Scope is an enclosing context where expressions are associated with their values. It is used to define the visibility or accessibility of variables from different parts of the program. Scopes may:

- contain declarations or definitions of identifiers;
- contain statements and/or expressions which define an executable algorithm or part thereof;
- nest or be nested.

In most programming languages, variables may appear in two different ways: as references or as declarations. A variable reference is a use of the variable. For example, in the LISP expression

\[(f \ x \ y)\]

all the variables f, x and y appear as references. However, in

\[(let \ ((x \ (+ \ y \ 7))) \ (+ \ x \ 3))\]

the first occurrence of x is a declaration as it introduces the variable as a name for some value.

We say that a variable declaration is bound by the declaration with which it is associated, and that it is bound to its value.

Declarations in most programming languages have a limited scope, so the same variable name may be used for different purposes in different parts of a program. In certain cases, the scope may be determined without executing the program. Such scoping rules are called lexical scoping rules. To find which declaration corresponds to a given use of a variable, we search outward from the use until we find a declaration of the variable. Here is a simple example in LISP:
(let ((x 3) (y 4))
   (+ (let ((x (+ y 5))
             (* x y))
       x))
)

Call this \textit{x1}

Call this \textit{x2}

Here \(x\) refers to \textit{x2}

Here \(x\) refers to \textit{x1}

\[
\left(\begin{array}{c}
\text{(let ((x 3) (y 4))}
\text{(+ (let ((x (+ y 5))
\text{(* x y))
\text{x})
\text{)})}
\end{array}\right)
\]

Under lexical scoping, we can create a hole in a scope by re-declaring a variable. Such an inner declaration shadows the outer one. For instance, in the example above, the inner \(x\) shadows the outer one in the multiplication \((* x y)\).

Lexical scoping may be illustrated by means of a contour diagram. Here is one for the preceding example.

The value of the foregoing expression may be computed as follows:

\[
\left(\begin{array}{c}
\text{(let ((x 9))}
\text{(+ (* x 4))
\text{3)}
\end{array}\right)
\]

\[
\left(\begin{array}{c}
\text{(+ 36}
\text{3)}
\end{array}\right)
\]

39
A more complicated program follows.

```
(lambda (x y)
  (let ((z (+ x y)))
    (lambda (x z)
      (let ((x (let ((x (+ x y z))
                    (y 11))
             (+ x y z)))
        (+ x y z))))
)
)
```

Following is the contour diagram.

The expression \((+ x y z)\) appears on lines 5, 7 and 8. Line 5 is within the scope of \(x_2\) and \(z_2\), which is within the scope of \(z_1\), which itself is within the scope of \(x_1\) and \(y_1\). So at Line 5, \(x\) refers to \(x_2\); \(y\) refers to \(y_1\); and \(z\) refers to \(z_2\). Line 7 is within the scope of \(x_4\) and \(y_2\), which is within the scope of \(x_2\) and \(z_2\), which is within the scope of \(z_1\), which itself is within the scope of \(x_1\) and \(y_1\). So at Line 7, \(x\) refers to \(x_4\); \(y\) refers to \(y_2\); and \(z\) refers to \(z_2\). Finally, Line 8 is within the scope of \(x_3\), which is within the scope of \(x_2\) and \(z_2\), which is within the scope of \(z_1\), which itself is within the scope of \(x_1\) and \(y_1\). So at Line 8, \(x\) refers to \(x_3\); \(y\) refers to \(y_1\); and \(z\) refers to \(z_2\).
Static scoping makes it easy to construct a piece of modular code and also to reason about it, since its binding structure can be understood in isolation. Consider the following program fragment in Pascal:

```pascal
program main
var i: integer;
  k: char;
  r: real;

procedure B
var k: real;
  L: Integer;

procedure C
var m: real;

begin // body of procedure C
  // (1)
  end;

begin // body of procedure B
  // (2)
  end;

begin // body of procedure A
  // (3)
  end.
```

The variable `i` is accessible as an integer at points (1), (2) and (3) in the program because its scope is global, and is not overridden by another variable of the same name. The variable `k` is accessible as a real at points (1) and (2) and as a character at (3). Also, because of the scope of `k`, the variable called `k` in `C` (at point (1)) and `B` (at point (2)) is not the same variable `k` in the main program at point (3). Variable `L` is accessible only in procedure `C` at point (1) and procedure `B` at point (2), and is not accessible from the main program. Variable `m` is only accessible in procedure `C` at point (1), and is not accessible either from Procedure `B` or the main program. Also, procedure `C` can only be called from procedure `B`; it cannot be called from the main program. Also, there could be yet another procedure `C` declared later in the program, and a reference to that procedure would be dependent upon where in the program code as to which procedure is being called, same as to which variable is being referenced in the above example.
Example 1: Assuming that static scoping is used, determine the scope of the variable \( x \) declared in the procedure `compute()` in the following program fragment in Pascal.

```pascal
program main
var y: real;

procedure compute()
var x: integer;

procedure initialize()
var z: real;
begin {initialize}
...
end; {initialize}

procedure transform()
var x: real;
begin {transform}
...
end; {transform}

begin {compute}
...
end; {compute}
begin {main}
...
end; {main}
```

**Answer:** The procedure in which the variable is declared is `compute()`, so `compute()` itself is in the scope of \( x \). Further, procedure `initialize()` is defined within `compute()` and since `initialize()` does not re-declare \( x \), `initialize()` is also within the scope of \( x \) declared in `compute()`.

The procedure `transform()` is defined within `compute()`. However, `transform()` does re-declare \( x \), so `transform()` is not within the scope of \( x \) declared in `compute()`. Also, `main()` is not in the scope of \( x \) declared in `compute()`.

Answer the following with respect to the preceding program fragment:

1. If procedure `transform()` prints the value of \( x \), which variable would it be referencing?
2. If procedure `initialize()` prints the value of \( x \), which variable would it be referencing?
3. If `program main()` attempts to use the value of \( x \), would it be legal?
Example 2: Present the output corresponding to execution of the following piece of code in C in which static scoping is used.

```c
int N;
void main() {N = 1; A(); printf("main: %ld", N);} 

void A()
int N;
{N = 2; B(); printf("A: %ld . ", N); } 

void B() {N = 3;}
```

Answer: A: 2. main 3

Example 3: Present the output corresponding to execution of the following piece of code in C in which static scoping is used.

```c
int x = 3; /* (1) */
void f(int x) /* (2) */
{
    g ();
}
void g ()
{
    printf (x);
}
void doit ()
{
    int x = 12;
    f(42);
    g();
}
```

Answer: This program would print ‘3’ twice.

Remark: Were this language to use dynamic scoping instead, it would print ‘42’, and then ‘12’.
**Example 4:** Present the output corresponding to execution of the following piece of code using (i) static scoping, and (ii) dynamic scoping.

```plaintext
val x = 1;
def f(y) = x + y;
def g() {val x = 2; f(x);}
g();
```

Static scoping: Result is 1 + 2  
Dynamic scoping: Result is 2 + 2

**Example 5:** Present the output corresponding to execution of the following piece of code using (i) static scoping, and (ii) dynamic scoping.

```plaintext
procedure p
  x: integer;
procedure q
  begin x := x+1; end;
procedure r
  x: integer;
  begin x := 1; q; write(x); end;
begin // procedure p
  x := 2;
r;
end;
```

Static scope: 1  
Dynamic scope: 2

**Further details on dynamic scoping**

With dynamic scope, each identifier 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. 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 runtime, which is why this type of scoping is called *dynamic* scoping.
Generally, certain blocks are defined to create bindings whose lifetime is the execution time of the block; this adds some features of static scoping to the dynamic scoping 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. As such, dynamic scoping can be dangerous and few modern languages use it. Some languages, like Perl and Common Lisp, allow the programmer to choose static or dynamic scoping when defining or redefining a variable. Logo and Emacs lisp are examples of languages that use dynamic scoping.

Dynamic scoping is fairly easy to implement. To find an identifier's value, the program could traverse the runtime stack, checking each activation record (each function's stack frame) for a value for the identifier. 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 scope. An alternate strategy that is considerably faster is to make use of a central reference table, which associates each name with its current meaning. This avoids a linear search during runtime to find a particular name, but maintaining this table is more complex. 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.

**Example:** Here is an example that compares the consequences of using static scope and dynamic scope. Consider the following piece of code in a C-like language:

```c
int x = 0;
int f() { return x; }
int g() { int x = 1; return f(); }
```

With static scoping, calling `g` will return 0 since it has been determined at compile time that the expression `x` in any invocation of `f` will yield the global `x` binding which is unaffected by the introduction of a local variable of the same name in `g`.

With dynamic scoping, the binding stack for the `x` identifier will contain two items when `f` is invoked from `g`: the global binding to 0, and the binding to 1 introduced in `g` (which is still present on the stack since the control flow hasn't left `g` yet). Since evaluating the identifier expression by definition always yields the top binding, the result is 1.
In the language Perl, variables can be defined with either static or dynamic scoping. Perl's keyword "my" defines a *statically* scoped local variable, while the keyword "local" defines a *dynamically* scoped local variable. This allows for further clarification with practical examples of each scoping model.

```perl
$x = 0;
sub f { return $x; }
sub g { my $x = 1; return f(); }
print g()."\n";
```

The example above uses "my" for static scoping of `g`'s local variable `$x`. As above, calling `g` returns 0 because `f` cannot see `g`'s variable `$x`, so it looks for the global `$x`.

```perl
$x = 0;
sub f { return $x; }
sub g { local $x = 1; return f(); }
print g()."\n";
```

In this alternative, "local" is used to make `g`'s `$x` dynamically-scoped. Now, calling `g` yields 1 because `f` sees `g`'s local variable by looking down the execution stack. In other words, the dynamically-scoped variable `$x` is resolved in the environment of execution, rather than the environment of definition.

**References:**
