Chapter 18
Parameters

18.1 Introduction
Ever since Chapter 5, the first ML chapter, you have been reading examples and doing exercises that involve calling functions or methods and passing parameters to them. It is now time for a closer look at this familiar operation. Exactly how are parameters passed from caller to callee? This chapter will look at seven different methods and compare their costs and dangers.

First, some basic terminology. Here is a method definition and call in a Java-like language, with the key parts labeled:

```
int plus(int a, int b) {
    return a + b;
}
int x = plus(1, 2);
```
It is important to distinguish between the actual parameters, the parameters passed at the point of call, and the formal parameters, the variables in the called method that correspond to the actual parameters.1

This chapter will use the word method instead of function and will use a Java-like syntax for most of the examples. Real Java cannot illustrate the many different parameter-passing mechanisms used in different languages. In fact, it implements only one of them. So be aware that most of the examples in this chapter are fictitious.

18.2 Correspondence

Before looking at the parameter-passing mechanisms, a preliminary question must be dealt with: how does a language decide which formal parameters go with which actual parameters? In the simplest case, as in ML, Java, and Prolog, the correspondence between actual parameters and formal ones is determined by their positions in the parameter list. Most programming languages use such positional parameters, but some offer additional parameter-passing features. Ada, for example, permits keyword parameters like this:

```
DIVIDE(DIVIDEND => X, DIVISOR => Y);
```

This call to an Ada procedure named DIVIDE passes two actual parameters, X and Y. It matches the actual parameter X to the formal parameter named DIVIDEND and the actual parameter Y to the formal parameter named DIVISOR, regardless of the order in which those formal parameters appear in the definition of DIVIDE. To call a procedure, the programmer does not have to remember the order in which it expects its parameters; instead, the programmer can use the names of the formal parameters to make the correspondence clear. (Of course, an Ada compiler would resolve the correspondence at compile time, so there is no extra runtime cost for using keyword parameters.) Other languages that support keyword parameters include Common Lisp, Dylan, Python, and recent dialects of Fortran. These languages also support positional parameters and allow the two styles to be mixed; the first parameters in a list can be positional, and the remainder can be keyword parameters.

1. Some authors use the word parameter to mean a formal parameter and use the word argument to mean an actual parameter. Many people also use the word parameter informally, referring to either a formal parameter or an actual parameter. This author will always say either formal parameter or actual parameter, thus avoiding any argument.

Another parameter-passing feature offered by some languages is a way to declare optional parameters with default values. The formal parameter list of a function can include default values to be used if the corresponding actual parameters are not given. This gives a very short way of writing certain kinds of overloaded function definitions. For example, consider a C++ definition like this one:

```
int f(int a=1, int b=2, int c=3) {
  function body
}
```

With this definition, the caller can provide zero, one, two, or three actual parameters. The actual parameters that are provided are matched with the formal parameters in order. Any formal parameters that are not matched with an actual parameter are initialized with their default values instead. In effect, C++ treats the definition above like the following overloaded collection of four definitions:

```
int f() {f(1,2,3);}
int f(int a) {f(a,2,3);}
int f(int a, int b) {f(a,b,3);}
int f(int a, int b, int c) {
  function body
}
```

A few languages, including C, C++, and most of the scripting languages like JavaScript, Python, and Perl, allow actual parameter lists of any length. In C, for example, an ellipsis can appear as the last item in a formal parameter list. The printf library function for C, which takes a format string followed by any number of additional parameters, would be declared like this:

```
int printf(char *format, ...) {
  function body
}
```

The function body must use C library routines to access the additional actual parameters. This is a weak spot in C’s static type checking, of course, since the types of the additional parameters cannot be checked statically.

18.3 By Value

The first parameter-passing mechanism this chapter will look at is the most common—passing parameters by value.

For by-value parameter passing, the formal parameter is just like a local variable in the activation record of the called method,
with one important difference: it is initialized using the value of
the corresponding actual parameter, before the called method
begins executing.

The by-value mechanism is the simplest. It is the only one used in real Java. The
actual parameter is used only to initialize the corresponding formal parameter.
After that, the called method can do anything it wants to with the formal para-
meter, and the actual parameter is not affected. For example:

```java
int plus(int a, int b) {
    a += b;
    return a;
}
void f() {
    int x = 3;
    int y = 4;
    int z = plus(x, y);
}
```

In this Java code, when the method f calls the method plus, the values of f's
variables x and y are used to initialize plus's variables a and b. When the plus
method begins executing, the activation records look like this:

```
current activation
record

<table>
<thead>
<tr>
<th>a: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>b: 4</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>previous activation record</td>
</tr>
<tr>
<td>result: ?</td>
</tr>
</tbody>
</table>
```

plus's assignment a += b changes only its own formal parameter a, not the
variable x in f that was the corresponding actual parameter on the call.

When parameters are passed by value, changes to the formal parameter do not
affect the corresponding actual parameter. That does not mean that the called

method is unable to make any changes that are visible to the caller. Consider the
ConsCell class from the previous chapter with this method added:

```java
/*
 * Mutator for the head field of this ConsCell.
 * @param h the new int for our head
 */
public void setHead(int h) {
    head = h;
}
```

The method setHead is a mutator—a method that changes the value of a field.
Now consider this method f:

```java
void f() {
    ConsCell x = new ConsCell(0, null);
    alter(3, x);
}
void alter(int newHead, ConsCell c) {
    c.setHead(newHead);
    c = null;
}
```

As you can see, the method f creates a ConsCell object and passes its reference to
the method alter. When alter begins to execute, the activation records look like
this:

```
current activation
record

<table>
<thead>
<tr>
<th>newHead: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>c:</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>previous activation record</td>
</tr>
</tbody>
</table>
```

As the illustration shows, the formal parameter c is a copy of the actual param-
eter x. This does not mean that there is a copy of the ConsCell object—only that
the reference c is a copy of the reference x, so both refer to the same object. Now
when the first statement of `alter` executes, it calls the `setHead` method of that object. The object's `head` becomes 3. That change is visible to the caller. When `alter` returns, the object to which `f`'s variable `x` refers will have a new `head` value. On the other hand, when the second statement of `alter` executes, it changes `c` to `null`. This has no effect on the object or on the actual parameter `x`. When `alter` is ready to return, this is the situation:

In general, when a Java method receives a parameter of a reference type, any change it makes to the `object` (like `c.setHead(3)`) is visible to the method's caller, while any change it makes to the `reference` (like `c = null`) is purely local. Another language that only has by-value parameters is C. C programmers often use the same kind of trick to get non-local effects. If a C function should be able to change a variable that is visible to the caller, a pointer to that variable is passed.

When a parameter is passed by value, the actual parameter can be any expression that yields a value suitable for initializing the corresponding formal parameter. It need not be a simple variable. It could be a constant (as in `c.setHead(3)`), an arithmetic expression (as in `c.setHead(1+2)`), or the value returned by another method call (as in `c.setHead(x.getHead())`). This may seem obvious, but it is pointed out here because it is not true of the next few parameter-passing methods.

### 18.4 By Result

A parameter that is passed `by result` is, in a way, the exact opposite of a parameter that is passed by value.

For by-result parameter passing, the formal parameter is just like a local variable in the activation record of the called method—it is uninitialized. After the called method finishes executing, the final value of the formal parameter is assigned to the corresponding actual parameter.

Notice that the actual parameter is not evaluated, only assigned to. No information is communicated from the caller to the called method. A by-result parameter works only in the opposite direction, to communicate information from the called method back to the caller. Here is an example, in a Java-like language but with a fictitious `by-result` keyword. (Parameters not otherwise declared are assumed to be passed by value, as in normal Java.)

```java
void plus(int a, int b, by-result int c) {
    c = a + b;
}

void f() {
    int x = 3;
    int y = 4;
    int z;
    plus(x, y, z);
}
```

In this example, the method `f` calls the method `plus`. The third parameter is passed by result. This means that the actual parameter `z` does not need to be initialized before the call, since its value is never called for. When `plus` starts, its formal parameter `c` is uninitialized.
When `plus` is ready to return, its formal parameter `c` has had a value assigned to it. This has had no immediate effect on the corresponding actual parameter `z`:

```
<table>
<thead>
<tr>
<th>current activation record</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 3</td>
</tr>
<tr>
<td>b: 4</td>
</tr>
<tr>
<td>c: 7</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>previous activation record</td>
</tr>
<tr>
<td>x: 3</td>
</tr>
<tr>
<td>y: 4</td>
</tr>
<tr>
<td>z: 7</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>previous activation record</td>
</tr>
</tbody>
</table>
```

Only when `plus` actually returns is the final value of `c` automatically copied to `z`:

```
<table>
<thead>
<tr>
<th>current activation record</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 3</td>
</tr>
<tr>
<td>b: 4</td>
</tr>
<tr>
<td>c: 7</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>previous activation record</td>
</tr>
<tr>
<td>x: 3</td>
</tr>
<tr>
<td>y: 4</td>
</tr>
<tr>
<td>z: 7</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>previous activation record</td>
</tr>
</tbody>
</table>
```

To use parameter passing by result, the actual parameter must be something that can have a value assigned to it; a variable, for example, and not a constant. In fact, the actual parameter must be an expression with an lvalue—something that could appear on the left-hand side of an assignment, as was discussed in Chapter 13.

By-result parameter passing is also sometimes called `copy-out`, for obvious reasons. Relatively few languages support pure by-result parameter passing. Algol W is one. Ada language systems also sometimes use the by-result mechanism.

### 18.5 By Value-Result

You have seen by-value parameters for communicating information from the caller to the called method, and you have seen by-result parameters for communicating information in the opposite direction. What about bidirectional communication? What if you want to pass a value into a method through a parameter, and get a different value out through that same parameter? One way to do this is to pass the parameter by `value-result`, which is a simple combination of by-value and by-result.

If you look at the descriptions of by-value and by-result, you will see that the first describes things that happen before the called method begins executing, while the second describes things that happen after it has finished. If you combine the two, you get value-result:

For passing parameters by value-result, the formal parameter is just like a local variable in the activation record of the called method. It is initialized using the value of the corresponding actual parameter, before the called method begins executing. Then, after the called method finishes executing, the final value of the formal parameter is assigned to the actual parameter.

This method behaves like by-value when the method is called and like by-result when the method returns. Because (like by-result) it assigns a value to the actual parameter, it needs the actual parameter to be an lvalue. For example:

```java
void plus(int a, by-value-result int b) {
    b += a;
}
void f() {
    int x = 3;
    plus(4, x);
}
```

When `plus` is called, but before it begins executing, the activation records look like this:
As the illustration shows, the formal parameter \( b \) has been initialized using the value of the actual parameter \( x \). When \( \text{plus} \) has finished, but not yet returned, its value for \( b \) has changed. But the value of the caller's \( x \) has not yet been changed:

Only when the method actually returns is the final value of the formal parameter copied back to the actual parameter, like this:

Value-result parameter passing is sometimes called \textit{copy-in/copy-out}, for obvious reasons. Ada language systems sometimes use the value-result mechanism.

18.6 By Reference

The three methods of parameter passing seen so far require copying values into and out of the called method's activation record. This can be a problem for languages in which values that take up a lot of space in memory can be passed as parameters. Copying a whole array, string, record, object, or some other large value to pass it as a parameter can be seriously inefficient, both because it slows down the method call and because it fattens up the activation record. This is not a problem in Java, because no primitive-type or reference-type value takes up more than 64 bits of memory. Objects, including arrays, can be large, but they are not passed as parameters—only references are. But languages other than Java sometimes need another method of parameter passing to handle large parameters more efficiently.

One solution is to pass the parameter \textit{by reference}:

For passing parameters by reference, the Ivale of the actual parameter is computed before the called method executes. Inside the called method, that Ivale is used as the Ivale for the corresponding formal parameter. In effect, the formal parameter is an alias for the actual parameter—another name for the same memory location.
Here is an example. It is the same as the example used for value-result parameter passing, except for the (fictional) keyword by-reference to indicate the parameter-passing technique.

```c
void plus(int a, by-reference int b) {
    b += a;
}
void f() {
    int x = 3;
    plus(4, x);
}
```

As in the value-result example, the final value of `x` seen by the method `f` is 7. But the mechanism is quite different. When `plus` is called, but before it begins executing, the activation records look like this:

There is no separate memory location for the value of `b`. The lvalue for `b` (that is, the address where `b`'s value is stored) is the same as the lvalue for `x`, which is indicated with an arrow in the illustration. The effect is that `b` is an alias—just another name for `x`. So when the `plus` method executes the expression `b += a`, the effect is to add 4 to `x`. Unlike value-result, the caller's actual parameter is affected even before the called method returns.

No extra action needs to be taken when the method returns, since the change to `x` has already been made. Notice that no copying of the values of the parameters ever took place. This makes little, if any, difference in efficiency in this example, since copying the value of `x` is probably no more expensive than setting up `b` to be an alias for `x`. But if `x` were some large value, like an array, passing it by reference would be less expensive than the previous parameter-passing methods.

The discussion of by-value parameter passing mentioned a trick that C programmers often use: passing a pointer to a variable rather than the variable itself. By-reference parameter passing is that same trick, really, except that the language system hides most of the details. Although C only has by-value parameter passing, a C program can exhibit the same behavior as the previous example, like this:

```c
void plus(int a, int *b) {
    *b += a;
}
void f() {
    int x = 3;
    plus(4, &x);
}
```

The declaration `int *b` means that `b` is a pointer to an integer. The expression `*b` refers to the integer to which `b` points. The expression `&x` gives a pointer to the integer `x`. An implementation of by-reference parameter passing might well work exactly like this C example. Passing by reference can be implemented simply by passing the actual parameter’s address by value.

By-reference parameter passing is the oldest parameter-passing technique in commercial high-level languages, since it was the only one implemented in early
The aliasing that can occur when parameters are passed by reference can be much more deceptive. Consider this example:

```c
void sigsum(by-reference int n, by-reference int ans) {
    ans = 0;
    int i = 1;
    while (i <= n) ans += i++;
}
```

This `sigsum` method takes two integer parameters, `n` and `ans`, by reference. When it is called properly, it stores the sum of the numbers 1 through `n` in `ans`. For example, this function uses it to compute the sum of the numbers 1 through 10:

```c
int f() {
    int x, y;
    x = 10;
    sigsum(x, y);
    return y;
}
```

The simple aliasing that occurs in this example is not a problem. `sigsum`'s variable `n` aliases `f`'s variable `x`, and `sigsum`'s variable `ans` aliases `f`'s variable `y`. Since the aliases have scopes that do not overlap, there is no danger from this. But consider what happens when `sigsum` is called this way:

```c
int g() {
    int x;
    x = 10;
    sigsum(x, x);
    return x;
}
```

You might expect this function `g` to return the same value as `f`, but it does not. Because the function `g` passes `x` for both parameters, `sigsum`'s variables `n` and `ans` are actually aliases for each other. (That's what makes this kind of aliasing so deceptive. The variables look innocent enough. It might not occur to you that, depending on how the function is called, they can actually have the same lvalue.) This is how the activation records look when `sigsum` is called, before it begins executing:
The first thing `sigsum` does is initialize `ans` to zero. Since `ans` and `n` are aliased, this also sets `n` to zero. That makes the loop guard for `sigsum`'s while loop, `(i<n)`, immediately false, so `sigsum` returns to its caller. This is how the activation records look:

The function `g` returns the value 0, instead of the sum of the numbers 1 through 10.
```c
int main() {
    int temp=1, b=2;
    intswap(temp, b);
    printf("%d, %d\n", temp, b);
}
```

This program prints the string "1, 2", showing that it does not swap the two variables. (See if you can figure out why, before reading on!)

The macro expansion shows why. Before the compiler sees the program, the preprocessing step expands the macro this way:

```c
int main() {
    int temp=1, b=2;
    {int temp= temp ; temp = b ; b =temp;}
    printf("%d, %d\n", temp, b);
}
```

The actual parameter `temp` is evaluated in an environment that has a new, local definition of a variable called `temp`. This is a kind of behavior that programmers usually find surprising. They have the habit of thinking that the names of local variables in the called method are irrelevant to the caller. But macros are not methods, and the names of local variables in a macro body may be critically important to the caller.

This phenomenon has a name—**capture**. In any program fragment, an occurrence of a variable that is not statically bound within the fragment is **free**. For example, in this fragment, the occurrences of `temp` are bound while the occurrences of `a` and `b` are free:

```c
{int temp= a ; a = b ; b =temp;}
```

In the problematic use of the `intswap` macro, `intswap(temp, b)`, the two actual parameters are program fragments with free variables, `temp` and `b`. When these program fragments are substituted into the body of the macro, the free variable `temp` is "captured" by the local definition of `temp` in the macro body. Capture can also occur when the macro body is substituted into the body of the caller. If the macro contains occurrences of a global variable and if the caller has a local definition of the same name, the macro's occurrences will be captured by the caller's definition.

The other parameter-passing methods you have seen can be mixed, and are mixed in some languages. In Pascal, for example, a procedure can take some of its parameters by reference and others by value. But macro expansion is really an all-or-nothing affair. Just substituting the text of an actual parameter for the formal parameter would be nearly useless, since the actual parameter could not then refer

to variables in the caller's context. To pass parameters by macro expansion, you must also substitute the text of the macro body back into the caller's code. When it is implemented by textual substitution like this, a macro does not have an activation record of its own, and generally cannot be recursive.

Macro expansion has been explained in terms of textual substitutions that happen before compilation and execution. This is the easiest way to think about it. But the trick of making textual substitution is just an implementation technique—one that is useful for compiled language systems like C, but not the only one that can be imagined. The important thing about macro expansion is not its implementation, it is its effect. The body of a macro must be evaluated in the caller's context, so that free variables in the macro body can be captured by the caller's definitions. Each actual parameter must be evaluated on every use of the corresponding formal parameter, in the context of that occurrence of the formal parameter, so that free variables in the actual parameter can be captured by the macro body's definitions.

Any implementation that achieves this can be said to pass parameters by macro expansion, even if it does not do textual substitution before compilation.

For passing parameters by macro expansion, the body of the macro is evaluated in the caller's context. Each actual parameter is evaluated on every use of the corresponding formal parameter, in the context of that occurrence of that formal parameter (which is itself in the caller's context).

### 18.8—By Name

The phenomenon of capture is a drawback for macro expansion. One way to eliminate it is to pass parameters by name. In this technique, each actual parameter is evaluated in the caller's context, on every use of the corresponding formal parameter. Macro expansion puts each actual parameter in the context of a use of the formal parameter, and then puts the whole macro body in the caller's context. In this way the actual parameters can get access to the caller's local definitions. But it happens in an indirect way that risks capture. Passing parameters by name skips the middle man; the actual parameter is evaluated directly in the caller's context.

For passing parameters by name, each actual parameter is evaluated in the caller's context, on every use of the corresponding formal parameter.
As with macro expansion, if a formal parameter passed by name is not used in the called method, the corresponding actual parameter is never evaluated.

Passing parameters by name is simple to describe in the abstract, but it is rather difficult to implement. It can be done by macro-style substitution, if the names used in the method body are changed to avoid capture. But this is not efficient enough for practical implementations. In practical implementations, the actual parameter is treated like a little anonymous function. Whenever the called method needs the value of the formal parameter (either its lvalue or its rvalue), it uses that little anonymous function to get it. Here is an example:

```c
void f(by-name int a, by-name int b) {
    b=a;
}
int g() {
    int i = 3;
    f(i+1,i);
    return i;
}
```

This is what the activation records look like when g calls f, before f starts executing:

![Activation Records Diagram]

This illustration shows how the formal parameters a and b are bound to two anonymous functions. The function for a knows how to compute the value for i+1, and the function for b knows how to compute i. These little anonymous functions need the caller’s context to get the variable i from the caller. As was shown in Chapter 12 when passing functions as parameters, the thing that is passed from caller to callee has two parts: the code for the actual parameter and the nesting link to use with it.

When f executes the expression b=a, it calls for the lvalue of b. The anonymous function for b supplies i’s lvalue. The result is that the value 5 is stored in the caller’s variable i. So far this seems to be working like a by-reference parameter—the change to the formal parameter immediately changed the corresponding actual parameter. But there is more to the story. When f executes the expression b=a, it calls again for the lvalue of b (which is recomputed) and for the rvalue of a. The anonymous function for a computes the value i+1 in the caller’s context—using the current value for i—which produces the value 6. This 6 is the value stored back in i and is the value that the function g returns.

Notice the similarities between by-name parameter passing and the anonymous functions experimented with in ML. A method call like f(i+1,i), if it passes parameters by name, is like a shorthand notation for passing two little anonymous functions without parameters. In ML, this might be written as f(fn () => i+1, fn () => i);

As in ML, the functions are passed with nesting links that allow them to access variables in the caller’s context.

By-name parameter passing was introduced in Algol 60 (which also introduced by-value parameter passing). It can be used to do some impressively tricky things, but on the whole it was not a successful invention. It is difficult to implement and can be inefficient. Moreover, most programmers prefer a parameter-passing technique that is easier to understand, even if it is not as flexible. By-name parameter passing is one of the few things introduced in Algol 60 that has not been widely copied in later languages. However, the variation shown next, by-need parameter passing, is used in many functional languages.

---

2. There is a customary name for the little-anonymous-function-and-nesting-link used to implement by-name parameter passing. It is called a thunk.
an actual parameter usually produce the same value as the first evaluation (except for unusual examples like the one above, in which the evaluation of the actual parameter is affected by some intervening side effect). Passing parameters by need eliminates this unnecessary recomputation.

For passing parameters by need, each actual parameter is evaluated in the caller's context, on the first use of the corresponding formal parameter. The value of the actual parameter is then cached, so that subsequent uses of the corresponding formal parameter do not cause reevaluation.

The previous example—the one that demonstrates parameters passed by name—would produce the same result if the parameters were passed by need. Each actual parameter would be evaluated only once. But the parameter \(i+1\) would be evaluated after the value 5 was assigned to \(i\), so the outcome would be the same.

On the other hand, here is an example that shows the difference between by-name and by-need parameters:

```java
void f(by-need int a, by-need int b) {
    b = a;
    b = a;
}
void g() {
    int i = 3;
    f(i, i+1);
    return i;
}
```

When \(f\) is called, its first assignment expression \(b = a\) has the same effect as evaluating \(i = i+1\) in the caller's context; it changes \(g\)'s variable \(i\) to 4. But the second assignment does not reevaluate the actual parameters. It just sets \(i\) to 4 again. By-name parameters would be reevaluated for the second assignment, so \(i\) would end up as 5. As you can see, the difference depends on the side effects. Without side effects, the only way to detect the difference between by-name and by-need parameter passing is by the difference in cost. If a formal parameter is used frequently in the called method and if the corresponding actual parameter is an expensive expression to evaluate, by-name parameters can be much slower than by-need parameters.

All three of the last parameter-passing methods—macro expansion, passing by name, and passing by need—have the property that if the called method does not use a formal parameter, the corresponding actual parameter is never evaluated. Consider this function, which implements the \&\& operator of Java:

```java
boolean andand(by-need boolean a, by-need boolean b) {
    if (a) return false;
    else return b;
}
```

This example short-circuits, just like Java's \&\& and ML's \(\text{andalso}\). If the first parameter is false, Java decides the result is false without ever evaluating the second parameter. This can make a big difference, in more than just efficiency. For example:

```java
boolean g() {
    while (true) {
    }
    return true;
}
void f() {
    andand(false, g());
}
```

When \(f\) calls \(\text{andand}\), it passes the expressions \(false\) and \(g()\) by need. Since the first parameter is false, \(\text{andand}\) never evaluates the second parameter; that is, it never calls \(g\). This is easily observable when the program runs, since the function \(g\) has an infinite loop. If \(f\) did call it, the program would hang. As it is, \(f\) completes normally.

By-need parameter passing is used in so-called lazy functional languages, such as Haskell. Such languages evaluate only as much of the program as necessary to get an answer. Consistent with that philosophy, they only evaluate an actual parameter if the corresponding formal parameter is really used. As in the example above, lazy languages can produce an answer where an eager language hangs or gets an exception.

---

### 18.10 Specification Issues

You have now seen seven different methods of passing parameters. Are these techniques part of the language specification? Does a programmer know and rely on the peculiarities of the parameter-passing technique used by the language system, or are they just hidden system-implementation details? The answer depends on the language.

In languages without side effects, the exact parameter-passing technique is often invisible to the programmer. For functional languages, the big question is whether actual parameters are always evaluated (eager evaluation) or whether they are...
evaluated on, — the corresponding formal parameter is actually used (lazy evaluation). ML uses eager evaluation. It guarantees to evaluate all actual parameters, whether they are used in the called method or not. It follows that an ML language system does not pass parameters by name, by need, or by macro expansion. But it is difficult to distinguish among the other possibilities without side effects, and there are no side effects in the subset of ML seen.

In imperative languages, it is possible to write a program whose behavior differs with each of our seven parameter-passing techniques. In that sense the technique is perfectly visible to the programmer. Nevertheless, some imperative language specifications define parameter passing abstractly, so that the language system is free to use one of several techniques. Ada, for example, has three parameter-passing "modes": in, out, and in out. An in parameter is used to pass values into the called method. The language treats in-mode formal parameters rather like constants in the called method and does not permit assignments to them. An out parameter is used to pass values out of the called method. The language requires assignment to out-mode formal parameters in the called method, and does not permit their values to be read. An in out parameter is used for two-way communication. It may be both read and assigned within the called method. None of this specifies whether the formal parameters are copies of the actual parameters or references to the actual parameters. For scalar values, the Ada standard specifies copying. But for aggregates like arrays and records it permits implementations to go either way. For example, an implementation might pass in out parameters by reference or by value-result. A program that can tell the difference (like some of the examples in this chapter) is simply not considered to be a valid Ada program.

This exemplifies the abstract definition of parameter passing. In this approach, the parameter-passing techniques in this chapter are all considered to be implementation details—they belong in the language system, not in the definition of the language itself. The language definition specifies parameter passing abstractly, not saying how parameters should be implemented, but only how they should be used. Any program that tries to make use of implementation-specific, parameter-passing properties not guaranteed by the language deserves what it gets.

### 18.11 Conclusion

The parameter-passing techniques described in this chapter are among the most commonly used, but they are certainly not the only techniques that have been tried. In particular, Prolog handles parameters in a completely different way, as you will see starting in the next chapter. Further, there are many minor variations on the