Chapter 4
Names and The Environment

The evolution of programming languages can be seen in large measure as a process which has led, by means of appropriate abstraction mechanisms, to the definition of formalisms that are increasingly distant from the physical machine. In this context, *names* play a fundamental role. A name, indeed, is nothing more than a (possibly meaningful) sequence of characters used to represent some other thing. They allow the abstraction either of aspects of data, for example using a name to denote a location in memory, or aspects of control, for example representing a set of commands with a name. The correct handling of names requires precise semantic rules as well as adequate implementation mechanisms.

In this chapter, we will analyse these rules. We will, in particular, look at the concept of *environment* and the constructs used to organise it. We will also look at visibility (or *scope*) rules. We leave until the next chapter treatment of the implementation of these concepts. Let us immediately observe how, in languages with procedures, in order to define precisely the concept of environment one needs other concepts, related to parameter passing. We will see these concepts in Chap. 7. In the case of object-oriented languages, finally, there are other specific visibility rules which we will consider in Chap. 10.

4.1 Names and Denotable Objects

When we declare a new variable in a program:

```
int fie;
```

or we define a new function:

```
int foo( ){
   fie = 1;
}
```

M. Gabbrielli, S. Martini, *Programming Languages: Principles and Paradigms,*
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-84882-914-5_4, © Springer-Verlag London Limited 2010

we introduce new names, such as `fie` and `foo` to represent an object (a variable and a function in our example). The character sequence `fie` can be used every time that we want to refer to the new variable, just as the character sequence `foo` allows us to call the function that assigns to `fie` the value 1.

A name is therefore nothing more than a sequence of characters used to represent, or denote, another object.[1]

In most languages, names are formed of identifiers, that is by alphanumeric tokens, moreover other symbols can also be names. For example, + and – are names which denote, in general, primitive operations.

Even though it might seems obvious, it is important to emphasise that a name and the object it denotes are *not* the same thing. A name, indeed, is just a character string, while its denotation can be a complex objects such as a variable, a function, a type, and so on. And in fact, a single object can have more than one name (in this case, one speaks of *aliasing*), while a single name can denote different objects at different times. When, therefore, we use, as we may, the phrase "the variable `fie`" or the phrase "the function `foo`", it should be remembered that the phrases are abbreviations for "the variable with the name `fie`" and "the function with the name `foo`". More generally, in programming practice, when a name is used, it is almost always meant to refer to the object that it denotes.

The use of names implements a first, elementary, *data abstraction* mechanism. For example, when, in an imperative language, we define a name using a variable, we are introducing a symbolic identifier for a memory location; therefore we are abstracting from the low-level details of memory addresses. If, then, we use the assignment command:

```
fie = 2;
```

the value 2 will be stored in the location reserved for the variable named `fie`. At the programming level, the use of the name avoids the need to bother with whatever this location is. The correspondence between name and memory location must be guaranteed by the implementation. We will use the term *environment* to refer to that part of the implementation responsible for the associations between names and the objects that they denote. We will see better in Sect. 6.2.1 what exactly constitutes a variable and how it can be associated with values.

Names are fundamental even for implementing a form of *control abstraction*. A procedure[2] is nothing more than a name associated with a set of commands, together with certain visibility rules which make available to the programmer its sole interface (which is composed of the procedure's name and possibly some parameters). We will see the specifics of control abstraction in Chap. 7.

[1] Here and in the rest of this chapter, "object" is intended in a wide sense, with no reference to technical terms used in the area of object-oriented languages.

[2] Here and elsewhere, we will use the generic term "procedure" for procedures as well as functions, methods and subprograms. See also Sect. 7.1.

4.1.1 Denotable Objects

The objects to which a name can be given are called *denotable objects*. Even if there are considerable differences between programming languages, the following is a non-exhaustive list of possible denotable objects:

- Objects whose names are defined by the user: variables, formal parameters, procedures (in the broad sense), user-defined types, labels, modules, user-defined constants, exceptions.
- Objects whose names are defined by the programming language: primitive types, primitive operations, predefined constants.

The association (or *binding*) between a name and an object it denotes can therefore be created at various times. Some names are associated with objects during the design of a language, while other associations are introduced only when a program is executed. Considering the entire process ranging from a programming language's definition to the execution of a specific program, we can identify the following phases for the creation of bindings of names to objects:

Design of language In this phase, bindings between primitive constants, types and operations of the language are defined (for example, + indicates addition, and `int` denotes the type of integers, etc.).

Program writing Given that the programmer chooses names when they write a program, we can consider this phase as one with the partial definition of some bindings, later to be completed. The binding of an identifier to a variable, for example, is defined in the program but is effectively created only when the space for the variable is allocated in memory.

Compile time The compiler, translating the constructs of the high-level language into machine code, allocates memory space for some of the data structures that can be statically processed. For example, this is the case for the global variables of a program. The connection between a variable's identifier and the corresponding memory location is formed at this time.

Runtime This term denotes the entire period of time between starting and termination of a program. All the associations that have not previously been created must be formed at runtime. This is the case, for example, for bindings of variable identifiers to memory locations for the local variables in a recursive procedure, or for pointer variables whose memory is allocated dynamically.

In the previous description we have ignored other important phases, such as linking and loading in which other bindings (for example for external names referring to objects in other modules). In practice, however, two principle phases are distinguished using the terms "static" and "dynamic". The term "static" is used to refer to everything that happens prior to execution, while "dynamic" refers to everything that happens during execution. Thus, for example, static memory management is performed by the compiler, while dynamic management is performed by appropriate operations executed by the abstract machine at runtime.

Fig. 4.1 A name denoting different objects

```
{int fie;
 fie = 2;
     {char fie;
      fie = a;
     }
}
```

4.2 Environments and Blocks

Not all associations between names and denotable objects are fixed once and for all at the start of program execution. Many can vary during execution. To be able to understand how these associations behave, we need to introduce the concept of environment.

Definition 4.1 (Environment) The set of associations between names and denotable objects which exist at runtime at a specific point in the program and at a specific time during execution, is called the (*referencing*) *environment*.

Usually, when we speak of environments, we refer only to associations that are not established by the language definition. The environment is therefore that component of the abstract machine which, for every name introduced by the programmer and at every point in the program, allows the determination of what the correct association is. Note that the environment does not exist at the level of the physical machine. The presence of the environment constitutes one of the principle characteristics of high-level languages which must be simulated in a suitable fashion by each implementation of the language.

A *declaration* is a construct that allows the introduction of an association in the environment. High-level languages often have explicit declarations, such as:

```
int x;
int f (){
   return 0;
}
type T = int;
```

(the first is a declaration of a variable, the second of a function named `f`, the third is declaration of a new type, `T`, which coincides with type `int`). Some languages allow implicit declarations which introduce an association in the environment for a name when it is first used. The denoted object's type is deduced from the context in which the name is used for the first time (or sometimes from the syntactic form of the name).

As we will see in detail below, there are various degrees of freedom in associations between names and denotable objects. First of all, a single name can denote different objects in different parts of the program. Consider, for example, the code of Fig. 4.1. The outermost name `fie` denotes an integer variable, while the inner one is of type character.

It is also possible that a single object is denoted by more than one name in different environments. For example, if we pass a variable by reference to a procedure, the variable is accessible using its name in the calling program and by means of the name of the formal parameter in the body of the procedure (see Sect. 7.1.2). Alternatively, we can use pointers to create data structures in which the same object is then accessible using different names.

While different names for the same object are used in different environments, no particular problems arise. The situation is more complicated when a single object is visible using different names in the same environment. This is called *aliasing* and the different names for the same object called *aliases*. If the name of a variable passed by reference to a procedure is also visible inside the same procedure, we have a situation of aliasing. Other aliasing situations can easily occur using pointers. If `X` and `Y` are variables of pointer type, the assignment `X = Y` allows us to access the same location using both `X` and `Y`.

Let us consider, for example, the following fragment of C program where, as we will do in the future, we assume that `write(Z)` is a procedure which allows us to print the value of the integer variable `Z`:

```
int *X, *Y;          // X,Y pointers to integers
X = (int *) malloc (sizeof (int));
                     // allocate heap memory
*X = 5;              // * dereference
Y=X;                 // Y points to the same object as X
*Y=10;
write(*X);
```

The names `X` and `Y` denote two different variables, which, however, after the execution of the assignment command `X = Y`, allow to access the same memory location (therefore, the next print command will output the value 10).

It is, finally, possible that a single name, in a single textual region of the program, can denote different objects according to the execution flow of the program. The situation is more common than it might seem at first sight. It is the case, for example, for a recursive procedure declaring a local name. Other cases of this type, which are more subtle, will be discussed below in this chapter when will discuss dynamic scope (Sect. 4.3.2).

4.2.1 Blocks

Almost all important programming languages today permit the use of blocks, a structuring method for programs introduced by ALGOL60. Block structuring is fundamental to the organisation of the environment.

Definition 4.2 (Block) A block is a textual region of the program, identified by a start sign and an end sign, which can contain declarations local to that region (that is, which appear within the region).

The start- and end-block constructs vary according to the programming language: `begin ... end` for languages in the ALGOL family, braces `{...}` for C and Java, round brackets `(...)` for LISP and its dialects, `let ... in ... end` in ML, etc. Moreover, the exact definition of block in the specific programming language can differ slightly from the one given above. In some cases, for example, one talks about block only when there are local declarations. Often, though, blocks have another important function, that of grouping a series of commands into a syntactic entity which can be considered as a single (composite) command. These distinctions, however, are not relevant as far as we are concerned. We will, therefore, use the definition given above and we distinguish two cases:

Block associated with a procedure This is a block associated with declarations local to a procedure. It corresponds textually to the body of the procedure itself, extended with the declarations of formal parameters.

In-line block This is a block which does not correspond to a declaration of procedure and which can appear (in general) in any position where a command can appear.

4.2.2 Types of Environment

The environment changes during the execution of a program. However, the changes occur generally at two precise times: on the entry and exit of a block. The block can therefore be considered as the construct of least granularity to which a constant environment can be associated.[3]

A block's environment, meaning by this terminology the environment existing when the block is executed, is initially composed of associations between names declared locally to the block itself. In most languages allowing blocks, blocks can be *nested*; that is, the definition of one block can be wholly included in that of another. An example of nested anonymous blocks is shown in Fig. 4.1. The overlapping of blocks so the last open block is not the first block to be closed is never permitted. In other words a sequence of commands of the following kind is not permitted in any language:

```
open block A;
    open block B;
close block A;
    close block B;
```

Different languages vary, then, in the type of nesting they permit. In C, for example, blocks associated with procedures cannot be nested inside each other (that is, there cannot be procedure declarations inside other procedures), while in Pascal and Ada this restriction is not present.[4]

[3] Declarations in the block are evaluated when the block is entered and are visible throughout the block. There exist many exceptions to this rule, some of which will be discussed below.

[4] The reasons for this restriction in C will be made clear in the next chapter when we will have discussed techniques for implementing scope rules.

Block nesting is an important mechanism for structuring the environment. There are mechanisms that allow the declarations local to a block to be visible in blocks nested inside it.

Remaining informal for the time being, we say that a declaration local to a block is *visible* in another block when the association created by such a declaration is present in the second block. Those mechanisms of the language which regulate how and when the declaration is visible are called *visibility rules*. The canonical visibility rule for languages with blocks is well known:

> A declaration local to a block is visible in that block and in all blocks listed within it, unless there is a new declaration of the same name in that same block. In this case, in the block which contains the redefinition the new declaration *hides* the previous one.

In the case in which there is a redefinition, the visibility rule establishes that only the last name declared will be visible in the internal block, while in the exterior one there is a visibility hole. The association for the name declared in the external block will be, in fact, deactivated for the whole of the interior block (containing the new declaration) and will be reactivated on exit from the inner block. Note that there is no visibility from the outside inwards. Every association introduced in the environment local to a block is not active (or rather the name that it defines is not visible) in an exterior block which contains the interior one. Analogously, if we have two blocks at the same nesting level, or if neither of the two contains the other, a name introduced locally in one block is not visible in the other.

The definition just given, although apparently precise, is insufficiently so to establish with precision what the environment will be at an arbitrary point in a program. We will assume this rule for the rest of this section, while the next will be concerned with stating the visibility rules correctly.

In general we can identify three components of an environment, as stated in the following definition.

Definition 4.3 (Type of environment) The environment associated with a block is formed of the following components:

Local environment This is composed of the set of associations for names declared locally to the block. In the case in which the block is for a procedure, the local environment contains also the associations for the formal parameters, given that they can be seen, as far as the environment is concerned, as locally declared variables.

Non-local environment This is the environment formed from the associations for names which are visible from inside a block but which have not been declared locally.

Global environment Finally, there is the environment formed from associations created when the program's execution began. It contains the associations for names which can be used in all blocks forming the program.

The environment local to a block can be determined by considering only the declarations present in the block. We must look outside the block to define the non-local environment. The global environment is part of the non-local environment. Names

```
A:{int a =1;

   B:{int b = 2;
      int c = 2;

      C:{int c =3;
         int d;
         d = a+b+c;
         write(d)
      }

      D:{int e;
         e = a+b+c;
         write(e)
      }
   }
}
```

Fig. 4.2 Nested blocks with different environments

introduced in the local environment can be themselves present in the non-local environment. In such cases, the innermost (local) declaration hides the outermost one.

The visibility rules specify how names declared in external blocks are visible in internal ones. In some cases, it is possible to import names from other, separately defined modules. The associations for these names are part of the global environment.

We will now consider the example in Fig. 4.2, where, for ease of reference, we assume that the blocks can be labelled (as before, we assume also that `write(x)` allows us to print an integer value). The labels behave as comments as far as the execution is concerned.

Let us assume that block `A` is the outermost. It corresponds to the main program. The declaration of the variable `a` introduces an association in the global environment.

Inside block `B` two variables are declared locally (`b` and `c`). The environment for `B` is therefore formed of the local environment, containing the association for the two names (`b` and `c`) and from the global environment containing the association for `a`.

Inside block `C`, 2 local variables (`c` and `d`) are declared. The environment of `C` is therefore formed from the local environment, which contains the association for the two names (`c` and `d`) and from the non-local environment containing the same global environment as above, and also the association for the name `b` which is inherited from the environment of block `B`. Note that the local declaration of `c` in block `C` hides the declaration of `c` present in block `B`. The print command present in block `C` will therefore print the value 6.

In block `D`, finally, we have a local environment containing the association for the local name `e`, the usual global environment and the non-local environment, which, in addition to the association for `a` contains the association for the names `b` and `c` introduced in block `B`. Given that variable `c` has not been internally re-declared, in this case, therefore, the variable declared in block `B` remains visible and the value printed will be 5. Note that the association for the name `d` does not appear in the

environment non-local to D, given that this name is introduced in an exterior block which does not contain D. The visibility rules, indeed, allows only the inheritance of names declared in exterior blocks from interior ones and not vice versa.

4.2.3 Operations on Environments

As we have seen, changes in the environment are produced at entry to and exit from a block. In more detail, during the execution of the program, when a new block is entered, the following modifications are made to the environment:

1. Associations between locally declared names and the corresponding denotable objects are created.
2. Associations with names declared external to and redefined inside the block are deactivated.

Also when the block is exited, the environment is modified as follows:

1. The associations for names declared locally to the block and the objects they denote are destroyed .
2. The associations are reactivated between names that existed external to the block and which were redefined inside it.

More generally, we can identify the following operations on names and on the environment:

Creation of associations between names and denoted object (naming) This is the elaboration of a declaration (or the connection of a formal to an actual parameter) when a new block containing local or parameter declarations is entered.

Reference to a denoted object via its name This is the use of the name (in an expression, in a command, or in any other context). The name is used to access the denoted object.

Deactivation of association between name and denoted object This happens when entering a block in which a new association for that name is created locally. The old association is not destroyed but remains (inactive) in the environment. It will be usable again when the block containing the new association is left.

Reactivation of an association between name and denoted object When leaving block in which a new association for that name is created locally, reactivation occurs. The previous association, which was deactivated on entry to the block, can now be used.

Destruction of an association between name and denoted object (unnaming) This is performed on local associations when the block in which these associations were created is exited. The association is removed from environment and can no longer be used.

Let us explicitly note, however, that any environment contains both active and inactive associations (they correspond to declarations that have been hidden by the

effects of the visibility rules). As far as denotable objects are concerned, the following operations are permitted:

Creation of a denotable object This operation is performed while allocating the storage necessary to contain the object. Sometimes, creation includes also the initialisation of the object.

Access to a denotable object Using the name, and hence the environment, we can access the denotable object and thus access its value (for example, to read the content of a variable). Let us observe that the set of rules which locate the environment has, as its aim, making the association between a name and the object which it refers one-to-one (at a given point in the program and during a given execution).

Modification of a denotable object It is always possible to access the denotable object via a name and then modify its value (for example, by assigning a value to a variable).

Destruction of a denotable object An object can be destroyed by reallocating the memory reserved for it.

In many languages, the operations of creating an association between the name and a denotable object and that of creating a denotable object take place at the same time. This is the case, for example, in a declaration of the form:

```
int x;
```

This declaration introduces into the environment a new association between the name `x` and an integer variable. At the same time, it allocates the memory for the variable.

Yet, this is not always the case and, in general, it is not stated that the *lifetime* of a denotable object, that is the time between the creation of the object and its destruction, coincides with the *lifetime* of the association between name and object. Indeed, a denotable object can have a lifetime that is greater than the association between a name and the object itself, as the case in which a variable is passed to a procedure by reference. The association between the formal parameter and associated variable has a lifetime less than that of the variable itself. More generally, a situation of this type occurs when a temporary name (for example, one local to a block) is introduced for an object which already has a name.

Note that the situation we are considering is *not* that shown in Fig. 4.2. In this case, indeed, the internal declaration of the variable, `c`, does not introduce a new name for an existing object, but introduces a new object (a new variable).

Even if, at first sight, this seems odd, it can also be the case that the lifetime of an association between name and a denoted object is greater than that of the object itself. More precisely, it can be the case that a name allows access to an object which no longer exists. Such an anomalous situation can occur, for example, if we call by reference an existing object and then deallocate the memory for it before the procedure terminates. The formal parameter to the procedure, in this case, will denote an object which no longer exists. A situation of this type, in which it is possible to access an object whose memory has been reallocated, is called a *dangling reference* and is a symptom of an error. We will return to the problem dangling references in Chap. 8, where we will present some methods to handle them.

4.3 Scope Rules

We have seen how, on block entry and exit, the environment can change as a result of the operations for the creation, destruction, activation and the deactivation of associations. These changes are reasonably clear for local environments, but are less clear where the non-local environment is concerned. The visibility rules stated in the previous section indeed lend themselves to at least two different interpretations. Consider for example the following program fragment:

```
A:{int x = 0;

   void fie(){
      x = 1;
   }

   B:{int x;
      fie();
   }

   write(x);
}
```

Which value will be printed? To answer this question, the fundamental problem is knowing which declaration of `x` refers to the non-local occurrence of this name appearing in the assignment in procedure `fie`'s body. On the one hand, we can reasonably think that the value 1 is printed, given that procedure `fie` is defined in block `A` and, therefore, the `x` which appears in the body of the procedure could be that defined on the first line of `A`. On the other hand, however, we can also reason as follows. When we call procedure `fie`, we are in block `B`, so the `x` that we are using in the assignment present in the body of the procedure is the one declared locally to block `B`. This local variable is now no longer visible when we exit block `B`, so `write(x)` refers to the variable `x` declared and initialised to 0 in block `A` and never again modified. Therefore the procedure prints the value 0.

Before the reader tries to find possible tricks in the above reasoning, we must assert that they are both legitimate. The result of the program fragment depends on the *scope rule* being used, as will become clear at the end of the section. The visibility rule that we have stated above establishes that a "declaration local to a block is visible in that block and all the blocks nested within it" but does not specify whether this concept of nesting must be considered in a static (that is based on the text of the program) or dynamic (that is based on the flow of execution) fashion. When the visibility rules, also called *scope rules*, depend only on the syntactic structure of the program, we will talk of a language with *static* or *lexical* scope. When it is influenced also by the control flow at runtime, we are dealing with a language with *dynamic* scope. In the following sections we will analyse these two concepts in detail.

4.3.1 Static Scope

In a language with static (or lexical) scope, the environment in force at any point of the program and at any point during execution depends uniquely on the syntactic structure of the program itself. Such an environment can then be determined completely by the compiler, hence the term "static".

Obviously there can be different static scope rules. One of the simplest, for example, is that of the first version of the Basic language which allowed a single global environment in which it was possible to use only a small number of names (some hundreds) and where declarations were not used.

Much more interesting is the static scope rule that is used in those block-structured languages that allow nesting. This was introduced in ALGOL60 and is retained, with few modifications, by many modern languages, including Ada, Pascal and Java. The rule can be defined as follows:

Definition 4.4 (Static Scope) The static scope rule, or the rule of nearest nested scope, is defined by the following three rules:

(i) The declarations local to a block define the local environment of that block. The local declarations of a block include only those present in the block (usually at the start of the block itself) and not those possibly present in blocks nested inside the block in question.

(ii) If a name is used inside a block, the valid association for this name is the one present in the environment local to the block, if it exists. If no association for the name exists in the environment local to the block, the associations existing in the environment local to the block immediately containing the starting block are considered. If the association is found in this block, it is the valid one, otherwise the search continues with the blocks containing the one with which we started, from the nearest to the furthest. If, during this search, the outermost block is reached and it contains no association for the name, then this association must be looked up in the language's predefined environment. If no association exists here, there is an error.

(iii) A block can be assigned a name, in which case the name is part of the local environment of the block which immediately includes the block to which the name has been assigned. This is the case also for blocks associated with procedures.

It can be immediately seen that this definition corresponds to the informal visibility rules that we have already discussed, suitably completed by a static interpretation of the concept of nesting.

Among the various details of the rule, the fact should not escape us that the declaration of a procedure introduces an association for name of a procedures in the environment local to the block containing the declaration (therefore, because of nesting, the association is also visible in the block which constitutes the body of the procedure, a fact which permits the definition of recursive procedures). The procedure's formal parameters, moreover, are present only in the environment local to

Fig. 4.3 An example of static scope

```
{int x = 0;
 void fie(int n){
    x = n+1;
 }
 fie(3);
 write(x);
    {int x = 0;
     fie(3);
     write(x);
    }
 write(x);
}
```

the procedure and are not visible in the environment which contains the procedure's declaration.

In a language with static scope, we will call the *scope of a declaration* that portion of the program in which the declaration is visible according to Definition 4.4.

We conclude our analysis of static scope by discussing the example in Fig. 4.3. The first and third occurrences of `write` print the value 4, while the second prints the value 0. Note that the formal parameter, `n`, is not visible outside of the body of the procedure.

Static scope allows the determination of all the environments present in a program simply by reading its text. This has two important consequences of a positive nature. First, the programmer has a better understanding of the program, as far as they can connect every occurrence of a name to its correct declaration by observing the textual structure of the program and without having to simulate its execution. Moreover, this connection can also be made by the compiler which can therefore determine each and every use of a name. This makes it possible, at compile time, to perform a great number of correctness tests using the information contained in types; it can also perform considerable number of code optimisations. For example, if the compiler knows (using declarations) that the variable `x` which occurs in a block is an integer variable, it will signal an error in the case in which a character is assigned to this variable. Similarly, if the compiler knows that the constant `fie` is associated with the value 10, it can substitute the value 10 for every reference to `fie`, so avoiding having to arrange for this operation to be performed at run-time, therefore, it updates the code. If, instead, the correct declaration for `x` and for `fie` can be determined only at execution time, it is clear that these checks and this optimisation are not possible as compilation time.

Note that, even with the static scope rules, the compiler cannot know in general which memory location will be assigned to the variable with name `x` nor what its value might be, given that this information depends on the execution of the program. When using the static scope rule, moreover, the compiler is in possession of some important information about the storage of variables (in particular it knows the offsets relative to a fixed position, as we will see in detail in the next chapter), that it uses to compile efficient accesses to variables. As we will see, this information is not available using dynamic scope, which, therefore, leads to less efficient exe-

Fig. 4.4 An example of dynamic scope

```
{const x = 0;
 void fie(){
    write(x);
 }
 void foo(){
    const x =1;
    fie();
 }
 foo();
}
```

cution. For these reasons, most current languages (for example ALGOL, Pascal, C, C++, Ada, scheme and Java) use some form of static scope.

4.3.2 Dynamic Scope

Dynamic scope was introduced in some languages, such as, for example, APL, LISP (some versions), SNOBOL and PERL, mainly to simplify runtime environment management. In fact it is true that static scope imposes a fairly complicated runtime regime because the various non-local environments are involved in a way that does not reflect the normal flow of activation and deactivation of blocks. We seek to understand the problem by considering the fragment of code in Fig. 4.4 and following its execution. First, the outermost block is entered and the association between the name `x` and the constant 0 is created, as well as that between the names `fie`, `foo` and associated procedures (as we said above, this association can be performed by the compiler). Next the call to the procedure `foo` is executed and control enters the block associated with the procedure. In this block, the link between the name `x` and the constant 1 is created; then the call to procedure `fie` is executed which causes entry to a new block (the one for the latter procedure). It is at this point that the command `write(x)` is executed and given that `x` is not a name local to the block introduced by the procedure `fie`, the association for the name `x` must be looked up in outer blocks. However, according to the rules of static scope, as presented in the last section, the first external block in which to look for the association for `x` is not the last block to be activated (it is the one for procedure `foo` in our example); such an external block depends on the structure of the program. In this case, then, the correct association for the name `x` used by `fie` is the one located in the first block and consequently the value 0 is printed. The block belonging to procedure `foo`, even if it contains a declaration for `x` and is still active, is not considered.

Generalising from the previous example, we can say that, under static scope, the sequence of blocks that must be considered to resolve references to non-local names is different from the sequence of blocks that is opened and exited during the program's normal flow of control. The opening and closing can be handled in a natural manner using the LIFO (Last In First Out) discipline, that is using a stack. The sequence of blocks that need examining to implement static scope depends on

Fig. 4.5 Another example of dynamic scope

```
{const x = 0;
 void fie(){
    write(x);
 }
 void foo(){
    const x = 1;
      {const x = 2;
      }
    foo();
 }
 foo();
}
```

the syntactic structure of the program and being able to handle it correctly at runtime depends upon the use of additional data structures, as we will see in detail in the next chapter.

To simplify the management of the runtime environment some languages use then the *dynamic scope* rule. This rule determines the associations between names and denoted objects using the backward execution of the program. In such languages, resolving non-local names requires only a stack dedicated to handling blocks at runtime. In our example, this means that, when the command `write(x)` is executed, the association for the name and `x` is sought in the second block (relative to procedure `foo`), rather than in the first block, because this is the last block, different from the current one, in which we entered and from which we have not yet exited. Given that in the second block, we find the declaration `const x = 1`, in the case of dynamic scope the preceding program prints the value 1.

With more precision the dynamic scope rule, also called the rule of the most recent association, can be defined as follows.

Definition 4.5 (Dynamic Scope) According to the rule of dynamic scope, the valid association for a name X, at any point P of a program, is the most recent (in the temporal sense) association created for X which is still active when control flow arrives at P.

It is appropriate to observe that this rule does not contradict the informal visibility rule that we stated in Sect. 4.2.2. A moment's reflection shows, indeed, that the dynamic scope rule expresses nothing other than the same visibility rule but the concept of block nesting is understood in a dynamic sense.

Let us again note how the difference between static and dynamic scope enters only into the determination of the environment which is currently *not local and not global*. For the local and global environment, the two rules coincide.

Let us conclude by discussing the example in Fig. 4.5. In a language with dynamic scope, the code prints the value 1 because when the command `write(x)` is executed, the last association created for `x` *which is still active* associates `x` with 1. The association which associates `x` to 2, even if it is the most recent one to be created, is no longer active when procedure `fie` is executed and is therefore not considered.

Note that dynamic scope allows the modification of the behaviour of procedure or subprogram without using explicit parameters but only by redefining some of the non-local variables used. To explain this point, assume that we have a procedure `visualise(text)` which can visualise text in various colours, according to the value of the non-local variable `colour`. If we assume that in the majority of cases, the procedure visualises text in black, it is natural to assume that we do not wish to introduce another parameter to the procedure in order to determine the colour. If the language uses dynamic scope, in the case in which the procedure has to visualise a text in red, it will be enough to introduce the declaration for the variable `colour` before the *call* of the procedure. We can therefore write:

```
...
{var colour = red;
 visualise(head);
}
```

Then the call to procedure *visualise* now will use the colour red, because of the effect of dynamic scope.

This flexibility of dynamic scope, is, on the one hand, advantageous, yet, on the other, it often makes programs more difficult to read, given that the same procedure call, in conditions differing by only one non-local variable can produce different results. If the variable (`colour` in our example) is modified in an area of program that is distant from the procedure call, understanding what has happened will probably turn out to be difficult.

For this reason, as well as for low runtime efficiency, dynamic scope remains little used in modern general-purpose languages, which instead use the static scope rule.

4.3.3 Some Scope Problems

In this section, we will discuss some questions about static scope. The major differences between the rules for static scope in various languages are based on where declarations can be introduced and what will be the exact visibility of the local variables. The scope rules just introduced, lend themselves to more than one interpretation and, in some cases, they can also be the cause of anomalous behaviour. Let us discuss here some different and important situations that can happen.

Let us, first, take the case of Pascal in which the static scope rule that we have already seen is extended with the following additional rules:

1. Declarations can appear only at the start of a block.
2. The scope of a name extends from the start to the end of the block in which the declaration of the name itself appears (excluding possible holes in scope) independent of the position of that declaration.

3. All names not predefined by the language must be declared before use.

It is an error therefore to write:

```
begin
   const fie = value;
   const value = 0;
...
end
```

This is because `value` is used before its definition. It could be assumed that such a fragment might be correct if it were inserted into a block already containing a definition of `value`. Also in this case, though, an error is produced. In fact, let us write:

```
begin
   const value = 1;
   procedure foo;
       const fie = value;
       const value = 0;
       ...
       begin
       ...
       end
...
end
```

Now, the rules that we have seen tell us that the declaration of the procedure `foo` introduces an internal block in which the local declaration of *value* (which initialises to the constant to 0) covers the external declaration (which initialises the constant to 1). Therefore the name `value` appearing in the declaration

```
const fie = value;
```

must refer to the declaration

```
const value = 0;
```

in the internal block. However this declaration occurs after the use of the name, therefore contravening Rule 3. In such a case, therefore, the more correct behaviour for a Pascal compiler is to raise a static semantic error as soon as it has analysed the declaration of `fie`. Some compilers, though, assign to `fie` the value 1; this is clearly incorrect for the reason that it violates the visibility rule.

To avoid this type of problem, some languages with static scope, such as C and Ada, limit the scope of declarations to that portion of the block between the point at which the declarations occur and the end of the book itself (excluding, as usual, holes in scope).

In these languages, therefore, we do not encounter the above problem where the name `value` appearing in the declaration

Fig. 4.6 A list

```
const fie = value;
```

would refer to the declaration in the external block, given that the name internally declared is no longer visible. However, also in these languages names must be declared before being used. So also in this case the following declarations

```
const fie = value;
const value = 0;
```

produce an error if `value` is not declared in an external block.

Rule 3, which prescribes declaration *before* use, is particularly burdensome in some cases. Indeed, it forbids the definition of recursive types or mutually recursive procedures.

Let us assume, for example, that we want to define a data type like a list which, as we know, is a variable-length data structure formed of a (possibly empty) ordered sequence of elements of some type and in which it is possible to add or remove elements. In a list, only the first element can be directly accessed (to access an arbitrary element, it is necessary to traverse the list sequentially). A list, as shown in Fig. 4.6, can be implemented using a sequence of elements, each of which is formed of two fields: the first will contain the information we want to store (for example, an integer); the second will contain a pointer to the next element of the list, if it exists, otherwise it stores the value `nil` if the list has ended. We can access the list using the pointer of the first element of the list (which is usually called the *head*). In Pascal, we can define the list type as follows:

```
type list = ^element;
type element = record
                 information: integer;
                 successor: list
                 end
```

Here `^T` denotes the type of pointers to objects of type `T`. A value of type `list` is a pointer to an arbitrary element of the list. The value of type `element` corresponds to an element of the list, which is composed of an integer and by a field the type `list` which allows it to connect to the next element. This declaration is incorrect according to Rule 3. In fact, whatever the order of the declarations of list and element may be, it can be seen that we use a name which has not yet been defined. This problem is resolved in Pascal by relaxing Rule 3. For data of a pointer type, and only for them, is it permitted to refer to a name that has not yet been declared. The declaration given above for list are therefore correct in Pascal.

In the case of C and Ada, on the other hand, the analogues of the previous declarations are not permitted. To specify mutually recursive types, it is necessary to

use *incomplete* declarations which introduce a name which will later be specified in full. For example, in Ada we can write:

```
type element;
type list is access element;
type element is record
                information: integer;
                successor: list;
                end record;
```

This solves the problem of using a name before its declaration.

The problem presents itself in the analogous case of the definition of mutually recursive procedures. Here, Pascal uses incomplete definitions. If procedure `fie` is to be defined in terms of procedure `foo` and vice versa, in Pascal, we must write:

```
procedure fie(A:integer); forward;
procedure foo(B: integer);
    begin
    ...
    fie(3);
    ...
    end
procedure fie;
    begin
    ...
    foo(4);
    ...
    end
```

In the case of function names, it is strange to observe that C, on the other hand, allows the use of an identifier before its declaration. The declaration of mutually recursive functions does not require any special treatment.

Rule 3 is relaxed in as many ways as there are programming languages. Java, for example, allows a declaration to appear at any point in a block. If the declaration is of a variable, the scope of the name being declared extends from the point of declaration to the end of the block (excluding possible holes in scope). If, on the other hand, the declaration refers to a member of a class (either a field or a method), it is visible in all classes in which it appears, independent of the order in which the declarations occur.

4.4 Chapter Summary

In this chapter we have seen the primary aspects of name handling in high-level languages. The presence of the environment, that is of a set of associations between names and the objects they represent, constitute one of the principal characteristics that differentiate high-level from low-level languages. Given the lack of environment in low-level languages, name management, as well as that of the environment,

is an important aspect in the implementation of a high-level language. We will see implementation aspects of name management in the next chapter. Here we are interested in those aspects which must be known to every user (programmer) of a high-level language so that they fully understand the significance of names and, therefore, of the behaviour of programs.

In particular, we have analysed the aspects that are listed below:

- *The concept of denotable objects.* These are the objects to which names can be given. Denotable objects vary according to the language under consideration, even if some categories of object (for example, variables) are fairly general.
- *Environment.* The set of associations existing at runtime between names and denotable objects.
- *Blocks.* In-line or associated with procedures, these are the fundamental construct for structuring the environment and for the definition of visibility rules.
- *Environment Types.* These are the three components which at any time characterise the overall environment: local environment, global environment and non-local environment.
- *Operations on Environments.* Associations present in the environment in addition to being created and destroyed, can also be deactivated, a re-activated and, clearly, can be used.
- *Scope Rules.* Those rules which, in every language, determine the visibility of names.
- *Static Scope.* The kind of scope rule typically used by the most important programming languages.
- *Dynamic Scope.* The scope rule that is easiest to implement. Used today in few languages.

In an informal fashion, we can say that the rules which define the environment are composed of rules for visibility between blocks and of scope rules, which characterise how the non-local environment is determined. In the presence of procedures, the rules we have given are not yet sufficient to define the concept of environment. We will return to this issue in Chap. 7 (in particular at the end of Sect. 7.2.1).

4.5 Bibliographical Notes

General texts on programming languages, such as for example [2, 3] and [4], treat the problems seen in this chapter, even if they are almost always viewed in the context of the implementation. For reasons of clarity of exposition, we have chosen to consider in this chapter only the semantic rules for name handling and the environment, while we will consider their implementation in the next chapter.

For the rules used by individual languages, it is necessary to refer to the specific manuals, some of which are mentioned in bibliographical notes for Chap. 13, even if at times, as we have discussed in Sect. 4.3.3, not all the details are adequately clarified.

The discussion in Sect. 4.3.3 draws on material from [1].

4.6 Exercises

Exercises 6–13, while really being centred on issues relating to scope, presuppose knowledge of parameter passing which we will discuss in Chap. 7.

1. Consider the following program fragment written in a pseudo-language which uses static scope and where the primitive `read(Y)` allows the reading of the variable `Y` from standard input.

```
  ...
  int X = 0;
  int Y;
void fie(){
    X++;
}
void foo(){
    X++;
    fie();
}
read(Y);
if Y > 0{int X = 5;
         foo();}
else foo();
write(X);
```

   State what the printed values are.
2. Consider the following program fragment written in a pseudo-language that uses dynamic scope.

```
 ...
int X;
X = 1;
int Y;
void fie() {
   foo();
   X = 0;
}
void foo(){
   int X;
   X = 5;
}
read(Y);
if Y > 0{int X;
         X = 4;
         fie();}
else     fie();
write(X);
```

   State which is (or are) the printed values.
3. Consider the following code fragment in which there are gaps indicated by (*) and (**). Provide code to insert in these positions in such a way that:

a. If the language being used employs static scope, the two calls to the procedure `foo` assign the same value to `x`.
b. If the language being used employs dynamic scope, the two calls to the procedure `foo` assign different values to `x`.

The function `foo` must be appropriately declared at (*).

```
{int i;
(*)
for (i=0; i<=1; i++){
   int x;
   (**)
   x= foo();
   }
}
```

4. Provide an example of a denotable object whose life is longer than that of the references (names, pointers, etc.) to it.
5. Provide an example of a connection between a name and a denotable object whose life is longer than that of the object itself.
6. Say what will be printed by the following code fragment written in a pseudo-language which uses static scope; the parameters are passed by a value.

```
{int x = 2;
 int fie(int y){
     x = x + y;
  }

  {int x = 5;
   fie(x);
   write(x);
  }
 write(x);
}
```

7. Say what is printed by the code in the previous exercise if it uses dynamic scope and call by reference.
8. State what is printed by the following fragment of code written in a pseudo-language which uses static scope and passes parameters by reference.

```
{int x = 2;
 void fie(reference int y){
     x = x + y;
     y = y + 1;
 }
 {int x = 5;
  int y = 5;
  fie(x);
  write(x);
 }
 write(x);
}
```

9. State what will be printed by the following code fragment written in a pseudo-language which uses static scope and passes its parameters by value (a command of the form `foo(w++)` passes the current value of `w` to `foo` and then increments it by one).

```
{int x = 2;
 void fie(value int y){
     x = x + y;
 }
 {int x = 5;
  fie(x++);
  write(x);
 }
 write(x);
}
```

10. State what will be printed by the following fragment of code written in a pseudo-language which uses static scope and call by name.

```
{int x = 2;
 void fie(name int y){
    x = x + y;
 }
 {int x = 5;
    {int x = 7
    }
  fie(x++);
  write(x);
 }
 write(x);
}
```

11. State what will be printed by the following code written in a pseudo-language which uses dynamic scope and call by reference.

```
{int x = 1;
 int y = 1;
 void fie(reference int z){
       z  = x + y + z;
 }
 {int y = 3;
   {int x = 3
   }
   fie(y);
   write(y);
 }
 write(y);
}
```

12. State what will be printed by the following fragment of code written in a pseudo-language which uses static scope and call by reference.

```
{int x = 0;
 int A(reference int y) {
         int x =2;
         y=y+1;
         return B(y)+x;
 }
 int B(reference int y){
        int C(reference int y){
            int x = 3;
            return A(y)+x+y;
        }
        if (y==1) return C(x)+y;
        else return x+y;
 }
 write (A(x));
}
```

13. Consider the following fragment of code in a language with static scope and parameter passing both by value and by name:

```
{int z= 0;
 int Omega(){
    return Omega();
 }
 int foo(int x, int y){
    if (x==0) return x;
    else return x+y;
 }
 write(foo(z, Omega()+z));
}
```

(i) State what will be the result of the execution of this fragment in the case in which the parameters to `foo` are passed by *name*.
(ii) State what will be the result of the execution of this fragment in the case in which the parameters to `foo` are passed by *value*.

References

1. R. Cailliau. How to avoid getting schlonked by Pascal. *SIGPLAN Not.*, 17(12):31–40, 1982. doi:10.1145/988164.988167.
2. T.W. Pratt and M.V. Zelkowitz. *Programming Languages: Design and Implementation*. Prentice-Hall, New York, 2001.
3. M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, San Mateo, 2000.
4. R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, Reading, 1996.