Languages such as Algol, Ada, C, and Pascal are statically scoped. A block defines a new scope. Variables can be declared in that scope, and aren't visible from the outside. However, variables outside the scope -- in enclosing scopes -- are visible unless they are overridden. In Algol and Pascal (but not C or Ada) these scope rules also apply to the names of functions and procedures.

In *lexical scoping* (or *lexical scope*; also called *static scoping* or *static scope*), if a variable name's scope is a certain function (that is, a particular variable is declared in that function), then **its scope is the program text of the function definition**: within that text, the variable name exists, and is bound to the variable's value, but outside that text, the variable name does not exist.

By contrast, in *dynamic scoping* (or *dynamic scope*), if a variable name's scope is a certain function, then **its scope is the time-period during which the function is executing**: while the function is running, the variable name exists, and is bound to its variable, but after the function returns, the variable name does not exist.

Consider the following example:

If function f invokes a separately defined function g, then under lexical scoping, function g does *not* have access to f's local variables (since the text of g is not inside the text of f), while under dynamic scoping, function g *does* have access to f's local variables (since the invocation of g is inside the invocation of f).

**STATIC SCOPING**

With **lexical scope**, a name always refers to its (more or less) local lexical environment. This is a property of the program text and is made independent of the runtime [call stack](http://en.wikipedia.org/wiki/Call_stack) by the language implementation. Because this matching only requires analysis of the static program text, this type of scoping is also called **static scoping.**

Example of static scoping:

const int var = 5;

int g()

{

   int a = var + 5;

   return a;

}

int f()

{

   int var = 2;

   return g();

}

int main()

{

   g(); // returns 10

   f(); // returns 10

   return 0;

}

**DYNAMIC SCOPING**

With **dynamic scope**, each identifier has a global [stack](http://en.wikipedia.org/wiki/Stack_%28data_structure%29) of bindings. Introducing a local variable with name x pushes a binding onto the global x stack (which may have been empty), which is popped off when the [control flow](http://en.wikipedia.org/wiki/Control_flow) leaves the scope. Evaluating x in any context always yields the top binding. In other words, a global identifier refers to the identifier associated with the most recent environment. Note that this cannot be done at compile-time because **the binding stack only exists at** [**run-time**](http://en.wikipedia.org/wiki/Run_time_%28program_lifecycle_phase%29), which is why this type of scoping is called *dynamic* scoping.

Example of Dynamic scoping:

const int var = 5;

int g()

{

   int a = var + 5;

   return a;

}

int f()

{

   int var = 2;

   return g();

}

int main()

{

   g(); // returns 10

   f(); // returns 7

   return 0;

}