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

Fig. 4.2 Nested blocks with different environments

```
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)
      }
   }
}
```

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

Fig. 4.4 An example of dynamic scope

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

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

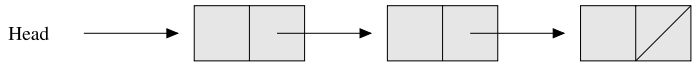


Fig. 4.6 A list

```
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.

Chapter 5
Memory Management

An important component of an abstract machine's interpreter is the one dealing with memory management. If this component can be extremely simple in a physical machine, memory management in an abstract machine for a high-level language is fairly complicated and can employ a range of techniques. We will see both static and dynamic management and will examine activation records, the system stack and the heap. One section in particular is dedicated to the data structures and mechanisms used to implement scope rules.

Conceptually, garbage-collection techniques, techniques for the automatic recovery of memory allocated in a heap, are included in memory management. However, to make the presentation more coherent, these techniques will be explained in Sect. 8.12, after having dealt with data types and pointers.

5.1 Techniques for Memory Management

As we said in Chap. 1, memory management is one of the functions of the interpreter associated with an abstract machine. This functionality manages the allocation of memory for programs and for data, that is determines how they must be arranged in memory, how much time they may remain and which auxiliary structures are required to fetch information from memory.

In the case of a low-level abstract machine, the hardware, for example, memory management is very simple and can be entirely *static*. Before execution of the program begins, machine language program and its associated data is loaded into an appropriate area of memory, where it remains until its execution ends.

In the case of a high-level language, matters are, for various reasons, more complicated. First of all, if the language permits recursion, static allocation is insufficient.[1] In fact, while we can statically establish the maximum number of active procedures at any point during execution in the case of languages without recursion,

[1] We will see below an exception to this general principle. This is the case of so-called tail recursion.

M. Gabbrielli, S. Martini, *Programming Languages: Principles and Paradigms,*
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-84882-914-5_5, © Springer-Verlag London Limited 2010

when we have recursive procedures this is no longer true because the number of simultaneously active procedure calls can depend on the parameters of the procedures or, generally, on information available only at runtime.

Example 5.1 Consider the following fragment:

```
int fib (int n) {
   if (n == 0) return 1;
   else if (n == 1) return 1;
        else return fib(n-1) + fib(n-2);
}
```

which, if called with an argument n, computes (in a very inefficient way) the value of the nth Fibonacci number. Let us recall that Fibonacci numbers are the terms of the sequence[2] defined inductively as follows: $Fib(0) = Fib(1) = 1$; $Fib(n) = Fib(n-1) + Fib(n-2)$, for $n > 1$. It is clear that the number of active calls to `Fib` depends, other than on the point of execution, on the value of the argument, n. Using a simple recurrence relation, it can be verified that the number, $C(n)$, of calls to `Fib` necessary to calculate the value of the term $Fib(n)$ (and, therefore, the simultaneously active calls) is exactly equal to this value. From a simple inspection of the code, indeed, it can be seen that $C(n) = 1$ for $n = 0$ and $n = 1$, while $C(n) = C(n-1) + C(n-2)$ for $n > 1$. It is known that the Fibonacci numbers grow exponentially, so the number of calls to `fib` is of the order of $O(2^n)$.

Given that every procedure call requires its own memory space to store parameters, intermediate results, return addresses, and so on, in the presence of recursive procedures, static allocation of memory is no longer sufficient and we have to allow *dynamic* memory allocation and deallocation operations, which are performed during the execution of the program. Such dynamic memory processing can be implemented in a natural fashion using a *stack* for procedure (or in-line block) activations since they follow a LIFO (Last In First Out) policy—the last procedure called (or the last block entered) will be the first to be exited.

There are, however, other cases which require dynamic memory management for which a stack is not enough. These are cases in which the language allows explicit memory allocation and deallocation operations, as happens, for example, in C with the `malloc` and `free` commands. In these cases, given that the allocation operation (`malloc`) and the one for deallocation (`free`) can be alternated in any order whatsoever, it is not possible to use a stack to manage the memory and, as we will better see as the chapter unfolds, a particular memory structure called a *heap* is used.

[2] The sequence takes the name of the Pisan mathematician of the same name, known also as Leonardo da Pisa (ca., 1175–1250), who seems to have encountered the sequence by studying the increase in a population of rabbits. For information on inductive definition, see the box on page 153.

5.2 Static Memory Management

Static memory management is performed by the complier before execution starts. Statically allocated memory objects reside in a fixed zone of memory (which is determined by the compiler) and they remain there for the entire duration of the program's execution. Typical elements for which it is possible statically to allocate memory are *global variables*. These indeed can be stored in a memory area that is fixed before execution begins because they are visible throughout the program. The *object code instructions* produced by the compiler can be considered another kind of static object, given that normally they do not change during the execution of the program, so in this case also memory will be allocated by the compiler. *Constants* are other elements that can be handled statically (in the case in which their values do not depend on other values which are unknown at compile time). Finally, various *compiler-generated tables*, necessary for the runtime support of the language (for example, for handling names, for type checking, for garbage collection) are stored in reserved areas allocated by the compiler.

In the case in which the language does not support recursion, as we have anticipated, it is possible statically to handle the memory for other components of the language. Substantially, this is done by statically associating an area of memory in which is stored the information local to the procedure with each procedure (or subroutine)[3] itself. This information is composed of local variables, possible parameters of the procedure (containing both arguments and results), the return address (or the address to which control must pass when the procedure terminates), possible temporary values used in complex calculations and various pieces of "bookkeeping" information (saved register values, information for debugging and so on).

The situation of a language with only static memory allocation is shown in Fig. 5.1.

It will be noted that successive calls to the same procedure share the same memory areas. This is correct because, in the absence of recursion, there cannot be two different calls to the same procedure that are active at the same time.

5.3 Dynamic Memory Management Using Stacks

Most modern programming languages allow block structuring of programs.[4]

Blocks, whether in-line or associated with procedures, are entered and left using the LIFO scheme. When a block A is entered, and then a block B is entered, before leaving A, it is necessary to leave B. It is therefore natural to manage the memory

[3] It would be more correct to speak of subroutines because this was the term used in languages that used static memory allocation, such as, for example, the first versions of FORTRAN from the 1960s and 70s.

[4] We will see below that important languages, such as C, though, do not offer the full potential of this mechanism, in that they do not permit the declaration of local procedures and functions in nested blocks.

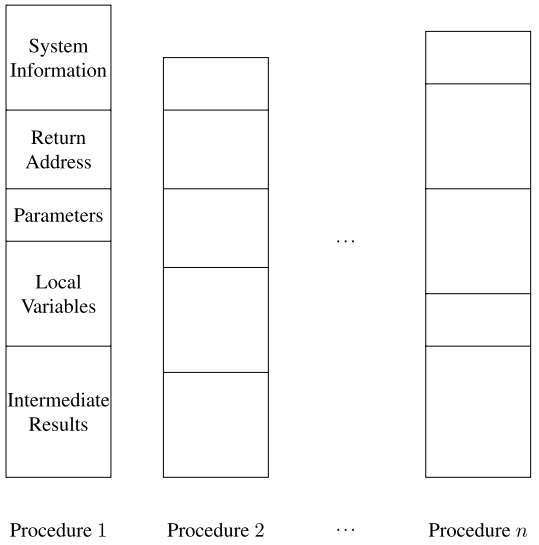


Fig. 5.1 Static memory management

space required to store the information local to each block using a stack. We will see an example.

Example 5.2 Let us consider the following program:

```
A:{int a = 1;
   int b = 0;

   B:{int c = 3;
      int b = 3;
     }
   b=a+1;
  }
```

At runtime, when block A is entered, a push operation allocates a space large enough to hold the variables `a` and `b`, as shown in Fig. 5.2. When block B is entered, we have to allocate a new space on the stack for the variables `c` and `b` (recall that the inner variable `b` is different from the outer one) and therefore the situation, after this second allocation, is that shown in Fig. 5.3. When block B exits, on the other hand, it is necessary to perform a pop operation to deallocate the space that had been reserved for the block from the stack. The situation after such a deallocation and after the assignment is shown in Fig. 5.4. Analogously, when block A exits, it will be necessary to perform another pop to deallocate the space for A as well.

The case of procedures is analogous and we consider it in Sect. 5.3.2.

The memory space, allocated on the stack, dedicated to an in-line block or to an activation of a procedure is called the *activation record*, or *frame*. Note that an acti-

Fig. 5.2 Allocation of an activation record for block A in Example 5.2

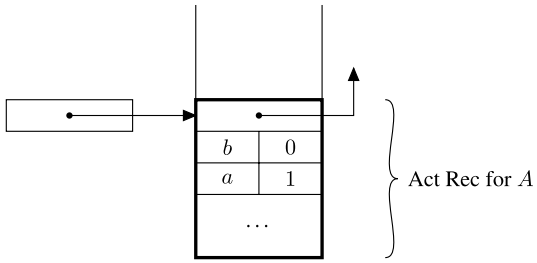


Fig. 5.3 Allocation of activation records for blocks A and B in Example 5.2

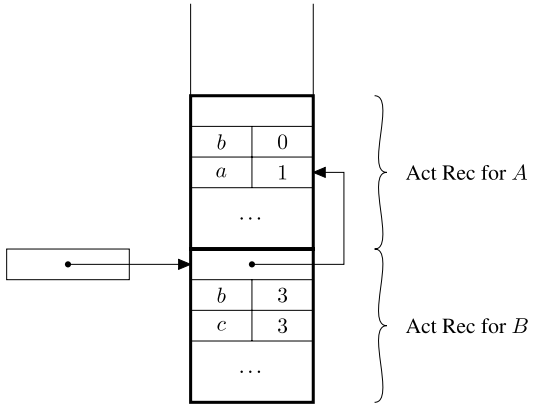
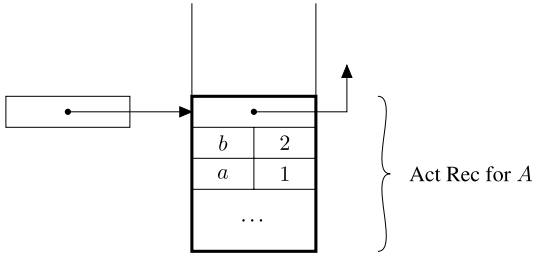


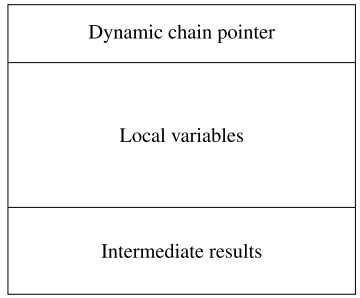
Fig. 5.4 Organisation after the execution of the assignment in Example 5.2



vation record is associated with a specific activation of a procedure (one is created when the procedure is called) and not with the declaration of a procedure. The values that must be stored in an activation record (local variables, temporary variables, etc.) are indeed different for the different calls on the same procedure.

The stack on which activation records are stored is called the *runtime* (*or system*) *stack*.

Fig. 5.5 Organisation of an activation record for an in-line block



It should finally be noted, that to improve the use of runtime memory, dynamic memory management is sometimes also used to implement languages that do not support recursion. If the average number of simultaneously active calls to the same procedure is less than the number of procedures declared in the program, using a stack will save space, for there will be no need to allocate a memory area for each declared procedure, as must be done in the case of entirely static management.

5.3.1 *Activation Records for In-line Blocks*

The structure of a generic activation record for an in-line block is shown in Fig. 5.5. The various sectors of the activation record contain the following information:

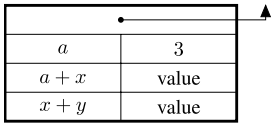
Intermediate results When calculations must be performed, it can be necessary to store intermediate results, even if the programmer does not assign an explicit name to them. For example, the activation record for the block:

```
{int a =3;
 b= (a+x)/ (x+y);}
```

could take the form shown in Fig. 5.6, where the intermediate results `(a+x)` and `(x+y)` are explicitly stored before the division is performed. The need to store intermediate results on the stack depends on the compiler being used and on the architecture to which one is compiling. On many architectures they can be stored in registers.

Local variables Local variables which are declared inside blocks, must be stored in a memory space whose size will depend on the number and type of the variables. This information in general is recorded by the compiler which therefore will be able to determine the size of this part of the activation record. In some cases, however, there can be declarations which depend on values recorded only at runtime (this is, for example, the case for dynamic arrays, which are present in some languages, whose dimensions depend on variables which are only instantiated at execution time). In these cases, the activation record also contains a variable-length

Fig. 5.6 An activation record with space for intermediate results



part which is defined at runtime. We will examine this in detail in Chap. 8 when discussing arrays.

Dynamic chain pointer This field stores a pointer to the previous activation record on the stack (or to the last activation record created). This information is necessary because, in general, activation records have different sizes. Some authors call this pointer the *dynamic link* or *control link*. The set of links implemented by these pointers is called the *dynamic chain*.

5.3.2 Activation Records for Procedures

The case of procedures and functions[5] is analogous to that of in-line blocks but with some additional complications due to the fact that, when a procedure is activated, it is necessary to store a greater amount of information to manage correctly the control flow. The structure of a generic activation record for a procedure is shown in Fig. 5.7. Recall that a function, unlike a procedure, returns a value to the caller when it terminates its execution. Activation records for the two cases are therefore identical with the exception that, for functions, the activation record must also keep tabs on the memory location in which the function stores its return value.

Let us now look in detail at the various fields of an activation record:

Intermediate results, local variables, dynamic chain pointer The same as for in-line blocks.

Static chain pointer This stores the information needed to implement the static scope rules described in Sect. 5.5.1.

Return address Contains the address of the first instruction to execute after the call to the current procedure/function has terminated execution.

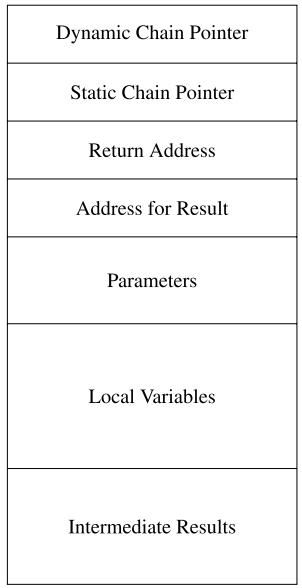
Returned result Present only in functions. Contains the address of the memory location where the subprogram stores the value to be returned by the function when it terminates. This memory location is inside the caller's activation record.

Parameters The values of actual parameters used to call the procedure or function are stored here.

The organisation of the different fields of the activation record varies from implementation to implementation. The dynamic chain pointer and, in general, every

[5]Here and below, we will almost always use the terms "function" and "procedure" as synonyms. Although there are is no agreement between authors, the term "procedure" should denote a subprogram which does not directly return a value, while a function is a subprogram that returns one.

Fig. 5.7 Structure of the activation record for a procedure



pointer to an activation record, points to a fixed (usually central) area of the activation record. The addresses of the different fields are obtained, therefore, by adding a negative or positive offset to the value of the pointer.

Variable names are not normally stored in activation records and the compiler substitutes references to local variables for addresses relative to a fixed position in (i.e., an offset into) the activation record for the block in which the variables are declared. This is possible because the position of a declaration inside a block is fixed statically and the compiler can therefore associate every local variable with an exact position inside the activation record.

In the case of references to non-local variables, also, as we will see when we discuss scope rules, it is possible to use mechanisms that avoid storing names and therefore avoid having to perform a runtime name-based search through the activation-record stack in order to resolve a reference.

Finally, modern compilers often optimise the code they produce and save some information in registers instead of in the activation record. For simplicity, in this book, we will not consider these optimisations. In any case for greater clarity, in the examples, we will assume that variable names are stored in activation records.

To conclude, let us note that all the observations that we have made about variable names, their accessibility and storage in activation records, can be extended to other kinds of denotable object.

5.3.3 Stack Management

Figure 5.8 shows the structure of a system stack which we assume growing downwards (the direction of stack growth varies according to the implementation). As shown in the figure, an external pointer to the stack points to the last activation record on the stack (pointing to a predetermined area of the activation record which is used as a base for calculating the offsets used to access local names). This pointer, which we call the activation record pointer, is also called the frame or current environment pointer (because environments are implemented using activation records). In the figure, we have also indicated where the first free location is. This second pointer, used in some implementations, can, in principle, also be omitted if the activation-record pointer always points to a position that is at a pre-defined distance from the start of the free area on the stack.

Activation records are stored on and removed from the stack at runtime. When a block is entered or a procedure is called, the associated activation record is pushed onto the stack; it is later removed from the stack when the block is exited or when the procedure terminates.

The runtime management of the system stack is implemented by code fragments which the compiler (or interpreter) inserts immediately before and after the call to a procedure or before the start and after the end of a block.

Let us examine in detail what happens in the case of procedures, given that the case of in-line blocks is only a simplification.

First of all, let us clarify the terminology that we are using. We use "caller" and "callee" to indicate, respectively, the program or procedure that performs a call (of a procedure) and the procedure that has been called.

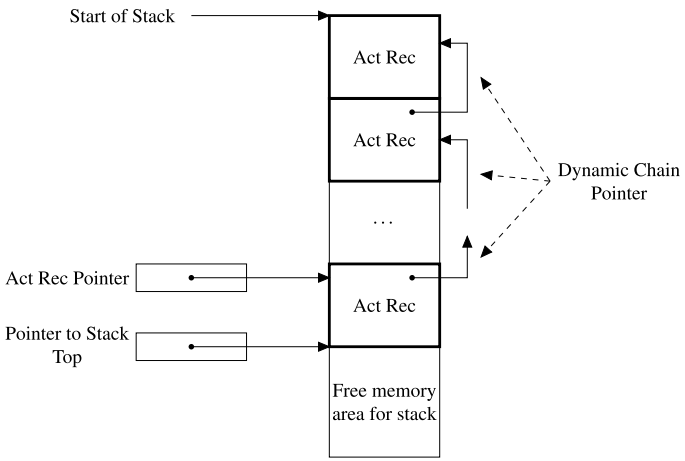


Fig. 5.8 The stack of activation records

Stack management is performed both by the caller and by the callee. To do this, as well as handling other control information, a piece of code called the *calling sequence* is inserted into the caller to be executed, in part, immediately before the procedure call. The remainder of this code is executed immediately after the call. In addition, in the callee two pieces of code are added: a *prologue*, to be executed immediately after the call, and an *epilogue*, which is executed when the procedure ends execution. These three code fragments manage the different operations needed to handle activation records and correctly implement a procedure call. The exact division of what the caller and callee do depends, as usual, on the compiler and on the specific implementation under consideration. Moreover, to optimise the size of code produced, it is preferable that the larger part of the activity is given to the callee, since the code is added only once (to the code associated with the declaration of the call) instead of many times (to the code associated with different calls). Without therefore further specifying the division of activities, they consist of the following tasks:

Modification of program counter This is clearly necessary to pass control to the called procedure. The old value (incremented) must be saved to maintain the return address.

Allocation of stack space The space for the new activation record must be preallocated and therefore the pointer to the first free location on the stack must be updated as a consequence.

Modification of activation record pointer The pointer must point to the new activation record for the called procedure; the activation record will have been pushed onto the stack.

Parameter passing This activity is usually performed by the caller, given that different calls of the same procedure can have different parameters.

Register save Values for control processing, typically stored in registers, must be saved. This is the case, for example, with the old activation record pointer which is saved as a pointer in the dynamic chain.

Execution of initialisation code Some languages require explicit constructs to initialise some items stored in the new activation record.

When control *returns to the calling program*, i.e. when the called procedure terminates, the *epilogue* (in the called routine) and the *calling sequence* (in the caller) must perform the following operations:

Update of program counter This is necessary to return control to the caller.

Value return The values of parameters which pass information from the caller to the called procedure, or the value calculated by the function, must be stored in appropriate locations usually present in the caller's activation record and accessible to the activation record of the called procedure.

Return of registers The value of previously saved registers must be restored. In particular, the old value of the activation record pointer must be restored.

Execution of finalisation code Some languages require the execution of appropriate finalisation code before any local objects can be destroyed.

Deallocation of stack space The activation record of the procedure which has terminated must be removed from the stack. The pointer to (the first free position on) the stack must be modified as a result.

It should be noted that in the above description, we have omitted the handling of the data structures necessary for the implementation of scope rules. This will be considered in detail in Sect. 5.5 of this chapter.

5.4 Dynamic Management Using a Heap

In the case in which the language includes explicit commands for memory allocation, as for example do C and Pascal, management using just the stack is insufficient. Consider for example the following C fragment:

```
int *p, *q; /* p,q  NULL pointers to integers */
p =  malloc (sizeof (int));
        /* allocates the memory pointed to by p */
q =  malloc (sizeof (int));
        /* allocates the memory pointed to by  q */
*p = 0;     /* dereferences and assigns */
*q = 1;     /* dereferences and assigns */
free(p);    /* deallocates the memory pointed to by p */
free(q);    /* deallocates the memory pointed to by q */
```

Given that the memory deallocation operations are performed in the same order as allocations (first `p`, then `q`), the memory cannot be allocated in LIFO order.

To manage explicit memory allocations, which can happen at any time, a particular area of memory, called a *heap*, is used. Note that this term is used in computing also to mean a particular type of data structure which is representable using a binary tree or a vector, used to implement efficiently priorities (and used also in the "heap sort" sorting algorithm, where the term "heap" was originally introduced). The definition of heap that we use here has nothing to do with this data structure. In the programming language jargon, a heap is simply an area of memory in which blocks of memory can be allocated and deallocated relatively freely.

Heap management methods fall into two main categories according to whether the memory blocks are considered to be of *fixed* or *variable* length.

5.4.1 Fixed-Length Blocks

In this case, the heap is divided into a certain number of elements, or blocks, of fairly small fixed length, linked into a list structure called the *free list*, as shown in Fig. 5.9. At runtime, when an operation requires the allocation of a memory block from the heap (for example using the `malloc` command), the first element of the free list is removed from the list, the pointer to this element is returned to the operation that

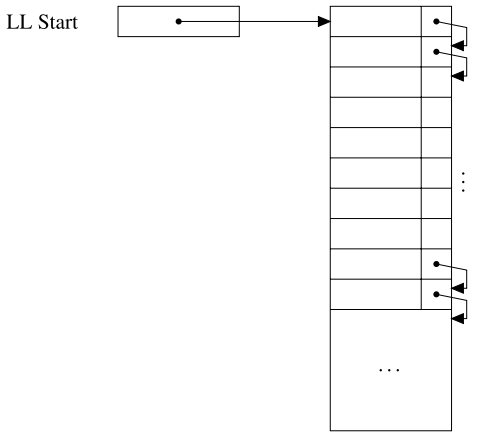


Fig. 5.9 Free list in a heap with fixed-size blocks

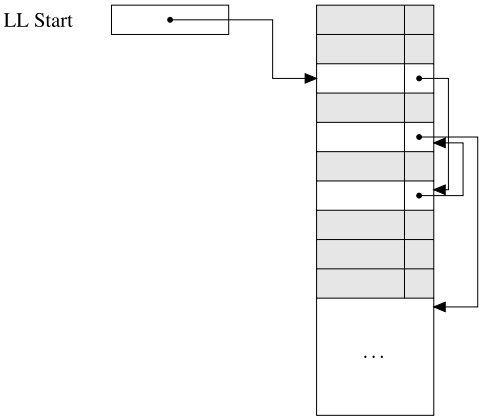


Fig. 5.10 Free list for heap of fixed-size blocks after allocation of some memory. *Grey blocks* are allocated (in use)

requested the memory and the pointer to the free list is updated so that it points to the next element.

When memory is, on the other hand, freed or deallocated (for example using `free`), the freed block is linked again to the head of the free list. The situation after some memory allocations is shown in Fig. 5.10. Conceptually, therefore, management of a heap with fixed-size blocks is simple, provided that it is known how to identify and reclaim the memory that must be returned to the free list easily. These operations of identification and recovery are not obvious, as we will see below.

5.4.2 Variable-Length Blocks

In the case in which the language allows the runtime allocation of variable-length memory spaces, for example to store an array of variable dimension, fixed-length blocks are no longer adequate. In fact the memory to be allocated can have a size greater than the fixed block size, and the storage of an array requires a contiguous region of memory that cannot be allocated as a series of blocks. In such cases, a heap-based management scheme using variable-length blocks is used.

This second type of management uses different techniques, mainly defined with the aim of increasing memory occupation and execution speed for heap management operations (recall that they are performed at runtime and therefore impact on the execution time of the program). As usual, these two characteristics are difficult to reconcile and good implementations tend towards a rational compromise.

In particular, as far as memory occupancy is concerned, it is a goal to avoid the phenomenon of memory *fragmentation*. So-called *internal fragmentation* occurs when a block of size strictly larger than the requested by the program is allocated. The portion of unused memory internal to the block clearly will be wasted until the block is returned to the free list. But this is not the most serious problem. Indeed, so-called *external fragmentation* is worse. This occurs when the free list is composed of blocks of a relatively small size and for which, even if the sum of the total available free memory is enough, the free memory cannot be effectively used. Figure 5.11 shows an example of this problem. If we have blocks of size x and y (words or some other unit—it has no relevance here) on the free list and we request the allocation of a block of greater size, our request cannot be satisfied despite the fact that the total amount of free memory is greater than the amount of memory that has been requested. The memory allocation techniques tend therefore to

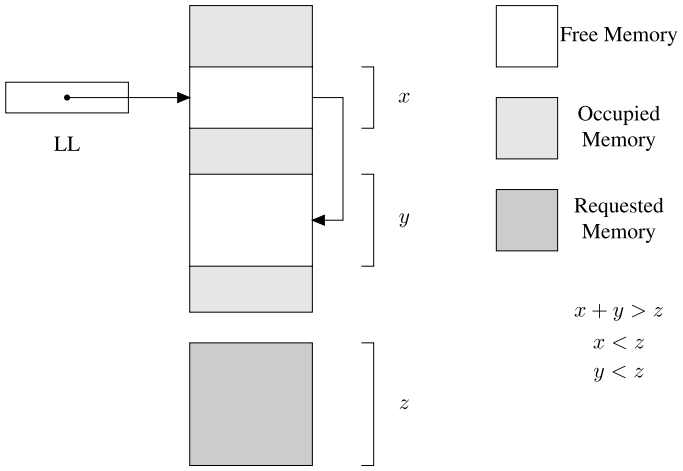


Fig. 5.11 External fragmentation

"compact" free memory, merging contiguous free blocks in such a way as to avoid external fragmentation. To achieve this objective, merging operations can be called which increase the load imposed by the management methods and therefore reduce efficiency.

Single free list The first technique we examine deals with a single free list, initially composed of a single memory block containing the entire heap. It is indeed convenient to seek to maintain blocks of largest possible size. It makes no sense, therefore, initially to divide the heap into many small blocks as, on the other hand, we did in the case of fixed-size blocks. When the allocation of a block of n words of memory is requested, the first n words are allocated and the pointer to the start of the head is incremented by n. Successive requests are handled in a similar way, in which deallocated blocks are collected on a free list. When the end of the heap's memory space is reached, is necessary to reuse deallocated memory and this can be done in the two following ways:

(i) **Direct use of the free list** In this case, a free list of blocks of variable size is used. When the allocation of a memory block n words in length is requested, the free list is searched for a block of size k words, where k is greater than or equal to n. The requested memory is allocated inside this block. The unused part of the block (of size $k-n$) if longer than some predefined threshold, is used to form a new block that is inserted into the free list (internal fragmentation is permitted below this threshold). The search for a block of sufficient size can be performed using one of two methods. Using *first fit*, the search is for the first block of sufficient size, while using *best fit*, the search is for a block whose size is the least of those blocks of sufficient size. The first technique favours processing time, while the second favours memory occupation. For both, however, the cost of allocation is linear with respect to the number of blocks on the free list. If the blocks are held in order of increasing block size, the two schemes are the same, because the list is traversed until a large enough block is found. Moreover, in this case, the cost of insertion of a block into the free list increases (from constant to linear), because is necessary to find the right place to insert it. Finally, when a deallocated block is returned to the free list, in order to reduce external fragmentation, a check is make to determine whether the physically adjacent blocks are free, in which case they are compacted into a single block. This type of compaction is said to be *partial* because it compacts only adjacent blocks.

(ii) **Free memory compaction** In this technique, when the end of the space initially allocated to the heap is reached, all blocks that are still active are moved to the end; they are the blocks that cannot be returned to the free list, leaving all the free memory in a single contiguous block. At this point, the heap pointer is updated so that it points to the start of the single block of free memory and allocation starts all over again. Clearly, for this technique to work, the blocks of allocated memory must be movable, something that is not always guaranteed (consider blocks whose addresses are stored in pointers on the stack). Some compaction techniques will be discussed in Sect. 8.12, when we discuss garbage collection.

Multiple free lists To reduce the block allocation cost, some heap management techniques use different free lists for blocks of different sizes. When a block of size n is requested, the list that contains blocks of size greater than or equal to n is chosen and a block from this list is chosen (with some internal fragmentation if the block has a size greater than n). The size of the blocks in this case, too, can be static or dynamic and, in the case of dynamic sizes, two management methods are commonly used: the *buddy system* and the *Fibonacci heap*. In the first, the size of the blocks in the various free lists are powers of 2. If a block of size n is requested and k is the least integer such that $2^k \geq n$, then a block of size 2^k is sought (in the appropriate free list). If such a free block is found it is allocated, otherwise, a search is performed in the next free list for a block of size 2^{k+1} and it is split into two parts. One of the two blocks (which therefore has size 2^k) is allocated, while the other is inserted into the free list for blocks of size 2^k. When a block resulting from a split is returned to the free list, a search is performed for its "buddy", that is the other half that was produced by the split operation, and it is free, the two blocks are merged to re-form the initial block of size 2^{k+1}. The *Fibonacci heap* method is similar but uses Fibonacci numbers instead of powers of 2 as block sizes. Given that the Fibonacci sequence grows more slowly than the series 2^n, this second method leads to less internal fragmentation.

5.5 Implementation of Scope Rules

The possibility of denoting objects, even complex ones, by names with appropriate visibility rules constitutes one of the most important aspects that differentiate high-level languages from low-level ones. The implementation of environments and scope rules discussed in Chap. 4 requires, therefore, suitable data structures. In this section, we analyse these structures and their management.

Given that the activation record contains the memory space for local names, when a reference to a non-local name is encountered, the activation records that are still active must be examined (that is, the ones present on the stack) in order to find the one that corresponds to the block where the name in question was declared; this will be the block that contains the association for our name. The order in which to examine the activation records varies according to the kind of scope under consideration.

5.5.1 Static Scope: The Static Chain

If the static scope rule is employed, as we anticipated in Chap. 4, the order in which activation records are consulted when resolving non-local references is not the one defined by their position on the stack. In other words, the activation record directly connected by the dynamic chain pointer is not necessarily the first activation record in which to look in order to resolve a non-local reference; the first activation record within which to look is defined by the textual structure of the program. Let us see an example.

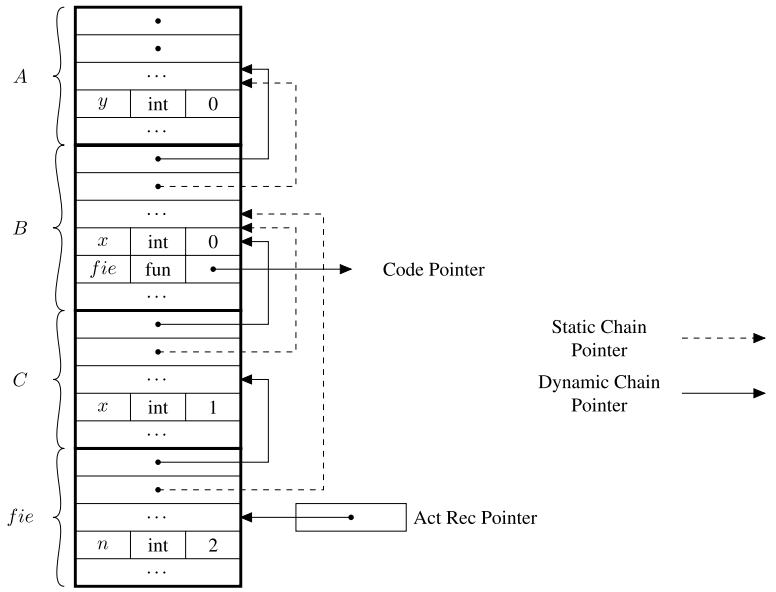


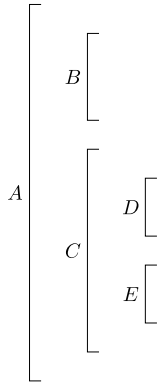
Fig. 5.12 Activation stack with static chain (see Example 5.3)

Example 5.3 Consider the following code (as usual, for ease of reference, we have labelled the blocks):

```
A:{int y=0;
   B:{int x = 0;
      void (int n){
         x = n+1;
         y = n+2;
      }
      C:{int x = 1;
         fie(2);
         write(x);
         }
      }
  write(y);
  }
```

After executing the call `fie(2)`, the situation on the activation-record stack is that shown in Fig. 5.12. The first activation record on the stack (the uppermost one in the figure) is for the outermost block; the second is the one for block `B`; the third is for `C` and finally the fourth is the activation record for the call to the procedure. The non-local variable, `x`, used in procedure `fie`, as we know from the static scope rule is not the one declared in block `C` but the one declared in block `B`. To be able

Fig. 5.13 A block structure



to locate this information correctly at runtime, the activation record for the call to the procedure is connected by a pointer, called the *static chain* pointer, to the record for the block containing the declaration of the variable. This record is linked, in its turn, by a static chain pointer to the record for block `A`, because this block, being the first immediately external to `B`, is the first block to be examined when resolving references non-local to `B`. When inside the call to procedure `fie`, the variables `x` and `y` are used, to access the memory area in which they are stored the static chain pointers are followed from `fie`'s activation record until first the record for `B` is encountered (for `x`) and then that for `A` (when searching for `y`).

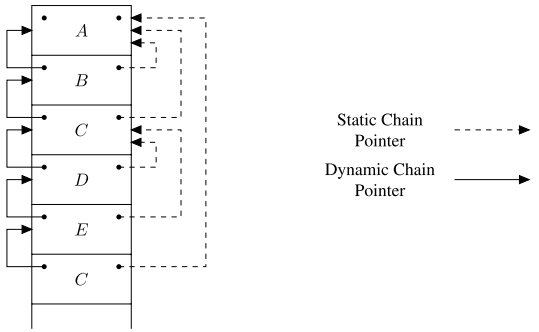
Generalising this example, we can say that, for the runtime management of static scope, the activation of the generic block `B` is linked by the *static chain pointer* to the record for the block immediately enclosing `B` (that is the nearest block that contains `B`). It should be noted that in the case in which `B` is the block for a procedure call, the block immediately enclosing `B` is the one containing the declaration of the procedure itself. Moreover if `B` is active, that is if its activation record is on the stack, then also the blocks enclosing `B` must be active and therefore can be located on the stack.

Hence, in addition to the *dynamic chain*, which is formed from the various records present on the system stack (linked in the order of the stack itself), there must exist a *static chain*, formed from the various static chain pointers used to represent the static nesting structure of the blocks within the program.

As an example, consider Fig. 5.13 which shows a generic structure of blocks which results from nested procedures. Consider now the sequence of calls: A, B, C, D, E, C, where it is intended that each call remains active when the next call is made. The situation on the activation-record stack, with is various static chain pointers, after such a sequence of calls is that shown in Fig. 5.14.

The runtime management of the static chain is one of the functions performed by the calling sequence, prologue and epilogue code, as we saw above. Such a management of the static chain can be performed by the caller and the callee in various ways. According to the most common approach, when a new block is entered, the caller calculates the static chain pointer and then passes it to the called routine. This

Fig. 5.14 Static chain for the previous structure and the sequence of calls A, B, C, D, E, C



computation is fairly simple and can be easily understood by separating the two cases:

The called routine is external to the caller In this case, by the visibility rules defined by static scope, for the called routine to be visible, it must be located in an outer block which includes the caller's block. Therefore, the activation record for such an outer block must already be stored on the stack. Assume that among the caller and the called routines, there are k levels of nesting in the program's block structure; if the caller is located on nesting level n and the called routine is on level m, we can assume therefore that $k = n - m$. This value of k can be determined by the compiler, because it depends only on the static structure of the program and therefore can be associated with the call in question. The caller can then calculate the static chain pointer for the called procedure simply by dereferencing its own static chain pointer k times (that is, it runs k steps along its own static chain).

Called inside calling routine In this case, the visibility rules ensure that the called routine is declared in same the block in which the call occurs and therefore the first block external to the called one is precisely that of the caller. The static chain pointer of the called routine must point to the caller's activation record. The caller can simply pass to the called routine the pointer to its own activation record as a pointer to the static chain.

Once the called routine has received the static chain pointer, it need only store it in the appropriate place in its activation record, an operation that can be performed by the prologue code. When a block exit occurs, the static chain requires no particular management actions.

We have hinted at the fact that the compiler, in order to perform runtime static-chain management, keeps track of the nesting level of procedure calls. This is done using the *symbol table*, a sort of dictionary where, more generally, the compiler stores all the names used in the program and all the information necessary to manage the objects denoted by the names (for example to determine the type) and to implement the visibility rules.

In particular, a number is maintained that depends on the nesting level and indicates the scope that contains the declaration of a name; this allows to associate to

each name a number indicating the scope when the declaration for such a name is made. Using this number, it is possible to calculate, at compile time, the distance between the scope of the call and that of the declaration which is necessary at runtime to handle the static chain.

It should be noted that this distance is calculated statically and it also allows the runtime resolution of non-local references without having to perform any name searches in the activation record on the stack. Indeed, if we use a reference to the non-local name, `x`, to find the activation record containing the memory space for `x` it suffices to start at the activation record corresponding to the block that contains the reference and follow the static chain for a number of links equal to the value of the distance. Inside the activation record that is thus found, the memory location for `x` is also fixed by the compiler and, therefore, at runtime, there is no need for a search but only the static offset of `x` with respect to the activation record pointer is needed.

However, it is clear that, in a static model, the compiler cannot completely resolve a reference to a non-local name and it is always necessary to follow the static-chain links at runtime. This is why, in general, it is not possible to know statically what the number of activation records present on the stack is.

As a concrete example of what has just been said, consider the code in Example 5.3. The compiler "knows" that to use variable `y` in procedure `fie`, it is necessary to pass two external blocks (`B` and `A`) to arrive at the one containing the declaration of the variable. It is enough, therefore, to store this value at compilation time so that it can subsequently be known, at runtime, that to resolve the name `y`, it is necessary to follow two pointers in the static chain. It is not necessary to store the name `y` explicitly because its position inside the activation record for the block `A` is fixed by the compiler. Analogously, the type information that we, for clarity, have included in Fig. 5.12, is stored in the symbol table, and after appropriate compile-time checks, can, in a large part, be omitted at runtime.

5.5.2 Static Scope: The Display

The implementation of static scope using the static chain has one inconvenient property: if we have to use a non-local name declared in an enclosing block, k levels of block away from the point at which we currently find ourselves, at runtime we have to perform k memory accesses to follow the static chain to determine the activation block that contains the memory location for the name of interest. This problem is not all that severe, given that in real programs it is rare that more than 3 levels of block and procedure nesting are required. The technique called the *display*, however, allows the reduction of the number of accesses to a constant (2).

This technique uses a vector, called the *display*, containing as many elements as there are levels of block nesting in the program, where the kth element of the vector contains the pointer to the activation record at nesting level k that is currently active. When a reference is made to a non-local object, declared in an block

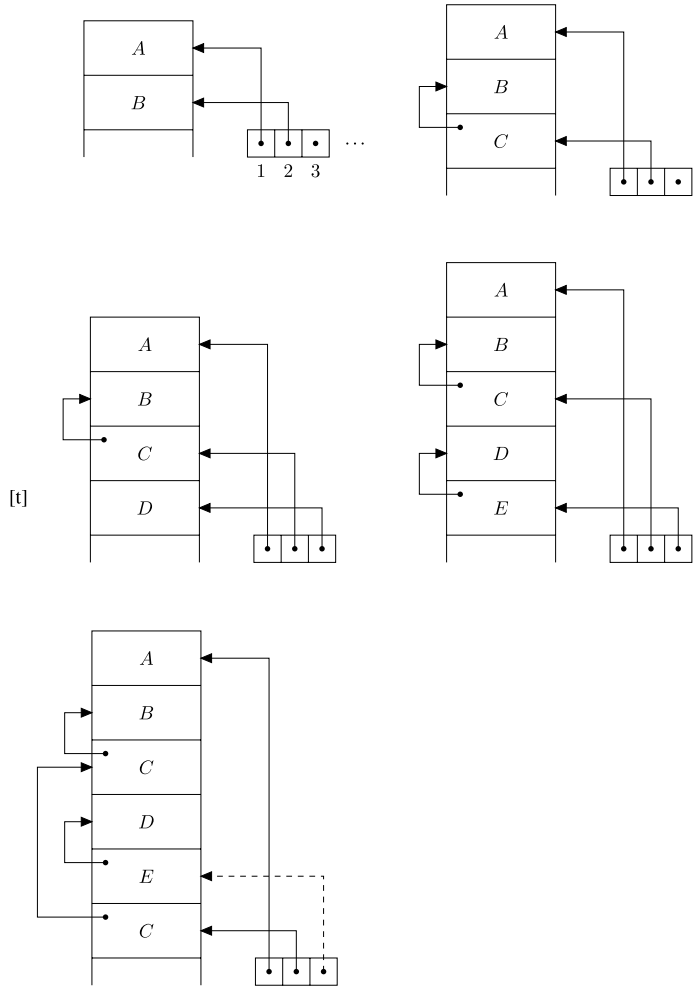


Fig. 5.15 Display for the structure in Fig. 5.13 and the call sequence A, B, C, D, E, C

external to level k, the activation record containing this object can be retrieved simply by accessing the kth position in the vector and following the pointer stored there.

Display processing is very simple, even if it is slightly more costly than static chain handling; when an environment is entered or left, in addition to updating the pointer stored in the vector, it is also necessary to save the old value. More precisely, when a procedure is called (or an in-line block is entered) at level k, position k in the

display will have to be updated with the value of the pointer to the activation record for the call, because this has become the new active block at level k. Before this update, however, it is necessary to save the preceding contents of the kth position of the display, normally storing it in the activation record of the call.

The need to save the old display value can be better understood by examining the following 2 possible cases:

The called routine is external to the caller Let use assume that the call is at nesting level n and the called routine is at level m, with $m < n$. The called routine and the caller therefore share the static structure up to level $m - 1$ and also the display up to position $m - 1$. Display element m is updated with the pointer to the activation record of the called routine and until the called routine terminates, the active display is the one formed of the first m elements. The old value contained in position m must be saved because it points to the activation record of the block which will be re-activated when the called routine terminates; thereafter, the display will go back to being the one used before the call.

The called routine is located inside the caller The nesting depth reached this far is incremented. If the caller is located at level n, caller and called routine share the whole current display up to position n and it is necessary to add a new value at position $n + 1$, so that it holds the pointer to the activation record for the caller. When we have the first activation of a block at level $n + 1$, the old value stored in the display is of no interest to us. However, in general, we cannot know if this is the case. Indeed, we could have reached the current call by a series of previous calls that also use level $n + 1$. In this case, as well, it will be necessary therefore to store the old value in the display at position $n + 1$.

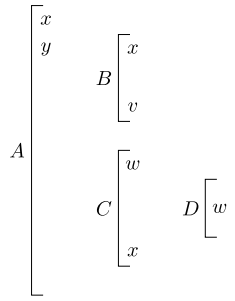
Both display update and the saving of the old value can be performed by the called procedure. Figure 5.15 shows the handling of the display for the call sequence A, B, C, D, E, C, using the block structure described in Fig. 5.13. The pointers on the left of the stack denote storage of the old display value in the activation record of the called routine, while the dotted pointer denotes a display pointer that is not currently active.

5.5.3 *Dynamic Scope: Association Lists and CRT*

Conceptually, the implementation of the dynamic scope rule is much simpler than the one for static scope. Indeed, given that non-local environments are considered in the order in which they are activated at runtime, to resolve a non-local reference to a name x, it suffices, at least in principle, to run backwards down the stack, starting with the current activation record until the activation record is found in which the name x is declared.

The various associations between names and the objects they denote which constitute the various local environment can be stored directly in the activation record. Let us consider, for example, the block structure shown in Fig. 5.16, where the

Fig. 5.16 A block structure with local declarations



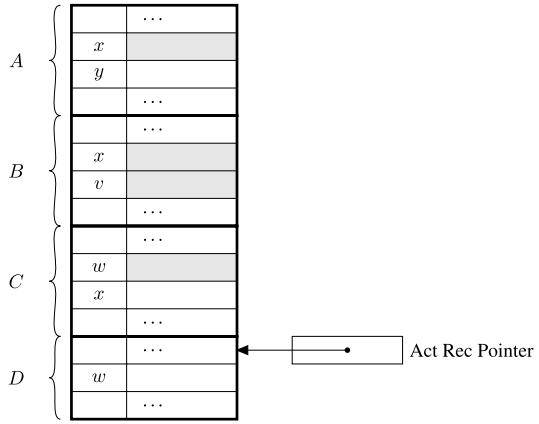


Fig. 5.17 Environment for block D in Fig. 5.16 after the call sequence A, B, C, D, with dynamic scope implemented using stored associations in the activation record. In *grey*: inactive associations

names denote local variable declarations (assuming the usual visibility rules). If we execute the call sequence A, B, C, D (where, as usual, all the calls remain active) when control reaches block D, we obtain the stack shown in Fig. 5.17 (the field on the right of each name contains the information associated with the object denoted by its name). The environment (local or otherwise) of D is formed from all the name-denoted object associations in which the information field is in white in the figure. The association fields that are no longer active are shown in grey. An association is not active either because the corresponding name is no longer visible (this is the case for v) or because it has been redefined in an inner block (this is the case for w and for the occurrences of x in A and B).

Other than direct storage in the activation record, name-object associations can be stored separately in an association list, called an *A-list*, which is managed like a stack. This solution is usually chosen for LISP implementations.

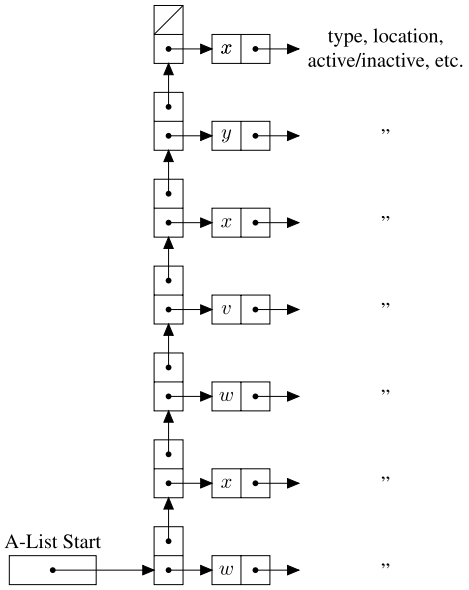


Fig. 5.18 Environment for block D in Fig. 5.16, after the call sequence A, B, C, D, with dynamic scope implemented using an A-list

When the execution of a program enters a new environment, the new local associations are inserted into the A-list. When an environment is left, the local associations are removed from the A-list. The information about the denoted objects will contain the location in memory where the object is actually stored, its type, a flag which indicates whether the association for this object is active (there can also be other information needed to make runtime semantic checks). Figure 5.18 shows how dynamic scope is implemented for the example in Fig. 5.16 using an A-list (the fields in grey are implemented using the flags described above and are omitted from the figure). Both using A-list and using direct storage in the activation record, the implementation of dynamic scope has two disadvantages.

First, names must be stored in structures present at runtime, unlike in the scheme that we saw for static scope. In the case of the A-list, this is clear (it depends on its definition). In the case, on the other hand, in which activation records are used to implement local environments, the need to store names depends on the fact that the same name, if declared in different blocks, can be stored in different positions in different activation records. Given that we are using the dynamic scope rule, we cannot statically determine which is the block (and therefore the activation record) that can be used to resolve a non-local reference; we cannot know the position in the activation record to access in order to search for the association belonging the name that we are looking for. The only possibility is therefore explicitly to store the name and perform a search (based on the name itself) at runtime.

The second disadvantage is due to the inefficiency of this runtime search. It can often be the case that is necessary to scan almost all of the list (which is either

an A-list or a stack of activation records) in the case reference is made to a name declared in one of the first active blocks (as for "global" names).

Central Referencing environment Table (CRT) To restrict the effects of these two disadvantages, at the cost of a greater inefficiency in the block entry and exit operations, we can implement dynamic scope in a different way. This alternative approach is based on the *Central Referencing environment Table* (CRT).

Using the CRT-based technique, environments are defined by arranging for all the blocks in the program to refer to an single central table (the CRT). All the names used in the program are stored in this table. For each name, there is a flag indicating whether the association for the name is active or not, together with a value composed of a pointer to information about the object associated with the name (memory location, type, etc.). If we assume that all the identifiers used in the program are not known at compile time, each name can be given a fixed position in the table. At runtime, access to the table can, therefore, take place in constant time by adding the memory address of the start of the table to an offset from the position of the name of interest. When, on the other hand, all names are not known at compile time, the search for a name's position in the table can be make use of runtime hashing for efficiency. The block entry and exit operations now are, however, more complicated. When entering block B from block A, the central table must be modified to describe B's new local environment, and, moreover, deactivated associations must be saved so that they can be restored when block B exits and control returns to block A. Usually a stack is the best data structure for storing such associations.

It should be observed that the associations for a block are not necessarily stored in contiguous locations within the CRT. To perform the operations required of the CRT on block entry and exit, it is necessary, therefore,to consider the individual elements of the table. This can be done in a convenient fashion by associating with each entry in the table (i.e., with every name present) a dedicated stack that contains the valid associations at the top and, in successive locations, the associations for this name that have been deactivated. This solution is shown in Fig. 5.19 (the second column contains the flags).

Alternatively, we can use a single hidden stack separate from the central table to store the deactivated associations for all names. In this case, for every name, the second column of the table contains a flag which indicates whether the association for this name is active or not, while the third column contains the reference to the object denoted by the name in question. When an association is deactivated, it is stored in the hidden stack to be removed when it becomes active again. Considering the structure in Fig. 5.16 and the call sequence A, B, C, D, the development of the CRT is shown in the upper part of Fig. 5.20; the lower portion of the Figure depicts the evolution of the hidden stack.

Using the CRT, with or without the hidden stack. access to an association in the environment requires one access to the table (either direct or by means of a hash function) and one access to another memory area by means of the pointer stored in the table. Therefore, no runtime search is required.

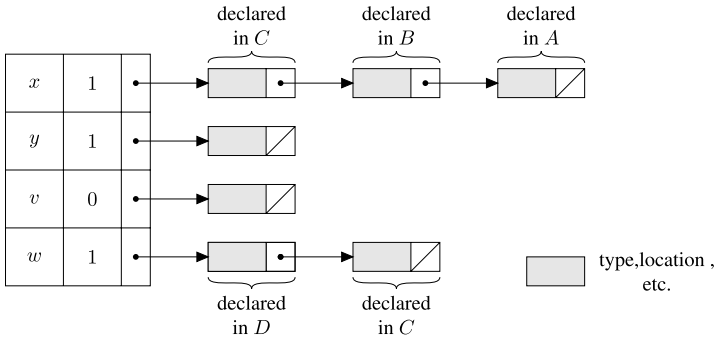
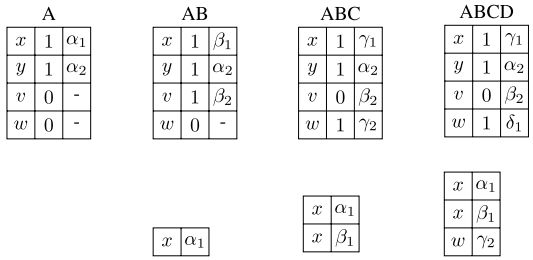


Fig. 5.19 Environment for block D in Fig. 5.16 after the call sequence A, B, C, D, with dynamic scope implemented using a CRT

Fig. 5.20 Environment for block D of Fig. 5.16 after the call sequence A, B, C, D, with dynamic scope implemented using a CRT and hidden stack



5.6 Chapter Summary

In this chapter, we have examined the main techniques for both static and dynamic memory management, illustrating the reasons for dynamic memory management using a stack and those that require the use of a heap. It remains to consider the important exception to this: in the presence of a particular type of recursion (called *tail recursion*) memory can be managed in a static fashion (this case will be given detailed consideration in the next chapter).

We have illustrated in detail the following on stack-based management:

- The format of activation records for procedures and in-line blocks.
- How the stack is managed by particular code fragments which are inserted into the code for the caller, as well as in the routine being called, and which act to implement the various operations for activation record allocation, initialisation, control field modification, value passing, return of results, and so on.

In the case of heap-based management, we saw:

- Some of the more common techniques for its handling, both for fixed- and variable-sized blocks.
- The fragmentation problem and some methods which can be used to limit it.

Finally, we discussed the specific data structures and algorithms used to implement the environment and, in particular, to implement scope rules. We examined the following in detail:

- The static chain.
- The display.
- The association list.
- The central referencing table.

This has allowed us better to understand our hint in Chap. 4 that it is more difficult to implement the static scope rules than those for dynamic scope. In the first case, indeed, whether static chain pointers or the display is used, the compiler makes use of appropriate information on the structure of declarations. This information is gathered by the compiler using symbol tables and associated algorithms, such as, for example, LeBlanc-Cook's, whose details fall outside the scope of the current text. In the case of dynamic scope, on the other hand, management can be, in principle, performed entirely at runtime, even if auxiliary structures are often used to optimise performance (for example the Central Referencing Table).

5.7 Bibliographic Notes

Static memory management is usually treated in textbooks on compilers, of which the classic is [1]. Determination of the (static) scopes to associate with the names in a symbol table can be done in a number of ways, among which one of the best known is due to LeBlanc and Cook [2]. Techniques for heap management are discussed in many texts, for example [4].

Stack-based management for procedures and for scope was introduced in ALGOL, whose implementation is described in [3].

For memory management in various programming languages, the reader should refer to texts specific to each language, some which are cited at the end of Chap. 13.

5.8 Exercises

1. Using some pseudo-language, write a fragment of code such that the maximum number of activation records present on the stack at runtime is not statically determinable.
2. In some pseudo-language, write a recursive function such that the maximum number of activation records present at runtime on the stack is statically determinable. Can this example be generalised?
3. Consider the following code fragment:

```
A:{int X= 1;
   ....
```

```
B:{ X  = 3;
    ....
  }

....
  }
```

Assume that B is nested one level deeper that A. To resolve the reference to X present in B, why is it not enough to consider the activation record which immediate precedes that of B on the stack? Provide a counter-example filling the spaces in the fragment with dots with appropriate code.

4. Consider the following program fragment written in a pseudo-language using static scope:

```
void P1 {
   void P2 { body-ofi-P2
           }
   void P3 {
            void P4 { body-of-P4
                    }
            body-of-P3
           }
   body-of-P1
   }
```

Draw the activation record stack region that occurs between the static and dynamic chain pointers when the following sequence of calls, `P1`, `P2`, `P3`,`P4`,`P2` has been made (is it understood that at this time they are all active: none has returned).

5. Given the following code fragment in a pseudo-language with `goto` (see Sect. 6.3.1), static scope and labelled nested blocks (indicated by `A: { ... }`):

```
A: { int x = 5;
     goto C;
        B: {int x = 4;
            goto E;
            }
        C: {int x = 3;
            D: {int x = 2;
            }
                 goto B;
            E: {int x = 1; // (**)
               }
            }
     }
```

The static chain is handled using a display. Draw a diagram showing the display and the stack when execution reaches the point indicated with the comment `(**)`. As far as the activation record is concerned, indicate what the only piece of information required for display handling is.

6. Is it easier to implement the static scope rule or the one for dynamic scope? Give your reasons.
7. Consider the following piece of code written in a pseudo-language using static scope and call by reference (see Sect. 7.1.2):

```
{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));
}
```

Assume that static scope is implemented using a display. Draw a diagram showing the state of the display and the activation-record stack when control enters the function A for the *second* time. For every activation record, just give value for the field that saves the previous value of the display.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, 1988.
2. R. P. Cook and T. J. LeBlanc. A symbol table abstraction to implement languages with explicit scope control. *IEEE Trans. Softw. Eng.*, 9(1):8–12, 1983.
3. B. Randell and L. J. Russell. *Algol 60 Implementation*. Academic Press, London, 1964.
4. C. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. Addison-Wesley, Reading, 1996.

Locally unsafe languages are those languages whose type system is well regulated and type checked but which contain some, limited, constructs which, when used, allow insecure programs to be written. The languages Algol, Pascal, Ada and many of their descendants belong to this category, provided that the abstract machine really checks types that need dynamic checking, such as checking the limits of interval types (which, it should be recalled, can be used as an index type for arrays). The unsafe part results from the presence of unions (uncontrolled variants) and from the explicit deallocation of memory. Of these two constructs, it is the second that has the greater impact in practice (it is much more common to encounter programs which deallocate memory than those which make significant use of variant records), but it is the former that is the more dangerous. As we will see in the next section, it is possible to equip the abstract machine with appropriate mechanisms to allow it to detect and prevent dangling pointers, even if, for reasons of efficiency, they are almost never used. We have already seen different examples of type-system violations which are possible when using variant records. We have postponed until last the nastiest possibility: that variant records can be used to access and manipulate the value of a pointer. In Pascal, we can write:[16]

```
var v : record
          case bool of
             true  : (i:integer);
             false : (p:^integer)
        end
```

Here, `^integer` is the Pascal notation for the type of pointers to integers. It is now possible to assign a pointer to the variant `p`, manipulate it as an integer using the variant `i` and then use it again as a pointer.

Finally, we have safe languages for which a theorem guarantees that the execution of a typed program can never generate a hidden error "induced by the violation of a type." In this category are languages with dynamic type checking like Lisp and Scheme, as well as languages with static checking such as ML, as well as languages with static checking but with many checks also performed at runtime (as in Java).

8.10 Avoiding Dangling References

In this section, we will tackle the question of which mechanisms can be included in an abstract machine that will dynamically prevent dereference of dangling pointers. This is a problem we first discussed in Sect. 8.4.5. We will first present a radical solution which works in the general case of pointers into the heap or into the stack. We will then consider a somewhat less demanding mechanism that, however, works only for pointers into a heap (and then under some probabilistic assumptions).

[16] In Pascal, pointers are allocated only on the heap by an allocation request (the `new` function).They can be assigned but there is no primitive method to access the value of a pointer.

8.10.1 Tombstone

Using tombstones, an abstract machine can detect every attempt at dereferencing a dangling reference. Every time an object is allocated in the heap (to be then accessed by pointer), the abstract machine also allocates an extra word in memory. This word is called the *tombstone*. In a similar fashion, a tombstone is allocated every time that a stack-referencing pointer is allocated (that is, to a certain approximation, every time that the "indirection-to" operator (`&`) is used). The tombstone is initialised with the address of the allocated object and the pointer receives the address of the tombstone. When a pointer is dereferenced, the abstract machine inserts a second level of indirection, so that it first accesses the tombstone and then uses what it finds there to access the object that is pointed to. When one pointer is assigned to another, it is the contents of the pointer (and not the tombstone) that is modified. Figure 8.10 shows this operation graphically.

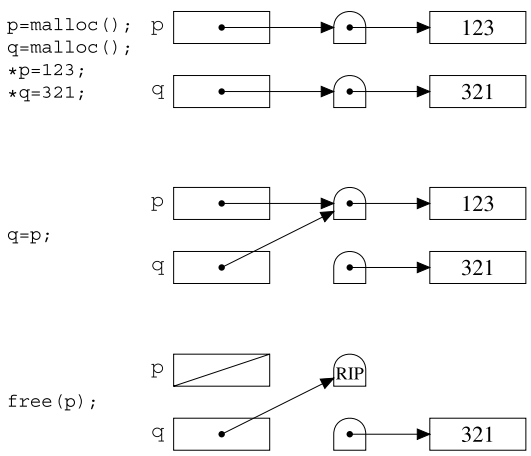
On deallocation of an object, or when an indirection on the stack becomes invalid because it is part of an activation record that has been popped,[17] the associated tombstone becomes invalidated, and a special value is stored in it to signal that the data to which the pointer refers is dead (this is where the name of this technique derives from). An appropriate choice of such a value must be made so that every attempt to access the contents of an invalid tombstone is captured by the address-protection mechanism of the underlying physical machine.

Tombstones are allocated in a particular area of memory in the abstract machine (this, appropriately enough, is called the *cemetery*). The cemetery can be managed more efficiently than the heap because all tombstones are of the same size.

For all its simplicity, the tombstone mechanism imposes a heavy cost in terms both of time and space. As far as time is concerned, we must consider the time required for the creation of tombstones, that for their checking (which can be reduced, as we have seen, if an underlying hardware protection mechanism can be used) and, most importantly, the cost of two levels of indirection. Tombstones are also expensive in terms of storage. They require a word of memory for every object allocated on the heap; a word is also required every time a pointer into the stack is created. If there are lots of allocations of small objects, the percentage of memory required by tombstones can be significant. Moreover, invalid tombstones always remain allocated, with the consequence that they might exhaust the space available in the cemetery, even though there might a great deal of space available on the heap. To prevent this last problem, it is possible to reuse those tombstones that are no longer in use (that is, those to which no pointer points) using a small reference-counting garbage collector (Sect. 8.11.1); this further increases the time cost of the mechanism.

[17] The operation can be quite opaque. Consider the case of a variable passed by reference to a function. Inside the function, a pointer is created to point to the variable. The pointer will be created using an associated tombstone. The tombstone must be invalidated when the lifetime of the variable forming the actual parameter ends, not when the function terminates.

Fig. 8.10 Tombstones



8.10.2 Locks and Keys

An alternative to the tombstone technique is called *locks and keys*. This solves the problem of dangling references into a heap using a mechanism which does not suffer from the accumulated problems of tombstones.

Every time an object is created on the heap, the object is associated with a *lock* which is a word of memory in which an arbitrary value is stored. (Strictly speaking, it should be a value that can be sequentially incremented every time an object is allocated, but which avoids values such as 0 and 1, the code of frequently used characters, and so on.) In this approach, a pointer is composed of a pair: the address proper and a *key* (a word of memory that will be initialised with the value of the lock corresponding to the object being pointed to). When one pointer is assigned to another, the whole pair is assigned. Every time the pointer is dereferenced, the abstract machine checks that the key opens the lock. In other words, it verifies that the information contained in the key coincides with that in the lock. In the case in which the values are not the same, an error is signalled. When an object is deallocated, its lock is invalidated and some canonical value (for example, zero) is stored, so that all the keys which previously opened it now cause an error (see Fig. 8.11). Clearly, it can happen that the area of memory previously used as a key is reallocated (for another lock or for some other structure). It is statistically highly improbable that an error will be signalled as a result of randomly finding an ex-lock that has the same value that it had prior to being clearing.

Locks and keys also have a non-negligible cost. In terms of space, they cost even more than tombstones because it is necessary to associate an additional word with each pointer. On the other hand, locks as well as keys are deallocated at the same time as the object or the pointer of which they are a part. From an efficiency viewpoint, it is necessary to take into account both the cost of creation, and, more