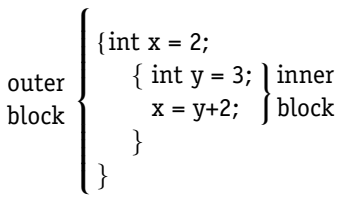
# 7

# Scope, Functions, and Storage Management

In this chapter storage management for block-structured languages is described by the run-time data structures that are used in a simple, reference implementation. The programming language features that make the association between program names and memory locations interesting are scope, which allows two syntactically identical names to refer to different locations, and function calls, which each require a new memory area in which to store function parameters and local variables. Some important topics in this chapter are parameter passing, access to global variables, and a storage optimization associated with a particular kind of function call called a *tail call*. We will see that storage management becomes more complicated in languages with nested function declarations that allow functions to be passed as arguments or returned as the result of function calls.

## 7.1 BLOCK-STRUCTURED LANGUAGES

Most modern programming languages provide some form of block. A *block* is a region of program text, identified by begin and end markers, that may contain declarations local to this region. Here are a few lines of C code to illustrate the idea:




In this section of code, there are two blocks. Each block begins with a left brace, {, and ends with a right brace, }. The outer block begins with the first left brace and ends with the last right brace. The inner block is nested inside the outer block. It begins with the second left brace and ends with the first right brace. The variable

x is declared in the outer block and the variable y is declared in the inner block. A variable declared within a block is said to be *local* to that block. A variable declared in an enclosing block is said to be *global* to the block. In this example, x is local to the outer block, y is local to the inner block, and x is global to the inner block.

C, Pascal, and ML are all block-structured languages. In-line blocks are delineated by { . . . } in C, begin...end in Pascal, and let...in...end in ML. The body of a procedure or function is also a block in each of these languages.

Storage management mechanisms associated with block structure allow functions to be called recursively.

The versions of Fortran in widespread use during the 1960s and 1970s were not block structured. In historical Fortran, every variable, including every parameter of each procedure (called a subroutine in Fortran) was assigned a fixed-memory location. This made it *impossible* to call a procedure recursively, either directly or indirectly. If Fortran procedure P calls Q, Q calls R, and then R attempts to call P, the second call to P is not allowed. If P were called a second time in this call chain, the second call would write over the parameters and return address for the first call. This would make it impossible for the call to return properly.

Block-structured languages are characterized by the following properties:

- New variables may be declared at various points in a program.
- Each declaration is visible within a certain region of program text, called a block. Blocks may be nested, but cannot partially overlap. In other words, if two blocks contain any expressions or statements in common, then one block must be entirely contained within the other.
- When a program begins executing the instructions contained in a block at run time, memory is allocated for the variables declared in that block.
- When a program exits a block, some or all of the memory allocated to variables declared in that block will be deallocated.
- An identifier that is not declared in the current block is considered global to the block and refers to the entity with this name that is declared in the closest enclosing block.

Although most modern general-purpose programming languages are block structured, many important languages do not provide full support for all combinations of block-structured features. Most notably, standard C and C++ do not allow local function declarations within nested blocks and therefore do not address implementation issues associated with the return of functions from nested blocks.

In this chapter, we look at the memory management and access mechanisms for three classes of variables:

- *local variables*, which are stored on the stack in the activation record associated with the block
- *parameters* to function or procedure blocks, which are also stored in the activation record associated with the block
- *global variables*, which are declared in some enclosing block and therefore must be accessed from an activation record that was placed on the run-time stack before activation of the current block.

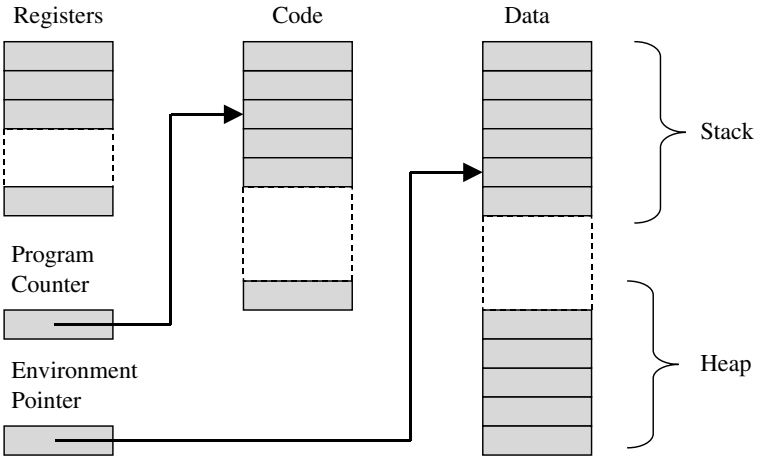



**Figure 7.1.** Program stack.

It may seem surprising that most complications arise in connection with access to global variables. However, this is really a consequence of stack-based memory management: The stack is used to make it easy to allocate and access local variables. In placing local variables close at hand, a global variable may be buried on the stack under any number of activation records.

### Simplified Machine Model

We use the simplified machine model in Figure 7.1 to look at the memory management in block-structured languages.

The machine model in Figure 7.1 separates code memory from data memory. The program counter stores the address of the current program instruction and is normally incremented after each instruction. When the program enters a new block, an *activation record* containing space for local variables declared in the block is added to the run-time stack (drawn here at the top of data memory), and the environment pointer is set to point to the new activation record. When the program exits the block, the activation record is removed from the stack and the environment pointer is reset to its previous location. The program may store data that will exist longer than the execution of the current block on the heap. The fact that the most recently allocated activation record is the first to be deallocated is sometimes called the *stack discipline*. Although most block-structured languages are implemented by a stack, higher-order functions may cause the stack discipline to fail.

Although Figure 7.1 includes some number of registers, generally used for short-term storage of addresses and data, we will not be concerned with registers or the instructions that may be stored in the code segment of memory.

***Reference Implementation.*** A reference implementation is an implementation of a language that is designed to define the behavior of the language. It need not be an efficient implementation. The goal in this chapter is to give you enough information about how blocks are implemented in most programming languages so that you can understand when storage needs to be allocated, what kinds of data are stored on the run-time stack, and how an executing program accesses the data locations it needs. We do this by describing a reference implementation. Because our goal is to understand programming languages, not build a compiler, this reference

implementation will be simple and direct. More efficient methods for doing many of the things described in this chapter, tailored for specific languages, may be found in compiler books.

#### A Note about C

The C programming language is designed to make C easy to compile and execute, avoiding several of the general scoping and memory management techniques described in this chapter. Understanding the general cases considered here will give C programmers some understanding of the specific ways in which C is simpler than other languages. In addition, C programmers who want the effect of passing functions and their environments to other functions may use the ideas described in this chapter in their programs.

Some commercial implementations of C and C++ actually do support function parameters and return values, with preservation of static scope by use of closures. (We will discuss closures in Section 7.4.) In addition, the C++ Standard Template Library (covered in Subsection 9.4.3) provides a form of function closure as many programmers find function arguments and return values useful.

## 7.2 IN-LINE BLOCKS

An in-line block is a block that is not the body of a function or procedure. We study in-line blocks first, as these are simpler than blocks associated with function calls.

### 7.2.1 Activation Records and Local Variables

When a running program enters an in-line block, space must be allocated for variables that are declared in the block. We do this by allocating a set of memory locations called an *activation record* on the run-time stack. An activation record is also sometimes called a *stack frame*.

To see how this works, consider the following code example. If this code is part of a larger program, the stack may contain space for other variables before this block is executed. When the outer block is entered, an activation record containing space for `x` and `y` is pushed onto the stack. Then the statements that set values of `x` and `y` will be executed, causing values of `x` and `y` to be stored in the activation record. On entry into the inner block, a separate activation record containing space for `z` will be added to the stack. After the value of `z` is set, the activation record containing this value will be popped off the stack. Finally, on exiting the outer block, the activation record containing space for `x` and `y` will be popped off the stack:

```
{ int x=0;
  int y=x+1;
      { int z=(x+y)*(x-y);
        };
};
```

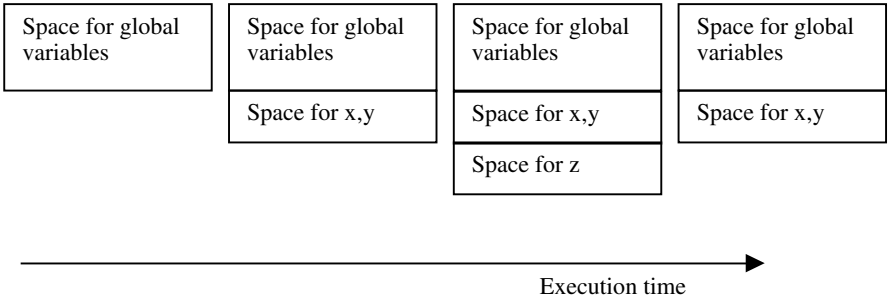



**Figure 7.2.** Stack grows and shrinks during program execution.

We can visualize this by using a sequence of figures of the stack. As in Figure 7.1, the stack is shown growing downward in memory in Figure 7.2.

A simple optimization involves combining small nested blocks into a single block. For the preceding example program, this would save the run time spent in pushing and popping the inner block for z, as z could be stored in the same activation record as that of x and y. However, because we plan to illustrate the general case by using small examples, we do not use this optimization in further discussion of stack storage allocation. In all of the program examples we consider, we assume that a new activation record is allocated on the run-time stack each time the program enters a block.

The number of locations that need to be allocated at run time depends on the number of variables declared in the block and their types. Because these quantities are known at compile time, the compiler can determine the format of each activation record and store this information as part of the compiled code.

### Intermediate Results

In general, an activation record may also contain space for intermediate results. These are values that are not given names in the code, but that may need to be saved temporarily. For example, the activation record for this block,

```
{ int z = (x+y)*(x-y);
}
```

may have the form

| Space for z |
|---|
| Space for x+y |
| Space for x-y |

because the values of subexpressions x+y and x-y may have to be evaluated and stored somewhere before they are multiplied.

On modern computers, there are enough registers that many intermediate results are stored in registers and not placed on the stack. However, because register

allocation is an implementation technique that does not affect programming language design, we do not discuss registers or register allocation.

### Scope and Lifetime

It is important to distinguish the scope of a declaration from the lifetime of a location:

*Scope*: a region of text in which a declaration is visible.

*Lifetime*: the duration, during a run of a program, during which a location is allocated as the result of a specific declaration.

We may compare lifetime and scope by using the following example, with vertical lines used to indicate matching block entries and exits.

```
{ int x = . . . ;
|   { int y = . . . ;
|   |   { int x = . . . ;
|   |   |   . . . .
|   |   };
|   };
};
```

In this example, the inner declaration of `x` hides the outer one. The inner block is called a *hole in the scope* of the outer declaration of `x`, as the outer `x` cannot be accessed within the inner block. This example shows that lifetime does not coincide with scope because the lifetime of the outer `x` includes time when inner block is being executed, but the scope of the outer `x` does not include the scope of the inner one.

### Blocks and Activation Records for ML

Throughout our discussion of blocks and activation records, we follow the convention that, whenever the program enters a new block, a new activation record is allocated on the run-time stack. In ML code that has sequences of declarations, we treat each declaration as a separate block. For example, in the code

```
fun f(x) = x+1;
fun g(y) = f(y) +2;
g(3);
```

we consider the declaration of `f` one block and the declaration of `g` another block inside the outer block. If this code is not inside some other construct, then these blocks will both end at the end of the program.

When an ML expression contains declarations as part of the *let-in-end* construct, we consider the declarations to be part of the same block. For example, consider this example expression:

```
let fun g(y) = y+3
    fun h(z) = g(g(z))
in
    h(3)
end;
```

This expression contains a block, beginning with let and ending with end. This block contains two declarations, functions g and h, and one expression, h(x), calling one of these functions. The construct let ... in ... end is approximately equivalent to { ... ; ... } in C. The main syntactic difference is that declarations appear between the keywords let and in, and expressions using these declarations appear between keywords in and end. Because the declarations of functions g and h appear in the same block, the names g and h will be given values in the same activation record.

### 7.2.2 Global Variables and Control Links

Because different activation records have different sizes, operations that push and pop activation records from the run-time stack store a pointer in each activation record to the top of the preceding activation record. The pointer to the top of the previous activation record is called the *control link*, as it is the link that is followed when control returns to the instructions in the preceding block. This gives us a structure shown in Figure 7.3. Some authors call the control link the *dynamic link* because the control links mark the dynamic sequence of function calls created during program execution.

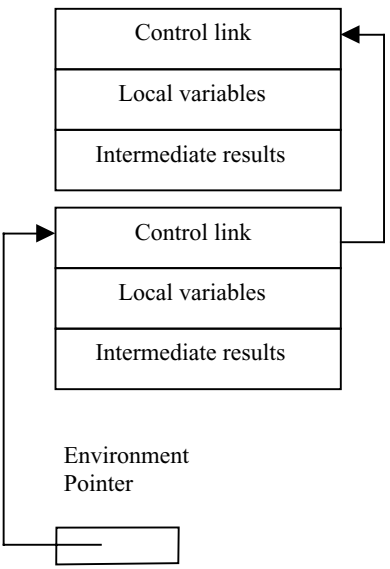



**Figure 7.3.** Activation records with control links.

When a new activation record is added to the stack, the control link of the new activation record is set to the previous value of the environment pointer, and the environment pointer is updated to point to the new activation record. When an activation record is popped off the stack, the environment pointer is reset by following the control link from the activation record.

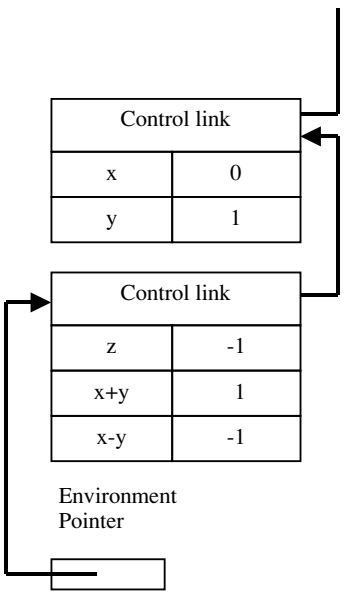
The code for pushing and popping activation records from the stack is generated by the compiler at compile time and becomes part of the compiled code for the program. Because the size of an activation record can be determined from the text of the block, the compiler can generate code for block entry that is tailored to the size of the activation record.

When a global variable occurs in an expression, the compiler must generate code that will find the location of that variable at run time. However, the compiler can compute the number of blocks between the current block and the block where the variable is declared; this is easily determined from the program text. In addition, the relative position of each variable within its block is known at compile time. Therefore, the compiler can generate lookup code that follows a predetermined number of links

**Example 7.1**

```
{ int x=0;
  int y=x+1;
      { int z=(x+y)*(x-y);
        };
        };
```

When the expressions x+y and x-y are evaluated during execution of this code, the run-time stack will have activation records for the inner and outer blocks as shown below:



On a register-based machine, the machine code generated by the compiler will find the variables x and y, load each into registers, and then add the two values. The code for loading x uses the environment pointer to find the top of the current activation, then computes the location of x by adding 1 to the location stored in the control link of the current activation record. The compiler generates this code by analyzing the program text at compile time: The variable x is declared one block out from the current block, and x is the first variable declared in the block. Therefore, the control link from the current activation record will lead to the activation record containing x, and the location of x will be one location down from the top of that block. Similar steps can be used to find y at the second location down from the top of its activation record. Although the details may vary from one compiler to the next, the main point is that the compiler can determine the number of control links to follow and the relative location of the variable within the correct block from the source code. In particular, it is *not* necessary to store variable names in activation records.

## 7.3 FUNCTIONS AND PROCEDURES

Most block-structured languages have procedures or functions that include parameters, local variables, and a body consisting of an arbitrary expression or sequence of statements. For example, here are representative Algol-like and C-like forms:

```
procedure P(<parameters>)
begin
    <local variables>;
    <procedure body>;
end;
```

```
<type> f(<parameters>)
{
        <local variables>;
<function body>;
};
```

The difference between a *procedure* and a *function* is that a function has a return value but a procedure does not. In most languages, functions and procedures may have side effects. However, a procedure has only side effects; a procedure call is a statement and not an expression. Because functions and procedures have many characteristics in common, we use the terms almost interchangeably in the rest of this chapter. For example, the text may discuss some properties of functions, and then a code example may illustrate these properties with a procedure. This should remind you that the discussion applies to functions and procedures in many programming languages, whether or not the language treats procedures as different from functions.

### 7.3.1 Activation Records for Functions

The activation record of a function or procedure block must include space for parameters and return values. Because a procedure may be called from different call sites, it is also necessary to save the return address, which is the location of the next instruction to execute after the procedure terminates. For functions, the activation record must also contain the location that the calling routine expects to have filled with the return value of the function.

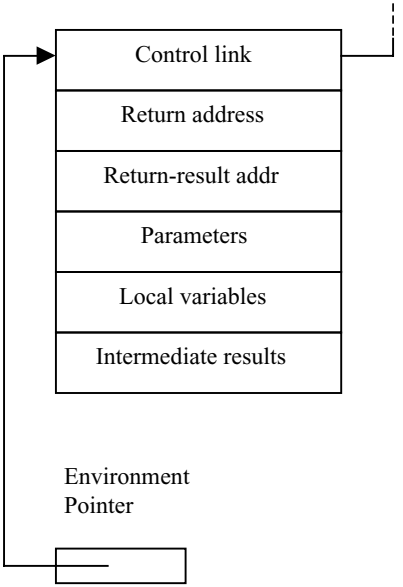



**Figure 7.4.** Activation record associated with funtion call.

The activation record associated with a function (see Figure 7.4) must contain space for the following information:

- *control link*, pointing to the previous activation record on the stack,
- *access link*, which we will discuss in Subsection 7.3.3,
- *return address*, giving the address of the first instruction to execute when the function terminates,
- *return-result address*, the location in which to store the function return value,
- *actual parameters* of the function,
- *local variables* declared within the function,
- *temporary storage* for intermediate results computed with the function executes.

This information may be stored in different orders and in different ways in different language implementations. Also, as mentioned earlier, many compilers perform optimizations that place some of these values in registers. For concreteness, we assume that no registers are used and that the six components of an activation record are stored in the order previously listed.

Although the names of variables are eliminated during compilation, we often draw activation records with the names of variables in the stack. This is just to make it possible for us to understand the figures.

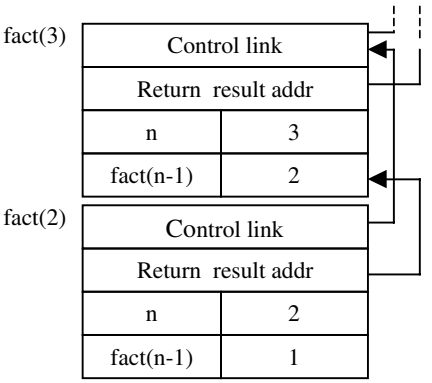
### Example 7.2

We can see how function activation records are added and removed from the run-time stack by tracing the execution of the familiar factorial function:

```
fun fact(n) = if n <= 1 then 1 else n * fact(n-1);
```

Note that in each of the lower activation records, the return-result address points to the space allocated in the activation record above it. This is so that, on return from fact(1), for example, the return result of this call can be stored in the activation record for fact(2). At that point, the final instruction from the calculation of fact(2) will be executed, multiplying local variable n by the intermediate result fact(1).

The final illustration of this example shows the situation during return from fact(2) when the return result of fact(2) has been placed in the activation record of fact(3), but the activation record for fact(2) has not yet been popped off the stack.




### 7.3.2 Parameter Passing

The parameter names used in a function declaration are called *formal parameters*. When a function is called, expressions called *actual parameters* are used to compute the parameter values for that call. The distinction between formal and actual parameters is illustrated in the following code:

```
proc p (int x, int y) {
    if (x > y) then ... else ... ;
    ...
    x := y*2 + 3;
    ...
}
p (z, 4*z+1);
```

The identifiers x and y are formal parameters of the procedure p. The actual parameters in the call to p are z and 4*z+1.

The way that actual parameters are evaluated and passed to the function depends on the programming language and the kind of parameter-passing mechanisms it uses. The main distinctions between different parameter-passing mechanisms are

- the time that the actual parameter is evaluated
- the location used to store the parameter value.

Most current programming languages evaluate the actual parameters before executing the function body, but there are some exceptions. (One reason that a language or

program optimization might wish to delay evaluation of an actual parameter is that evaluation might be expensive and the actual parameter might not be used in some calls.) Among mechanisms that evaluate the actual parameter before executing the function body, the most common are

- *Pass-by-reference*: pass the L-value (address) of the actual parameter
- *Pass-by-value*: pass the R-value (contents of address) of the actual parameter

Recall that we discussed L-values and R-values in Subsection 5.4.5 in connection with ML reference cells (assignable locations) and assignment. We will discuss how pass-by-value and pass-by-reference work in more detail below. Other mechanisms such as *pass-by-value-result* are covered in the exercises.

The difference between pass-by-value and pass-by-reference is important to the programmer in several ways:

***Side Effects.*** Assignments inside the function body may have different effects under pass-by-value and pass-by-reference.

***Aliasing.*** Aliasing occurs when two names refer to the same object or location. Aliasing may occur when two parameters are passed by reference or one parameter passed by reference has the same location as the global variable of the procedure.

***Efficiency.*** Pass-by-value may be inefficient for large structures if the value of the large structure must be copied. Pass-by-reference may be less efficient than pass-by-value for small structures that would fit directly on stack, because when parameters are passed by reference we must dereference a pointer to get their value.

There are two ways of explaining the semantics of call-by-reference and call-by-value. One is to draw pictures of computer memory and the run-time program stack, showing whether the stack contains a copy of the actual parameter or a reference to it. The other explanation proceeds by translating code into a language that distinguishes between L- and R-values. We use the second approach here because the rest of the chapter gives you ample opportunity to work with pictures of the run-time stack.

### Semantics of Pass-by-Value

In pass-by-value, the actual parameter is evaluated. The value of the actual parameter is then stored in a new location allocated for the function parameter. For example, consider this function definition and call:

```
function f (x) = { x := x+1; return x };
....f(y)...;
```

If the parameter is passed by value and y is an integer variable, then this code has the same meaning as the following ML code:

```
fun f (z : int) = let   x = ref z   in   x := !x+1; !x   end;
...f(!y)...;
```

As you can see from the type, the value passed to the function f is an integer. The integer is the R-value of the actual parameter y, as indicated by the expression !y in the call. In the body of f, a new integer location is allocated and initialized to the R-value of y.

If the value of y is 0 before the call, then the value of f(!y) is 1 because the function f increments the parameter and returns its value. However, the value of y is still 0 after the call, because the assignment inside the body of f changes the contents of only a temporary location.

### Semantics of Pass-by-Reference

In pass-by-reference, the actual parameter must have an L-value. The L-value of the actual parameter is then bound to the formal parameter. Consider the same function definition and call used in the explanation of pass-by-value:

```
function f (x) = { x := x+1; return x };
....f(y) ...;
```

If the parameter is passed by reference and y is an integer variable, then this code has the same meaning as the following ML code:

```
fun f (x : int ref) = ( x := !x+1; !x );
...f(y)
```

As you can see from the type, the value passed to the function f is an integer reference (L-value).

If the value of y is 0 before the call, then the value of f(!y) is 1 because the function f increments the parameter and returns its value. However, unlike the situation for pass-by-value, the value of y is 1 after the call because the assignment inside the body of f changes the value of the actual parameter.

### Example 7.3

Here is an example, written in an Algol-like notation, that combines pass-by-reference and pass-by-value:

```
fun f(pass-by-ref x : int, pass-by-value y : int)
    begin
        x := 2;
        y := 1;
        if x = 1 then return 1 else return 2;
    end;
var z : int;
z := 0;
print f(z,z);
```

Translating the preceding pseudo-Algol example into ML gives us

```
fun f(x : int ref, y : int) =
    let val yy = ref y in
        x := 2;
        yy := 1;
        if (!x = 1) then 1 else 2
    end;
val z = ref 0;
f(z,!z);
```

This code, which treats L- and R-values explicitly, shows that for pass-by-reference we pass an L-value, the integer reference z. For pass-by-value, we pass an R-value, the contents !z of z. The pass-by-value is assigned a new temporary location.

With y passed by value as written, z is assigned the value 2. If y is instead passed by reference, then x and y are aliases and z is assigned the value 1.

**Example 7.4**

Here is a function that tests whether its two parameters are aliases:

```
function (y,z){
    y := 0;
    z :=0;
    y := 1;
    if z=1 then y :=0; return 1 else y :=0; return 0
}
```

If y and z are aliases, then setting y to 1 will set z to 1 and the function will return 1. Otherwise, the function will return 0. Therefore, a call f(x,x) will behave differently if the parameters are pass-by-value than if the parameters are pass-by-reference.

### 7.3.3 Global Variables (First-Order Case)

If an identifier x appears in the body of a function, but x is not declared inside the function, then the value of x depends on some declaration outside the function. In this situation, the location of x is outside the activation record for the function. Because x must be declared in some other block, access to a global x involves finding an appropriate activation record elsewhere on the stack.

There are two main rules for finding the declaration of a global identifier:

- *Static Scope*: A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text.
- *Dynamic Scope*: A global identifier refers to the identifier associated with the most recent activation record.

These definitions can be tricky to understand, so be sure to read the examples below carefully. One important difference between static and dynamic scope is that finding a declaration under static scope uses the static (unchanging) relationship between blocks in the program text. In contrast, dynamic scope uses the actual sequence of calls that are executed in the dynamic (changing) execution of the program.

Although most current general-purpose programming languages use static scope for declarations of variables and functions, dynamic scoping is an important concept that is used in special-purpose languages and for specialized constructs such as exceptions.

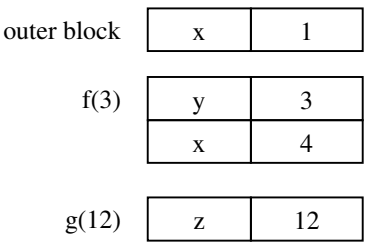
| Dynamically Scoped | Statically Scoped |
|---|---|
| Older Lisps | Newer Lisps, Scheme |
| TeX/LaTeX document languages | Algol and Pascal |
| Exceptions in many languages | C |
| Macros | ML |
| | Other current languages |

### Example 7.5

The difference between static and dynamic scope is illustrated by the following code, which contains two declarations of x:

```
int x=1;
function g(z) = x+z;
function f(y) = {
      int x = y+1;
      return g(y*x)
};
f(3);
```

The call f(3) leads to a call g(12) inside the function f. This causes the expression x+z in the body of g to be evaluated. After the call to g, the run-time stack will contain activation records for the outer declaration of x, the invocation of f, and the invocation of g, as shown in the following illustration.




At this point, two integers named x are stored on the stack, one from the outer block declaring x and one from the local declaration of x inside f. Under dynamic scope, the identifier x in the expression x+z will be interpreted as the one from the most

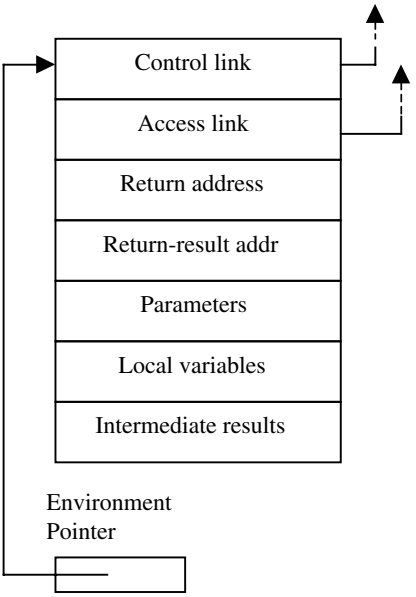



**Figure 7.5.** Activation record with access link for functions call with static scope.

recently created activation record, namely `x=4`. Under static scope, the identifier `x` in `x+z` will refer to the declaration of `x` from the closest program block, looking upward from the place that `x+z` appears in the program text. Under static scope, the relevant declaration of `x` is the one in the outer block, namely `x=1`.

### Access Links are Used to Maintain Static Scope

The *access link* of an activation record points to the activation record of the closest enclosing block in the program. In-line blocks do not need an access link, as the closest enclosing block will be the most recently entered block – for in-line blocks, the control link points to the closest enclosing block. For functions, however, the closest enclosing block is determined by where the function is declared. Because the point of declaration is often different from the point at which a function is called, the access link will generally point to a different activation record than the control link. Some authors call the access link the *static link*, as the access links represent the static nesting structure of blocks in the source program.

The general format for activation records with an access link is shown in Figure 7.5.

### Example 7.6

Let us look at the activation records and access links for the example code from Example 7.5, treating each ML declaration as a separate block.

Figure 7.6 shows the run-time stack after the call to `g` inside the body of `f`. As always, the control links each point to the preceding activation record on the stack.

The control links are drawn on the left here to leave room for the access links on the right. The access link for each block points to the activation record of the closest enclosing block in the program text. Here are some important points about

As for in-line blocks, the compiler can determine how many access links to follow and where to find a variable within an activation record at compile time. These properties are easily determined from the structure of the source code.

To summarize, the *control* link is a link to the activation record of the previous (calling) block. The *access link* is a link to the activation record of the closest enclosing block in program text. The control link depends on the dynamic behavior of program whereas the access link depends on only the static form of the program text. Access links are used to find the location of global variables in statically scoped languages with nested blocks at run time.

Access links are needed only in programming languages in which functions may be declared inside functions or other nested blocks. In C, in which all functions are declared in the outermost global scope, access links are not needed.

### 7.3.4 Tail Recursion (First-Order Case)

In this subsection we look at a useful compiler optimization called tail recursion elimination. For tail recursive functions, which are subsequently described, it is possible to reuse an activation record for a recursive call to the function. This reduces the amount of space used by a recursive function.

The main programming language concept we need is the concept of tail call. Suppose function f calls function g. Functions f and g might be different functions or f and g could be the same function. A call to f in the body of g is a *tail call* if g returns the result of calling f without any further computation. For example, in the function

```
fun g(x) = if x=0 then f(x) else f(x)*2
```

the first call to f in the body of g is a tail call, as the return value of g is exactly the return value of the call to f. The second call to f in the body of g is not a tail call because g performs a computation involving the return value of f before g returns.

A function f is *tail recursive* if all recursive calls in the body of f are tail calls to f.

**Example 7.7**

Here is a tail recursive function that computes factorial:

```
fun tlfact(n,a) = if n <= 1 then a else tlfact(n-1, n * a);
```

More specifically, for any positive integer n, tlfact(n,a) returns n!. We can see that tlfact is a tail recursive function because the only recursive call in the body of tlfact is a tail call.

The advantage of tail recursion is that we can use the same activation record for all recursive calls. Consider the call tlfact(3,1). Figure 7.7 shows the parts of each activation record in the computation that are relevant to the discussion.