WIKIPEDIA

Scope (computer science)

In computer programming, the **scope** of a name binding—an association of a name to an entity, such as a variable—is the part of a program where the name binding is valid, that is where the name can be used to refer to the entity. In other parts of the program the name may refer to a different entity (it may have a different binding), or to nothing at all (it may be unbound). The scope of a name binding is also known as the **visibility** of an entity, particularly in older or more technical literature—this is from the perspective of the referenced entity, not the referencing name.

The term "scope" is also used to refer to the set of *all* name bindings that are valid within a part of a program or at a given point in a program, which is more correctly referred to as *context* or *environment*.[a]

Strictly speaking[b] and in practice for most programming languages, "part of a program" refers to a portion of source code (area of text), and is known as **lexical scope**. In some languages, however, "part of a program" refers to a portion of run time (time period during execution), and is known as **dynamic scope**. Both of these terms are somewhat misleading—they misuse technical terms, as discussed in the definition—but the distinction itself is accurate and precise, and these are the standard respective terms. Lexical scope is the main focus of this article, with dynamic scope understood by contrast with lexical scope.

In most cases, name resolution based on lexical scope is relatively straightforward to use and to implement, as in use one can read backwards in the source code to determine to which entity a name refers, and in implementation one can maintain a list of names and contexts when compiling or interpreting a program. Difficulties arise in name masking, forward declarations, and hoisting, while considerably subtler ones arise with non-local variables, particularly in closures.

Contents

Definition

The strict definition of the (lexical) "scope" of a name (identifier) is unambiguous—it is "the portion of source code in which a binding of a name with an entity applies"—and is virtually unchanged from its 1960 definition in the specification of ALGOL 60. Representative language specifications follow.

ALGOL 60 (1960)[1]
> The following kinds of quantities are distinguished: simple variables, arrays, labels, switches, and procedures. The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid.

C (2007)[2]
> An identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. The same identifier can denote different entities at different points in the program. [...] For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope.*

Go (2013)[3]
> A declaration binds a non-blank identifier to a constant, type, variable, function, label, or package. [...] The scope of a declared identifier is the extent of source text in which the identifier denotes the specified constant, type, variable, function, label, or package.

Most commonly "scope" refers to when a given name can refer to a given variable—when a declaration has effect—but can also apply to other entities, such as functions, types, classes, labels, constants, and enumerations.

Lexical scope vs. dynamic scope

A fundamental distinction in scope is what "part of a program" means. In languages with **lexical scope** (also called **static scope**), name resolution depends on the location in the source code and the *lexical context* (also called *static context*), which is defined by where the named variable or function is defined. In contrast, in languages with **dynamic scope** the name resolution depends upon the program state when the name is encountered which is determined by the *execution context* (also called *runtime context*, *calling context* or *dynamic context*). In practice, with lexical scope a name is resolved by searching the local lexical context, then if that fails by searching the outer lexical context, and so on, whereas with dynamic scope a name is resolved by searching the local execution context, then if that fails by searching the outer execution context, and so on, progressing up the call stack.[4]

Most modern languages use lexical scope for variables and functions, though dynamic scope is used in some languages, notably some dialects of Lisp, some "scripting" languages, and some template languages. [c] Perl 5 offers both lexical and dynamic scope. Even in lexically scoped languages, scope for closures can be confusing to the uninitiated, as these depend on the lexical context where the closure is defined, not where it is called.

Lexical resolution can be determined at compile time, and is also known as *early binding*, while dynamic resolution can in general only be determined at run time, and thus is known as *late binding*.

Related concepts

In object-oriented programming, dynamic dispatch selects an object method at runtime, though whether the actual name binding is done at compile time or run time depends on the language. De facto dynamic scope is common in macro languages, which do not directly do name resolution, but instead expand in place.

Some programming frameworks like AngularJS use the term "scope" to mean something entirely different than how it is used in this article. In those frameworks the scope is just an object of the programming language that they use (JavaScript in case of AngularJS) that is used in certain ways by the framework to emulate dynamic scope in a language that uses lexical scope for its variables. Those AngularJS scopes can themselves be in context or not in context (using the usual meaning of the term) in any given part of the program, following the usual rules of variable scope of the language like any other object, and using their own inheritance and transclusion rules. In the context of AngularJS, sometimes the term "$scope" (with a dollar sign) is used to avoid confusion, but using the dollar sign in variable names is often discouraged by the style guides.[5]

Use

Scope is an important component of name resolution,[d] which is in turn fundamental to language semantics. Name resolution (including scope) varies between programming languages, and within a programming language, varies by type of entity; the rules for scope are called **scope rules** (or **scoping rules**). Together with namespaces, scope rules are crucial in modular programming, so a change in one part of the program does not break an unrelated part.

Overview

When discussing scope, there are three basic concepts: *scope, extent,* and *context*. "Scope" and "context" in particular are frequently confused: scope is a property of a name binding, while context is a property of a part of a program, that is either a portion of source code (*lexical context* or *static context*) or a

portion of run time (*execution context, runtime context, calling context* or *dynamic context*). Execution context consists of lexical context (at the current execution point) plus additional runtime state such as the call stack.[e] Strictly speaking, during execution a program enters and exits various name bindings' scopes, and at a point in execution name bindings are "in context" or "not in context", hence name bindings "come into context" or "go out of context" as the program execution enters or exits the scope.[f] However, in practice usage is much looser.

Scope is a source-code level concept, and a property of name bindings, particularly variable or function name bindings—names in the source code are references to entities in the program—and is part of the behavior of a compiler or interpreter of a language. As such, issues of scope are similar to pointers, which are a type of reference used in programs more generally. Using the value of a variable when the name is in context but the variable is uninitialized is analogous to dereferencing (accessing the value of) a wild pointer, as it is undefined. However, as variables are not destroyed until they go out of context, the analog of a dangling pointer does not exist.

For entities such as variables, scope is a subset of lifetime (also known as extent)—a name can only refer to a variable that exists (possibly with undefined value), but variables that exist are not necessarily visible: a variable may exist but be inaccessible (the value is stored but not referred to within a given context), or accessible but not via the given name, in which case it is not in context (the program is "out of the scope of the name"). In other cases "lifetime" is irrelevant—a label (named position in the source code) has lifetime identical with the program (for statically compiled languages), but may be in context or not at a given point in the program, and likewise for static variables—a static global variable is in context for the entire program, while a static local variable is only in context within a function or other local context, but both have lifetime of the entire run of the program.

Determining which entity an name refers to is known as name resolution or name binding (particularly in object-oriented programming), and varies between languages. Given a name, the language (properly, the compiler or interpreter) checks all entities that are in context for matches; in case of ambiguity (two entities with the same name, such as a global and local variable with the same name), the name resolution rules are used to distinguish them. Most frequently, name resolution relies on an "inner-to-outer context" rule, such as the Python LEGB (Local, Enclosing, Global, Built-in) rule: names implicitly resolves to the narrowest relevant context. In some cases name resolution can be explicitly specified, such as by the `global` and `nonlocal` keywords in Python; in other cases the default rules cannot be overridden.

When two identical names are in context at the same time, referring to different entities, one says that *name masking* is occurring, where the higher-priority name (usually innermost) is "masking" the lower-priority name. At the level of variables, this is known as variable shadowing. Due to the potential for logic errors from masking, some languages disallow or discourage masking, raising an error or warning at compile time or run time.

Various programming languages have various different scope rules for different kinds of declarations and names. Such scope rules have a large effect on language semantics and, consequently, on the behavior and correctness of programs. In languages like C++, accessing an unbound variable does not have well-defined semantics and may result in undefined behavior, similar to referring to a dangling pointer; and declarations or names used outside their scope will generate syntax errors.

Scopes are frequently tied to other language constructs and determined implicitly, but many languages also offer constructs specifically for controlling scope.

Levels of scope

Scope can vary from as little as a single expression to as much as the entire program, with many possible gradations in between. The simplest scope rule is global scope—all entities are visible throughout the entire program. The most basic modular scope rule is two-level scope, with a global scope anywhere in the program, and local scope within a function. More sophisticated modular programming allows a separate module scope, where names are visible within the module (private to the module) but not visible outside it. Within a function, some languages, such as C, allow block scope to restrict scope to a subset of a function; others, notably functional languages, allow expression scope, to restrict scope to a single expression. Other scopes include file scope (notably in C) which behaves similarly to module scope, and block scope outside of functions (notably in Perl).

A subtle issue is exactly when a scope begins and ends. In some languages, such as C, a name's scope begins at its declaration, and thus different names declared within a given block can have different scopes. This requires declaring functions before use, though not necessarily defining them, and requires forward declaration in some cases, notably for mutual recursion. In other languages, such as JavaScript or Python, a name's scope begins at the start of the relevant block (such as the start of a function), regardless of where it is defined, and all names within a given block have the same scope; in JavaScript this is known as *variable hoisting*. However, when the name is bound to a value varies, and behavior of in-context names that have undefined value differs: in Python use of undefined names yields a runtime error, while in JavaScript undefined names declared with `var` (but not names declared with `let` nor `const`) are usable throughout the function because they are bound to the value `undefined`.

Expression scope

The scope of name is an expression, which is known as **expression scope**. Expression scope is available in many languages, especially functional languages which offer a feature called *let-expressions* allowing a declaration's scope to be a single expression. This is convenient if, for example, an intermediate value is needed for a computation. For example, in Standard ML, if `f()` returns `12`, then `let val x = f() in x * x end` is an expression that evaluates to `144`, using a temporary variable named `x` to avoid calling `f()` twice. Some languages with block scope approximate this functionality by offering syntax for a block to be embedded into an expression; for example, the aforementioned Standard ML expression could be written in Perl as `do { my $x = f(); $x * $x }`, or in GNU C as `({ int x = f(); x * x; })`.

In Python, auxiliary variables in generator expressions and list comprehensions (in Python 3) have expression scope.

In C, variable names in a function prototype have expression scope, known in this context as **function protocol scope**. As the variable names in the prototype are not referred to (they may be different in the actual definition)—they are just dummies—these are often omitted, though they may be used for generating documentation, for instance.

Block scope

The scope of a name is a block, which is known as **block scope**. Block scope is available in many, but not all, block-structured programming languages. This began with ALGOL 60, where "[e]very declaration ... is valid only for that block.",[6] and today is particularly associated with languages in the Pascal and C families and traditions. Most often this block is contained within a function, thus restricting the scope to a part of a function, but in some cases, such as Perl, the block may not be within a function.

```c
unsigned int sum_of_squares(const unsigned int N) {
  unsigned int ret = 0;
  for (unsigned int n = 1; n <= N; n++) {
    const unsigned int n_squared = n * n;
    ret += n_squared;
  }
  return ret;
}
```

A representative example of the use of block scope is the C code shown here, where two variables are scoped to the loop: the loop variable *n*, which is initialized once and incremented on each iteration of the loop, and the auxiliary variable *n_squared*, which is initialized at each iteration. The purpose is to avoid adding variables to the function scope that are only relevant to a particular block—for example, this prevents errors where the generic loop variable *i* has accidentally already been set to another value. In this example the expression `n * n` would generally not be assigned to an auxiliary variable, and the body of the loop would simply be written `ret += n * n` but in more complicated examples auxiliary variables are useful.

Blocks are primarily used for control flow, such as with if, while, and for loops, and in these cases block scope means the scope of variable depends on the structure of a function's flow of execution. However, languages with block scope typically also allow the use of "naked" blocks, whose sole purpose is to allow fine-grained control of variable scope. For example, an auxiliary variable may be defined in a block, then used (say, added to a variable with function scope) and discarded when the block ends, or a while loop might be enclosed in a block that initializes variables used inside the loop that should only be initialized once.

A subtlety of several programming languages, such as Algol 68 and C (demonstrated in this example and standardized since C99), is that block-scope variables can be declared not only within the body of the block, but also within the control statement, if any. This is analogous to function parameters, which are declared in the function declaration (before the block of the function body starts), and in scope for the whole function body. This is primarily used in for loops, which have an initialization statement separate from the loop condition, unlike while loops, and is a common idiom.

Block scope can be used for shadowing. In this example, inside the block the auxiliary variable could also have been called *n*, shadowing the parameter name, but this is considered poor style due to the potential for errors. Furthermore, some descendants of C, such as Java and C#, despite having support for block scope (in that a local variable can be made to go out of context before the end of a function), do not allow one local variable to hide another. In such languages, the attempted declaration of the second *n* would result in a syntax error, and one of the *n* variables would have to be renamed.

If a block is used to set the value of a variable, block scope requires that the variable be declared outside of the block. This complicates the use of conditional statements with single assignment. For example, in Python, which does not use block scope, one may initialize a variable as such:

```python
if c:
    a = "foo"
else:
    a = ""
```

where `a` is accessible after the `if` statement.

In Perl, which has block scope, this instead requires declaring the variable prior to the block:

```
my $a;
if (c) {
    $a = 'foo';
} else {
    $a = '';
}
```

Often this is instead rewritten using multiple assignment, initializing the variable to a default value. In Python (where it is not necessary) this would be:

```
a = ""
if c:
    a = "foo"
```

while in Perl this would be:

```
my $a = '';
if (c) {
    $a = 'foo';
}
```

In case of a single variable assignment, an alternative is to use the ternary operator to avoid a block, but this is not in general possible for multiple variable assignments, and is difficult to read for complex logic.

This is a more significant issue in C, notably for string assignment, as string initialization can automatically allocate memory, while string assignment to an already initialized variable requires allocating memory, a string copy, and checking that these are successful.

Some languages allow the concept of block scope to be applied, to varying extents, outside of a function. For example, in the Perl snippet at right, `$counter` is a variable name with block scope (due to the use of the `my` keyword), while `increment_counter` is a function name with global scope. Each call to `increment_counter` will increase the value of `$counter` by one, and return the new value. Code outside of this block can call `increment_counter`, but cannot otherwise obtain or alter the value of `$counter`. This idiom allows one to define closures in Perl.

```
sub increment_counter {
    my $counter = 0;
    return sub
    {
        return ++$counter;
    }
}
```

Function scope

The scope of a name is a function, which is known as **function scope**. Function scope is available in most programming languages which offer a way to create a *local variable* in a function or subroutine: a variable whose scope ends (that goes out of context) when the function returns. In most cases the lifetime of the variable is the duration of the function call—it is an automatic variable, created when the function starts (or the variable is declared), destroyed when the function returns—while the scope of the variable is within the function, though the meaning of "within" depends on whether scope is lexical or dynamic. However, some languages, such as C, also provide for static local variables, where the lifetime of the variable is the entire lifetime of the program, but the variable is only in context when inside the function. In the case of static local variables, the variable is created when the program initializes, and destroyed only when the program terminates, as with a static global variable, but is only in context within a function, like an automatic local variable.

Importantly, in lexical scope a variable with function scope has scope only within the *lexical context* of the function: it goes out of context when another function is called within the function, and comes back into context when the function returns—called functions have no access to the local variables of calling functions, and local variables are only in context within the body of the function in which they are declared. By contrast, in dynamic scope, the scope extends to the *execution context* of the function: local variables *stay in context* when another function is called, only going out of context when the defining function ends, and thus local variables are in context of the function in which they are defined *and all called functions*. In languages with lexical scope and nested functions, local variables are in context for nested functions, since these are within the same lexical context, but not for other functions that are not lexically nested. A local variable of an enclosing function is known as a non-local variable for the nested function. Function scope is also applicable to anonymous functions.

For example, in the snippet of Python code on the right, two functions are defined: `square` and `sum_of_squares`. `square` computes the square of a number; `sum_of_squares` computes the sum of all squares up to a number. (For example, `square(4)` is $4^2 = 16$, and `sum_of_squares(4)` is $0^2 + 1^2 + 2^2 + 3^2 + 4^2 = 30$.)

```
def square(n):
    return n * n

def sum_of_squares(n):
    total = 0
    i = 0
    while i <= n:
        total += square(i)
        i += 1
    return total
```

Each of these functions has a variable named *n* that represents the argument to the function. These two *n* variables are completely separate and unrelated, despite having the same name, because they are lexically scoped local variables with function scope: each one's scope is its own, lexically separate function and thus, they don't overlap. Therefore, `sum_of_squares` can call `square` without its own *n* being altered. Similarly, `sum_of_squares` has variables named *total* and *i*; these variables, because of their limited scope, will not interfere with any variables named *total* or *i* that might belong to any other function. In other words, there is no risk of a *name collision* between these names and any unrelated names, even if they are identical.

No name masking is occurring: only one variable named *n* is in context at any given time, as the scopes do not overlap. By contrast, were a similar fragment to be written in a language with dynamic scope, the *n* in the calling function would remain in context in the called function—the scopes would overlap—and would be masked ("shadowed") by the new *n* in the called function.

Function scope is significantly more complicated if functions are first-class objects and can be created locally to a function and then returned. In this case any variables in the nested function that are not local to it (unbound variables in the function definition, that resolve to variables in an enclosing context) create a closure, as not only the function itself, but also its context (of variables) must be returned, and then potentially called in a different context. This requires significantly more support from the compiler, and can complicate program analysis.

File scope

The scope of name is a file, which is known as **file scope**. File scope is largely particular to C (and C++), where scope of variables and functions declared at the top level of a file (not within any function) is for the entire file—or rather for C, from the declaration until the end of the source file, or more precisely translation unit (internal linking). This can be seen as a form of module scope, where modules are identified with files, and in more modern languages is replaced by an explicit module scope. Due to the presence of include statements, which add variables and functions to the internal context and may themselves call further include statements, it can be difficult to determine what is in context in the body of a file.

In the C code snippet above, the function name `sum_of_squares` has file scope.

Module scope

The scope of a name is a module, which is known as **module scope**. Module scope is available in modular programming languages where modules (which may span various files) are the basic unit of a complex program, as they allow information hiding and exposing a limited interface. Module scope was pioneered in the Modula family of languages, and Python (which was influenced by Modula) is a representative contemporary example.

In some object-oriented programming languages that lack direct support for modules, such as C++, a similar structure is instead provided by the class hierarchy, where classes are the basic unit of the program, and a class can have private methods. This is properly understood in the context of dynamic dispatch rather than name resolution and scope, though they often play analogous roles. In some cases both these facilities are available, such as in Python, which has both modules and classes, and code organization (as a module-level function or a conventionally private method) is a choice of the programmer.

Global scope

The scope of a name is an entire program, which is known as **global scope**. Variable names with global scope—called *global variables*—are frequently considered bad practice, at least in some languages, due to the possibility of name collisions and unintentional masking, together with poor modularity, and function scope or block scope are considered preferable. However, global scope is typically used (depending on the language) for various other sorts of names, such as names of functions, names of classes and names of other data types. In these cases mechanisms such as namespaces are used to avoid collisions.

Lexical scope vs. dynamic scope

The use of local variables — of variable names with limited scope, that only exist within a specific function — helps avoid the risk of a name collision between two identically named variables. However, there are two very different approaches to answering this question: What does it mean to be "within" a function?

In **lexical scope** (or **lexical scoping**; also called **static scope** or **static scoping**), if a variable name's scope is a certain 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 scope** (or **dynamic scoping**), 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 value, but after the function returns, the variable name does not exist. This means that if function `f` invokes a separately defined function `g`, then under lexical scope, function `g` does *not* have access to `f`'s local variables (assuming the text of `g` is not inside the text of `f`), while under dynamic scope, function `g` *does* have access to `f`'s local variables (since `g` is invoked during the invocation of `f`).

Consider, for example, the program on the right. The first line, `x=1`, creates a global variable `x` and initializes it to `1`. The second line, `function g() { echo $x ; x=2 ; }`, defines a function `g` that

```
$ # bash language
$ x=1
$ function g() { echo $x ; x=2 ; }
```

prints out ("echoes") the current value of `x`, and then sets `x` to `2` (overwriting the previous value). The third line, `function f() { local x=3 ; g ; }` defines a function `f` that creates a local variable `x` (hiding the identically named global variable) and initializes it to `3`, and then calls `g`. The fourth line, `f`, calls `f`. The fifth line, `echo $x`, prints out the current value of `x`.

```
$ function f() { local x=3 ; g ; }
$ f # does this print 1, or 3?
3
$ echo $x # does this print 1, or 2?
1
```

So, what exactly does this program print? It depends on the scope rules. If the language of this program is one that uses lexical scope, then `g` prints and modifies the global variable `x` (because `g` is defined outside `f`), so the program prints `1` and then `2`. By contrast, if this language uses dynamic scope, then `g` prints and modifies `f`'s local variable `x` (because `g` is called from within `f`), so the program prints `3` and then `1`. (As it happens, the language of the program is Bash, which uses dynamic scope; so the program prints `3` and then `1`. If the same code was run with ksh93 which uses lexical scope, the results would be different.)

Lexical scope

With **lexical scope**, a name always refers to its lexical context. This is a property of the program text and is made independent of the runtime call stack by the language implementation. Because this matching only requires analysis of the static program text, this type of scope is also called **static scope**. Lexical scope is standard in all ALGOL-based languages such as Pascal, Modula-2 and Ada as well as in modern functional languages such as ML and Haskell. It is also used in the C language and its syntactic and semantic relatives, although with different kinds of limitations. Static scope allows the programmer to reason about object references such as parameters, variables, constants, types, functions, etc. as simple name substitutions. This makes it much easier to make modular code and reason about it, since the local naming structure can be understood in isolation. In contrast, dynamic scope forces the programmer to anticipate all possible execution contexts in which the module's code may be invoked.

For example, Pascal is lexically scoped. Consider the Pascal program fragment at right. The variable `I` is visible at all points, because it is never hidden by another variable of the same name. The `char` variable `K` is visible only in the main program because it is hidden by the `real` variable `K` visible in procedure `B` and `C` only. Variable `L` is also visible only in procedure `B` and `C` but it does not hide any other variable. Variable `M` is only visible in procedure `C` and therefore not accessible either from procedure `B` or the main program. Also, procedure `C` is visible only in procedure `B` and can therefore not be called from the main program.

```
program A;
var I:integer;
    K:char;

    procedure B;
    var K:real;
        L:integer;

        procedure C;
        var M:real;
        begin
         (*scope A+B+C*)
        end;

     (*scope A+B*)
    end;

 (*scope A*)
end.
```

There could have been another procedure `C` declared in the program outside of procedure `B`. The place in the program where "`C`" is mentioned then determines which of the two procedures named `C` it represents, thus precisely analogous with the scope of variables.

Correct implementation of lexical scope in languages with first-class nested functions is not trivial, as it requires each function value to carry with it a record of the values of the variables that it depends on (the pair of the function and this context is called a closure). Depending on implementation and computer architecture, variable lookup *may* become slightly inefficient when very deeply lexically nested functions are used, although there are well-known techniques to mitigate this.[7][8] Also, for nested functions that only refer to their own arguments and (immediately) local variables, all relative locations can be known at compile time. No overhead at all is therefore incurred

when using that type of nested function. The same applies to particular parts of a program where nested functions are not used, and, naturally, to programs written in a language where nested functions are not available (such as in the C language).

History

Lexical scope was used for the imperative language ALGOL 60 and has been picked up in most other imperative languages since then.[4]

Languages like Pascal and C have always had lexical scope, since they are both influenced by the ideas that went into ALGOL 60 and ALGOL 68 (although C did not include lexically nested functions).

Perl is a language with dynamic scope that added static scope afterwards.

The original Lisp interpreter (1960) used dynamic scope. *Deep binding*, which approximates static (lexical) scope, was introduced in LISP 1.5 (via the Funarg device developed by Steve Russell, working under John McCarthy).

All early Lisps used dynamic scope, at least when based on interpreters. In 1982, Guy L. Steele Jr. and the Common LISP Group publish *An overview of Common LISP*,[9] a short review of the history and the divergent implementations of Lisp up to that moment and a review of the features that a *Common Lisp* implementation should have. On page 102, we read:

> Most LISP implementations are internally inconsistent in that by default the interpreter and compiler may assign different semantics to correct programs; this stems primarily from the fact that the interpreter assumes all variables to be dynamically scoped, while the compiler assumes all variables to be local unless forced to assume otherwise. This has been done for the sake of convenience and efficiency, but can lead to very subtle bugs. The definition of Common LISP avoids such anomalies by explicitly requiring the interpreter and compiler to impose identical semantics on correct programs.

Implementations of Common LISP were thus required to have lexical scope. Again, from *An overview of Common LISP*:

> In addition, Common LISP offers the following facilities (most of which are borrowed from MacLisp, InterLisp or Lisp Machines Lisp): (...) Fully lexically scoped variables. The so-called "FUNARG problem"[10][11] is completely solved, in both the downward and upward cases.

By the same year in which *An overview of Common LISP* was published (1982), initial designs (also by Guy L. Steele Jr.) of a compiled, lexically scoped Lisp, called *Scheme* had been published and compiler implementations were being attempted. At that time, lexical scope in Lisp was commonly feared to be inefficient to implement. In *A History of T*,[12] Olin Shivers writes:

> *All* serious Lisps in production use at that time were dynamically scoped. No one who hadn't carefully read the Rabbit[13] thesis (written by Guy Lewis Steele Jr. in 1978) believed lexical scope would fly; even the few people who *had* read it were taking a bit of a leap of faith that this was going to work in serious production use.

The term "lexical scope" dates at least to 1967,[14] while the term "lexical scoping" dates at least to 1970, where it was used in Project MAC to describe the scope rules of the Lisp dialect MDL (then known as "Muddle").[15]

Dynamic scope

With **dynamic scope**, a name refers to execution context. It is uncommon in modern languages.[4] In technical terms, this means that each name has a global stack 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 leaves the scope. Evaluating x in any context always yields the top binding. Note that this cannot be done at compile-time because the binding stack only exists at run-time, which is why this type of scope is called *dynamic* scope.

Generally, certain blocks are defined to create bindings whose lifetime is the execution time of the block; this adds some features of static scope to the dynamic scope process. However, since a section of code can be called from many different locations and situations, it can be difficult to determine at the outset what bindings will apply when a variable is used (or if one exists at all). This can be beneficial; application of the principle of least knowledge suggests that code avoid depending on the *reasons* for (or circumstances of) a variable's value, but simply use the value according to the variable's definition. This narrow interpretation of shared data can provide a very flexible system for adapting the behavior of a function to the current state (or policy) of the system. However, this benefit relies on careful documentation of all variables used this way as well as on careful avoidance of assumptions about a variable's behavior, and does not provide any mechanism to detect interference between different parts of a program. Some languages, like Perl and Common Lisp, allow the programmer to choose static or dynamic scope when defining or redefining a variable. Examples of languages that use dynamic scope include Logo, Emacs Lisp, LaTeX and the shell languages bash, dash, and PowerShell.

Dynamic scope is fairly easy to implement. To find an name's value, the program could traverse the runtime stack, checking each activation record (each function's stack frame) for a value for the name. In practice, this is made more efficient via the use of an association list, which is a stack of name/value pairs. Pairs are pushed onto this stack whenever declarations are made, and popped whenever variables go out of context.[16] *Shallow binding* is an alternative strategy that is considerably faster, making use of a *central reference table*, which associates each name with its own stack of meanings. This avoids a linear search during run-time to find a particular name, but care should be taken to properly maintain this table.[16] Note that both of these strategies assume a last-in-first-out (LIFO) ordering to bindings for any one variable; in practice all bindings are so ordered.

An even simpler implementation is the representation of dynamic variables with simple global variables. The local binding is performed by saving the original value in an anonymous location on the stack that is invisible to the program. When that binding scope terminates, the original value is restored from this location. In fact, dynamic scope originated in this manner. Early implementations of Lisp used this obvious strategy for implementing local variables, and the practice survives in some dialects which are still in use, such as GNU Emacs Lisp. Lexical scope was introduced into Lisp later. This is equivalent to the above shallow binding scheme, except that the central reference table is simply the global variable binding context, in which the current meaning of the variable is its global value. Maintaining global variables isn't complex. For instance, a symbol object can have a dedicated slot for its global value.

Dynamic scope provides an excellent abstraction for thread local storage, but if it is used that way it cannot be based on saving and restoring a global variable. A possible implementation strategy is for each variable to have a thread-local key. When the variable is accessed, the thread-local key is used to access

the thread-local memory location (by code generated by the compiler, which knows which variables are dynamic and which are lexical). If the thread-local key does not exist for the calling thread, then the global location is used. When a variable is locally bound, the prior value is stored in a hidden location on the stack. The thread-local storage is created under the variable's key, and the new value is stored there. Further nested overrides of the variable within that thread simply save and restore this thread-local location. When the initial, outermost override's context terminates, the thread-local key is deleted, exposing the global version of the variable once again to that thread.

With referential transparency the dynamic scope is restricted to the argument stack of the current function only, and coincides with the lexical scope.

Macro expansion

In modern languages, macro expansion in a preprocessor is a key example of de facto dynamic scope. The macro language itself only transforms the source code, without resolving names, but since the expansion is done in place, when the names in the expanded text are then resolved (notably free variables), they are resolved based on where they are expanded (loosely "called"), as if dynamic scope were occurring.

The C preprocessor, used for macro expansion, has de facto dynamic scope, as it does not do name resolution by itself. For example, the macro:

```
#define ADD_A(x) x + a
```

will expand to add `a` to the passed variable, with this name only later resolved by the compiler based on where the macro `ADD_A` is "called" (properly, expanded), is in dynamic scope, and is independent of where the macro is defined. Properly, the C preprocessor only does lexical analysis, expanding the macro during the tokenization stage, but not parsing into a syntax tree or doing name resolution.

For example, in the following code, the `a` in the macro is resolved (after expansion) to the local variable at the expansion site:

```
#define ADD_A(x) x + a

void add_one(int *x) {
  const int a = 1;
  *x = ADD_A(*x);
}

void add_two(int *x) {
  const int a = 2;
  *x = ADD_A(*x);
}
```

Qualified names

As we have seen, one of the key reasons for scope is that it helps prevent name collisions, by allowing identical names to refer to distinct things, with the restriction that the names must have separate scopes. Sometimes this restriction is inconvenient; when many different things need to be accessible throughout a program, they generally all need names with global scope, so different techniques are required to avoid name collisions.

To address this, many languages offer mechanisms for organizing global names. The details of these mechanisms, and the terms used, depend on the language; but the general idea is that a group of names can itself be given a name — a prefix — and, when necessary, an entity can be referred to by a *qualified name* consisting of the name plus the prefix. Normally such names will have, in a sense, two sets of scopes: a scope (usually the global scope) in which the qualified name is visible, and one or more narrower scopes in which the *unqualified name* (without the prefix) is visible as well. And normally these groups can themselves be organized into groups; that is, they can be *nested*.

Although many languages support this concept, the details vary greatly. Some languages have mechanisms, such as *namespaces* in C++ and C#, that serve almost exclusively to enable global names to be organized into groups. Other languages have mechanisms, such as *packages* in Ada and *structures* in Standard ML, that combine this with the additional purpose of allowing some names to be visible only to other members of their group. And object-oriented languages often allow classes or singleton objects to fulfill this purpose (whether or not they *also* have a mechanism for which this is the primary purpose). Furthermore, languages often meld these approaches; for example, Perl's packages are largely similar to C++'s namespaces, but optionally double as classes for object-oriented programming; and Java organizes its variables and functions into classes, but then organizes those classes into Ada-like packages.

By language

Scope rules for representative languages follow.

C

In C, scope is traditionally known as **linkage** or **visibility**, particularly for variables. C is a lexically scoped language with global scope (known as *external linkage*), a form of module scope or file scope (known as *internal linkage*), and local scope (within a function); within a function scopes can further be nested via block scope. However, standard C does not support nested functions.

The lifetime and visibility of a variable are determined by its storage class. There are three types of lifetimes in C: static (program execution), automatic (block execution, allocated on the stack), and manual (allocated on the heap). Only static and automatic are supported for variables and handled by the compiler, while manually allocated memory must be tracked manually across different variables. There are three levels of visibility in C: external linkage (global), internal linkage (roughly file), and block scope (which includes functions); block scopes can be nested, and different levels of internal linkage is possible by use of includes. Internal linkage in C is visibility at the translation unit level, namely a source file after being processed by the C preprocessor, notably including all relevant includes.

C programs are compiled as separate object files, which are then linked into an executable or library via a linker. Thus name resolution is split across the compiler, which resolves names within a translation unit (more loosely, "compilation unit", but this is properly a different concept), and the linker, which resolves names across translation units; see linkage for further discussion.

In C, variables with block scope enter context when they are declared (not at the top of the block), go out of context if any (non-nested) function is called within the block, come back into context when the function returns, and go out of context at the end of the block. In the case of automatic local variables, they are also allocated on declaration and deallocated at the end of the block, while for static local variables, they are allocated at program initialization and deallocated at program termination.

The following program demonstrates a variable with block scope coming into context partway through the block, then exiting context (and in fact being deallocated) when the block ends:

```
#include <stdio.h>

int main(void) {
  char x = 'm';
  printf("%c\n", x);
  {
    printf("%c\n", x);
    char x = 'b';
    printf("%c\n", x);
  }
  printf("%c\n", x);
}
```

The program outputs:

```
m
m
b
m
```

There are other levels of scope in C.[17] Variable names used in a function prototype have function prototype visibility, and exit context at the end of the function prototype. Since the name is not used, this is not useful for compilation, but may be useful for documentation. Label names for GOTO statement have function scope, while case label names for switch statements have block scope (the block of the switch).

C++

All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that a, b, and result were of type int. A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.

Modern versions allow nested lexical scope.

Swift

Swift has a similar rule for scopes with C++, but contains different access modifiers.

Modifier	Immediate scope	File	Containing module/package	Rest of the world
open	Yes	Yes	Yes	Yes, allows subclass
public	Yes	Yes	Yes	Yes, disallows subclass
internal	Yes	Yes	Yes	No
fileprivate	Yes	Yes	No	No
private	Yes	No	No	No

Go

Go is lexically scoped using blocks.[3]

Java

Java is lexically scoped.

A Java class can contain three types of variables:[18]

Local variables
: are defined inside a method, or a particular block. These variables are local to where they were defined and lower levels. For example, a loop inside a method can use that method's local variables, but not the other way around. The loop's variables (local to that loop) are destroyed as soon as the loop ends.

Member variables
: also called *fields* are variables declared within the class, outside of any method. By default, these variables are available for all methods within that class and also for all classes in the package.

Parameters
: are variables in method declarations.

In general, a set of brackets defines a particular scope, but variables at top level within a class can differ in their behavior depending on the modifier keywords used in their definition. The following table shows the access to members permitted by each modifier.[19]

Modifier	Class	Package	Subclass	World
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
(no modifier)	Yes	Yes	No	No
private	Yes	No	No	No

JavaScript

JavaScript has simple *scope rules*,[20] but variable initialization and name resolution rules can cause problems, and the widespread use of closures for callbacks means the lexical context of a function when defined (which is used for name resolution) can be very different from the lexical context when it is called (which is irrelevant for name resolution). JavaScript objects have name resolution for properties, but this is a separate topic.

JavaScript has lexical scope [21] nested at the function level, with the global context being the outermost context. This scope is used for both variables and for functions (meaning function declarations, as opposed to variables of function type).[22] Block scope with the `let` and `const` keywords is standard since ECMAScript 6. Block scope can be produced by wrapping the entire block in a function and then executing it; this is known as the immediately-invoked function expression (IIFE) pattern.

While JavaScript scope is simple—lexical, function-level—the associated initialization and name resolution rules are a cause of confusion. Firstly, assignment to a name not in scope defaults to creating a new global variable, not a local one. Secondly, to create a new local variable one must use the `var` keyword; the variable is then created at the top of the function, with value `undefined` and the variable is assigned its value when the assignment expression is reached:

> A variable with an *Initialiser* is assigned the value of its *AssignmentExpression* when the *VariableStatement* is executed, not when the variable is created.[23]

This is known as *variable hoisting*[24]—the declaration, but not the initialization, is hoisted to the top of the function. Thirdly, accessing variables before initialization yields `undefined`, rather than a syntax error. Fourthly, for function declarations, the declaration and the initialization are both hoisted to the top of the function, unlike for variable initialization. For example, the following code produces a dialog with output `undefined`, as the local variable declaration is hoisted, shadowing the global variable, but the initialization is not, so the variable is undefined when used:

```
a = 1;
function f() {
  alert(a);
  var a = 2;
}
f();
```

Further, as functions are first-class objects in JavaScript and are frequently assigned as callbacks or returned from functions, when a function is executed, the name resolution depends on where it was originally defined (the lexical context of the definition), not the lexical context or execution context where it is called. The nested scopes of a particular function (from most global to most local) in JavaScript, particularly of a closure, used as a callback, are sometimes referred to as the **scope chain**, by analogy with the prototype chain of an object.

Closures can be produced in JavaScript by using nested functions, as functions are first-class objects.[25] Returning a nested function from an enclosing function includes the local variables of the enclosing function as the (non-local) lexical context of the returned function, yielding a closure. For example:

```
function newCounter() {
  // return a counter that is incremented on call (starting at 0)
  // and which returns its new value
  var a = 0;
  var b = function() { a++; return a; };
  return b;
}
c = newCounter();
alert(c() + ' ' + c());  // outputs "1 2"
```

Closures are frequently used in JavaScript, due to being used for callbacks. Indeed, any hooking of a function in the local context as a callback or returning it from a function creates a closure if there are any unbound variables in the function body (with the context of the closure based on the nested scopes of the current lexical context, or "scope chain"); this may be accidental. When creating a callback based on parameters, the parameters must be stored in a closure, otherwise it will accidentally create a closure that refers to the variables in the enclosing context, which may change.[26]

Name resolution of properties of JavaScript objects is based on inheritance in the prototype tree—a path to the root in the tree is called a *prototype chain*—and is separate from name resolution of variables and functions.

Lisp

Lisp dialects have various rules for scope.

The original Lisp used dynamic scope; it was Scheme, inspired by ALGOL, that introduced static (lexical) scope to the Lisp family.

Maclisp used dynamic scope by default in the interpreter and lexical scope by default in compiled code, though compiled code could access dynamic bindings by use of `SPECIAL` declarations for particular variables.[27] However, Maclisp treated lexical binding more as an optimization than one would expect in modern languages, and it did not come with the closure feature one might expect of lexical scope in modern Lisps. A separate operation, `*FUNCTION`, was available to somewhat clumsily work around some of that issue.[28]

Common Lisp adopted lexical scope from Scheme,[29] as did Clojure.

ISLISP has lexical scope for ordinary variables. It also has dynamic variables, but they are in all cases explicitly marked; they must be defined by a `defdynamic` special form, bound by a `dynamic-let` special form, and accessed by an explicit `dynamic` special form.[30]

Some other dialects of Lisp, like Emacs Lisp, still use dynamic scope by default. Emacs Lisp now has lexical scope available on a per-buffer basis.[31]

Python

For variables, Python has function scope, module scope, and global scope. Names enter context at the start of a scope (function, module, or global scope), and exit context when a non-nested function is called or the scope ends. If a name is used prior to variable initialization, this raises a runtime exception. If a variable is simply accessed (not assigned to), name resolution follows the LEGB (Local, Enclosing, Global, Built-in) rule which resolves names to the narrowest relevant context. However, if a variable is assigned to, it defaults to declaring a variable whose scope starts at the start of the level (function, module, or global), not at the assignment. Both these rules can be overridden with a `global` or `nonlocal` (in Python 3) declaration prior to use, which allows accessing global variables even if there is a masking nonlocal variable, and assigning to global or nonlocal variables.

As a simple example, a function resolves a variable to the global scope:

```
>>> def f():
...     print(x)
...
>>> x = "global"
>>> f()
global
```

Note that `x` is defined before `f` is called, so no error is raised, even though it is defined after its reference in the definition of `f`. Lexically this is a forward reference, which is allowed in Python.

Here assignment creates a new local variable, which does not change the value of the global variable:

```
>>> def f():
...     x = "f"
...     print(x)
...
```

```
>>> x = "global"
>>> print(x)
global
>>> f()
f
>>> print(x)
global
```

Assignment to a variable within a function causes it to be declared local to the function, hence its scope is the entire function, and thus using it prior to this assignment raises an error. This differs from C, where the scope of the local variable start at its declaration. This code raises an error:

```
>>> def f():
...     print(x)
...     x = "f"
...
>>> x = "global"
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'x' referenced before assignment
```

The default name resolution rules can be overridden with the `global` or `nonlocal` (in Python 3) keywords. In the below code, the `global x` declaration in `g` means that `x` resolves to the global variable. It thus can be accessed (as it has already been defined), and assignment assigns to the global variable, rather than declaring a new local variable. Note that no `global` declaration is needed in `f`—since it does not assign to the variable, it defaults to resolving to the global variable.

```
>>> def f():
...     print(x)
...
>>> def g():
...     global x
...     print(x)
...     x = "g"
...
>>> x = "global"
>>> f()
global
>>> g()
global
>>> f()
g
```

`global` can also be used for nested functions. In addition to allowing assignment to a global variable, as in an unnested function, this can also be used to access the global variable in the presence of a nonlocal variable:

```
>>> def f():
...     def g():
...         global x
...         print(x)
...     x = "f"
...     g()
...
>>> x = "global"
>>> f()
global
```

For nested functions, there is also the `nonlocal` declaration, for assigning to a nonlocal variable, similar to using `global` in an unnested function:

```
>>> def f():
...     def g():
...         nonlocal x  # Python 3 only
...         x = "g"
...     x = "f"
...     g()
...     print(x)
...
>>> x = "global"
>>> f()
g
>>> print(x)
global
```

R

R is a lexically scoped language, unlike other implementations of S where the values of free variables are determined by a set of global variables, while in R they are determined by the context in which the function was created.[32] The scope contexts may be accessed using a variety of features (such as `parent.frame()`) which can simulate the experience of dynamic scope should the programmer desire.

There is no block scope:

```
a <- 1
{
  a <- 2
}
message(a)
## 2
```

Functions have access to scope they were created in:

```
a <- 1
f <- function() {
  message(a)
}
f()
## 1
```

Variables created or modified within a function stay there:

```
a <- 1
f <- function() {
  message(a)
  a <- 2
  message(a)
}
f()
## 1
## 2
message(a)
## 1
```

Variables created or modified within a function stay there unless assignment to enclosing scope is explicitly requested:

```
a <- 1
f <- function() {
  message(a)
  a <<- 2
  message(a)
}
f()
## 1
## 2
message(a)
## 2
```

Although R has lexical scope by default, function scopes can be changed:

```
a <- 1
f <- function() {
  message(a)
}
my_env <- new.env()
my_env$a <- 2
f()
## 1
environment(f) <- my_env
f()
## 2
```

See also

- Closure (computer science)
- Global variable
- Local variable
- Let expression
- Non-local variable
- Name binding
- Name resolution (programming languages)
- Variables (scope and extent)
- Information hiding
- Immediately-invoked function expressions in Javascript
- Object lifetime

Notes

a. See definition for meaning of "scope" versus "context".

b. "Dynamic scope" bases name resolution on *extent* (lifetime), not *scope*, and thus is formally inaccurate.

c. For example, the Jinja template engine for Python by default uses both lexical scope (for imports) and dynamic scope (for includes), and allows behavior to be specified with keywords; see Import Context Behavior (http://jinja.pocoo.org/docs/templates/#import-context-behavior).

d. "Name resolution" and "name binding" are largely synonymous; narrowly speaking "resolution" determines to which name a particular use of a name refers, without associating it with any meaning, as in higher-order abstract syntax, while "binding" associates the name with an actual meaning. In practice the terms are used interchangeably.

e. For self-modifying code the lexical context itself can change during run time.

f. By contrast, *"a name binding's context", *"a name binding coming into scope" or *"a name binding going out of scope" are all incorrect—a name binding has scope, while a part of a program has context.

References

1. "Report on the Algorithmic Language Algol 60", 2.7. Quantities, kinds and scopes
2. WG14 N1256 (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf) (2007 updated version of the C99 standard), 6.2.1 Scopes of identifiers, 2007-09-07
3. The Go Programming Language Specification (https://golang.org/ref/spec): Declarations and scope (https://golang.org/ref/spec#Declarations_and_scope), Version of Nov 13, 2013
4. Borning A. CSE 341 -- Lexical and Dynamic Scoping (https://web.archive.org/web/20150207225438/http://courses.cs.washington.edu/courses/cse341/08au/general-concepts/scoping.html). University of Washington.
5. Crockford, Douglas. "Code Conventions for the JavaScript Programming Language" (https://docs.angularjs.org/guide/scope). Retrieved 2015-01-04.
6. Backus, J. W.; Wegstein, J. H.; Van Wijngaarden, A.; Woodger, M.; Bauer, F. L.; Green, J.; Katz, C.; McCarthy, J.; Perlis, A. J.; Rutishauser, H.; Samelson, K.; Vauquois, B. (1960). "Report on the algorithmic language ALGOL 60". *Communications of the ACM*. **3** (5): 299. doi:10.1145/367236.367262 (https://doi.org/10.1145%2F367236.367262). S2CID 278290 (https://api.semanticscholar.org/CorpusID:278290).
7. "Programming Language Pragmatics (http://booksite.elsevier.com/9780123745149/appendices/data/chapters/3a_impsc.pdf)", LeBlank-Cook symbol table
8. "A Symbol Table Abstraction to Implement Languages with Explicit Scope Control (http://origin-www.computer.org/csdl/trans/ts/1983/01/01703006.pdf)", LeBlank-Cook, 1983
9. Louis Steele, Guy (August 1982). "An overview of Common LISP". *LFP '82: Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*: 98–107. doi:10.1145/800068.802140 (https://doi.org/10.1145%2F800068.802140). ISBN 0897910826. S2CID 14517358 (https://api.semanticscholar.org/CorpusID:14517358).
10. Joel, Moses (June 1970). "The Function of FUNCTION in LISP". *MIT AI Memo 199*. MIT Artificial Intelligence Lab.
11. Steele, Guy Lewis Jr.; Sussman, Gerald Jay (May 1978). "The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One and Two)". *MIT AI Memo 453*. MIT Artificial Intelligence Lab.
12. Shivers, Olin. "History of T" (http://www.paulgraham.com/thist.html). *Paul Graham*. Retrieved 5 February 2020.
13. Steele, Guy Lewis Jr. (May 1978). "RABBIT: A Compiler for SCHEME". MIT. hdl:1721.1/6913 (https://hdl.handle.net/1721.1%2F6913).
14. "lexical scope (https://books.google.com/books?id=qk0jAQAAMAAJ&q=%22lexical+scope%22)", *Computer and Program Organization, Part 3 (https://books.google.com/books?id=qk0jAQAAMAAJ&pg=PA18)*, p. 18, at Google Books, University of Michigan. Engineering Summer Conferences, 1967
15. "lexical scoping (https://books.google.com/books?id=m0IdAQAAMAAJ&q=%22lexical+scoping%22)", *Project MAC Progress Report, Volume 8 (https://books.google.com/books?id=m0IdAQAAMAAJ&pg=PA80)*, p. 80, at Google Books, 1970.
16. Scott 2009, 3.4 Implementing Scope, p. 143.
17. "Scope (http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8l.doc%2Flanguage%2Fref%2Fzexscope_c.htm)", *XL C/C++ V8.0 for Linux,* IBM
18. "Declaring Member Variables (The Java™ Tutorials > Learning the Java Language > Classes and Objects)" (https://docs.oracle.com/javase/tutorial/java/javaOO/variables.html). *docs.oracle.com*. Retrieved 19 March 2018.

19. "Controlling Access to Members of a Class (The Java™ Tutorials > Learning the Java Language > Classes and Objects)" (https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html). *docs.oracle.com*. Retrieved 19 March 2018.
20. "Everything you need to know about Javascript variable scope (http://www.coolcoder.in/2014/03/everything-you-need-to-know-about.html)", Saurab Parakh (http://www.coolcoder.in/p/about-us.html), *Coding is Cool (http://www.coolcoder.in/)*, 2010-02-08
21. "Annotated ES5" (https://es5.github.io/#x10.2). *es5.github.io*. Retrieved 19 March 2018.
22. "Functions" (https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope). *MDN Web Docs*. Retrieved 19 March 2018.
23. "12.2 Variable Statement (https://es5.github.io/#x12.2)", Annotated ECMAScript 5.1, Last updated: 2012-05-28
24. "JavaScript Scoping and Hoisting (http://www.adequatelygood.com/JavaScript-Scoping-and-Hoisting.html)", Ben Cherry (http://www.adequatelygood.com/about.html), *Adequately Good (http://www.adequatelygood.com/)*, 2010-02-08
25. Javascript Closures (http://jibbering.com/faq/notes/closures/), Richard Cornford. March 2004
26. "Explaining JavaScript Scope And Closures (http://robertnyman.com/2008/10/09/explaining-javascript-scope-and-closures/)", Robert Nyman, October 9, 2008
27. Pitman, Kent (December 16, 2007). "The Revised Maclisp Manual (The Pitmanual), Sunday Morning Edition" (http://maclisp.info/pitmanual/complr.html#23.1.2). *MACLISP.info*. HyperMeta Inc. Declarations and the Compiler, Concept "Variables". Retrieved October 20, 2018. "If the variable to be bound has been declared to be special, the binding is compiled as code to imitate the way the interpreter binds variables"
28. Pitman, Kent (December 16, 2007). "The Revised Maclisp Manual (The Pitmanual), Sunday Morning Edition" (http://www.maclisp.info/pitmanual/eval.html#3.7.1). *MACLISP.info*. HyperMeta Inc. The Evaluator, Special Form `*FUNCTION`. Retrieved October 20, 2018. "`*FUNCTION` is intended to help solve the "funarg problem," however it only works in some easy cases."
29. Pitman, Kent; et al. (webbed version of ANSI standard X3.226-1994) (1996). "Common Lisp HyperSpec" (http://www.lispworks.com/documentation/lw50/CLHS/Body/01_ab.htm). *Lispworks.com*. LispWorks Ltd. 1.1.2 History. Retrieved October 20, 2018. "MacLisp improved on the Lisp 1.5 notion of special variables ... The primary influences on Common Lisp were Lisp Machine Lisp, MacLisp, NIL, S-1 Lisp, Spice Lisp, and Scheme."
30. "Programming Language ISLISP, ISLISP Working Draft 23.0" (http://www.islisp.info/Documents/PDF/islisp-2007-03-17-pd-v23.pdf) (PDF). *ISLISP.info*. 11.1 The lexical principle. Retrieved October 20, 2018. "Dynamic bindings are established and accessed by a separate mechanism (i.e., `defdynamic`, `dynamic-let`, and `dynamic`)."
31. "Lexical Binding" (https://www.emacswiki.org/emacs/LexicalBinding). *EmacsWiki*. Retrieved October 20, 2018. "Emacs 24 has optional lexical binding, which can be enabled on a per-buffer basis."
32. "R FAQ" (https://cran.r-project.org/doc/FAQ/R-FAQ.html#Lexical-scoping). *cran.r-project.org*. Retrieved 19 March 2018.

- Abelson, Harold; Sussman, Gerald Jay; Sussman, Julie (1996) [1984]. *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press. ISBN 0-262-51087-1.
  - "Lexical addressing" (https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book-Z-H-35.html#%_sec_5.5.6)
- Scott, Michael L. (2009) [2000]. *Programming Language Pragmatics* (https://www.cs.rochester.edu/~scott/pragmatics/) (Third ed.). Morgan Kaufmann Publishers. ISBN 978-0-12-374514-9.
  - Chapter 3: Names, Scopes, and Bindings, pp. 111–174
  - Section 13.4.1: Scripting Languages: Innovative Features: Names and Scopes, pp. 691–699