. But the function that returns must know which call to f created it, so it can look up the correct value of x! For this reason, function values within frames are represented using a special structure called a *closure*. We actually talked about closures before too, in the context of Scheme. A closure contains two things as well:

- The actual function definition, or *function body*.
- The frame where the function was created, which is called the *referencing environment*.

Each function in a frame will now just be a pointer to a closure, which in turn points to another frame. The result can be very simple, or we can get some really interesting and gnarly diagrams. We might affectionately call these "boxes and balls" diagrams, since each frame is drawn in a box, and each closure is drawn as a pair of circles.

Here is the resulting diagram from the program above:



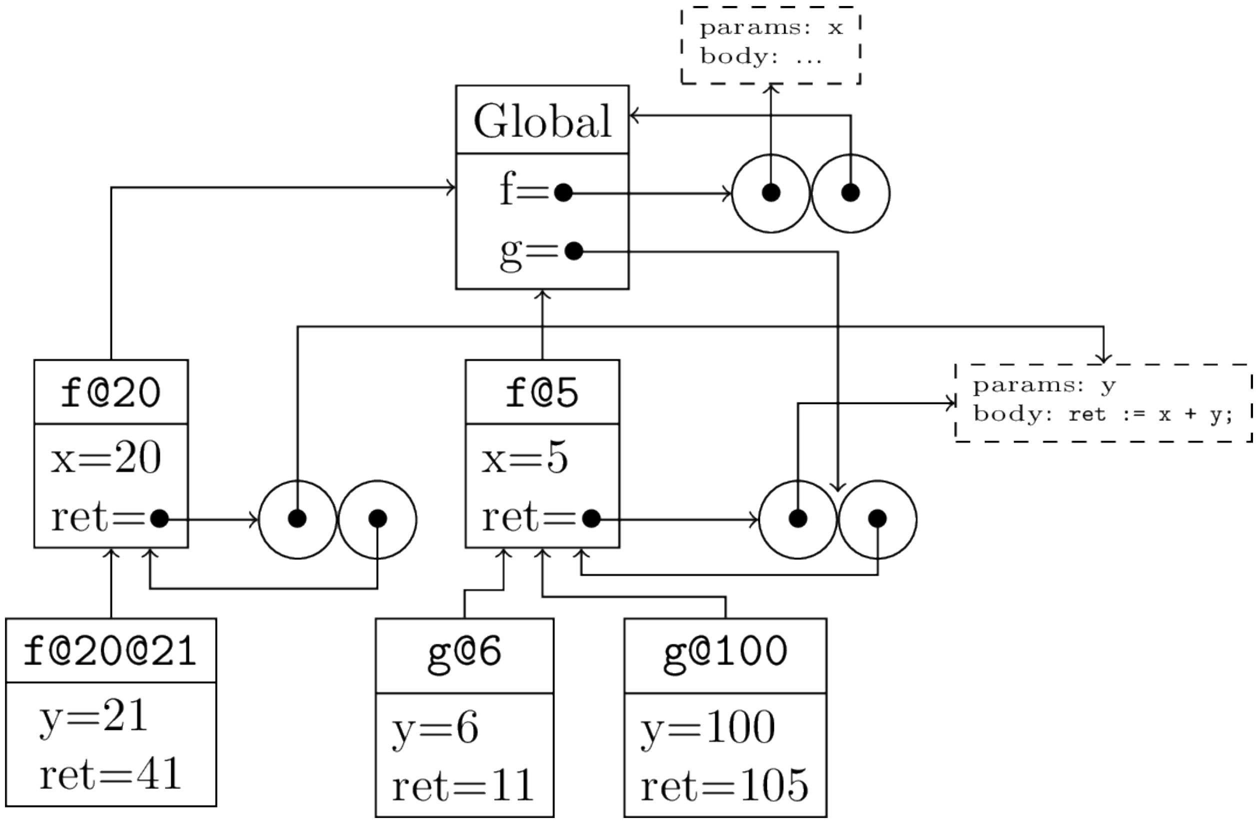
Now consider if two more lines of code were executed in this program:

```
write f@20@21;
write g@100;
```

This first line starts with f@20, creating a new frame, and within the function execution of f@20 a new closure is also created because of the lambda. This closure will point to the new frame we just made as the referencing environment, but the *procedure* this closure points to will be the same one was created before when calling f@5. The code of that procedure hasn't changed; the difference here is the referencing environment. Then the returned closure is called with argument 21, which results in one additional new frame, and 41 is printed out.

After this, the old closure assigned to name g in the global frame is called with argument 100, which again creates a new frame, with a *different parent frame* according to that closure's referencing environment. After this line, 105 is printed out. (Notice, not 120 or anything else!)

The resulting diagram after these two additional lines of code is:



Here are the most important things to remember about how this works:

1) Each function *definition* creates a new closure. The referencing environment of the closure is the frame in which the function is defined.
2) Each function *call* creates a new frame. The parent frame will be the referencing environment of that closure.

We will see a few examples of such diagrams in class. There are many more in the very-well written section of the SICP book mentioned above. I recommend you look at it (it's free and online and uses Scheme) for more examples.

The difference between dynamic and lexical scoping can be summarized by the following: With dynamic scoping, each binding is determined by where the function is *called*, but in lexical scoping, each binding is determined by where each function is *defined*. Make sure you understand what this means now! It's important to recognize that frames are the most general technique for implementing scoping rules - we could even use them for dynamic scoping! But they are only *needed* when we have lexical scoping and first-class functions, like in Scheme… or SPL!