Chapter 7
Control Abstraction

The concept of abstraction is a recurrent theme in this text. Right from the first chapter, we have encountered *abstract* machines and their hierarchies. For them, we used the terms "abstract" rather than "physical", denoting by abstract machine, a set of algorithms and data structures not directly present in any specific physical machine, but executable on it by means of interpretation. Moreover, the fact that the concept of abstract machine, to some extent, hides the underlying machine is fundamental.

"To abstract" means simply to hide something. Often, abstracting from some concrete data relating to some object, one succeeds in bringing out with more clarity a concept common to that object. Each description of a phenomenon (natural, artificial, physical, etc.) is not based on the set of *all* data relating to the phenomenon itself, otherwise it would be like a geographical map of scale 1:1, extremely precise but useless. Every scientific discipline describes a certain phenomenon, concentrating only on some aspects, those which have been found to be the most relevant to the agreed aims. It is for this reason that scientific language uses appropriate mechanisms to express these "abstractions". Programming languages, themselves the abstractions of the physical machine, are no exception. Rather, expressiveness depends in an essential way on the mechanisms of abstraction which the languages provide. These mechanisms are the principal instruments available to the designer and programmer for describing in an accurate, but also simple and suggestive, way the complexity of the problems to be solved.

In a programming language, in general, two classes of abstraction mechanisms are distinguished. That which provides *control abstraction* and that which provides *data abstraction*. The former provides the programmer the ability to hide procedural data; the latter allow the definition and use of sophisticated data types without referring to how such types will be implemented. In this chapter we will be concerned with the mechanisms for control abstraction, while data abstraction will be the subject of Chap. 9, after we have seen mechanisms for data structuring in the next chapter.

M. Gabbrielli, S. Martini, *Programming Languages: Principles and Paradigms,*
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-84882-914-5_7, © Springer-Verlag London Limited 2010

```
int foo (int n, int a) {
   int tmp=a;
   if (tmp==0) return n;
   else return n+1;
}
...
int x;
x = foo(3,0);
x = foo(x+1,1);
```

Fig. 7.1 Definition and use of a function

7.1 Subprograms

Every program of any complexity is composed of many components, each of which serves to provide part of a global solution. The decomposition of a problem into subproblems allows better management of complexity. A more restrictive problem is easier to solve. The solution to the global problem is obtained by appropriate composition of the solutions to these subproblems.

In order for all of this to be efficient, however, it is necessary that the programming language provides linguistic support which facilitates and makes possible such subdivisions and, therefore, re-composition. This linguistic support allows the expression of decomposition and re-composition directly in the language, transforming these concepts from simple methodological suggestions into real and genuine instruments for design and programming.

The key concept provided by all modern languages is that of the subprogram, procedure, or function.[1] A *function* is a piece of code identified by name, is given a local environment of its own and is able to exchange information with the rest of the code using *parameters*. This concept translates into two different linguistic mechanisms. The first, *definition* (or declaration) of function, and its *use* (or call). In Fig. 7.1, the first five lines constitute the definition of the function named `foo`, whose local environment is composed from three names `n`, `a` and `tmp`.[2] The first line is the *header*, while the remaining lines constitute the *body* of the function. The last two lines of Fig. 7.1 are the uses (or calls) of `foo`.

A function exchanges information with the rest of the program using three principal mechanisms: parameters, return value, nonlocal environment.

Parameters We distinguish between *formal parameters*, which appear in the definition of a function, and *actual parameters*, which appear, instead, in the call. The formal parameters are always names which, as far as the environment is concerned, behave as declarations local to the function itself. They behave, in particular, as *bound variables*, in the sense that their consistent renaming has no effect on the

[1] These three terms assume different meanings in different languages. For example, "subprogram" is usually the most general term. In languages of the Algol family and their descendants, a procedure is a subprogram which modifies the state, while a function is a subprogram that returns a value. In this chapter, at least, we will use the three terms as synonyms.

[2] The name `foo` is part of the nonlocal environment of the function.

Fig. 7.2 Renaming of formal parameters

```
int foo (int m, int b){
   int tmp=b;
   if (tmp==0) return m;
   else return m+1;
}
```

semantics of the function. For example, the function `foo` in Fig. 7.1 and that in Fig. 7.2 are indistinguishable, even though the second has different names for its formal parameters.

The number and type of actual and formal parameters must, in general, coincide, although many type compatibility rules can be brought into play (see Sect. 8.7). There is sometimes also the possibility of declaring functions with a variable number of parameters.

Return value Some functions exchange information with the rest of the program by returning a value as a result of the function itself, as well as through the use of parameters. Our function, `foo`, for example, returns an integer. The language makes available, in this case, the mechanism which allows the "return of value" to be expressed (for example the `return` construct which has also the effect of terminating the execution of the current function). In some languages, the name "function" is reserved for subprograms which return a value, while those subprograms which interact with the caller just via parameters or the non-local environment are called "procedures".

In languages which derive their syntax from C, all subprograms are, linguistically, functions. If the result type of a function is `void`, the function returns no meaningful value (the command to return such a value and to terminate execution is `return`).

Nonlocal environment This is a less sophisticated mechanism with which a function can exchange information with the rest of the program. If the body of the function modifies a nonlocal variable, it is clear that such a modification is felt in all parts of the program where this variable is visible.

7.1.1 Functional Abstraction

From a pragmatic viewpoint, subprograms are mechanisms which allow the software designer to obtain *functional abstraction*. A software component is an entity which provides services to its environment. The clients of such a component are not interested *how* the services are provided, only how to use them. The possibility of associating a function with every component allows separation of what the client needs to know (expressed in the header of the function: its name, its parameters, its result type, if present) from what it does not need to know (which is in the body). We have real functional abstraction when the client does not depend on the body

"Static" variables

In all we have said, we have always assumed that the local environment of a function has the same lifetime as the function itself. In such a case, there is no primitive mechanism which one instance of a function can use for communicating information to another instance of the *same* function. The only way for this to happen is through the use of a nonlocal variable.

In some languages, though, it is possible to arrange for that a variable (which must be local to a function) to maintain its value between one invocation of the function and another. In C, for example, this is achieved using the `static` modifier (Fortran uses `save`, Algol `own`, etc.). A `static` variable allows programmers to write functions with memory. The following function, for example, returns the number of times it has been called:

```
int how_many_times(){
   static int count;
          /* C guarantees that static variables are
             initialised to zero by the first
             activation of the function */
   return count++;
}
```

The declaration of a `static` variable introduces an association with unlimited lifetime into the environment (in the lifetime of the program, clearly).

It should be observed that a `static` variable provides greater data abstraction than a global variable. It is not, in fact, visible from outside the function. The visibility mechanisms guarantee, therefore, that it is only modifiable inside the function body.

of a function, only on its header. In this case, the substitution (for example for efficiency reasons) of the body by another one with the same semantics is transparent to the system software in its entirety. If a system is based on functional abstraction, the three acts of specification, implementation and use of a function can occur independent of one another without knowledge of the context in which the other actions are performed.

Functional abstraction is a methodological principle, whose functions provide linguistic support. It is clear that this is not a definitive support. It is necessary that the programmer correctly uses these functions, for example by limiting the interactions between function and call to parameter passing, because use of the nonlocal environment to exchange information between functions and the rest of the program destroys functional abstraction. On the other hand, functional abstraction is increasingly guaranteed by greater limitation of interaction between components to external behaviour, as expressed by function headers.

7.1.2 Parameter Passing

The way in which actual parameters are paired with formal parameters, and the semantics which results from this, is called the *parameter passing discipline*. According to what is now traditional terminology, a specific mode is composed of the kind of communication that it supports, together with the implementation that produces this form of communication. The mode is fixed when the function is defined and can be different for each parameter; it is fixed for all calls of the function.

From a strictly semantic viewpoint, the classification of the type of communication permitted by a parameter is simple. From a subprogram's viewpoint, three parameter classes can be discerned:

- Input parameters.
- Output parameters.
- Input/output parameters.

A parameter is of input type if it allows communication which is only in the direction from the caller to the function (the "callee"). It is of output type if it permits communication only in direction from the callee to the caller. Finally, it is input/output when it permits bidirectional communication.

Note that this is a linguistic classification, part of the definition of the language; it is not derived from the use to which parameters are put. An input/output parameter remains that way even if it is used only in a unidirectional fashion (e.g., from caller to callee).

It is clear that each of these types of communications can be obtained in different ways. The specific implementation technique constitutes, exactly, the "call mode", which we will now subject to analysis, describing for each mode:

- What type of communication it allows.
- What form of actual parameter is permitted.
- The semantics of the mode.
- The canonical implementation.
- Its cost.

Of the modes that we will discuss, the first two (by value and by reference) are the most important and are widely used. The others are little more than variations on the theme of call by value. An exception is call by name, which we will discuss last. Although call by name is now disused as a parameter-passing mechanism, nevertheless, it allows us to present a simple case of what it means to "pass an environment" into a procedure.

Call by value Call by value is a mode that corresponds to an input parameter. The local environment of the procedure is extended with an association between the formal parameter and a new variable. The actual parameter can be an expression. When called, the actual parameter is evaluated and its r-value obtained and associated with the formal parameter. On termination of the procedure, the formal parameter is destroyed, as is the local environment of the procedure itself. During the execution of

Fig. 7.3 Passing by value

```
int y = 1;
void foo (int x) {
   x = x+1;
   }
...
y = 1;
foo(y+1);
   // here y =  1
```

the body, there is no link between the formal and the actual parameter. There is no way of make use of a value parameter to transfer information from the callee to the caller.

Figure 7.3 shows a simple example of passing by value. Like in C, C++, Pascal and Java, when we do not explicitly indicate any parameter-passing method for a formal parameter, it is to be understood that parameter is to be passed by value. The variable `y` never changes its value (it always remains 1). During the execution of `foo`, `x` assumes the initial value 2 by the effect of passing the parameter. It is then incremented to 3, finally it is destroyed with the entire activation record for `foo`.

If we assume a stack-based allocation scheme, the formal parameter corresponds to a location in the procedure's activation record in which the value of the actual parameter is stored during the calling sequence of the procedure.

Let us note how this is an expensive method when the value parameter is bound to a large data structure. In such a case, the entire structure is copied to the formal.[3] On the other hand, the cost of accessing the formal parameter is minimal, since it is the same as the cost of accessing a local variable in the body.

Passing by value is a very simple mechanism with clear semantics. It is the default mechanism in many languages (e.g., Pascal) and is the only way to pass parameters in C and Java.

Call by reference Call by reference (also called *by variable*) implements a mechanism in which the parameter can be used for both input and output. The actual parameter *must* be an expression with l-value (recall the definition of l-value on page 132). At the point of call, the l-value of the actual parameter is evaluated and the procedure's local environment is extended with an association between the formal parameter and the actual parameter's l-value (therefore creating an *aliasing* situation). The most common case is that in which the actual parameter is a variable. In this case, the formal and the actual are two names for the same variable. At the end of the procedure, the connection between the formal parameter and the actual parameter's l-value is destroyed, as is the environment local to the procedure. It is clear that call by reference allows bidirectional communication: each modification of the formal parameter is a modification of the actual parameter.

[3] The reader who knows C should not be misled. In a language with pointers, as we will discuss below, often the passing of a complex structure consists of passing (by value) a pointer to the data structure. In such a case, it is the pointer that is copied, not the data structure.

Fig. 7.4 Passage by reference

```
int y = 0;
void foo (reference int x) {
   x = x+1;
   }
y=0;
foo(y);
   // here y = 1
```

Fig. 7.5 Another example of passage by reference

```
int[] V = new V[10];
int i=0;
void foo (reference int x) {
   x = x+1;
   }
...
V[1] = 1;
foo(V[i+1]);
   // here V[1] = 2
```

Figure 7.4 shows a simple example of call by reference (which we have notated in the pseudocode with the `reference` modifier). During the execution of `foo`, `x` is a name for `y`. Incrementing `x` in the body is, to all effects, the incrementing of `y`. After the call, the value of `y` is therefore 1.

It can be seen that, as shown in Fig. 7.5, the actual parameter need not necessarily be a variable but can be an expression whose l-value is determined at call time. In a way similar to the first case, during the execution of `foo`, `x` is a name for `v[1]` and the increment of `x` in the body, is an increment of `v[1]`. After the call, the value of `v[1]` is, therefore, 2.

In the stack-based abstract machine model, each formal is associated with a location in the procedure's activation record. During the calling sequence, the l-value of the actual is stored in the activation record. Access to the formal parameter occurs via an indirection which uses this location.

This is a parameter passing mode of very low cost. At the point of call, only an address need be stored; every reference to the formal is achieved by an indirect access (implemented implicitly by the abstract machine) which can be implemented at very low cost on many architectures.

Call by reference is a low-level operation. It is possible in Pascal (`var` modifier) and in many other languages. It has been excluded from more modern languages. In these languages, however, some form of bidirectional communication between caller and callee can be obtained by exploiting the interaction between parameter passing and other mechanisms (the most important being the model chosen for variables) or the availability of pointers in the language. Two boxes show simple examples in C (and C++) and Java. The moral of the examples is that in an imperative language, passing by value is always accompanied by other mechanisms so that procedures are really a versatile programming technique.

Call by constant We have already seen how call by value is expensive for large-sized data. When, however, the formal parameter is not modified in the body of

Call by reference in C

C admits only call by value but also allows the free manipulation of pointers and addresses. Making use of this fact, it is not difficult to simulate call by reference. Let us consider the problem of writing a simple function which swaps the values of two integer variables which are passed to the function as parameters. With only call by value, there is no way to do this. We can, though, combine call by value with pointer manipulation, as in the following example:

```
void swap (int *a, int *b) {
   int tmp = *a; *a=*b; *b=tmp;
   }
int v1 = ...;
int v2 = ...;
swap(&v1, &v2);
```

The formal parameters to `swap` (both by value) are of type pointer to integer (`int *`). The values of the actual parameters are the addresses (that is, the l-value) of `v1` and `v2` (obtained using the operator `&`). In the body of `swap`, the use of the `*` operator performs dereferencing of the parameters. For example, we can paraphrase the meaning of `*a = *b` as: take the value contained in the location whose address is contained in `b` and store it in the location whose address is stored in `a`. Our swap therefore simulates call-by-reference.

the function, we can imagine maintaining the semantics of passing by value, implementing it using call by reference. This is what constitutes the *read-only* parameter method.

This is a method that establishes an input communication and in which arbitrary expressions are permitted as actual parameters. The formal parameters passed by this method are subject to the static constraint that they cannot be modified in the body, either directly (by assignment) or indirectly (via calls to functions that modify them). From a semantic viewpoint, call by constant coincides completely with call by value, while the choice of implementation is left to the abstract machine. For data of small sizes, call by constant can be implemented as call by value. For larger structures, an implementation using a reference, without copy, will be chosen.

Call by constant is an optimum way to "annotate" a given parameter to a procedure. By reading the header, one immediately has information about the fact that this parameter is input only. Furthermore, one can count on static semantic analysis to verify that this annotation is really guaranteed.

Call by result Call by result is the exact dual of call by value. This a mode which implements *output-only* communication. The procedure's local environment is extended with an association between the formal parameter and a new variable. The actual parameter must be an expression that evaluates to an l-value. When the procedure terminates (normally), immediately before the destruction of the local environ-

Bidirectional communication in Java

A function like `swap` (in the box "Call by reference in C") cannot be written in Java. However, we can make use of the fact that Java uses a reference-based model for variables (of class type) to obtain some form of bidirectional communication. Let us consider for example the following simple definition of a class:

```
class A {
   int v;
   }
```

We can certainly write a method which swaps the values of the field `v` in two objects of class `A`:

```
void swap (A a, A b) {
   int tmp = a.v; a.v= b.v; b.v=tmp;
   }
```

In this case, what is passed (by value) to `swap` are two references to objects of class `A`. Using the reference model for variables of class type, `swap` effectively swaps the values of the two fields. It can be seen, however, that a true simulation of call by reference is not possible, as it was with C.

ment, the current value of the formal parameter is assigned to the location obtained using the l-value from the actual parameter. It should be clear that, as in call by value, the following questions about evaluation order must be answered. If there is more than one result parameter, in which order (for example, from left to right) should the corresponding "backward assignment" from the actual to formal be performed? Finally, when is the actual parameter's l-value determined? It is reasonable to determine it both when the function is called and when it terminates.[4]

It can be seen that, during the execution of the body, there is no link between the formal and actual parameter. There is no way to make use of a result parameter to transfer information from the caller to the callee. An example of call by result is shown in Fig. 7.6. The implementation of call by result is analogous to call by value, with which it shares its advantages and disadvantages. From a pragmatic viewpoint, the by-result mode simplifies the design of functions which must return (that is provide as result) more than one value, each in a different variable.

Call by value-result The combination of call by value and call by result produces a method called call by value-result. This is a method that implements bidirectional communication using the formal parameter as a variable local to the procedure.

[4]Construct an example which gives different results if the l-value of the actual is determined at call time or when the procedure terminates.

Fig. 7.6 Call by result

```
void foo (result int x) {x = 8;}
...
int y = 1;
foo(y);
   // here y is 8
```

Fig. 7.7 Call by value-result

```
 void foo (valueresult int x){
    x = x+1;
 }
 ...
 y = 8;
 foo(y);
    // here y is 9
```

Fig. 7.8 Call by value-result is not call by reference

```
void foo (reference/valueresult int x,
          reference/valueresult int y,
          reference int z){
    y = 2;
    x = 4;
    if (x == y) z = 1;
}
...
int a = 3;
int b = 0;
foo(a,a,b);
```

The actual parameter must be an expression that can yield an l-value. At the call, the actual parameter is evaluated and the r-value thus obtained is assigned to the formal parameter. At the end of the procedure, immediately before the destruction of the local environment, the current value of the formal parameter is assigned to the location corresponding to the actual parameter. An example of call by value-result is shown in Fig. 7.7.

The canonical implementation of call by value-result is analogous to that of call by value, even if some languages (Ada, for example) choose to implement it as call by reference in the case of large-sized data, so that the problems of cost of call by value can be avoided.

The implementation of call by value-result using a reference is, however, not semantically correct. Consider, for a moment, the fragment in Fig. 7.8. At first sight, the conditional command present in the body of `foo` seems useless, for `x` and `y` have just received distinct values. The reality is that `x` and `y` have different values only *in the absence of aliasing*. If, on the other hand, `x` and `y` are two different names for the same variable, it is clear that the condition `x == y` is always true.

If, therefore, `x` and `y` are passed by value-result (there is no aliasing), the call `foo(a,a,b)` terminates without the value of `b` being modified. If, on the other hand, `x` and `y` are passed by reference (where there *is* aliasing), `foo(a,a,b)` terminates by assigning the value 1 to `b`.

```
int x=0;
int foo (name int y){
   int x = 2;
   return x + y;
}
...
int a = foo(x+1);
```

Fig. 7.9 Which environment should be used to evaluate `x+1` in the body of `foo`?

Call by name Call by name, introduced in ALGOL60, is a semantically cleaner way of passing parameters than by reference. It is no longer used by any major imperative language. However, it is a conceptually important method worth the effort of studying in detail because of its properties and its implementation.

The problem that the designers of ALGOL60 set themselves was to give a *precise* semantics. The semantics should specify in an elementary fashion what the effect of a call to a function with specified parameters would be. The solution that they chose was to define the semantics of function call using the so-called *copy rule*. Without loss of generality, we will state it in the case of a function of one argument:

> *Let* `f` *be a function with a single formal parameter,* `x`*, and let* `a` *be an expression of a type that is compatible with that of* `x`*. A call to* `f` *with an actual parameter* `a` *is semantically equivalent to the execution of the body of* `f` *in which all occurrences of the formal parameter,* `x`*, have been replaced by* `a`*.*

As can easily be seen, it is a very simple rule. It reduces the semantics of function call to the syntactic operation of expanding the body of the function after a textual substitution of the actual for the formal parameter. This notion of substitution, however, is not a simple concept because it must take into account the fact it might have to deal with several different variables with the same name. Consider, for example, the function in Fig. 7.9. If we blindly apply the copy rule, the call `foo(x+1)` results in the execution of `return x+x+1`. This command, executed in the local environment of `foo`, returns the value 5. But it is clear that this is an incorrect application of the copy rule because it makes the result of the function depend on what the name of the local variable is. If the body of `foo` had been:

```
{int z=2; return z+y;}
```

the same call would result in the execution of `return z+x+1`, with the result 3.

In the first substitution that we suggested, we say that the actual parameter, `x`, was *captured* by the local declaration. The substitution of which the copy rule talks must therefore be a substitution that *does not capture variables*. It is not possible to avoid having different variables with the same name, so we can obtain a non-capturing substitution by requiring that the formal parameter, even after substitution, is evaluated in the environment of the caller and not in that of the callee.

We can therefore define call by name as that method whose semantics is given by the copy rule, where the concept of substitution is always understood without capture. Equivalently, we can say that what is substituted is not merely the actual parameter, but the actual parameter *together with* its own evaluation environment which is fixed at the moment of call.

Fig. 7.10 Side effects of call by name

```
int i = 2;
int fie (name int y) {
   return y+y;
}
...
int a = fie(i++);
   // here  i has value 4; a has value 5
```

```
void fiefoo (valueresult/name int x, valueresult/name int y) {
   x = x+1;
   y = 1;
}
...
int i = 1;
int[] A = new int[5];
A[1]=4;
fiefoo(i,A[i]);
```

Fig. 7.11 Call by name is not call by value-result

Note how the copy rule requires that the actual parameter must be evaluated *every time* that the formal parameter is encountered during execution. Consider the example of Fig. 7.10, where the construct `i++` has the semantics of returning the current value of the variable `i` and then incrementing the value of the variable by 1.

The copy rule requires that the `i++` construct must be evaluated twice. Once for every occurrence of the formal parameter `y` in `fie`. The first time, its evaluation returns the value 2 and increments the value of `i` by one. The second time, it returns the value 3 and increments `i` again.

The example we have just discussed shows how it is an error to consider call by name as a complicated way of describing call by value-result. The two modes are semantically different, as Fig. 7.11 shows.

Let us assume, first, that we are to execute `fiefoo` with parameters passed by value-result. On termination, we will have `A[1]` with value 1 and `i` with value 2, while the rest of the array `A` has not been touched. If, on the other hand, we execute the same procedure with the two parameters passed by name, on termination, we will have `A[1]` with value 4, `i` with value 2 and, what is more important, the element `A[2]` will have been updated to the value 1. It can be seen that, in this case, value-result and call by reference will have exhibited the same behaviour.

It remains to describe how call by name can be implemented. We have already discussed the necessity for the caller to pass not only the textual expression forming its actual parameter but also the environment in which this expression must be evaluated. We call a pair, (expression, environment), in which the environment includes (at least) all the free variables in the expression a *closure*.[5] We can therefore

[5]The term "closure" comes from mathematical logic. A formula is closed when it does not contain free variables. A closure is a canonical method for transforming a piece of code containing nonlocal (that is, "free") variables in a completely specified code.

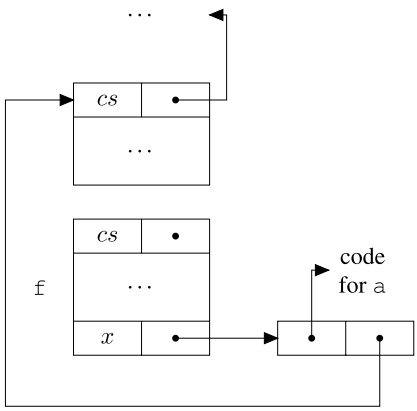


Fig. 7.12 Implementation of call by name

say that, in the case of call by name, the caller passes a closure, formed from the actual parameter (in the form of a textual expression) and the current environment. When the callee encounters a reference to the formal parameter, it is resolved using the evaluation of the first component of the closure in the environment given by the second component. Figure 7.12 describes this situation in the particular case of an abstract machine with a stack. In this case, a closure is a pair formed from two pointers: the first points to the code that evaluates the expression of the formal parameter, the second is a pointer to the static chain, which indicates the block which forms the local environment in which to evaluate the expression. When a procedure `f` with a formal name parameter, `x`, is called with actual parameter `a`, the call constructs a closure whose first component is a pointer to the code for `a` and whose second component is a pointer to the (caller's) actual activation record. It then binds this closure (for example, using another pointer) to the formal parameter `x` which resides in the called procedure's activation record.

We can finally summarise what we know on call by name. It is a method which supports input and output parameters. The formal parameter does *not* correspond to a variable that is local to the procedure; the actual parameter can be an arbitrary expression. It is possible that the actual and formal parameters can be aliased. The actual parameter must evaluate to an l-value if the formal parameter appears on the left of an assignment, The semantics of call by name was established by the copy rule which allows the maintenance of a constant link between formal and actual parameters during execution. The canonical implementation uses a closure. The procedure's local environment is extended with an association between the formal and a closure, the latter being composed of the actual parameter and the environment in which the call occurs. Every access to the formal parameter is performed via the *ex novo* evaluation of the actual parameter in the environment stored in the closure. This is a very expensive parameter-passing method, both because of the need to pass a complex structure and, in particular, because of the repeated evaluation of the actual parameter in an environment that is not the current one.

Jensen's Device

Call by name allows side effects to be exploited to obtaining elegant and compact code, even though it often results in code that is difficult to understand and maintain. This is the case with the so-called Jensen's Device which makes use of pass by name to pass a complex expression and, at the same time, a variable which appears in the same expression, in such a way that modifications to variable change the value of the expression. Let us consider the following example:

```
int summation (name int exp; name int i;
               int start; int stop) {
   int acc = 0;
   for (i=start, i<= stop, i++)
      acc = acc + exp;
   return acc;
}
int x = ...;
...
int y = summation(2*x*x - 2*x + 1, x, 1, 10);
```

The side effects of passing a parameter by name are such that, in the body of the loop in `summation`, the value of `exp` can depend upon the value of `i`. A moment's reflection shows that the call on the last line is equivalent to the calculation of the sum:

$$y = \sum_{x=1}^{10} 2x^2 - 2x + 1.$$

Jensen's Device allows call by name to be used as a way to derive powerful and specialisable "higher-order" procedures at call time (in the case of the example, by indicating the expression in which to calculate the sum).

7.2 Higher-Order Functions

A function is *higher order* when it accepts as parameters, or returns another function as its result. Although there is no unanimous agreement in the literature, we will say that a programming language is higher-order when it allows functions either as parameters or as results of other functions. Languages with functions as parameters are fairly common. On the other hand, languages that allow functions to return functions as a result are less common. This last type of operation, however, is one of the fundamental mechanisms of *functional* programming languages (which we will deal with extensively in Chap. 11). We will, in this section, discuss linguistic and implementation problems in these two cases. We treat each of them separately.

Fig. 7.13 Functional parameters

```
{int x = 1;
int f(int y){
     return x+y;
}
void g (int h(int b)){
     int x = 2;
     return h(3) + x;
}
...
 {int x = 4;
  int z = g(f);
 }
}
```

7.2.1 Functions as Parameters

The general case, the one we want to analyse, is that of a language with functional parameters, nested environments and the ability to define functions at every nesting level.[6] Let us consider the example shown in Fig. 7.13.

Using the notation `void g (int h(int n)){ ... }`, we mean, in our pseudo-language, the declaration of the function `g` with a single formal parameter, `h`, which, in its turn, is specified to be a function returning an `int` with its own formal parameter of type `int`.[7] The two key points of the example are: (i) the fact that `f` is passed as an actual parameter to `g` and later called through the formal parameter `h`; and (ii) the name `x` is defined more than once, so it is necessary to establish which is the (nonlocal) environment in which `f` will be evaluated. Concerning this second question, the reader will not be surprised if we observe that there are two possibilities for selecting the nonlocal environment to use when executing a function `f` invoked using a formal parameter `h`:

- Use the environment that is active at the time when the *link between `h` and `f` is created* (which happens on line 11). We say, in this case, that the language uses a *deep binding* policy.
- Use the environment that is active when the *call* of `f` using `h` occurs (which happens on line 7). In this case, we say that the language uses a *shallow binding* policy.[8]

Although the two alternatives for binding immediately recall the distinction between static and dynamic scope, we emphasise that the binding policy (in the case

[6] C allows functions as parameters, but it is possible to define a function only in the global environment. With this limitation, the problem becomes considerably simplified (and this simplicity is just one of the reasons why C does not allow nested functions).

[7] The name of the formal parameter of `h` (in this case `n`), is of no relevance and there is no way in which the programmer can use it in the body of `g`.

[8] The terminology, however, is not uniform across the literature. In particular, the terms deep and shallow binding are also used, in a special way, in the LISP community to indicate two different implementation techniques for dynamic scope.

of higher-order functions) should be considered independent of scope policy. All common languages that use static scope also use deep binding (because the choice of a shallow policy appears contradictory at the methodological level). The matter is not as clear for languages with dynamic scope, among which there are languages with deep as well as shallow binding.

Returning to the example of Fig. 7.13, the different scope and binding policies yield the following behaviours:

- Under static scope and deep binding, the call `h(3)` returns 4 (and `g` returns 6). The `x` in the body of `f` when it is called using `h` is the one in the outermost block;
- Under dynamic scope and deep binding, the call `h(3)` returns 7 (and `g` returns 9). The `x` in the body of `f` when it is called using `h` is the one local to the block in which the call `g(f)` occurs;
- Under dynamic scope and shallow binding, the call `h(3)` returns 5 (and `g` returns 7). The `x` in the body of `f` at the moment of its call through `h` is the one local to `g`.

Implementation of deep binding Shallow binding does not pose additional implementation problems to the technique used to implement dynamic scope. It is enough, at least conceptually, to look for every name's last association in the environment. Things are different, though, for deep binding, which requires auxiliary data structures in addition to the usual static and dynamic chains.

So as to fix our ideas, let us consider the case of a language with static scope and deep binding (the case of dynamic scope is left to the reader, see Exercise 6). From Sect. 5.5.1, we already know that to any direct invocation (one that is not of a call to a formal parameter) of a function `f`, there is statically associated information (an integer) which expresses the nesting level of the definition of `f` with respect to the block in which the call occurs. This information is used dynamically by the abstract machine to initialise the static chain pointer (that is, the nonlocal environment) in the activation record for `f`. When, on the other hand, a function `f` is invoked using a formal parameter, `h`, no information can be associated to the call because it is called via a formal parameter. Indeed, in the course of different activations of the procedure in which it is located, the formal can be associated with different functions (this is the case, for example, with the call `h(3)` in Fig. 7.13).

In other words, it is clear that with deep binding, the information about the static chain pointer must be determined at the moment the association between the formal and actual parameters is created. With the formal `h` must be associated not only the code for `f` but also the nonlocal environment in which the body of `f` is to be evaluated. Such a nonlocal environment can be determined in a simple fashion: corresponding to a call of the form `g(f)` (the procedure `g` is called with the functional actual parameter `f`), we can statically associate with the parameter `f` the information about the nesting level of the definition of `f` within the block in which the call `g(f)` occurs. When this call is performed, the abstract machine will use this information to associate with the formal parameter corresponding to `f` *both* the code for `f`, *and* a pointer to the activation record of the block inside which `f` is declared (this pointer is determined using the same rules that were discussed in Sect. 5.5.1).

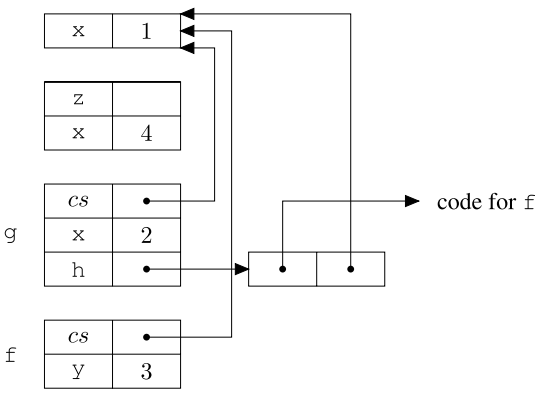


Fig. 7.14 Closure and static chain

The formal function parameter is therefore associated logically with a (text, environment) pair, represented at the implementation level by a pair (pointer to code, pointer to an activation record). We have already seen when we discussed the implementation of call by name that such a data structure is called a *closure*. Therefore, a closure is associated with the formal functional parameter. When the formal parameter is used to invoke a function (that is statically unknown), the abstract machine finds in the first component of the pair the code to which to transfer control and assigns the content of the second component of the closure to the static chain pointer of the activation record for the new invocation.

Figure 7.14 shows the situation on the activation stack for the code in Fig. 7.13 at the moment at which the function `f` is entered (it is called using `h` from the body of `g`). When the function `g` is called with actual parameter `f`, a closure is linked to `h`. `f` is declared at distance 1 from the place in which it appears as an actual parameter (it is in fact declared inside the block which *contains* the one in which it appears as an actual parameter). The second component of the closure is therefore determined by 1 step along the static chain (obtaining a pointer to the outer block). When `f` is called through the name `h`, the corresponding activation record is pushed onto the stack. The value of the static chain pointer is taken from the second component of the closure.

The reader will recall, once more, that the problems we are discussing appear only when the language allows the definition of functions with nonlocal environments, that is allows the definition of functions *inside nested blocks*. In the contrary case, for example in C, there is no nonlocal environment, so there is no need for closures. To pass a function as a parameter, it is enough to pass a pointer to its code. All nonlocal references in the body of the function will be resolved in the global environment.

Binding policy and static scope We have already observed how all languages with static scope use deep binding. At first sight it could rather seem that deep or shallow binding make no difference in the case of static scope. After all, the nonlocal environment of a function is determined from the (static) position of its declaration and not by the way in which it is invoked. In the case in Fig. 7.13, it is the scope

Fig. 7.15 The binding policy is necessary for determining the environment

```
{void foo (int f(), int x){
    int fie(){
       return x;
    }
    int z;
    if (x==0) z=f();
    else foo(fie,0);
 }
 int g(){
    return 1;
 }
 foo(g,1);
}
```

Thunks

The parameter-passing rule required for functional parameters is similar to call by name. In fact, a formal name parameter can be considered as a kind of functional parameter (without arguments). Analogously, the corresponding actual parameter can be considered as the definition of an anonymous argumentless function. During execution of the body, every occurrence of the name parameter corresponds to an *ex novo* evaluation of the actual parameter in the environment fixed at the moment the association between the actual and formal name parameter is made. A process that is analogous to a new call to the anonymous function corresponding to the actual parameter.

In ALGOL60 jargon, the name *thunk* was used for a structure like this. A function without arguments and associated evaluation environment. In call by name, therefore a connection between the formal parameter and a thunk is introduced.

(and not binding) rule that establishes that every invocation of `f` (whether direct, using its name, or indirect, using a formal parameter) is evaluated in the outermost nonlocal environment.

In general, however, it is not like this. The reason for this is that there can be many activation records for the same function simultaneously present on the stack (this clearly happens when we have recursive or mutually recursive functions). If a procedure is passed out from one of these activations, it is possible to create a situation in which the scope rules alone are not enough to determine which nonlocal environment to use in invoking the functional parameter. As an example, we will discuss the code in Fig. 7.15, which, as usual, we assume was written in a pseudo-language with static scope.

The heart of the problem is the (nonlocal) reference to `x` inside `fie`. The scope rules tell us that such an `x` refers to the formal parameter to `foo` (which, as it happens, is the only `x` declared in the program). But when `fie` is called (using the formal `f`), there are two active instances of `foo` (and therefore two instances of its local environment). A first activation from the call `foo(g,1)`, in which `x` is

The environment in C

The structure of the environment in C is particularly simple. A C program consists, in fact, of a sequence of variable and function declarations. The variables declared in this way (which in C jargon are called *external* variables) are visible at any point in the program. They are global variables, according to the terminology of Sect. 4.2.2. Functions are structured internally as blocks, and in each block local variables can be declared, but the definition of functions inside other functions is not permitted.

The environment of a function, therefore, is composed of a local and a global part. Each reference to a nonlocal name is resolved in a unique fashion in the global environment. With this simplified structure, environment handling is very simple. The static chain does not have to be maintained and to pass a function as a parameter to another function, it is sufficient to pass a pointer to its code.

For reasons of efficiency, furthermore, there is no management of in-line blocks. Variables declared in any block in a function are allocated in the activation record of the function.

Execution efficiency is one of the primary objectives for C. To avoid the cost of activation record stacks, the compiler can choose to translate a function call using the expansion of its body (in the case of a recursive function, the expansion happens once only).

associated (to a location which contains) the value 1, and a second one from the (recursive) call `foo(fie, 0)`, in which `x` is associated with the value 0. It is inside this second activation that the call to `fie` through `f` is made. The scope rules say nothing about which of the instances of `x` should be used in the body of `f`. It is at this point that the binding policy intervenes. Using deep binding, the environment is established when the association between `fie` and `f` is created, that is when `x` is associated with the value 1. The variable `z` will therefore be assigned the value 1. To help in understanding this example, Fig. 7.16 shows the stack and the closures when `fie` is executed.

In the case of shallow binding (which, let us repeat, is not used with static scope), the environment would be determined at the time `f` is invoked and `z` would be assigned the value 0.

What defines the environment Before closing this section, let us consider again the problem encountered in Chap. 4, of which rules are used to determine the environment. We can finally complete the ingredients which contribute to the correct determination of the evaluation environment for a block-structured language. The following are necessary:

- Visibility rules, normally guaranteed by block structure.
- Exceptions to the visibility rules (which take into account, for example, redefinitions of names and the possibility or not of using a name before its declaration).
- Scope rules.

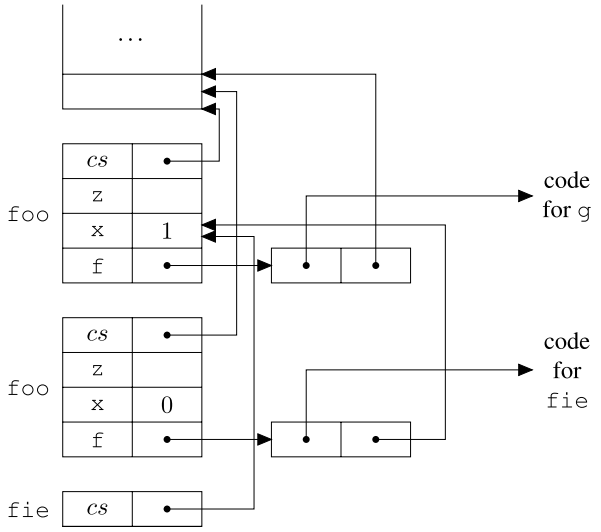


Fig. 7.16 Activation stack for Fig. 7.15

- The rules for the parameter passing method.
- The binding policy.

7.2.2 Functions as Results

The possibility of generating functions as the results of other functions allows the dynamic creation of functions at runtime. It is clear how, in general, a function returned as a result cannot be represented at execution time by its code alone, it will also be necessary to include the environment in which the function will be evaluated. Let us consider a first simple example in Fig. 7.17. Let us fix, first of all, the notation: by `void->int` we denote the *type* of the functions which take no argument (`void`) and return a value of type integer (`int`). The second line of the code is therefore the declaration of a function `F` which returns a function of no arguments which also returns an integer (note that `return g` returns the "function", not its application). The first line after the body of `F` is the declaration of the name `gg` with which the result of the evaluation of `F` is dynamically associated.

It should not be difficult to convince ourselves that the function `gg` returns the successor of the value of `x`. Using the static scope regime, this `x` is fixed by the structure of the program and not by the position of the call to `gg`, which could appear in an environment in which another definition of the name `x` occurs.

We can, therefore, say that, in general, when a function returns a function as result, this result is a *closure*. In consequence, the abstract machine must be appropriately modified to take into the account calls to closures. Analogous to what happens

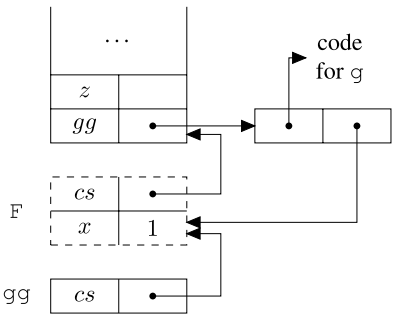
Fig. 7.17 Functions as results

```
{int x = 1;
void->int  F () {
   int g () {
      return x+1;
   }
   return g;
}
void->int gg = F();
int z = gg();
}
```

Fig. 7.18 Functions as result and stack discipline

```
void->int  F () {
   int x = 1;
   int g () {
      return x+1;
   }
   return g;
}
void->int gg = F();
int z = gg();
```

Fig. 7.19 Activation records for Fig. 7.18



when a function is called via a formal parameter, when a function whose value is obtained dynamically (like gg), the static chain pointer of its activation record is determined using the associated closure (and not the canonical rules discussed in Sect. 5.5.1, which would be of no help).

The general situation, moreover, is much more complex. If it is possible to return a function from the inside of a nested block, it is possible to arrange that its evaluation environment will refer to a name that, according to stack discipline, is going to be destroyed. Let us consider, indeed, the code in Fig. 7.18, where we have moved the declaration of x to the inside of F. Figure 7.19 shows the activation record arrangement when gg() is called.

When the result of F() is assigned to gg, the closure which forms its value points to an environment containing the name x. But this environment is local to F and will therefore be destroyed on its termination. How is it possible, then, to call gg later than this without producing a dangling reference to x? The reply can only be drastic: abandon stack discipline for activation records, so that they can then stay

alive indefinitely, because they could constitute the environments for functions that will be evaluated subsequently. In languages with the characteristics that we have just discussed, local environments have an *unlimited* lifetime.

The most common solution in this case is to allocate all activation records in the heap and to leave it to a garbage collector to deallocate them when it is discovered that there are no references to the names they contain.

Every functional language is constructed around the possibility of returning functions as results. They must therefore take this problem head on. On the contrary, returning functions as results in imperative languages is rare; this is purely to maintain a stack discipline for activation records. In imperative languages which do permit functions as results, there are, generally, many restrictions aimed at guaranteeing that it is never possible to create a reference to an environment that has become deactivated (for example: no nested functions (C, C++), return only non-nested functions (Modula-2, Modula-3), appropriately restrict the scope of those nested functions that are returned (Ada), etc.).

7.3 Exceptions

An exception is an event that is checked during the execution of a program and which must not (or cannot) be handled in the normal flow of control. Such events could be checking that a dynamic semantic error has occurred (e.g., a division by zero, overflow, etc.) or checking that a situation has occurred for which the programmer explicitly decides to terminate the current computation and transfer control to another point in the program, often outside the currently executing block.

First-generation languages did not provide structures for handling such situations. They typically treat them by means of jumps (`gotos`). On the other hand, many modern languages such as C++, Ada and Java have *structural* mechanisms for exception handling which appear as real abstraction constructs. These constructs allow the interruption of a computation and the shifting of control outside of the current construct, block or procedure. Often, this mechanism also allows data to be passed across the jump, resulting in a very flexible (and also often efficient) tool for handling those cases of exceptional termination of a computation which the normal control constructs (loops and conditionals) are unable to handle properly. Devised for handling the unusual or exceptional cases which can present themselves in a program, exceptions are also useful, as we will see, when giving compact and efficient definitions of some ordinary algorithms.

The mechanisms for handling exceptional situations vary greatly from language to language. We will restrict ourselves here to describing some common approaches and we do not pretend to be exhaustive. In general, we can say that, in order to correctly handle exceptions, a language must:

1. Specify *which* exceptions can be handled and how they can be defined.
2. Specify how an exception can be *raised*, that is which mechanisms cause the exceptional termination of a computation.

```
class EmptyExcp extends Throwable {int x=0;};

int average(int[] V) throws EmptyExcp(){
   if (length(V)==0) throw new EmptyExcp();
   else {int s=0; for (int i=0, i<length(V), i++) s=s+V[i];}
   return s/length(V);
};
...
try{...
    average(W);
    ...
}
catch (EmptyExcp e) {write('Array␣empty);␣}
```

Fig. 7.20 Exception handling

3. Specify how an exception can be *handled*, that is what are the actions to perform to determine an exception has occurred and where to transfer control of execution.

On the first point, we find both exceptions raised directly by the abstract machine (when some dynamic semantic condition is violated) and exceptions defined by the user. The latter can be values of a special type (as is the case in Ada and in ML), or any value whatsoever (as in C++) or something in the middle (in Java, an exception is an instance of some subclass of `Throwable`). When an exception is of any type whatsoever, in general it can contain a dynamically generated value which is passed to the handler.

Once an exception is defined, it can be raised implicitly if it is an abstract machine exception, or explicitly by the programmer using an appropriate construct.

Finally, for point (3), the handling of an exception in general requires two different constructs:

- A mechanism that defines a capsule around a portion of code (the *protected block*), with the aim of intercepting the exceptions that are to be handled inside the capsule itself.
- The definition of a handler for the exception, statically linked to the protected block. Control is transferred to the handler when the capsule intercepts the exception.

Let us examine the example in Fig. 7.20 (written in the usual pseudo-language inspired by Java and C++). The first line is the definition of the exception. In the case we are considering, all instances of class `EmptyExcp` can be an exception. To an approximation, we can imagine an instance of such a class as a record with a single internal field, labelled by `x`. Passing an exception involves the creation of a such a value and then raising the exception proper.

The second line is the definition of the `average` function. The keyword `throws` introduces the list of exceptions that *can* be thrown in the body of the function (in our case `EmptyExcp`). This clause (necessary in Java but optional in C++) has an important function as documentation: it shows the clients of the func-

tion that, in addition to the integer result, it could result in anomalous termination as signalled by the exception itself.

The construct that raises an exception is `throw`. A protected block is introduced by the keyword `try`, while the handler is contained in a block introduced by `catch`. The `average` function computes the arithmetic mean of the elements of the vector `V`. In the case in which the vector is empty, the function, instead of returning something arbitrary, raises an exception of class `EmptyExcp`. In checking such an event, the language's abstract machine interrupts the execution of the current command (in this case, the conditional command) and propagates the exception. All blocks entered during execution are exited until a `try` block trapping (`catch`) *this* exception is found. In the case of Fig. 7.20, the `average` function would be terminated, as would every block appearing between the only `try` present and the call to `average`. When the exception is intercepted by a `try`, control passes to the code in the `catch` block. If no explicit `try` trapping the exception is encountered, it is captured by a default handler which then terminates execution of the program and prints some error message or other.

The handler (the code in the `catch`) block is statically linked to the protected block. When an exception is detected, execution of the handler replaces the part of the protected block which has still to be executed. In particular, after execution of the handler, control flows normally to the first instruction which follows the handler itself.[9]

As far as these questions are strongly dependent on the language, let us make two important observations:

1. An exception is not an anonymous event. It has a name (often, rather, as in our case, it is a value in one of the language's types) which is explicitly mentioned in the `throw` construct and is used by constructs of the form `try-catch` to trap a specific class of exception.
2. Although the example does not show it, the exception could include a value which the construct that raised the exception passes in some way to the handler as an argument on which to operate in order to react to the exception that has just occurred (in our case, when it is raised, the value of the field `x` in the exception could be modified).

It is easy to convince ourselves that the propagation of an exception is not a simple jump. Let us assume that the exception is detected inside some procedure. If the exception is not handled inside the currently executing procedure, it is necessary to terminate the current procedure and to re-raise the exception at the point at which the current procedure was invoked. If the exception is not handled by the caller either, the exception is propagated along the procedure's call chain until it reaches

[9] This way of working is, in the literature, called "handling with termination" (because the construct where the exception is determined is terminated). Some older languages, PL/1 for example, (one of the first languages to introduce exception-handling mechanisms) follow a different approach, called "handling with resumption". In this case, the handler can arrange that control returns to the point where the exception was raised. The scheme with resumption can lead to flagging errors that are very difficult to locate.

Fig. 7.21 Exceptions propagate along the dynamic chain

```
{
void f() throws X {
   throw new X();
   }

void g (int sw) throws X {
   if (sw == 0) {f();}
   try {f();} catch (X e) {write("in g");}
   }
...
try {g(1);}
catch (X e)  {write("in main");}
}
```

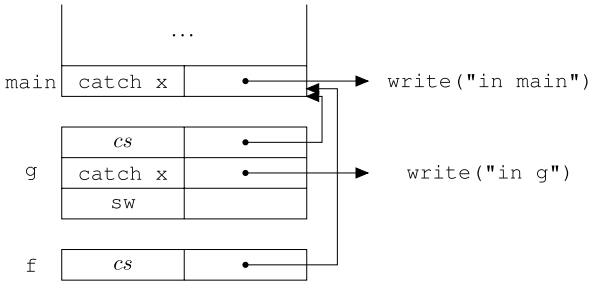


Fig. 7.22 System stack for Fig. 7.21

a compatible handler or reaches the highest level which provides a default handler (which results in an error termination).

A subtle aspect which is worth explicitly considering is that exceptions propagate along the dynamic chain, even if the handler is statically associated with the protected block. To illustrate this point, let us consider the code in Fig. 7.21, where we have assumed that we have already declared an exception class `X`.

The exception `X` is raised inside two protected blocks. The outer `try` is the one in the body of `g`. This is the one that traps the exception. The program prints the string "in `g`". Figure 7.22 shows the stack of activation records and handlers when the body of function `f` is executed. Considering Fig. 7.21, note that, in the case in which the argument of `g` in the protected block is a variable whose value is execution-dependent, it is not statically determinable which will be the handler to invoke.

Summarising, an exception is handled by the last handler that has been placed on the execution stack. This is a reasonable behaviour. The exception is trapped at the closest possible point to the one at which it was detected.

Pragmatics We have discussed the use of exceptions in handling error cases. There are ordinary cases, though, in which the cautious use of exceptions makes for more elegant and efficient programs. We will limit ourselves to presenting one example, that of walking a binary tree where it is desired to calculate the product of the integers which label the nodes. The most obvious code is that shown in Fig. 7.23,

```
type Node = {int key;
             Node FS;
             Node FD;
            }

int mul (Node alb){
   if (alb == null) {return 1;}
   return alb.key * mul(alb.FS) * mul(alb.FD);
}
```

Fig. 7.23 Anticipated visit to a binary tree

```
int mulAus (Node alb) throws Zero{
   if (alb == null) {return 1;}
   if (alb.key == 0) {throw new Zero();}
   return alb.key * mulAus(alb.FS) * mulAus(alb.FD);
}

int mulEff (Node alb){
   try {return mulAus (alb);}
   catch (Zero e) {return 0;};
}
```

Fig. 7.24 A more efficient traversal

where a depth-first search of the tree is used and it is assumed that the type `Tree` is a structure (a record or a class) with three fields of which the first is an integer and the others are of type `Tree` and which link the structure together. The generic null value is represented by `null`.

The function `mul` correctly returns the product of the nodes, but is inefficient in the cases of very large trees where there is significant chance that some nodes are zero, given that in such a case the function could immediately terminate with the result zero. We can make use of the exception mechanism to force this expected termination without needing to produce the traversal code.

Figure 7.24 shows the function `mulAus` which raises an exception (of class `Zero`, which we assume to be defined elsewhere) when it encounters a node labelled with zero. `mullAus` is called by `mulEff`[10] inside to the protected block which handles the exception returning zero.

7.3.1 Implementing Exceptions

The simplest and intuitive way in which an abstract machine can implement exceptions is one that uses the stack of activation records. When a block is entered at

[10]The function `mulEff` is the function "exported" to clients in this program. In jargon, we say that `mulEff` is a *wrapper* of `mulAus`.

runtime, a pointer to the corresponding handler (together with the type of exception to which it refers) is inserted into the activation record (of the current procedure or of the current anonymous block). When a normal exit is made from a protected block (that is, because control transfers from it in the usual way and not through raising an exception), the reference to the handler is removed from the stack. Finally, when an exception is raised, the abstract machine looks for a handler for this exception in the current activation record. If one is not found, it uses the information in the record to reset the state of the machine, removes the record from the stack and rethrows the exception. This is a conceptually very simple solution but it has a not insignificant defect: each time that a protected block is entered, or is left, the abstract machine must manipulate the stack. This implementation, therefore, requires explicit work even in the normal (and more frequent) case in which the exception *is not* raised.

A more efficient runtime solution is obtained by anticipating a little of the work at compile time. In the first place, each procedure is associated with a hidden protected block formed from the entire procedure body and whose hidden handler is responsible only for clearing up the state and for rethrowing the exception unchanged. The compiler then prepares a table, *EH*, in which, for each protected block (including the hidden ones) it inserts two addresses (*ip*, *ig*) which correspond to the start of the protected block and the start of its handler. Let us assume that the start of the handler coincides with the end of the protected block. The table, ordered by the first component of the address, is passed to the abstract machine. When a protected block is entered or exited normally, nothing need be done. When, on the other hand, an exception is raised, a search is performed in the table for a pair of addresses (*ip*, *ig*) such that *ip* is the greatest address present in *EH* for which $ip \leq pc \leq ig$, where *pc* is the value of the program counter when the exception is detected. Since *EH* is ordered, a binary search can be performed. The search locates a handler for the exception (recall that a hidden handler is added to each procedure). If this handler re-throws the exception (because it does not capture this exception or because it is a hidden handler), the procedure starts again, with the current value of the program counter (if it was a handler for another exception) or with the return address of the procedure (if it was a hidden handler). This solution is more expensive than the preceding one when an exception is detected, at least by a logarithmic factor in the number of handlers (and procedures) present in the program (the reduction in performance depends on the need to perform a search through a table every time; the cost is logarithmic because a binary search is performed). On the other hand, it costs nothing at protected block entry and exit, where the preceding solution imposed a constant overhead on both of these operations. Since it is reasonable to assume that the detection of an exception is a rarer event than simple entry to a protected block, we have a solution that is on average more efficient than the preceding one.

7.4 Chapter Summary

The chapter has dealt with a central question for every programming language: mechanisms for functional abstraction. It has discussed, in particular, two of the

```
class X extends Exception {};
class P{                                                    2
    void f() throws X{
        throw new X();                                      4
    }
}                                                           6

class Q{                                                    8
    class X extends Exception {};
    void g(){                                               10
        P p = new P();
        try {p.f();} catch (X e){                           12
                     System.out.println("in g");
                     }                                      14
    }
}                                                           16
```

Fig. 7.25 Static scope and exception names

Exceptions and Static Scope

Languages like Java and C++ combine static scope for name definitions (and therefore also for exception names) with the dynamic association of handlers with protected blocks, as we have just seen. Such a combination is sometimes the cause of confusion, as the Java code in Fig. 7.25 shows. On a superficial reading, the code seems syntactically correct. Moreover, it could be said that an invocation of `g` results in the string "`in g`" being printed.

Yet, if the compilation of the code is attempted, the compiler finds two static semantic errors around line 12: (i) the exception `X` which must be caught by the corresponding `catch` is not raised in the `try`; (ii) the exception `X`, raised by `f`, is not declared in the (absent) `throws` clause of `g`.

The fact is that exception names (`X` in this case) have normal static scope. Method `f` raises the exception `X` declared on line 1; while the `catch` on line 12 traps the exception `X` declared on line 9 (and which is more correctly denoted by `Q.X`, since it is a nested class within `Q`). Just so as to avoid errors caused by situations of this kind, Java imposes the requirement on every method that it must declare all the exceptions that can be generated in its body in its `throws` clause.

The analogous situation can be reproduced in C++, *mutatis mutandis*. In C++, however, the `throws` clause is optional. If we compile the C++ code corresponding to that in Fig. 7.25, in which `throws` clauses are omitted, compilation terminates properly. But clearly, an invocation of the method `f` throws an exception different from that caught in the body of `g`. An invocation of `g` results in an exception `X` (of the class declared on line 1) which is then propagated upwards.

most important linguistic mechanisms to support functional abstraction in a high-level language: procedures and constructs for handling exceptions. The main concepts introduced are:

- *Procedures*: The concept of procedure, or function, or subprogram, constitutes the fundamental unit of program modularisation. Communication between procedures is effected using return values, parameters and the nonlocal environment.
- *Parameter passing method*: From a semantic viewpoint, there are input, output and input-output parameters. From an implementation viewpoint, there are different ways to pass a parameter:
  - By value.
  - By reference.
  - By means of one of the variations on call by value: by result, by constant or by value-result.
- *Higher-order functions*: Functions that take functions as arguments or return them as results. The latter case, in particular, has a significant impact on the implementation of a language, forcing the stack discipline for activation records to be abandoned.
- *Binding policy*: When functions are passed as arguments, the binding policy specifies the time at which the evaluation environment is determined. This can be when the link between the procedure and the parameter is established (deep binding) or when the procedure is used via the parameter (shallow binding).
- *Closures*: Data structures composed of a piece of code and an evaluation environment, called *closures*, are a canonical model for implementing call by name and all those situations in which a function must be passed as a parameter or returned as a result.
- *Exceptions*: Exceptional conditions which can be detected and handled in high-level languages using *protected blocks* and a *handler*. When an exception is detected, the appropriate handler is found by ascending the dynamic chain.

7.5 Bibliographical Notes

All the principal modes for parameter passing originate in the work of the Algol committee and were subsequently explored in other languages such as Algol-W and Pascal. The original definition of Algol60 [1] is a milestone in programming language design. The preparatory work on Algol-W can be seen in [10] and its mature form in the reference manual [7]. Algol-W included call by name (as default), call by value, by result and by value-result, as well as pointers and garbage collection. Pascal, which adopts as default call by reference, was first defined in [9]; the reference manual for the ISO Standard is [4].

The problems with determining the environment in the case of higher-order functions are often known as the *funarg problem* (the *fun*ctional *arg*ument problem). The *downward* funarg problem refers to the case of functions passed as arguments and therefore to the necessity of handling deep binding. The *upward* funarg problem refers to the case in which a function can be returned as a result [6]. The relations between binding policy and scope rules are discussed in [8].

One of the first languages with exceptions was PL/1 which used resumption handling (see [5]). More modern handling with termination (which anticipates the static link between protected blocks and handling) descends from Ada, which, in its turn, was inspired by [3].

7.6 Exercises

1. On page 166, commenting on Fig. 7.1, it can be seen that the environment of the function `foo` includes (as a nonlocal) the name `foo`. What purpose does the presence of this name serve inside the block?
2. State what will be printed by the following code fragment written in a pseudo-language permitting reference parameters (assume `Y` and `J` are passed by reference).

```
int X[10];
int i = 1;
X[0] = 0;
X[1] = 0;
X[2] = 0;
void foo (reference int Y,J){
     X[J] = J+1;
     write(Y);
     J++;
     X[J]=J;
     write(Y);
}
foo(X[i],i);
write(X[i]);
```

3. State what will be printed by the following code fragment written in a pseudo-language which allows *value-result* parameters:

```
int X = 2;
void foo (valueresult int Y){
    Y++;
    write(X);
    Y++;
}
foo(X);
write(X);
```

4. The following code fragment, is to be handed to an *interpreter* for a pseudo-language which permits constant parameters:

```
int X = 2;
void foo (constant int Y){
    write(Y);
    Y=Y+1;
```

```
}
foo(X);
write(X);
```

What is the most probable behaviour of the interpreter?

5. Say what will be printed by the following code fragment which is written in a pseudo-language allowing *name* parameters:

```
int X = 2;
void foo (name int Y){
   X++;
   write(Y);
   X++;
}
foo(X+1);
write(X);
```

6. Based on the discussion of the implementation of deep binding using closures, describe in detail the case of a language with dynamic scope.
7. Consider the following fragment in a language with exceptions and call by value-result and call by reference:

```
{int y=0;
 void f(int x){
    x = x+1;
    throw E;
    x = x+1;
 }
 try{ f(y); } catch E {};
 write(y);
}
```

State what is printed by the program when parameters are passed: (i) by value-result; (ii) by reference.

8. In a pseudo-language with exceptions, consider the following block of code:

```
void ecc() throws X {
   throw new X();
}
void g (int para) throws X {
   if (para == 0) {ecc();}
   try {ecc();} catch (X) {write(3);}
}
void main () {
   try {g(1);} catch (X) {write(1);}
   try {g(0);} catch (X) {write(0);}
}
```

Say what is printed when `main()` is executed.

9. The following is defined in a pseudo-language with exceptions:

```
int f(int x){
   if (x==0) return 1;
   else if (x==1) throw E;
        else if (x==2) return f(1);
             else try {return f(x-1);} catch E {return x+1;}
}
```

What is the value that is returned by `f(4)`?

10. The description of the implementation of exceptions in Sect. 7.3.1 assumes that the compiler has access (direct or through the linkage phase) to the entire code of the program. Propose a modification to the implementation scheme based on the handler table for a language in which separate compilation of program fragments is possible (an example is Java, which allows separate compilation of classes).

References

1. J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Commun. ACM*, 3(5):299–314, 1960.
2. M. Broy and E. Denert, editors. *Software Pioneers: Contributions to Software Engineering*. Springer, Berlin, 2002.
3. J. B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.
4. K. Jensen and N. Wirth. *Pascal-User Manual and Report*. Springer, Berlin, 1991.
5. M. D. MacLaren. Exception handling in PL/I. In *Proc. of an ACM Conf. on Language Design for Reliable Software*, pages 101–104, 1977.
6. J. Moses. The function of FUNCTION in LISP, or why the FUNARG problem should be called the environment problem. Technical report, MIT AI Memo 199, 1970. Disposable online at http://hdl.handle.net/1721.1/5854.
7. R. L. Sites. ALGOL W reference manual. Technical report, Stanford, CA, USA, 1972.
8. T. R. Virgilio and R. A. Finkel. Binding strategies and scope rules are independent. *Computer Languages*, 7(2):61–67, 1982.
9. N. Wirth. The programming language Pascal. *Acta Informatica*, 1(1):35–63, 1971. Reprinted in [2].
10. N. Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Commun. ACM*, 9(6):413–432, 1966.