Lab 8: Scope

Baheem Ferrell

Scope

So what does this name refer to again?

The following is based on Ray Toal's notes on scope in programming languages.

Definition

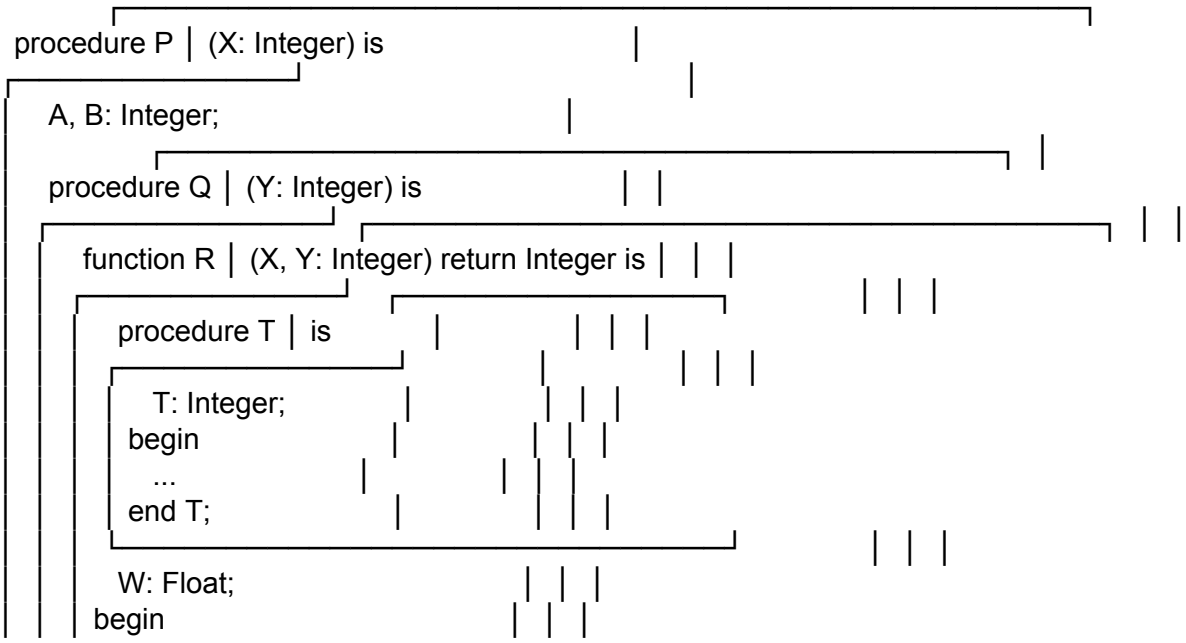
The scope of a binding is the region in a program in which the binding is active.

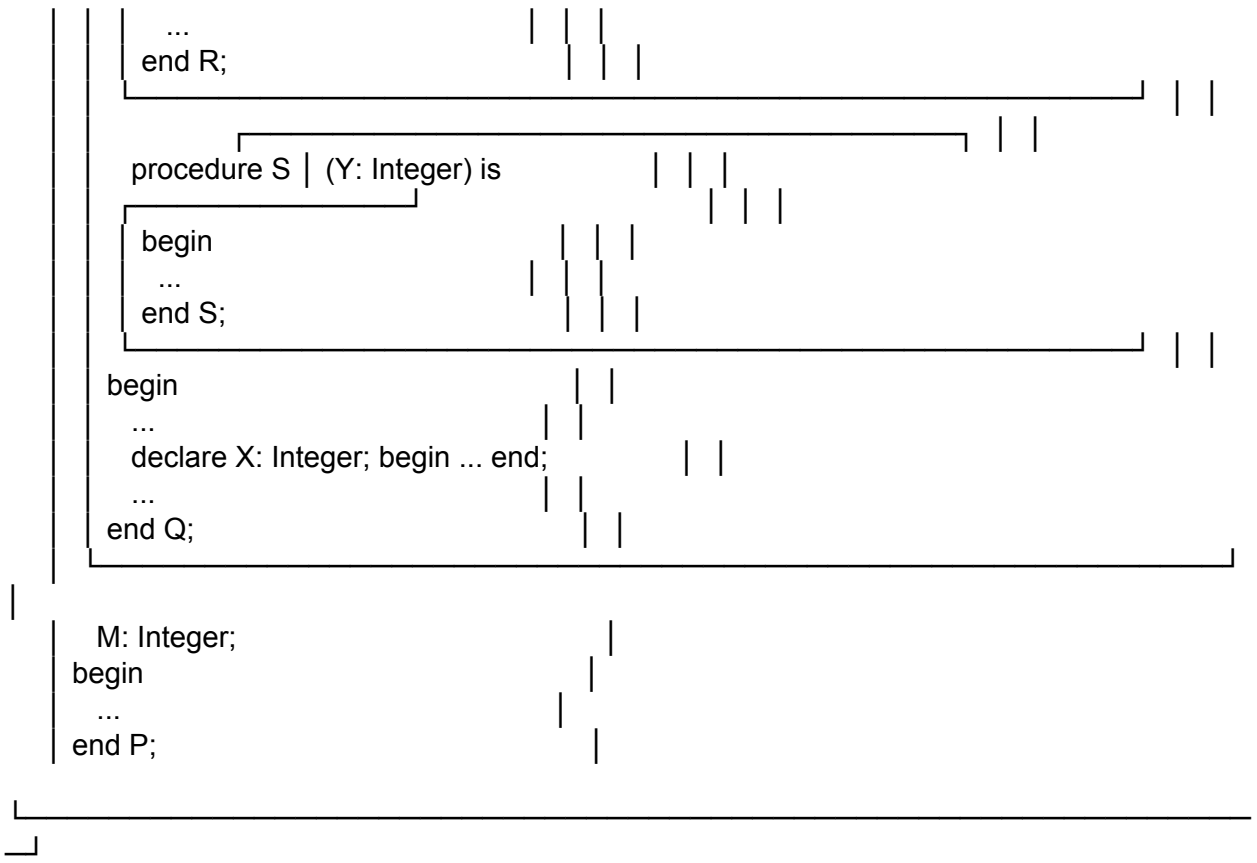
- Most languages employ static scoping (also called lexical scoping), meaning all scopes can be determined at compile time.
- Some languages (APL, Snobol, early LISP) employ dynamic scoping, meaning scopes depend on runtime execution.
- Perl lets you have both!

Regardless of whether scoping is static or dynamic, the definition implies that when you leave a scope, bindings are deactivated.

Static Scope

Static scoping is determined by the structure of the source code:





The general rule is that bindings are determined by finding the “innermost” declaration, searching outward through the *enclosing* scopes if needed.

Interestingly, not everything is obvious! There are reasonable questions regarding:

- Where *exactly* the scope of a particular binding begins and ends
- Whether bindings that have been *hidden* in inner scopes are still accessible
- What actually counts as a *declaration* versus just a use? Can declarations be implicit?
- The effect of redeclarations within a scope

Scope Range

Does the scope consist of a whole block or just from the declaration onward?

Consider this pseudocode:

```
var x = 3
var y = 5
function f() {
```

```
  print y
  var x = x + 1
  var y = 7
  print x
}
f()
```

Exercise: Translate this pseudocode into your favorite programming language, *e.g.* Python, Java, JavaScript, Ruby, *etc.*. Determine if the output corresponds to the output you expected for each of your implementations.

Here is the the example translated into C code:

```
#include <stdio.h>

int x = 3;
int y = 5;

int main() {
  printf("%d\n", y);
  int x = x + 1;
  int y = 7;
  printf("%d\n", x);
}
main();
```

Here is what you get when you compile and run this code:

```
$ gcc foo.c
foo.c:12:1: warning: type specifier missing, defaults to 'int' [-Wimplicit-int]
main();
^
1 warning generated.
$ ./a.out
5
2
$
```

Is this what you expected? Hint: insert some additional local variables into the 'main' function and observe what happens.

In python:

```
x = 3
y = 5

def main():
   print(y)
   x = x + 1
   y = 7
   print(x)
```

main()

Here is the result.

```
python p1.py
Traceback (most recent call last):
  File "p1.py", line 12, in <module>
        main()
  File "p1.py", line 6, in main
        print(y)
UnboundLocalError: local variable 'y' referenced before assignment
```

Inside main() function the outer variables (global variables) x, y cannot be referenced. They can only be referenced inside main() function by using global keywords. Otherwise they are treated like local variables and hence y is referenced before it is assigned a value.

Scope Holes

In static scoping, “inner” declarations hide, or shadow outer declarations of the same identifier, causing a hole in the scope of the outer identifier’s binding. But when we’re in the scope of the inner binding, can we still *see* the outer binding? Sometimes, we can:

In C++:

```
#include <iostream>

int x = 1;
namespace N {
  int x = 2;
  class C {
    int x = 3;
    void f() {
      int x = 4;
      std::cout << ::x;      // global
      std::cout << N::x;     // namespace
      std::cout << this->x;  // class/object
      std::cout << x;        // function local
      std::cout << '\n';
    }
  };
}
```

Here is code that works fine in C with the expected shadowing but will fail in Java:

```
void f()
{
```

```
  int x = 1;

  if (true) {
    int x = 2;
  }
}
```

Java does not allow more than one declaration of a local variable within a method.

Exercise: Find out the rationale for this rule.
If more than one declaration of a local variable is allowed within a method, the programmer must follow every declaration and reference to make sure that each variable is referenced as it's supposed to be. For example, in the above C code, if there are lines that reference the variable x, depending on where it is referenced (whether before or after int x = 2;) they carry different meanings. Therefore, the programmer must make sure that each variable is referenced within its valid scope without interfering with other variables with the same identifier, which is difficult in developing a large-scope project and even more impossible for someone else to comprehend.

JavaScript has a somewhat interesting take on the interplay of scopes and nested blocks, with very different semantics between declarations introduced with var, const, or let.

Exercise: Research the difference between var and let in JavaScript, as it pertains to scope. You should run across something called the temporal dead zone (TDZ). Explain the TDZ in your own words.
Variables that are declared using let keyword is limited to the scope of a block statement or an expression where it is used, while those declared using var keyword are usually global or local to the entire source file or the entire function. The other difference is that variables that are declared using let keyword must be initialized only when they are parsed.
Because variables that are declared using let keyword aren't initialized until they are parsed, they can't be accessed until their declaration is parsed. Therefore unlike var, variables that are declared using let keyword are in a temporal dead zone from the start of the corresponding block, where they cannot be referenced until they are declared explicitly.

Indicating Declarations

Generally, languages indicate declarations with a keyword, such as var, val, let, const, my, our, or def. Or sometimes we see a type name, as in:

```
int x = 5;              // C, Java
```

or

```
X: Integer := 3;        -- Ada
```

When we use these **explicit declarations** in a local scope, things are pretty easy to figure out, as in this JavaScript example:

```
let x = 1, y = 2;
function f() {
  let x = 3;          // local declaration
  console.log(x);     // uses local x
  console.log(y);     // uses global y, it's automatically visible
}
f();                  // writes 3 and 2
```

JS

Easy to understand, for sure, but be super careful, leaving off the declaration keyword in JavaScript updates, *and introduces if necessary*, a global variable:

```
let x = 1;
function f() {
  x = 2;              // DID YOU FORGET let?
}                     // Updated global var! Accident or intentional?
f()
console.log(x);
```

JS

Forgetting the declarating keywords has happened in the real world, with very sad consequences.

But in Python, there are no keywords and no types. Declarations are **implicit**. How does this work?

```
x = -5
def f():
    print(abs(x))
f()                   # prints 5
```

Sure enough we can see the globals abs and x inside f. Now consider:

```
x = -5
def f():
    x = 3
f()
print(x) # prints -5
```

Aha, so if we assign to a variable inside a function, that variable is implicitly declared to be a local variable. Now how about:

```
x = -5
def f():
    print(x)
    x = 3
f()
print(x)
```

This gives UnboundLocalError: local variable 'x' referenced before assignment. The scope of the local x is the whole function body!

Exercise: Why do the Python rules “make sense”?
In Python local variables have the scope of the entire block or function. In the above Python code, x=3 declares a local variable x implicitly which has the scope of function f(). Therefore accessing it before its declaration/initialization gives an error because it is not initialized. Compared to C this rule eliminates the confusion between variables with the same identifier and furthermore the possibility of accidentally changing global variables where local variables change is intended.
Exercise: Do some research for this one: How then do you write to a global variable from inside a function? Is there ever a good reason to do so? Hint: See first exercise above.
In order to write to a global variable inside a function, the variable must be explicitly declared using a global keyword inside the function and local variables cannot have the same identifier as the global variable. By doing so we can eliminate the possibility of accidentally changing global variables where local variables change is intended.

Now Ruby’s declarations are also implicit. Let’s see what we can figure out here:

```
x = 3
def f(); puts x; end      # Error!
f()
```

WHOA! NameError: undefined local variable or method `x' for main:Object. We can’t see that outer x inside of f! If we want to play this game, we must use global variables, which in Ruby start with a $:

```
$x = 3
def f(); x = 5; y = 4; p [$x, x, y] end    # writes [3, 5, 4]
f()
```

So assigning to a local declares it, and variables starting with a $ are global.

CoffeeScript also has no explicit declarations. But it has a controversial design. If you assign to a variable inside a function, and that variable does not already have a binding in an outer scope, a new local variable is created. But if an outer binding does exist, you *get an assignment to the outer variable*.

```
x = 3
f = () -> x = 5
f()               # Plasters the global x
console.log(x)      # Writes 5

g = () -> y = 8
g()               # Defines a local y which then goes away
console.log(y)      # Throws ReferenceError: y is not defined
```

Exercise: Research this issue discussion on Github. Summarize the "problem" here. What major problems is Dmitry Soshnikov pointing out? Why does Jeremy Ashkenas think these are not real problems? Or are they?

Dmitry Soshnikov points out that shadowing global variables is not a bad choice. The major problem of not shadowing global variables is that someone might reuse third-party functions in their code without knowing what local variables they are using and how they are conflicting with the global variables in their own code. Hence he argues that functions must be like black-box allowing them to use whatever variables they want without worrying about how they might conflict with other variables in outer scopes. Jeremy Ashkenas suggests that this isn't a real problem because high-quality code should use a minimum number of global variables and shadowing the global variables prevents the remaining scope from using the original values. One can always rename the variables.

On one hand Jeremy is right to state that there should be a minimum number of global variables and all variables must have specific meanings in their names. On the other hand, Dmitry's argument is correct to allow different developers to use any variable names they want without worrying about variable collision.

In my opinion, correctness should come before convenience and it should be preferred to reference variables with their full scope so that there is no clash between variables in different scopes. Renaming every global variable or using unnecessarily long names for temporary local variables cannot be the solution to this argument and a good programmer should always reference variables with their full scopes so as not to confuse him/herself as well as to make their code more reusable in any contexts.

Redeclarations Within a Scope

If an identifier is redeclared within a scope, a language designer can choose to:

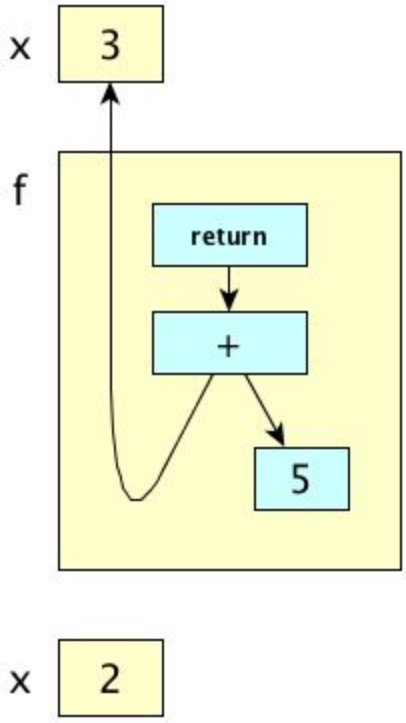
- Reject it (as a compile-time or run-time error).
- Make a fresh new binding and retain the old ones.
- Treat the redeclaration like an assignment.

Perl and ML actually retain the old bindings:

```
# Perl
my $x = 3;              # declare $x and bind it
sub f {return $x + 5;}    # uses the existing binding of $x
my $x = 2;              # declare a NEW entity and thus a new binding
print f(), "\n";
```

```
(* SML *)
val x = 3;              (* declare x and bind it *)
fun f() = x + 5;         (* uses existing binding of x *)
val x = 2;              (* declare a NEW entity and thus a new binding *)
f();
```

Both of these programs print 8, because the bindings are new:



But in some languages, the second declaration is really just an assignment, rather than the creation of a new binding. In such a language, the code implementing the above example would print 7, not 8.

Exercise: Find a language in which the direct translation of the code about would cause 7 to be printed. Write the program.

In Python, the following code prints out 7.

```
x = 3

def f():
    return x + 5

x = 2
print(f())
```

Dynamic Scope

With dynamic scoping, the current binding for a name is the one **most recently encountered during execution**. The following script prints 2, not 1:

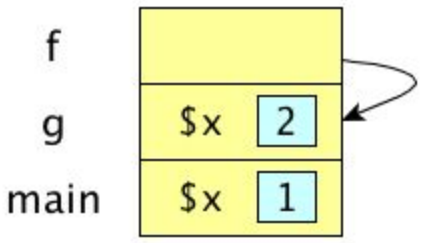
```
# Perl
our $x = 1;

sub f{
    print "$x\n";
}

sub g {
    local $x = 2;     # local makes dynamic scope (use my for static)
    f();
}

g();
print "$x\n";
```

At each call, a new frame is created for each subroutine invocation and pushed on the call stack. A frame holds each of the locally declared variables for that invocation. In dynamic scoping, when looking up bindings for identifiers not in the current frame, we search the call stack. In our Perl example, while executing f and looking for $x, we'll find it in g's frame:



Dynamic scopes tend to require **lots of runtime work**: type checking, binding resolution, argument checking, etc.

They are also prone to redeclaration problems.

Exercise: Explain this.
When dynamic scoping is used, variables at the same line of code might have different values depending on the call stacks at the time of execution. This is different from local variables in functions having different values depending on the parameter values passed to it. In the latter case, local variables have different values because it is intended by the programmer and he/she clearly knows about this. In the former case, it is determined by the compiler/parser and one might have a difficult time following all possible tracks of execution and design/debug the logic for all these cases. For example in the above Perl script, depending on whether f() is called directly from the global scope or indirectly from inside g(), x can have different values. If this was intended it is not a problem. But as the call stack gets deeper and the program gets larger, it becomes harder and harder to predict/follow all possible call stacks and variable scopes which makes the code prone to logical errors.

Exercise: Explain this.

The principal argument in favor of dynamic scoping is the way they allow subroutines to be customized by using an “environment variable” style that allows selective overriding of a default (more or less). Michael Scott gives this example:

```
# Perl
our $print_base = 10;

sub print_integer {
   ...
   # some code using $print_base
   ...
}

# So generally we want to use 10, but say at one point
# we were in a hex mood.  We wrap the call in a block like this:
{
   local $print_base = 16;
   print_integer($n);
}
# At the end of the block the old value is essentially restored
```

Exercise: Rip this argument apart from the context of modern programming practice. Hints: Consider terms like “global variables” and “default parameters.”

In the above Perl script, default base is set at 10 using a global variable. And by using a dynamic scoping this global variable is temporarily overridden to set the base at 16.

Hence it is argued that dynamic overriding of default variables has the advantage of using a selectively customizable execution environment.
However, the same effect can be achieved using default parameters in function definitions without introducing the complications that are caused by dynamic scoping. When these default parameters are not given explicitly during function call, the default parameters are used implicitly inside the function. By explicitly specifying the default parameters in function calls, different execution environments can be achieved. Furthermore, unnecessary complications such as redeclaration issues and more runtime work discussed above can be successfully avoided.