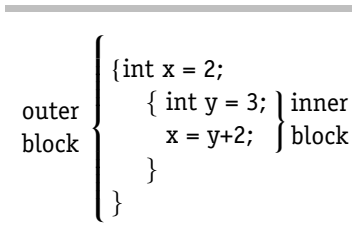
7

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

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

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.

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

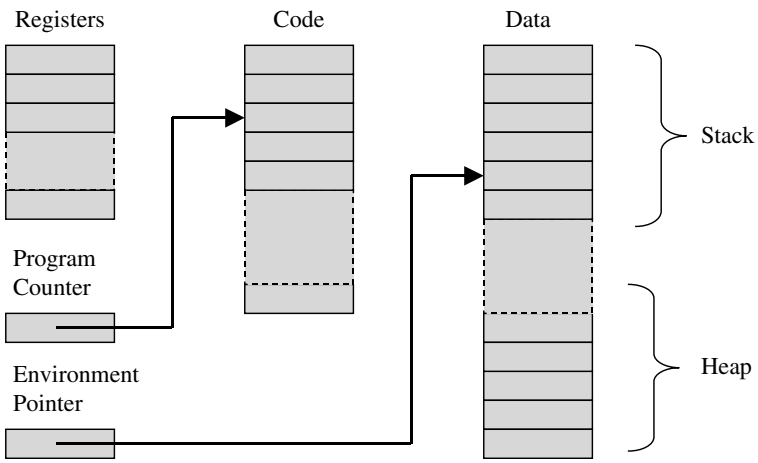


Figure 7.1. Program stack.

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

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

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);
        };
};
```

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

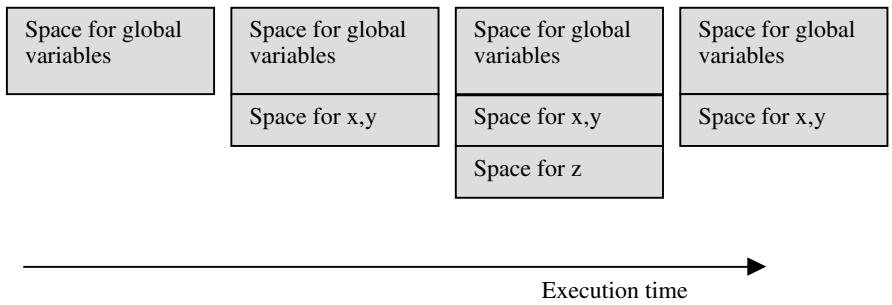


Figure 7.2. Stack grows and shrinks during program execution.

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.

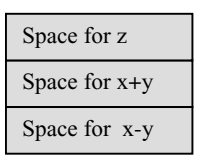
The number of locations that need to be allocated at run time depends on the number of variables declared in the block and their types. Because these quantities are known at compile time, the compiler can determine the format of each activation record and store this information as part of the compiled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



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

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

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:

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

```
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.

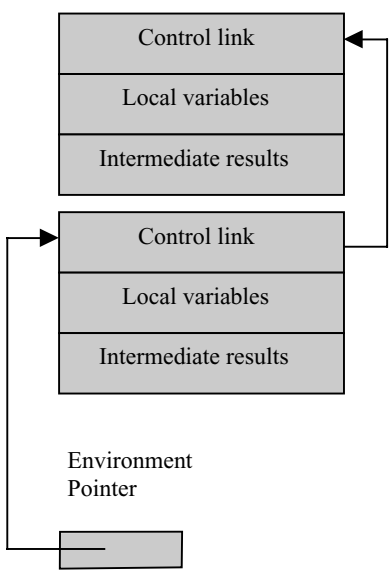


Figure 7.3. Activation records with control links.

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

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.

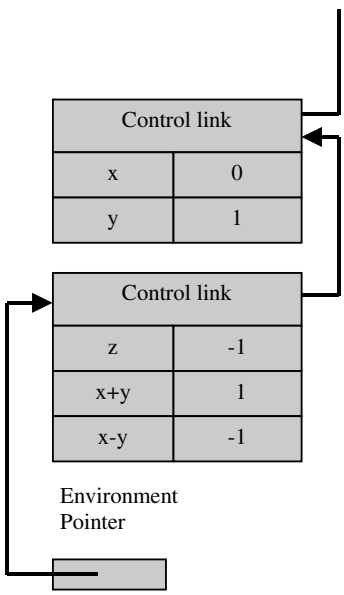
The code for pushing and popping activation records from the stack is generated by the compiler at compile time and becomes part of the compiled code for the program. Because the size of an activation record can be determined from the text of the block, the compiler can generate code for block entry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



https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

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.

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

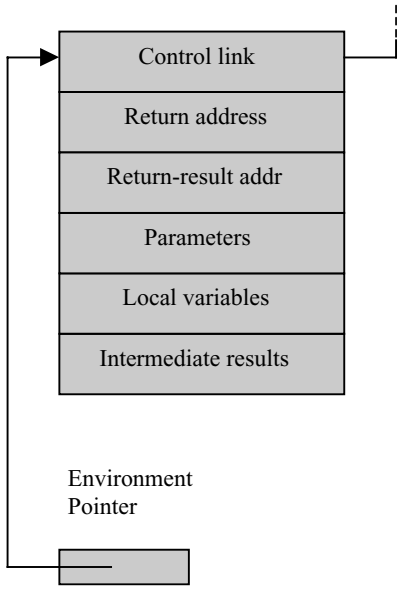


Figure 7.4. Activation record associated with funtion call.

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:

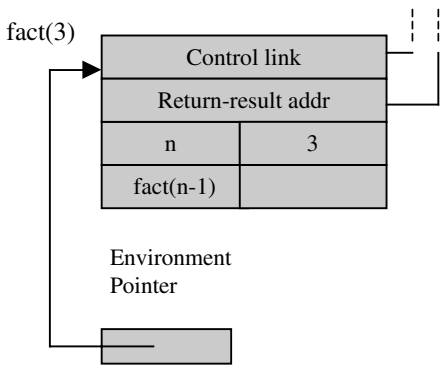
```
fun fact(n) = if n <= 1 then 1 else n * fact(n-1);
```

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

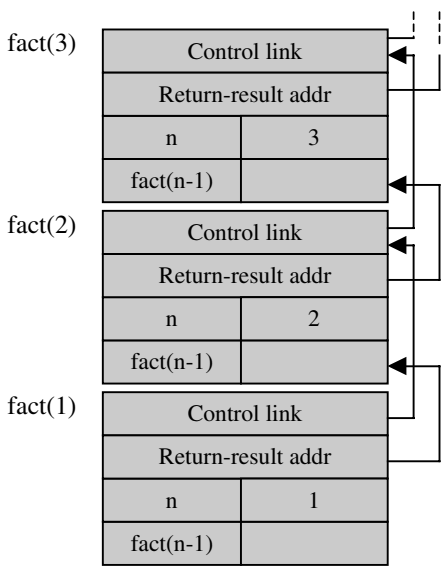
Suppose that some block contains the expression fact(3)+1, leading to a call of fact(3). Let us assume that the activation record of the calling block contains a location that will be used to store the value of fact(3) before computing fact(3)+1.

The next activation record that is placed on the stack will be an activation record for the call fact(3). In this activation record, shown after the list,

- the control link points to the activation record of the block containing the call fact(3),
- the return-result link points to the location in the activation record of the calling block that has been allocated for the intermediate value fact(3) of the calling expression fact(3)+1,
- the actual parameter value 3 is placed in the location allocated for the formal parameter n,
- a location is allocated for the intermediate value fact(n-1) that will be needed when n>0.



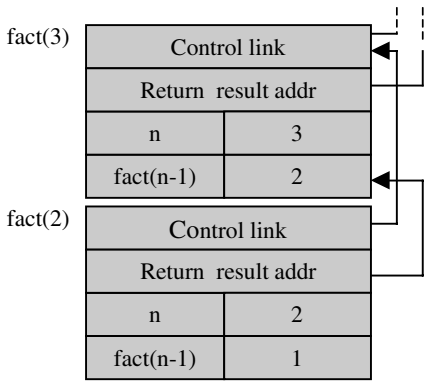
After this activation record is allocated on the stack, the code for factorial is executed. Because n>0, there is a recursive call fact(2). This leads to a recursive call fact(1), which results in a series of activation records, as shown in the subsequent figure.



https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

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

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

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)...;
```

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

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);
```

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

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.

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

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.

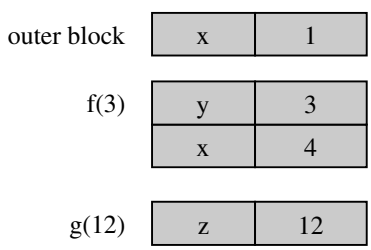
Dynamically Scoped	Statically Scoped
Older Lisps	Newer Lisps, Scheme
TeX/LaTeX document languages	Algol and Pascal
Exceptions in many languages	C
Macros	ML
	Other current languages

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

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

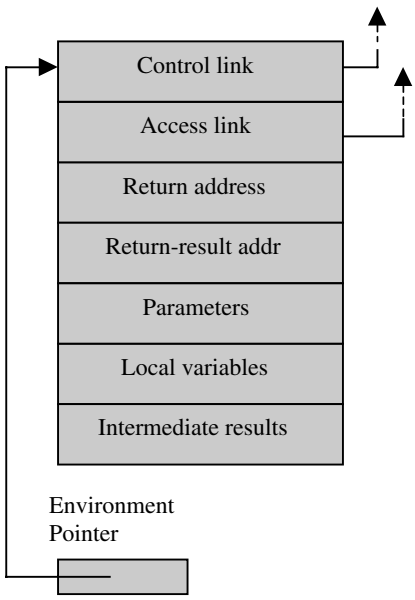


Figure 7.5. Activation record with access link for functions call with static scope.

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

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

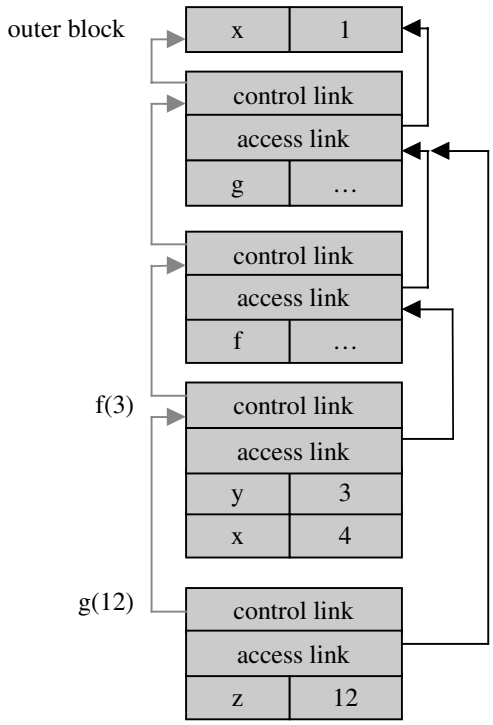


Figure 7.6. Run-time stack after call to g inside f.

this illustration, which follows our convention that we begin a new block for each ML declaration.

- The declaration of g occurs inside the scope of the declaration of x. Therefore, the access link for the declaration of g points to the activation record for the declaration of x.
- The declaration of f is similarly inside the scope of the declaration of g. Therefore, the access link for the declaration of f points to the activation record for the declaration of g.
- The call f(3) causes an activation record to be allocated for the scope associated with the body of f. The body of f occurs inside the scope of the declaration of f. Therefore, the access link for f(3) points to the activation record for the declaration of f.
- The call g(12) similarly causes an activation record to be allocated for the scope associated with the body of g. The body of g occurs inside the scope of the declaration of g. Therefore, the access link for g(12) points to the activation record for the declaration of g.
- We evaluate the expression x+z by adding the value of the parameter z, stored locally in the activation record of g, to the value of the global variable x. We find the location of the global variable x by following the access link of the activation of g to the activation record associated with the declaration of g. We then follow the access link in that activation record to find the activation record containing the variable x.

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

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.

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

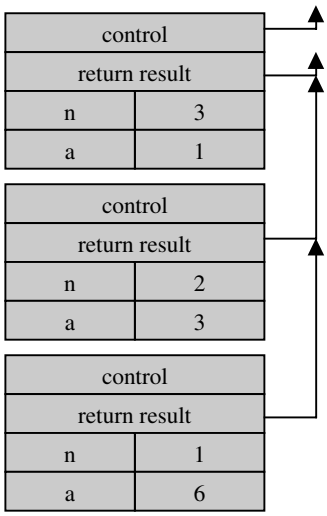


Figure 7.7. Three calls to tail recursive tlfact without optimization.

After the third call terminates, it passes its return result back to the second call, which then passes the return result back to the first call. We can simplify this process by letting the third call return its result to the activation that made the original call, tlfact(3,1). Under this alternative execution, when the third call finishes, we can pop the activation record for the second call as well, as we will no longer need it. In fact, because the activation record for the first call is no longer needed when the second call begins, we can pop the first activation record from the stack before allocating the second. Even better, instead of deallocating the first and then allocating a second activation record identical to the first, we can use one activation record for all three calls.

Figure 7.8 shows how the same activation record can be used used three times for three successive calls to tail recursive tlfact. The figure shows the contents of this activation record for each call. When the first call, tlfact(3,1), begins, an activation record with parameters (1,3) is created. When the second call, tlfact(2,3), begins, the parameter values are changed to (2,3). Tail recursion elimination reuses a single activation record for all recursive calls, using assignment to change the values of the function parameters on each call.

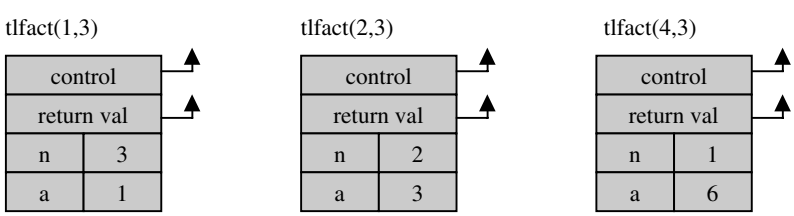


Figure 7.8. Three calls to tail recursive tlfact *with* optimization.

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

Tail Recursion as Iteration

When a tail recursive function is compiled, it is possible to generate code that computes the function value by use of an iterative loop. When this optimization is used, tail recursive functions can be evaluated by only one activation record for a first call and all resulting recursive calls, not one additional activation record per recursive call. For example, the preceding tail recursive factorial function may be compiled into the same code as that of the following iterative function:

```
fun itfact(n,a) = ( while !n > 0 do (a := !n * !a; n := !n - 1 ); !a );
```

An activation record for itfact looks just like an activation record for tlfact . If we look at the values of n and a on each iteration of the loop, we find that they change in exactly the same way is for tail recursive calls to tlfact. The two functions compute the same result by exactly the same sequence of instructions. Put another way, tail recursion elimination compiles tail recursive functions into iterative loops.

7.4 HIGHER-ORDER FUNCTIONS

7.4.1 First-Class Functions

A language has *first-class functions* if functions can be

- declared within any scope,
- passed as arguments to other functions, and
- returned as results of functions.

In a language with first-class functions and static scope, a function value is generally represented by a *closure*, which is a pair consisting of a pointer to function code and a pointer to an activation record.

Here is an example ML function that requires a function argument:

```
fun map (f, nil) = nil
|   map(f, x::xs) = f(x) :: map(f, xs)
```

The map function take a function f and a list m as arguments, applying f to each element of m in turn. The result of map(f, m) is the list of results f(x) for elements x of the list m. This function is useful in many programming situations in which lists are used. For example, if we have a list of expiration times for a sequence of events and we want to increment each expiration time, we can do this by passing an increment function to map.

We will see why closures are necessary by considering interactions between static scoping and function arguments and return values. C and C++ do not support closures because of the implementation costs involved. However, the implementation of objects in C++ and other languages is related to the implementation of function

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

values discussed in this chapter. The reason is that a closure and an object both combine data with code for functions.

Although some C programmers may not have much experience with passing functions as arguments to other functions, there are many situations in which this can be a useful programming method. One recognized use for functions as function arguments comes from an influential software organization concept. In systems programming, the term *upcall* refers to a function call up the stack. In an important paper called "The Structuring of Systems Using Upcalls," (ACM Symp. Operating Systems Principles, 1985) David Clark describes a method for arranging the functions of a system into layers. This method makes it easier to code, modularize, and reason about the system. As in the network protocol stack, higher layers are clients of the services provided by lower layers. In a layered file system, the file hierarchy layer is built on the vnode, which is in turn built over the inode and disk block layers. In Clark's method, which has been widely adopted and used, higher levels pass handler functions into lower levels. These handler functions are called when the lower level needs to notify the higher level of something. These calls to a higher layer are called upcalls. This influential system design method shows the value of language support for passing functions as arguments.

7.4.2 Passing Functions to Functions

We will see that when a function f is passed to a function g, we may need to pass the closure for f, which contains a pointer to its activation record. When f is called within the body of g, the environment pointer of the closure is used to set the access link in the activation record for the call to f correctly. The need for closures in this situation has sometimes been called the *downward funarg problem*, because it results from passing function as arguments downward into nested scopes.

Example 7.8

An example program with two declarations of a variable x and a function f passed to another function g is used to illustrate the main issues:

```
val x = 4;
fun f(y) = x*y;
fun g(h) = let val x=7 in h(3) + x;
g(f);
```

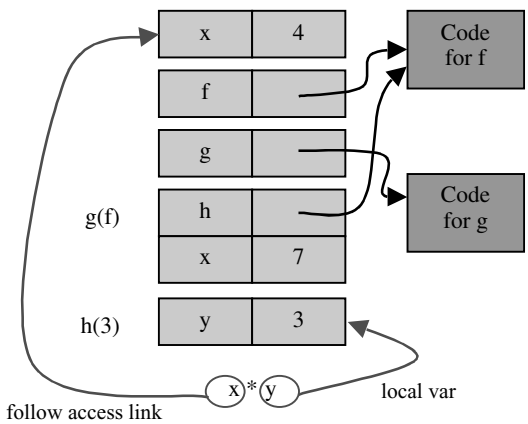
In this program, the body of f contains a global variable x that is declared outside the body of f. When f is called inside g, the value of x must be retrieved from the activation record associated with the outer block. Otherwise the body of f would be evaluated with the local variable x declared inside g, which would violate static scope. Here is the same program written with C-like syntax (except for the type expression int →int) for those who find this easier to read:

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

```
{ int x = 4;
      { int f(int y) {return x*y;}
            { int g(int → int h) {
                  int x=7;
                  return h(3) + x;
                  }
            g(f);
} } }
```

The C-like version of the code reflects a decision, used for simplicity throughout this book, to treat each top-level ML declaration as the beginning of a separate block.

We can see the variable-lookup problem by looking at the run-time stack after the call to f from the invocation of g.



This simplified illustration shows only the data contained in each activation record. In this illustration, the expression x*y from the body of f is shown at the bottom, the activation record associated with the invocation of f (through formal parameter h of g). As the illustration shows, the variable y is local to the function and can therefore be found in the current activation record. The variable x is global, however, and located several activation records above the current one. Because we find global variables by following access links, the access link of the bottom activation record should allow us to reach the activation record at the top of the illustration.

When functions are passed to functions, we must set the access link for the activation record of each function so that we can find the global variables of that function correctly. We cannot solve this problem easily for our example program without extending some run-time data structure in some way.

Use of Closures

The standard solution for maintaining static scope when functions are passed to functions or returned as results is to use a data structure called a closure. A closure is a pair consisting of a pointer to function code and a pointer to an activation record. Because each activation record contains an access link pointing to the record for the

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

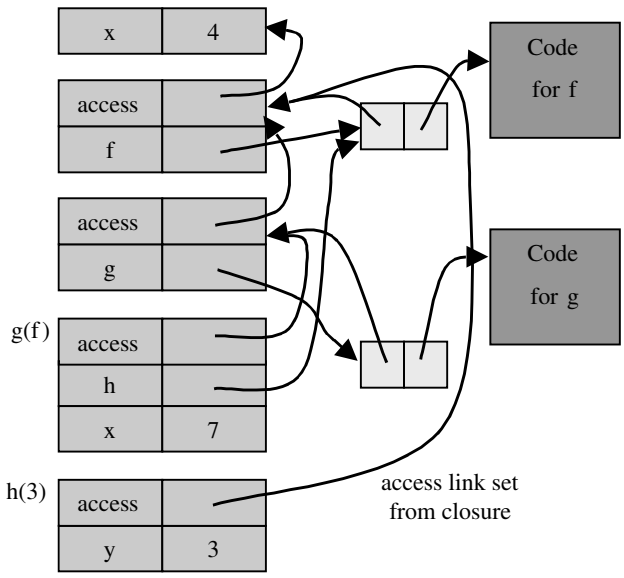


Figure 7.9. Access link set from closure.

closest enclosing scope, a pointer to the scope in which a function is declared also provides links to activation records for enclosing blocks.

When a function is passed to another function, the actual value that is passed is a pointer to a closure. The following steps are used for calling a function, given a closure:

- Allocate an activation record for the function that is called, as usual.
- Set the access link in the activation record by using the activation record pointer from the closure.

We can see how this solves the variable-access problem for functions passed to functions by drawing the activation records on the run-time stack when the program in Figure 7.8 is executed. These are shown in Figure 7.9.

We can understand Figure 7.9 by walking through the sequence of run-time steps that lead to the configuration shown in the figure.

1. *Declaration of* `x`: An activation record for the block where `x` is declared is pushed onto the stack. The activation record contains a value for `x` and a control link (not shown).
2. *Declaration of* `f`: An activation record for the block where `f` is declared is pushed onto the stack. This activation record contains a pointer to the run-time representation of `f`, which is a closure with two pointers. The first pointer in the closure points to the activation record for the static scope of `f`, which is the activation record for the declaration of `f`. The second closure pointer points to the code for `f`, which was produced during compilation and placed at some location that is known to the compiler when code for this program is generated.

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

3. *Declaration of* g: As with the declaration of f, an activation record for the block where g is declared is pushed onto the stack. This activation record contains a pointer to the run-time representation of g, which is a closure.
4. *Call to* g(f): The call causes an activation record for the function g to be allocated on the stack. The size and the layout of this record are determined by the code for g. The access link is set to the activation record for the scope where g is declared; the access link points to the same activation record as the activation record in the closure for g. The activation record contains space for the parameter h and local variable x. Because the actual parameter is the closure for f, the parameter value for h is a pointer to the closure for f. The local variable x has value 7, as given in the source code.
5. *Call to* h(3): The mechanism for executing this call is the main point of this example. Because h is a formal parameter to g, the code for g is compiled without knowledge of where the function h is declared. As a result, the compiler cannot insert any instructions telling how to set the access link for the activation record for the call h(3). However, the use of closures provides a value for the access link – the access link for this activation record is set by the activation record pointer from the closure of h. Because the actual parameter is f, the access link points to the activation record for the scope where f is declared. When the code for f is executed, the access link is used to find x. Specifically, the code will follow the access link up to the second activation record of the illustration, follow one additional access link because the compiler knew when generating code for f that the declaration of x lies one scope above the declaration of f, and find the value 4 for the global x in the body of f.

As described in step 5, the closure for f allows the code executed at run time to find the activation record containing the global declaration of x.

When we can pass functions as arguments, the access links within the stack form a tree. The structure is not linear, as the activation record corresponding to the function call h(3) has to skip the intervening activation record for g(f) to find the necessary x. However, all the access links point up. Therefore, it remains possible to allocate and deallocate activation records by use of a stack (last allocated, first deallocated) discipline.

7.4.3 Returning Functions from Nested Scope

A related but more complex problem is sometimes called the *upward funarg problem*, although it might be more accurate to call it the *upward fun-result problem* because it occurs when returning a function value from a nested scope, generally as the return value of a function.

A simple example of a function that returns a function is this ML code for function composition:

```
fun compose(f,g) = (fn x => g(f x));
```

Given two function arguments f and g, function compose returns the function

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

composition of f and g. The body of compose is code that requires a function parameter x and then computes g(f(x)). This code is useful only if it is associated with some mechanism for finding values of f and g. Therefore, a closure is used to represent compose(f,g). The code pointer of this closure points to compiled code for "get function parameter x and then computes g(f(x))" and the activation record pointer of this closure points to the activation record of the call compose(f,g) because this activation record allows the code to find the actual functions f and g needed to compute their composition.

We will use a slightly more exciting program example to see how closures solve the variable-lookup problem when functions are returned from nested scopes. The following example may give some intuition for the similarity between closures and objects.

Example 7.9

In this example code, a "counter" is a function that has a stored, private integer value. When called with a specific integer increment, the counter increments its internal value and returns the new value. This new value becomes the stored private value for the next call. The followng ML function, make_counter, takes an integer argument and returns a counter initialized to this integer value:

```
fun make_counter (init : int) =
    let val count = ref init    (* private variable count *)
        fun counter(inc:int) = (count := !count + inc; !count)
    in
        counter     (* return function counter from make_counter *)
    end;
val c = make_counter(1); (* c is a new counter *)
c(2) + c(2);             (* call counter c twice *)
```

Function make_counter allocates a local variable count, initialized to the value of the parameter init. Function make_counter then returns a function that, when called, increments count's value by parameter inc, and then returns the new value of count.

The types and values associated with these declarations, as printed by the compiler, are

```
val make_counter = fn : int ⟶ (int ⟶ int)
val c = fn : int ⟶ int
8 : int
```

Here is the same program example written in a C-like notation for those who prefer this syntax:

```
{int⟶ int mk_counter (int init) {
        int count = init;
```

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

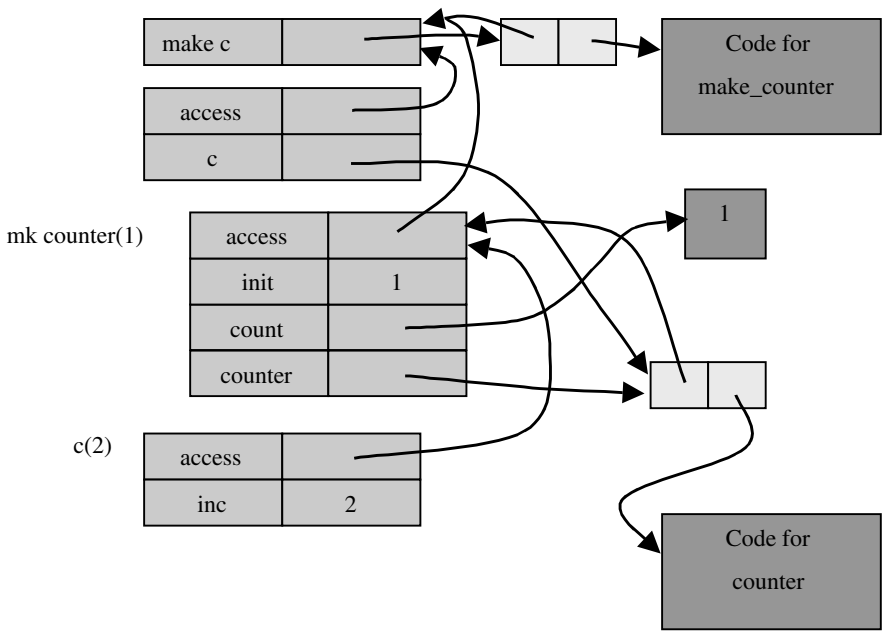


Figure 7.10. Activation records for function closure returned from function.

```
        int counter(int inc) { return count += inc;}
        return counter
    }
    int⟶ int c = mk_counter(1);
    print c(2) + c(2);
}
```

If we trace the storage allocation associated with this compilation and execution, we can see that the stack discipline fails. Specifically, it is necessary to save an activation record that would be popped off the stack if we follow the standard last allocated–first deallocated policy.

Figure 7.10 shows that the records are allocated and deallocated in execution of the code from Example 7.9. Here are the sequence of steps involved in producing the activation records shown in Figure 7.10:

1. *Declaration of* `make_counter`: An activation record for the block consisting of the declaration of function `make_counter` is allocated on the run-time stack. The function name `make_counter` is abbreviated here as `make_c` so that the name fits easily into the figure. The value of `make_counter` is a closure. The activation record pointer of this closure points to the activation record for `make_counter`. The code pointer of this closure points to code for `make_counter` produced at compile time.
2. *Declaration of* `c`: An activation record for the block consisting of the declaration of function `c` is allocated on the run-time stack. The access pointer of this activation record points to the first activation record, as the declaration of

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

make_counter is the previous block. The value of c is a function, represented by a closure. However, the expression defining function c is not a function declaration. It is an expression that requires a call to make_counter. Therefore, after this activation record is set up as the program executes, it is necessary to call make_counter.

3. *Call to* make_counter: An activation record is allocated in order to execute the function make_counter. This activation record contains space for the formal parameter init of make_counter, local variable count and function counter that will be the return result of the call. The code pointer of the closure for counter points to code that was generated and stored at compile time. The activation record pointer points to the third activation record, because the global variables of the function reside here. The program still needs activation record three after the call to make_counter returns because the function counter returned from the call to make_counter refers to variables init and count that are stored in (or reachable through) the activation record for make_counter. If activation record three (used for the call to make_counter) is popped off the stack, the counter function will not work properly.
4. *First call to* c(2): When the first expression c(2) is evaluated, an activation record is created for the function call. This activation record has an access pointer that is set from the closure for c. Because the closure points to activation record three, so does the access pointer for activation record four. This is important because the code for counter refers to variable count, and count is located through the pointer in activation record three.

There are two main points to remember from this example:

- Closures are used to set the access pointer when a function returned from a nested scope is called.
- When functions are returned from nested scopes, activation records do *not* obey a stack discipline. The activation record associated with a function call cannot be deallocated when the function returns if the return value of the function is another function that may require this activation record.

The second point here is illustrated by the preceding example: The activation record for the call to make_counter cannot be deallocated when make_counter returns, as this activation record is needed by the function count returned from make_counter.

Solution to Storage Management Problem

You may have noted that, after the code in Example 7.9 is executed, following the steps just described, we have several activation records left on the stack that might or might not be used in the rest of the program. Specifically, if the function c is called again in another expression, then we will need the activation records that maintain its static scope. However, if the function c is not used again, then we do not need these activation records. How, you may ask, does the compiler or run-time system determine when an activation record can be deallocated?

There are a number of methods for handling this problem. However, a full discussion is complicated and not necessary for understanding the basic language design trade-offs that are the subject of this chapter. One solution that is relatively

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

straightforward and not as inefficient as it sounds is simply to use a garbage-collection algorithm to find activation records that are no longer needed. In this approach, a garbage collector follows pointers to activation records and collects unreachable activation records when there are no more closure pointers pointing to them.

7.5 CHAPTER SUMMARY

A *block* is a region of program text, identified by begin and end markers, that may contain declarations local to this region. Blocks may occur inside other blocks or as the body of a function or procedure. Block-structured languages are implemented by activation records that contain space for local variables and other block-related information. Because the only way for one block to overlap another is for one to contain the other, activation records are generally managed on a run-time stack, with the most recently allocated activation record deallocated first (last allocated–first deallocated).

Parameters passed to functions and procedures are stored in the activation record, just like local variables. The activation record may contain the actual parameter value (in pass-by-value) or its address (in pass-by-reference). Tail calls may be optimized to avoid returning to the calling procedure. In the case of tail recursive functions that do not pass function arguments, this means that the same activation record may be used for all recursive calls, eliminating the need to allocate, initialize, and then deallocate an activation record for each call. The result is that a tail recursive function may be executed with the same efficiency as an iterative loop.

Correct access to statically scoped global variables involves three implementation concepts:

- Activation records of functions and procedures contain *access* (or *static scoping*) *links* that point to the activation record associated with the immediately enclosing block.
- Functions passed as parameters or returned as results must be represented as *closures*, consisting of function code together with a pointer to the correct lexical environment.
- When a function returns a function that relies on variables declared in a nested scope, static scoping requires a deviation from stack discipline: An activation record may need to be retained until the function value (closure) is no longer in use by the program.

Each of the implementation concepts discussed in this chapter may be tied to a specific language feature, as summarized in the following table.

Language Feature	Implementation construct
Block	Activation record with local memory
Nested blocks	Control link
Functions and procedures	Return address, return-result address
Functions with static scope	Access link
Function as arguments and results	Closures
Functions returned from nested scope	Failure of stack deallocation order for activation records

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

EXERCISES

7.1 Activation Records for In-Line Blocks

You are helping a friend debug a C program. The debugger `gdb`, for example, lets you set breakpoints in the program, so the program stops at certain points. When the program stops at a breakpoint, you can examine the values of variables. If you want, you can compile the programs given in this problem and run them under a debugger yourself. However, you should be able to figure out the answers to the questions by thinking about how activation records work.

(a) Your friend comes to you with the following program, which is supposed to calculate the absolute value of a number given by the user:

```
1:    int main( )
2:    {
3:            int num, absVal;
4:
5:            printf("Absolute Value\n");
6:            printf("Please enter a number:");
7:            scanf("%d",&num);
8:
9:             if (num <= 0)
10:           {
11:                   int absVal = num;
12:           }
13:           else
14:           {
15:                   int absVal = -num;
16:           }
17:
18:       printf("The absolute value of %d is %d.\n \n",num,absVal);
19:
20:       return 0;
21:   }
```

Your friend complains that this program just doesn't work and shows you some sample output:

```
cardinal:~> gcc -g test.c
cardinal:~> ./a.out
Absolute Value
Please enter a number: 5
The absolute value of 5 is 4.

cardinal:~> ./a.out
Absolute Value
Please enter a number: -10
The absolute value of -10 is 4.
```

Being a careful student, your friend has also used the debugger to try to track down the problem. Your friend sets a breakpoints at line 11 and at line 18 and looks at the address of absval. Here is the debugger output:

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

```
cardinal:~> gdb a.out
(gdb) break test.c : 11
(gdb) break test.c : 18
(gdb) run
Starting program:
Absolute Value
Please enter a number: 5

Breakpoint 1, main (argc=1, argv=0xffbefb04) at test.c : 11
11              int absVal = num;
(gdb) print &absVal
$1 = (int *) 0xffbefa84
(gdb) c
Continuing.

Breakpoint 2, main (argc=1, argv=0xffbefb04) at test.c : 18
18              printf("The absolute value of %d is %d.\n\n",num absVal);
(gdb) print &absVal
$2 = (int *) 0xffbefa88
```

Your friend swears that the computer must be broken, as it is changing the address of a program variable. Using the information provided by the debugger, explain to your friend what the problem is.

(b) Your explanation must not have been that good, because your friend does not believe you. Your friend brings you another program:

```
1:    int main( )
2:    {
3:            int num;
4:
5:            printf("Absolute Value\n");
6:            printf("Please enter a number:");
7:            scanf("%d",&num);
8:
9:            if (num >= 0)
10:           {
11:                   int absVal = num;
12:           }
13:           else
14:           {
15:                   int absVal = -num;
16:           }
17:
18:           {
19:           int absVal;
20:           printf("The absolute value of %d is %d.\n\n",num, absVal);
21:           }
22:   }
```

This program works. Explain why.

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

(c) Imagine that line 17 of the program in part (b) was split into three lines:

```
17a:        {
17b:        ...
17c:        }
```

Write a single line of code to replace the ... that would guarantee this program would NEVER be right. You may not declare functions or use the word absVal in this line of code.

(d) Explain why the change you made in part (c) breaks the program in part (b).

7.2 Tail Recursion and Iteration

The following recursive function multiplies two integers by repeated addition:

```
fun mult(a,b) =
    if a=0 then 0
      else if a = 1 then b
         else b + mult(a-1,b);
```

This is not tail recursive as written, but it can be transformed into a tail recursive function if a third parameter, representing the result of the computation done so far, is added:

```
fun mult(a,b) =
  let fun mult1(a,b,result) =
        if a=0 then 0
          else if a = 1 then b + result
             else mult1(a-1, b, result+b)
in
  mult1(a, b, 0)
end;
```

Translate this tail recursive definition into an equivalent ML function by using a while loop instead of recursion. An ML while has the usual form, described in Subsection 5.4.5.

7.3 Time and Space Requirements

This question asks you to compare two functions for finding the middle element of a list. (In the case of an even-length list of $2n$ elements, both functions return the n + 1st.) The first uses two local recursive functions, len and get. The len function finds the length of a list and get(n,l) returns the nth element of list l. The second middle function uses a subsidiary function m that recursively traverses two lists, taking two elements off the first list and one off the second until the first list is empty. When this occurs, the first element of the second list is returned:

```
exception Empty;

fun middle1(l) =
    let fun len(nil) = 0
        |   len(x : : l) = 1+len(l)
        and get(n, nil) = raise Empty
        |   get(n, x : : l) = if n=1 then x else get(n-1, l)
    in
```

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

```
        get((len(l) div 2)+1, l)
    end;

fun middle2(l) =
    let fun m(x, nil) = raise Empty
        |   m(nil,x : : l) = x
        |   m([y],x : : l) = x
        |   m(y::(z : : l1),x : : l2) = m(l1, l2)
    in
        m(l, l)
    end;
```

Assume that both are compiled and executed by a compiler that optimizes use of activation records or "stack frames."

(a) Describe the approximate running time and space requirements of `middle1` for a list of length n. Just count the number of calls to each function and the maximum number of activation records that *must* be placed on the stack at any time during the computation.

(b) Describe the approximate running time and space requirements of `middle2` for a list of length n. Just count the number of calls to `m` and the maximum number of activation records that *must* be placed on the stack at any time during the computation.

(c) Would an iterative algorithm with two pointers, one moving down the list twice as fast as the other, be significantly more or less efficient than `middle2`? Explain briefly in one or two sentences.

7.4 Parameter Passing

Consider the following procedure, written in an Algol/Pascal-like notation:

```
proc power(x, y, z : int)
begin
      z := 1
         while y > 0 do
              z := z*x
              y := y-1
         end
end
```

The code that makes up the body of `power` is intended to calculate x^y and place the result in `z`. However, depending on the actual parameters, `power` may not behave correctly for certain combinations of parameter-passing methods. For simplicity, we only consider call-by-value and call-by-reference.

(a) Assume that `a` and `c` are assignable integer variables with distinct L-values. Which parameter-passing methods make $c = a^a$ *after* a call `power(a, a, c)`. You may assume that the R-values of `a` and `c` are nonnegative integers.

(b) Suppose that `a` and `c` are formal parameters to some procedure `P` and that the preceding expression `power(a, a, c)` is evaluated inside the body of `P`. If `a` and `c` are passed to `P` by reference and become aliases, then what parameter-passing

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

method(s) will make c = a^a *after* a call power(a, a, c)? If, after the call, c = a^a, does that mean that power actually performed the correct calculation?

7.5 Aliasing and Static Analysis

The designers of the systems programming language Euclid wanted programs written in Euclid to be easy to verify formally. They based the language on Pascal, changing those those features that made verification complicated. One set of changes eliminates *aliasing*, in which two different names denote the same location. Aliasing can arise in several ways in Pascal. For example, suppose that a program contains the declaration

```
procedure P(var x,y: real);
```

Later in the body of the program the procedure call P(z,z) appears. While executing the body of P for this call, both x and y refer to the same location. In Pascal this call is perfectly legal, but in Euclid, the program is not considered syntactically correct.

(a) Explain briefly why aliasing might make it difficult to verify (reason about) programs.

(b) What problems do you think the designers of Euclid had to face in prohibiting aliasing? Would it be easy to implement a good compile-time test for aliasing?

7.6 Pass-by-Value-Result

In *pass-by-value-result*, also called call-by-value-result and copy-in/copy-out, parameters are passed by value, with an added twist. More specifically, suppose a function f with a pass-by-value-result parameter u is called with actual parameter v. The activation record for f will contain a location for formal parameter u that is initialized to the R-value of v. Within the body of f, the identifier u is treated as an assignable variable. On return from the call to f, the actual parameter v is assigned the R-value of u.

The following pseudo-Algol code illustrates the main properties of pass-by-value-result:

```
var x : integer;
x := 0;
procedure p(value-result y : integer)
    begin
        y := 1;
        x := 0;
    end;
p(x);
```

With pass-by-value-result, the final value of x will be 1: Because y is given a new location distinct from x, the assignment to x does not change the local value of y. When the function returns, the value of y is assigned to the actual parameter x. If the parameter were passed by reference, then x and y would be aliases and the assignment to x would change the value of y to 0. If the parameter were passed by value, the assignment to y in the body of p would not change the global variable x and the final value of x would also be 0.

Translate the preceding program fragment into ML (or pseudo-ML if you prefer) in a way that makes the operations on locations, and the differences between L-values and R-values, explicit. Your solution should have the form

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

```
val x = ref 0;
fun p(y' : int ref) =
    ...;
p(x);
```

Note that, in ML, like C and unlike Algol or Pascal, a function may be called as a procedure.

7.7 Parameter-Passing Comparison

For the following Algol-like program, write the number printed by running the program under each of the listed parameter passing mechanisms. Pass-by-value-result, also sometimes called copy-in/copy-out, is explained in problem 6:

```
begin
    integer i;

    procedure pass ( x, y );
        integer x, y; // types of the formal parameters
        begin
            x := x + 1;
            y := x + 1;
            x := y;
            i := i + 1
        end

    i := 1;
    pass (i, i);
    print    i
end
```

(a) pass-by-value

(b) pass-by-reference

(c) pass-by-value-result

7.8 Static and Dynamic Scope

Consider the following program fragment, written both in ML and in pseudo-C:

```
1     let x = 2 in                        int x = 2; {
2       let val fun f(y) = x + y in         int f (int y) { return x + y; } {
3         let val x = 7 in                    int x = 7; {
4           x +                                 x +
5                f(x)                               f(x);
6         end                                   }
7       end                                   }
8     end;                                  }
```

The C version would be legal in a version of C with nested functions.

(a) Under static scoping, what is the value of x + f(x) in this code? During the execution of this code, the value of x is needed three different times (on lines 2, 4, and 5). For each line where x is used, state what numeric value is used when the value of x is requested and explain why these are the appropriate values under static scoping.

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

(b) Under dynamic scoping, what is the value of x + f(x) in this code? For each line in which x is used, state which value is used for x and explain why these are the appropriate values under dynamic scoping.

7.9 Static Scope in ML

ML uses static scope. Consider two functions of this form:

```
fun f(x) = ...;
fun g(x) = ...f...;
```

If we treat a sequence of declarations as if each begins a new block, static scoping implies that any mention of f in g will always refer to the declaration that precedes it, not to future declarations in the same interactive session.

(a) Explain how ML type inference relies on static scope. Would the current typing algorithm work properly if the language were changed to use dynamic scope?

(b) Suppose that, instead of using the language interactively, we intend to build a "batch" compiler. This compiler should accept any legal interactive session as input. The difference is that the entire program will be read and compiled before printing any type information or output. Will the problem with type inference and dynamic scope become simpler in this context?

7.10 Eval and Scope

Many compilers look at programs and eliminate any unused variables. For example, in the following program, x is unused so it could be eliminated:

```
let x = 5 in f(0) end
```

Some languages, including Lisp and Scheme, have a way to construct and evaluate expressions at run time. Constructing programs at run time is useful in certain kinds of problems, such as symbolic mathematics and genetic algorithms.

The following program evaluates the string bound to s, inside the scope of two declarations:

```
let s = read_text_from_user( ) in
   let x = 5 and y = 3 in eval s end
end
```

If s were bound to 1+x*y then eval would return 16. Assume that eval is a special language feature and not simply a library function.

(a) The "unused variable" optimization and the "eval" construct are not compatible. The identifiers x and y do not appear in the body of the inner let (the part between in and end), yet an optimizing compiler cannot eliminate them because the eval *might* need them. In addition to the values 5 and 3, what information does the language implementation need to store for eval that would not be needed in a language without eval?

(b) A clever compiler might look for eval in the scope of the let. If eval does *not* appear, then it may be safe to perform the optimization. The compiler could eliminate any variables that do not appear in the scope of the let declaration. Does this optimization work in a statically scoped language? Why or why not?

(c) Does the optimization suggested in part (b) work in a dynamically scoped language? Why or why not?

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

7.11 Lambda Calculus and Scope

Consider the following ML expression:

```
fun foo(x : int) =
    let fun bar(f) = fn x => f (f (x))
  in
    bar(fn y => y + x)
end;
```

In β reduction on lambda terms, the function argument is substituted for the formal parameter. In this substitution, it is important to rename bound variables to avoid capture. This question asks about the connection between names of bound variables and static scope. Using a variant of substitution that does not rename bound variables, we can investigate dynamic scoping.

(a) The following lambda term is equivalent to the function foo:

$$\lambda x.((\lambda f.\lambda x.\, f(\, fx))\ (\lambda y.y + x))$$

Use β reduction to reduce this lambda term to normal form.

(b) Using the example reduction from part (a), explain how renaming bound variables provides static scope. In particular, say which preceding variable (x, f, or y) must be renamed and how some specific variable reference is therefore resolved statically.

(c) Under normal ML static scoping, what is the value of the expression foo(3)(2)?

(d) Give a lambda term in normal form that corresponds to the function that the expression in part (a) would define under dynamic scope. Show how you can reduce the expression to get this normal form by *not* renaming bound variables when you perform substitution.

(e) Under dynamic scoping, what is the value of the expression foo(3)(2)?

(f) In the usual statically scoped lambda calculus, α conversion (renaming bound variables) does not change the value of an expression. Use the example expression from part (a) to explain why α conversion may change the value of an expression if variables are dynamically scoped. (This is a general fact about dynamically scoped languages, not a peculiarity of lambda calculus.)

7.12 Function Calls and Memory Management

This question asks about memory management in the evaluation of the following statically scoped ML expression:

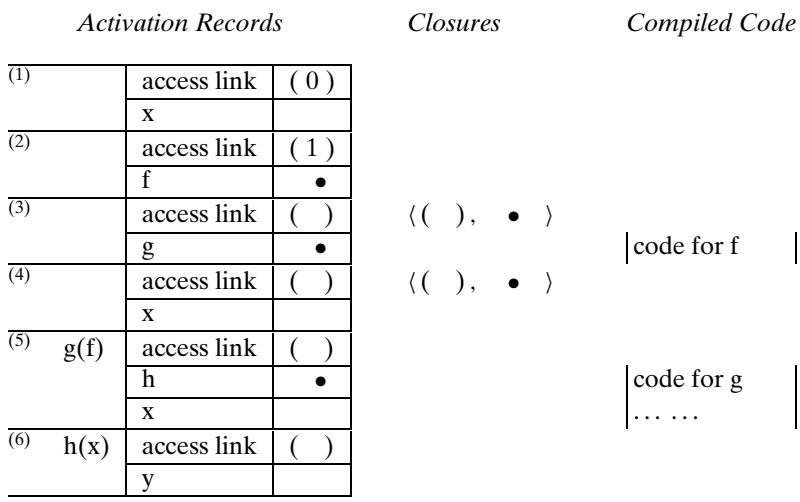
```
val x = 5;
fun f(y) = (x+y) -2;
fun g(h) = let val x = 7 in h(x) end;
let val x = 10 in g(f) end;
```

(a) Fill in the missing information in the following illustration of the run-time stack after the call to h inside the body of g. Remember that function values are represented by closures and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code.

In this figure, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure or compiled code. Because the pointers to activation records cross and could become difficult to read, each activation

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link." The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.



(b) What is the value of this expression? Why?

7.13 Function Returns and Memory Management

This question asks about memory management in the evaluation of the following statically scoped ML expression:

```
val x = 5;
fun f(y) =
    let val z = [1, 2, 3]    (* declare list *)
        fun g(w) = w+x+y (* declare local function *)
    in
        g                    (* return local function *)
    end;
val h = let val x=7 in f(3) end;
h(2);
```

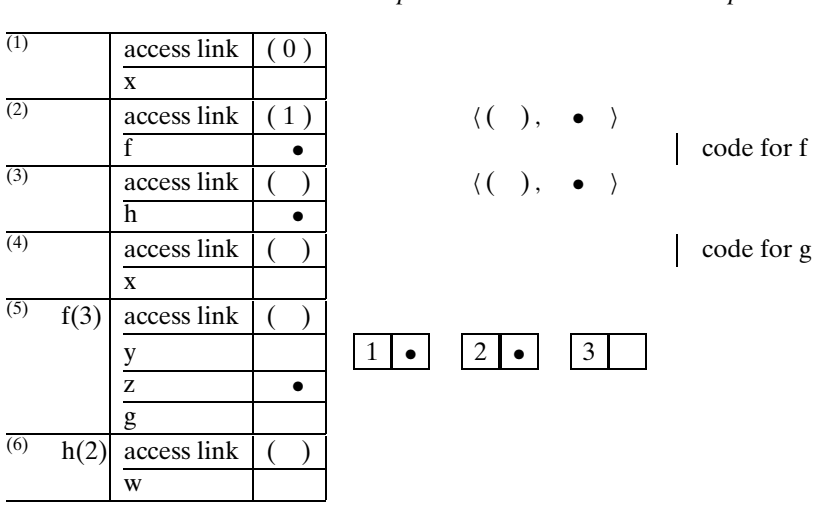
(a) Write the type of each of the declared identifiers (x, f, and h).

(b) Because this code involves a function that returns a function, activation records cannot be deallocated in a last-in/first-out (LIFO) stacklike manner. Instead, let us just assume that activation records will be garbage collected at some point. Under this assumption, the activation record for the call f in the expression for h will still be available when the call h(2) is executed.

Fill in the missing information in the following illustration of the run-time stack after the call to h at the end of this code fragment. Remember that function values are represented by closures and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code.

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

In this figure, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure, compiled code, or list cell. Because the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link." The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.



(c) What is the value of this expression? Explain which numbers are added together and why.

(d) If there is another call to `h` in this program, then the activation record for this closure cannot be garbage collected. Using the definition of garbage given in the Lisp chapter (see Chapter 3), explain why, as long as `h` is reachable, mark-and-sweep will fail to collect some garbage that will never be accessed by the program.

7.14 Recursive Calls and Memory Management

This question asks about memory management in the evaluation of the following ML expression (with line numbers):

```
1>   fun myop(x,y) = x*y;
2>   fun recurse(n) =
3>       if n=0 then 1
4>       else myop(n, recurse(n-1));
5>   let
6>       fun myop(x,y) = x + y
7>   in
8>       recurse(1)
9>   end;
```

(a) Assume that expressions are evaluated by static scope. Fill in the missing information in the following illustration of the run-time stack after the last call to `myop` on line 4 of this code fragment. Remember that function values are

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

represented by closures and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code. Remember also that in ML function arguments are evaluated before the function is called.

In this figure, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure, compiled code, or list cell. Because the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link" and the number of the activation record of the dynamically enclosing scope in the slot labeled "control link." The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters in the activation records. Write the line numbers of code inside the brackets [ ].

Activation Records			*Closures*	*Compiled Code*
(1)	access link	(0)		
	control link	(0)		
	myop	•		
(2)	access link	(1)	⟨ ( ), • ⟩	code for myop defined line [ ]
	control link	(1)		
	recurse	•		
(3)	access link	( )		
	control link	( )	⟨ ( ), • ⟩	code for recurse
	myop	•		
(4) recurse(1)	access link	( )		
	control link	( )		
	n		⟨ ( ), • ⟩	code for myop defined line [ ]
(5) myop(1,0)	access link	( )		
	control link	( )		
	x			
	y			
(6) recurse(0)	access link	( )		
	control link	( )		
	n			

(b) What is the value of this expression under static scope? Briefly explain how your stack diagram from part (a) allows you to find which value of `myop` to use.

(c) What would be the value if dynamic scope were used? Explain how the stack diagram would be different from the one you have drawn in part (a) and briefly explain why.

7.15 Closures

In ANSI C, we can pass and return pointers to functions. Why does the implementation of this language not require closures?

7.16 Closures and Tail Recursion

The function f in this declaration of factorial is formally tail recursive: The only calls in the body of f are tail calls:

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

```
fun fact(n) =
    let f(n, g) = if n=0 then g(1)
                  else let h(i) = g(i*n)
                       in
                           f(n-1, h)
                       end
    in
        f(n, fn x => x)
    end
```

This question asks you to consider the problem of applying the ordinary tail recursion elimination optimization to this function.

(a) Fill in the missing information in the following outline of the activation records resulting from the unoptimized execution of f(2, fn x => x). You may want to draw closures or other data.

f(2, fn x => x)	access link	
	n	
	g	
f(1, h)	access link	
	n	
	g	
f(0, h)	access link	
	n	
	g	

(b) What makes this function more difficult to optimize than the other examples discussed in this chapter? Explain.

7.17 Tail Recursion and Order of Operations

This asks about the order of operations when converting a function to tail recursive form and about passing functions from one scope to another. Here are three versions of our favorite recursive function, *factorial*:

Normal recursive version

$\mathbf{fact}_A\ (n)$ = (if n=0 then 1 else n*fact_A(n-1))

Tail-recursive version

$\mathbf{fact}'(n,a)$ = (if n=0 then a else fact'(n-1,(n*a)))

$\mathbf{fact}_B(n)$ = fact'(n,1)

Another tail-recursive version

$\mathbf{fact}''(n,rest)$ = (if n=0 then rest(1)
else fact''(n-1,(fn(r)⇒ rest(n*r))))

$\mathbf{fact}_C(n)$ = fact''(n,fn(answer)⇒answer)

Here is the evaluation of $\mathbf{fact}_A$ (3) and $\mathbf{fact}_B$ (3):

$\mathbf{fact}_A(3)$ = (if 3=0 then 1 else 3*fact_A(2))
= (if false then 1 else 3*fact_A(2))
= (3*fact(2))
= (3*(if 2=0 then 1 else 2*fact_A(1)))
= (3*(2*fact(1)))

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press

= (3*(2*(if 1=0 then 1 else 1*$\mathbf{fact}_A$(0))))
= (3*(2*(1*fact(0))))
= (3*(2*(1*(if 0=0 then 1 else 0*$\mathbf{fact}_A$(-1)))))
= (3*(2*(1*(1))))

$\mathbf{fact}_B$(3) = fact′(3,1)
= (if 3=0 then 1 else fact′(2,(3*1)))
= (fact′(2,(3*1)))
= (if 2=0 then (3*1) else fact′(1,(2*(3*1))))
= (fact′(1,(2*(3*1))))
= (if 1=0 then (2*(3*1)) else fact′(0,(1*(2*(3*1)))))
= (fact′(0,(1*(2*(3*1)))))
= (if 0=0 then (1*(2*(3*1))) else fact′(-1,(0*(1*(2*(3*1))))))
= (1*(2*(3*1)))

The multiplications are not carried out in these symbolic calculations so that we can compare the order in which the numbers are multiplied. The evaluations show that $\mathbf{fact}_A$ (3) multiplies by 1 first and by 3 last; $\mathbf{fact}_B$ (3) multiplies by 3 first and 1 last. The order of multiplications has changed.

(a) Show that $\mathbf{fact}_C$ (3) multiplies the numbers in the correct order by a symbolic calculation similar to those above that does not carry out the multiplications.

(b) We can see that **fact″** is tail recursive in the sense that there is nothing left to do after the recursive call. If these functions were entered into a compiler that supported tail recursion elimination, would $\mathbf{fact}_C$ (n) be closer in efficiency to $\mathbf{fact}_A$ (n) or $\mathbf{fact}_B$ (n)? First, ignore any overhead associated with creating functions, then discuss the cost of passing functions on the recursive calls.

(c) Do you think it is important to preserve the order of operations when translating an arbitrary function into tail recursive form? Why or why not?

https://doi.org/10.1017/CBO9780511804175.008 Published online by Cambridge University Press