Chapter 6
Control Structure

In this chapter, we will tackle the problem of managing sequence control, an important part in defining the execution of program instructions in a generic abstract machine's interpreter.

In low-level languages, sequence control is implemented in a very simple way, just by updating the value of the PC (Program Counter) register. In high-level languages, however, there are special language-specific constructs which permit the structuring of control and the implementation of mechanisms that are much more abstract than those available on the physical machine. One thinks, for example, of the simple evaluation of an arithmetic expression: even if we find them obvious and natural, operations of this kind requires the use of control mechanisms that specify the order in which operands are evaluated, and operator precedence, and so on.

In this chapter, we will consider the constructs used in programming languages for the explicit or implicit specification of sequence control. We will first consider expressions, spending some time on the syntactic aspects of the usual notation for representing expressions, as well as the semantic aspects of their evaluation. We will then move on to commands and, after discussing the concepts of variable and assignment, we will see the main commands for sequence control present in modern languages, showing the difference between structured and unstructured control and briefly illustrating the principles of structured programming. We will finally examine some aspects that are significant to recursion and clarify an important terminological distinction between imperative and declarative languages.

We will leave the examination of those constructs that allow the implementation of mechanisms for control abstraction until the next chapter.

6.1 Expressions

Expressions, together with commands and declarations, are one of the basic components of every programming language. We can say that expressions are the essential component of every language because, although there exist declarative languages in which commands are absent, expressions, numeric or symbolic, are present in every language.

First, let us try to clarify what sorts of object we are talking about.

Definition 6.1 (Expressions) An expressions is a syntactic entity whose evaluation either produces a value or fails to terminate, in which case the expression is undefined.

The essential characteristic of an expression, that which differentiates it from a command, is therefore that its evaluation produces a value. Examples of numerical expressions are familiar to all: `4+3*2`, for example, is an expression whose evaluation is obvious. Moreover, it can be seen that, even in such a simple case, in order to obtain the correct result, we have made an implicit assumption (derived from the mathematical convention) about operator precedence. This assumption, which tells us that `*` has precedence over `+` (and that, therefore, the result of the evaluation is 10 and not 14), specifies a control aspect for evaluation of expressions. We will see below other more subtle aspects that can contribute to modify the result of the evaluation of an expression.

Expressions can be non-numeric, for example in LISP, we can write `(cons a b)` to denote an expression which, if it is evaluated, returns the so-called pair formed by `a` and `b`.

6.1.1 Expression Syntax

In general, an expression is composed of a single entity (constant, variable, etc.) or even of an operator (such as `+`, `cons`, etc.), applied to a number of arguments (or operands) which are also expressions. We saw in Chap. 2 how expression syntax can be precisely described by a context-free grammar and that an expression can be represented by a derivation tree in which, in addition to syntax, there is also semantic information relating to the evaluation of the expression. Tree structures are also often used to represent an expression internally inside the computer. However, if we want to use expressions in a conventional way in the text of a program, linear notations allow us to write an expression as a sequence of symbols. Fundamentally, the various notations differ from each other by how they represent the application of an operator to its operands. We can distinguish three main types of notation.

Infix Notation In this notation, a binary operation symbol is placed between the expressions representing its two operands. For example, we write `x+y` to denote than the addition of `x` and `y`, or `(x+y)*z` to denote the multiplication by `z` of the result of the addition of `x` and `y`. It can be seen that, in order to avoid ambiguity in the application of operator to operands, brackets and precedence rules are required. For operators other than binary ones, we must basically fall back on their representation in terms of binary symbols, even if, in this case, this representation is not

Lisp and S-expressions

The programming language Lisp (an acronym of LISt Processor), which was developed at the beginning of the 1960s by John McCarthy and by a group of researchers at MIT (Massachusetts Institute of Technology), is a language designed for symbolic processing, which has been particularly important in Artificial Intelligence. In particular, in the 1970s, the language Scheme was developed from a dialect of Lisp; Scheme is still in use in academic circles.

A Lisp program is composed of sequences of expressions to be evaluated by the language's interpreter. Some expressions are used to define functions which are then called in other expressions. Control is exercised using recursion (there is no iterative construct).

As the very name of the language implies, a Lisp program mainly handles expressions constructed from lists. The basic data structures in Lisp, in fact, is the dotted pair, or rather a pair of data items written with a dot separating the two components: for example, `(A.B)`. A pair like this is implemented as a *cons* cell, or rather by the application of the `cons` operator to two arguments, as in `(cons A B)`. As well as atomic types (integers, floating point numbers, character strings), the two arguments of `cons` can be other dotted pairs so this data structure allows the implementation of symbolic expressions, so-called S-expressions. S-expressions are binary trees which allow the representation of lists as a particular case. For example, it is possible to construct the list `A B C` as `(cons A (cons B (cons C nil)))`, where `nil` is a particular value denoting the empty list. Among the many interesting characteristics of Lisp, programs and data are represented using the same syntax and the same internal representation. This allows the evaluation of data structures as if they were programs and to modify programs as if they were data.

the most natural. A programming language which insists on infix notation even for user-defined functions is Smalltalk, an object oriented language.

Infix notation is the one most commonly used in mathematics, and, as a consequence is the one used by most programming languages, at least for binary operators and for user syntax. Often, in fact, this notation is only an abbreviation or, as we say, a *syntactic sugar* used to make code more readable. For example, in Ada, `a + b` is an abbreviation for `+(a,b)`, while in C++ the same expression is an abbreviation for `a.operator+(b)`.

Prefix Notation Prefix notation is another type of notation. It is also called *prefix Polish notation.*[1] The symbol which represents the operation precedes the symbols representing the operands (written from left to right, in the same way as text). Thus, to write the sum of `x` and `y`, we can write `+(x,y)`, or, without using parentheses,

[1]This terminology derives from the fact that the Polish mathematician W. Łukasiewicz was the person to make prefix notation without parentheses fashionable.

`+ x y`, while if we want to write the application of the function `f` to the operands `a` and `b`, we write `f(a b)` or `fab`.

It is important to note that when using this kind of notation, parentheses and operator precedence rules are of no relevance, provided that the arity (that is the number of operands) of every operator is already known. In fact there is no ambiguity about which operator to apply to any operands, because it is always the one immediately preceding the operands. For example if we write:

```
*(+(a b)+(c d))
```

or even

```
* + a b + c d
```

we mean the expression represented by `(a+b)*(c+d)` in normal infix notation.

The majority of regular languages use prefix notation for unary operators (often using parentheses to group arguments) and for user-defined functions. Some programming languages even use prefix notation for binary operators. LISP represents functions using a particular notation known as *Cambridge Polish*, which places operators inside parentheses. In this notation, for example the last expression becomes:

```
(*(+ a b)(+ c d)).
```

Postfix Notation Postfix notation is also called *Reverse Polish*. It is similar to the last notation but differs by placing the operator symbol after the operands. For example, the last expression above when written in postfix notation is:

```
a b + c d + *.
```

Prefix notation is used in the intermediate code generated by some compilers. It is also used in programming languages (for example Postscript).

In general, an advantage of Polish notation (prefix or otherwise) over infix is that the former can be used in a uniform fashion to represent operators with any number of operands. In infix notation, on the other had, representing operators with more than two operands means that we have to introduce auxiliary operators. A second advantage, already stated, is that there is the possibility of completely omitting parentheses even if, for reasons of readability, both mathematical prefix notation `f(a b)` and Cambridge Polish `(f a b)` use parentheses. A final advantage of Polish notation, as we will see in the next subsection is that it makes the evaluation of an expression extremely simple. For this reason, this notation became rather successful during the 1970s and 80s when it was used for the first pocket-sized calculators.

6.1.2 Semantics of Expressions

According to the way in which an expression is represented, the way in which its semantics is determined changes and so, consequently, does its method of evaluation. In particular, in infix representation the absence of parentheses can cause ambiguity problems if the precedence rules for different operators and the associativity of every binary operator are not defined clearly. When considering the most common programming languages, it is also necessary to consider the fact that expressions are often represented internally in the form of a tree. In this section we will discuss these problems, starting with the evaluation of expressions in each of the three notations that we saw above.

Infix Notation: Precedence and Associativity When using infix notation, we pay for the facility and naturalness of use with major complication in the evaluation mechanism for expressions. First of all, if parentheses are not used systematically, it is necessary to clarify the precedence of each operator.

If we write `4 + 3 * 5`, for example, clearly we intend the value of 19 as the result of the expression and not 35: mathematical convention, in fact, tells us that we have to perform the multiplication first, and the addition next; that is, the expression is to be read as `4 + (5 * 3)` and not as `(4 + 3) * 5`. In the case of less familiar operators, present in programming languages, matters are considerably more complex. If, for example, in Pascal one writes:

```
x=4 and y=5
```

where the `and` is the logical operator, contrary to what many will probably expect, we will obtain an error (a static type error) because, according to Pascal's precedence rules, this expression can be interpreted as

```
x=(4 and y)=5
```

and not as

```
(x=4) and (y=5).
```

In order to avoid excessive use of parentheses (which, when in doubt it is good to use), programming languages employ *precedence rules* to specify a hierarchy between the operators used in a language based upon the relative evaluation order. Various languages differ considerably in their definition of such rules and the conventions of mathematical notation are not always respected to the letter.

A second problem in expression evaluation concerns operator associativity. If we write `15-5-3`, we could intend it to be read as either `(15-5)-3` or as `15-(5-3)`, with clearly different results. In this case, too, mathematical convention says that the usual interpretation is the first. In more formal terms, the operator "−" associates from left to right.[2] In fact, the majority of arithmetic operators in programming languages associate *from left to right* but there are exceptions. The exponentiation operator, for example, often associates from right to left, as in mathematical notation. If we write 5^{3^2}, or, using a notation more familiar to programmers, `5 ** 3 ** 2`, we mean $5^{(3^2)}$, or `5 ** (3 ** 2)`, and not $(5^3)^2$, or `((5 ** 3) ** 2)`. Thus, when an operator is used, it is useful to include parentheses when in doubt about precedence and associativity. In fact, there is no lack of special languages that in this respect have rather counter-intuitive behaviour.

In APL, for example, the expression `15-5-3` is interpreted as `15 -(5 - 3)` rather than what we would ordinarily expect. The reason for this apparent strangeness is that in APL there are many new operators (defined to operate on matrices) that do not have an immediate equivalents in other formalisms. Hence, it was decided to abandon operator precedence and to evaluate all expressions from right to left.

Even if there is no difficulty in conceiving of a direct algorithm to evaluate an expression in infix notation, the implicit use of precedence and associativity rules, together with the explicit presence of parentheses, complicates matters significantly. In fact, it is not possible to evaluate an expression in a single left-to-right scan (or one from right to left), given that in some cases we must first evaluate the rest of the expression and then return to a sub-expression of interest. For example, in the case of `5+3*2`, when the scan from left to right arrives at `+`, we have to suspend the evaluation of this operator but divert to the evaluation of `3*2` and then go back to the evaluation of `+`.

Prefix Notation Expressions written in prefix Polish notation lend themselves to a simple evaluation strategy which proceeds by simply walking the expression from left to right using a stack to hold its components. It can be assumed that the sequence of symbols that forms the expression is syntactically correct and initially not empty. The evaluation algorithm is described by the following steps, where we use an ordinary stack (with the push and pop operations) and a counter C to store the number of operands requested by the last operator that was read:

1. Read in a symbol from the expression and push it on the stack;
2. If the symbol just read is an operator, initialise the counter C with the number of arguments of the operator and go to step 1.
3. If the symbol just read it is an operand, decrement C.
4. If $C \neq 0$, go to 1.
5. If $C = 0$, execute the following operations:
   (a) Apply the last operator stored on the stack to the operands just pushed onto the stack, storing the results in R, eliminate operator and operands from the stack and store the value of R on the stack.
   (b) If there is no operator symbol in the stack go to 6.

[2]A binary operator, `op`, is said to be associative if `x op (y op z) = (x op y) op z` holds; that is, whether it associates from the right to the left or left to right makes no difference.

   (c) Initialise the counter C to $n - m$, where n is the number of the argument of the topmost operator on the stack, and m is number of operands present on the stack above this operator.
   (d) Go to 4.
6. If the sequence remaining to be read is not empty, go to 1.

The result of the evaluation is located on the stack when the algorithm finishes. It should be noted that the evaluation of an expression using this algorithm assumes that we know in advance the number of operands required by each operator. This requires that we syntactically distinguish unary from binary operators. Furthermore, it is generally necessary to check that the stack contains enough operands for the application of the operator (Step 5.(c) in the algorithm above). This check is not required when using postfix notation, as we see below.

Postfix Notation The evaluation of expression in Polish notation is even simpler. In fact, we do not need to check that all the operands for the last operator have been pushed onto the stack, since the operands are read (from left to right) before the operators. The evaluation algorithm is then the following (as usual, we assume that the symbol sequence is syntactically correct and is not empty):

1. Read the next symbol in the expression and push it on the stack.
2. If the symbol just read is an operator apply it to the operands immediately below it on the stack, store the result in R, pop operator and operands from the stack and push the value in R onto the stack.
3. If the sequence remaining to be read is not empty, go to 1.
4. If the symbol just read is an operand, go to 1.

This algorithm also requires us to know in advance the number of operands required by each operator.

6.1.3 Evaluation of Expressions

As we saw the start of the Chap. 2 and as we extensively discussed in that chapter, expressions, like the other programming language constructs, can be conveniently represented by trees. In particular, can be represented by a tree (called the expression's *syntax tree*) in which:

- Every non-leaf node is labelled with an operator.
- Every subtree that has as root a child of a node N constitutes an operand for the operator associated with N.
- Every leaf node is labelled with a constant, variable or other elementary operand.

Trees like this can be directly obtained from the derivation trees of an (unambiguous) grammar for expressions by eliminating non-terminal symbols and by appropriate rearrangement of the nodes. It can be seen also that, given the tree representation, the linear infix, prefix and postfix representations can be obtained by traversing

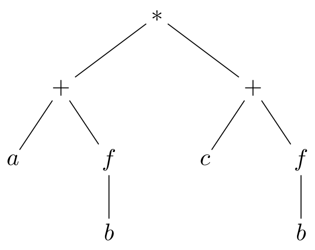

Fig. 6.1 An expression

the tree in a symmetric, prefix or postfix order, respectively. The representation of expressions as trees clarifies (without needing parentheses) precedence and associativity of operators. The subtrees found lower in the tree constitute the operands and therefore operators at lower levels must be evaluated before those higher in the tree. For example the tree shown in Fig. 6.1 represents the expression:

```
(a+f(b))*(c+f(b))
```

This expression can be obtained (parentheses apart) from the symmetric-order traversal of the tree (`f` is here an arbitrary unary operation).

For languages with a compilative implementation, as we have seen, the parser implements syntactic analysis by constructing a derivation tree. In the specific case of expressions then, infix representation in the source code is translated into a tree-based representation. This representation is then used by successive phases of the compilation procedure to generate the object code implementing runtime expressions evaluation. This object code clearly depends on the type of machine for which the compiler is constructed. In the case in which we have a traditional physical machine, for example, code of a traditional kind (i.e. in the form `opcode operand1 operand2`) is generated which uses registers as well as a temporary memory locations to store intermediate results of evaluation.

In some particular cases, on the other hand, object code can be represented using a prefix or postfix form which is subsequently evaluated by a stack architecture. This is the case for example in the executable code for many implementations of SNOBOL4 programs.

In the case of languages with an interpretative implementation, it is also convenient to translate expressions, normally represented in the source code in infix notation, into a tree representation which can then be directly evaluated using a tree traversal. This is the case, for example, in interpreted implementations of LISP, where the entire program is represented as a tree.

It is beyond the scope of the present text to go into details on mechanisms for generating code or for evaluating expression in an interpreter. However, it is important to clarify some difficult points which often cause ambiguity. For convenience, we will fix on the evaluation of expressions represented in infix form. We will see that what we have to say applies equally to the direct evaluation of expressions represented as a tree, as well as to code generation-mechanisms.

Subexpression Evaluation Order Infix notation precedence and associativity rules (or the structure, when expressions are represented as trees) do not hint at the order to evaluate an operator's operands (i.e., nodes at the same level). For example, in the expression in Fig. 6.1, nothing tells us that it is necessary first to evaluate either `a+f(b)` or `c+f(b)`. There is also nothing explicit about whether the evaluation of operands or operator should come first; nor, in general, whether expressions which are mathematically equivalent can be inter-substituted without modifying the result (for example, `(a-b+c)` and `(a+c-b)` could be considered equivalent).

While in mathematical terms these differences are unimportant (the result does not change), from our viewpoint these questions are extremely relevant and for the following five reasons.

Side effects In imperative programming languages, expression evaluation can modify the value of any variables through so-called side effects. A side effect is an action that influences the result (partial or final) of a computation without otherwise explicitly returning a value in the context in which it is found. The possibility of side effects renders the order of evaluation of operands relevant to the final result. In our example in Fig. 6.1, if the evaluation of the function `f` were to modify the value of its operand through side effects, first executing `a+f(b)` rather than `c+f(b)`, could change the value produced by the evaluation (see Exercise 1). As far as side effects are concerned, languages follow various approaches. On the one hand, pure declarative languages do not permit side effects at all, while languages which do allow them in some cases forbid the use in expressions of functions that can cause side effects. In other, more common cases where the presence of side effects is permitted, the order with which expressions are evaluated is, though, clearly stated in the definition of the language. Java, for example, imposes left-to-right evaluation of expressions (while C fixes no order at all).

Finite arithmetic Given the set of numbers represented in a computer is finite (see also Sect. 8.3), reordering expressions can cause overflow problems. For example, if `a` has, as its value, the maximum integer representable and `b` and `c` are positive numbers such that `b > c`, right-to-left evaluation of `(a-b+c)` does not produce overflow, while we have an overflow resulting from the evaluation from left to right of `(a+c-b)`. Moreover, when we do not have overflow, the limited precision of computer arithmetic implies that changing the order of the operands can lead to different results (this is particularly relevant in cases of floating point computation).

Undefined operands When the application of operator to operands is considered, two evaluation strategies can be followed. The first, called *eager evaluation*, consists of first evaluating all the operands and then applying the operator to the values thus obtained. The strategy probably seems the most reasonable when reasoning in terms of normal arithmetic operators. The expressions that we use in programming languages, however, pose problems over and above those posed by arithmetic expressions, because some can be defined even when some of the operands are missing. Let us consider the example of a conditional expression of the form:

```
a == 0 ? b : b/a
```

We can write this in C to denote the value of `b/a` when `a` is non-zero and `b`, otherwise. This expression results from the application of a single operator (expressed in infix notation using two binary operators `?` and `:`) to three operands (the boolean expression, `a==0`, and the two arithmetic expressions `b` and `b/a`). Clearly here we cannot use eager evaluation for such conditional expressions because the expression `b/a` would have to be evaluated even when `a` is equal to zero and this would produce an error.

In such a case, it is therefore better to use a *lazy evaluation* strategy which mainly consists of *not* evaluating operands before the application of the operator, but in passing the un-evaluated operands to the operator, which, when it is evaluated, will decide which operands are required, and will only evaluate the ones it requires.

The lazy evaluation strategy, used in some declarative languages, is much more expensive to implement than eager evaluation and for this reason, most languages use eager evaluation (with the significant exception of conditional expressions as we will see below). There are languages which use a mix of both the techniques (ALGOL, for example). We will discuss the various strategies for evaluating expressions in greater detail when we consider functional languages in Chap. 11.

Short-circuit evaluation The problem detailed in the previous point presents itself with particular clarity when evaluating Boolean expressions. For example, consider the following expression (in C syntax):

```
a == 0 || b/a > 2
```

If the value of `a` is zero and both operands of `||` are evaluated at the same time, it is clear that an error will result (in C, "`||`" denotes the logical operation of disjunction). To avoid this problem, and to improve the efficiency of the code, C, like other languages uses a form of lazy evaluation, also called *short-circuiting evaluation*, of boolean expressions. If the first operand of a disjunction has the value *true* then the second is not evaluated, given that the overall result will certainly have the value *true*. In such a case, the second expression is *short-circuited* in the sense that we arrive at the final value before knowing the value of all of the operands. Analogously, if the first operand of a conjunction has the value *false*, the second is not evaluated, given that the overall result can have nothing other than the value *false*.

It is opportune to recall that not all languages use this strategy for boolean expressions. Counting on the presence of a short-circuited evaluation, without being certain that the language uses it, is dangerous. For example, we can write in Pascal

```
p := list;
while (p <> nil ) and (p^.value <> 3) do
    p := p^.next;
```

The intention of this code is to traverse a list until we have arrived at the end or until we have encountered the value 3. This is badly written code that can produce a runtime error. Pascal, in fact, does not use short-circuit evaluation. In the case in which we have `p = nil`, the second operand of the conjunction `(p^.value <> 3)` yields an error when it dereferences a null pointer. Similar code, on the other hand, *mutatis mutandis*, can be written in C without causing problems. In order to avoid ambiguity, some languages (for example C and Ada), explicitly provide different boolean operators for short-circuit evaluation. Finally, it should be noted that this kind of evaluation can be simulated using a conditional command (see Exercise 2).

Optimisation Frequently, the order evaluation of subexpressions influences the efficiency of the evaluation of an expression for reasons relating to the organisation of the physical machine. For example, consider the following code:

```
a = vector[i];
b = a*a + c*d;
```

In the second expression, it is probably better first to evaluate `c*d`, given that the value of `a` has to be read from memory (with the first instruction) and might not be yet available; in such a case, the processor would have to wait before calculating `a * a`. In some cases, the compiler can change the order of operands expressions to obtain code that is more efficient but semantically equivalent.

The last point explains many of the semantic problems that appear while evaluating expressions. Given the importance of the efficiency of the object code produced by the compiler, it is given considerable liberty in the precise definition of its expression evaluation method, without it being specified at the level of semantic description of the language (as we have already said, Java is a rare exception). The result of this kind of approach is that, sometimes, different implementations of the same language produce different results for the same expression, or have errors at runtime whose source is hard to determine.

Wishing to capitalise in a pragmatic prescription, given what has been said so far, if we do not know the programming language well and the specific implementation we are using, if we want to write correct code, it is wise to use all possible means at our disposal to eliminate as many sources of ambiguity as possible in expression evaluation (such as brackets parentheses, specific boolean operations, auxiliary variables in expressions, etc.).

6.2 The Concept of Command

If, as we were saying above, expressions are present in all programming languages, the same is not true for commands. They are constructs that are typically present (but not entirely restricted to them) in so-called imperative languages.

Definition 6.2 (Command) A command is a syntactic entity whose evaluation does not necessarily return a value but can have a side effect.

A command, or more generally, any other construct, has a side-effect if it influences the result of the computation but its evaluation returns no value to the context in which it is located.

This point is fairly delicate and merits clarification with an example. If the `print` command in a hypothetical programming language can print character strings supplied as an argument, when the command `print "pippo"` is evaluated, we will not obtain a value but only a side-effect which is composed of the characters "pippo" appearing on the output device.

The attentive reader will be aware that the definition of command, just as the previous definition of expression, it is not very precise, given that we have referred to an informal concept of evaluation (the one performed by the abstract machine of the language to which the command or the expression belongs). It is clear that we can always modify the interpreter so that we obtain some value as a result of the evaluation of the command. This is what happens in some languages (for example in C assignment also returns the value to the right of =, see Sect. 6.2.2).

A precise definition and, equally, an exact distinction, between expressions and commands on the basis of their semantics is possible only in the setting of a formal definition of the semantics of language. In such a context, the difference between the two concepts derives from the fact that, once a starting state has been fixed, the result of the evaluation of an expression is a value (together with possible side-effects). On the other hand, the result of evaluating a command is a new state which differs from the start state precisely in the modifications caused by the side-effects of the command itself (and which are due principally to assignments). Command is therefore a construct whose purpose is the modification of the *state*. The concept of state can be defined in various ways; in Sect. 2.5, we saw a simple version, one which took into account the value of all the variables present in the program.

If the aim of a command is to modify the state, it is clear that the assignment command is the elementary construct in the computational mechanism for languages with commands. Before dealing with them, however, it is necessary to clarify the concept of variable.

6.2.1 The Variable

In mathematics, a variable is an unknown which can take on all the numerical values in a predetermined set. Even if we keep this in mind, in programming languages, it is necessary to specify this concept in more detail because, as we will see also in Sects. 11.1 and 12.3, the imperative paradigm uses a model for variables which is substantially different from that employed the in logic and functional programming paradigms.

The classical imperative paradigm uses *modifiable variables*. According to this model, the variable is seen as a sort of container, or location (clearly referring to physical memory), to which a name can be given and which contains values (usually of a homogeneous type, for example integers real, characters etc.). These values can

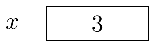
Fig. 6.2 A modifiable variable

be changed over time, by execution of assignment commands (whence comes the adjective "modifiable"). This terminology might seem tautological to the average computer person, who is almost always someone who knows an imperative language and is therefore used to modifiable variables. The attentive reader, though, will have noted that, in reality, variables are not always modifiable. In mathematics a variable represents a value that is unknown but when such a value is defined the link thus created cannot be modified later.

Modifiable variables are depicted in Fig. 6.2. The small box which represents the variable with the name x can be re-filled with a value (in the figure, the value is 3). It can be seen that the variable (the box) is different from the name x which denotes it, even if it is common to say "the variable x" instead of "the variable with the name x".

Some imperative languages (particularly object-oriented ones) use a model that is different from this one. According to this alternative model, a variable is not a container for a value but is a *reference* to (that is a mechanism which allows access to) a value which is typically stored in the heap. This is a new concept analogous to that of the pointer (but does not permit the usual pointer-manipulation operations). We will see this in the next section after we have introduced assignment commands. This variable model is called, in [8], the "reference model", while in [6], where it is discussed in the context of the language CLU, is called the "object model". Henceforth, we will refer to this as the *reference model* of variables.

(Pure) functional languages, as we will see in more detail in Sect. 11.1, use a concept of variable similar to the mathematical one: a variable is nothing more than an identifier that stands for a value. Rather, it is often said that functional languages "do not have variables", meaning that (in their pure forms) they do not have any modifiable variables.

Logic languages also use identifiers associated with values as variables and, as with functional languages, once a link between a variable identifier and a value is created, it can never be eliminated. There is however a mode in which the value associated with a variable can be modified without altering the link, as will be seen in Sect. 12.3.

6.2.2 Assignment

Assignment is the basic command that allows the modification of the values associated with modifiable variables. It also modifies the state in imperative languages. It is an apparently very simple command. However, as will be seen, in different programming languages, there are various subtleties to be taken into account.

Let us first see the case that will probably be most familiar to the reader. This is the case of an imperative language which uses modifiable variables and in which assignment is considered only as a command (and not also as an expression). One example is Pascal, in which we can write

```
X := 2
```

to indicate that the variable `X` is assigned the value `2`. The effect of such a command is that, after its execution, the container associated with the variable (whose name is) `X` will contain the value `2` in place of the value that was there before. It should be noted that this is a side effect, given that the evaluation of the command does not on its own, return any kind of value. Furthermore, every access to `X` in the rest of the program will return the value `2` and not the one previously stored.

Consider now the following command:

```
X := X+1
```

The effect of this assignment, as we know, is that of assigning to the variable `X` its previous value incremented by `1`. Let us observe the different uses of the name, `X`, of the variable in the two operands of the assignment operator. The `X` appearing to the left of the := symbol is used to indicate the container (the location) inside which the variable's value can be found. The occurrence of the `X` on the right of the := denotes the value inside the container. This important distinction is formalised in programming languages using two different sets of values: *l-values* are those values that usually indicate locations and therefore are the values of expressions that can be on the left of an assignment command. On the other hand, *r-values* are the values that can be stored in locations, and therefore are the values of expressions that can appear on the right of an assignment command. In general, therefore, the assignment command has the syntax of a binary operator in infix form:

```
exp1 OpAss exp2
```

where `OpAss` indicates the symbol used in the particular language to denote assignment (:= in Pascal, = in C, FORTRAN, SNOBOL and Java, $\leftarrow$ in APL, etc.). The meaning of such a command (in the case of modifiable variables) is as follows: compute the l-value of `exp1`, determining, thereby, a container *loc*; compute the r-value of `exp2` and modify the contents of *loc* by substituting the value just calculated for the one previously there.[3] Which expressions denote (in the context on the left of an assignment) an l-value depends on the programming language: the usual cases are variables, array elements, record fields (note that, as a consequence, calculation of an l-value can be arbitrarily complex because it could involve function calls, for example when determining an array index).

In some languages, for example C, assignment is considered to be an operator whose evaluation, in addition to producing a side effect, also returns the r-value thus computed. Thus, if we write in C:

[3] Some languages, e.g., Java, allow the left-hand side to be evaluated before the right-hand side; others (e.g., C), leave this decision to the implementer.

```
x = 2;
```

the evaluation of such a command, in addition to assigning the value 2 to x, returns the value 2. Therefore, in C, we can also write:

```
y = x = 2;
```

which should be interpreted as:

```
(y = (x = 2));
```

This command assigns the value 2 to x as well as to y. In C, as in other languages, there are other assignment operators that can be used, either for increasing code legibility or avoiding unforeseen side effects. Let us take up the example of incrementing a variable. Once again we have:

```
x = x+1;
```

This command, unless optimised by the compiler, requires, in principle, two accesses to the variable x: one to determine the l-value, and one to obtain the r-value. If, from the efficiency viewpoint, this is not serious (and can be easily optimised by the compiler), there is a question which is much more important and which is again related to side-effects. Let us then consider the code:

```
b = 0;
a[f(3)] = a[f(3)]+1;
```

where a is a vector and f is a function defined as follows:

```
int f (int n){
   if b == 0{
      b=1;
      return 1;
      }
   else return 2;
}
```

This function is defined in such a way that the non-local reference to b in the body of f refers to the same variable b that is cleared in the previous fragment.

Given that f modifies the non-local variable b, it is clear that the assignment

```
a[f(3)] = a[f(3)]+1
```

does not have the effect of incrementing the value of the element a[f(3)] of the array, as perhaps we wanted it to do. Instead, it has the effect of assigning the value of a[1]+1 to a[2] whenever the evaluation of the left-hand component of the assignment precedes the evaluation of the right-hand one. It should be noted, on the other hand, that the compiler cannot optimise the computation of r-values, because the programmer might have wanted this apparently anomalous behaviour.

To avoid this problem we can clearly use an auxiliary variable and write:

```
int j = f(3);
a[j]  = a[j]+1;
```

Doing this obscures the code and introduces a variable which expresses very little. To avoid all of this, languages like C provide assignment operators which allow us to write:

```
a[f(3)] += 1;
```

This add to the r-value of the expression present on the left the quantity present on the right of the += operator, and then assigns the result to the location obtained as the l-value of the expression on the left. There are many specific assignment commands that are similar to this one. The following is an incomplete list of the assignment commands in C, together with their descriptions:

- `X = Y`: assign the r-value of `Y` to the location obtained as the l-value of `X` and return the r-value of `X`;
- `X += Y` (or `X -= Y`): increment (decrement) `X` by the quantity given by the r-value of `Y` and return of the new r-value;
- `++X` (or `-X`): increment (decrement) `X` by and return the new r-value of `X`;
- `X++` (or `X-`): return the r-value of `X` and then increment (decrement) `X`.

We will now see how the reference model for variables differs from the traditional modifiable-variable one. In a language which uses the reference model (for example, CLU and, as we will see, in specific cases, Java) after an assignment of the form:

```
x=e
```

`x` becomes a reference to an object that is obtained from the evaluation of the expression `e`. Note that this does not copy the value of `e` into the location associated with `x`. This difference becomes clear if we consider an assignment between two variables using the reference model.

```
x=y
```

After such an assignment, `x` and `y` are two references to the same object. In the case in which this object is modifiable (for example, record or array), a modification performed using the variable `x` becomes visible through variable `y` and vice versa. In this model, therefore, variables behave in a way similar to variables of a pointer type in languages which have that type of data. As we will more clearly see in Sect. 8.4.5, a value of a pointer type is no more than the location of some data item (or, equivalently, its address in some area of memory). In many languages which have pointer types, the values of such types can be explicitly manipulated (which causes several problems as we will also see in Chap. 8). In the case of the reference model, however, these values can be manipulated only implicitly using assignments

Environment and Memory

In Chap. 2, we defined the semantics of a command by referring to a simple notion of state, which we defined as a function associating with every variable present in the program, the value it takes. This concept of state, although adequate for didactic purposes in mini-languages, is not sufficient when we want to describe the semantics of real programming languages which use modifiable variables and assignments. In fact, we have already seen in Chap. 4 (and will see in more detail in the next two chapters) that parameter-passing mechanisms as well as pointers can easily create situations in which two different names, for example `X` and `Y`, refer to the same variable, or rather the same location in memory. Such case of aliasing cannot be described using a simple function *State: Names* $\rightarrow$ *Values* because with a simple function it is not possible to express the fact that a modification of the value associated with (the variable denoted by) `X` also reflects on the value associated with `Y`. To correctly express the meaning of modifiable variables, we therefore use two separate functions. The first, called the *environment*, mostly corresponds to the concept of environment introduced in Chap. 4: in other words, it is a function *Environment: Names* $\rightarrow$ *DenotableValues* which maps names to the values they denote. The set (or, as one says in semantics jargon, the *domain*) of names often coincides with that of identifiers. The domain *DenotableValues*, instead, includes all values to which a name can be given; what these values are depends on the programming language but if the language provides modifiable variables then this domain certainly includes memory locations.

The values associated with locations are, on the other hand, expressed by a function *Memory: Locations* $\rightarrow$ *StorableValues* which (informally) associates every location with the value stored in it. In this case also, what exactly is a storable value depends on the specific language.

Therefore, when we say that “in the current state the variable `X` has the value 5”, formally we mean to say that we have an environment ρ and a memory σ such that $\sigma(\rho(\mathtt{X})) = 5$. Note that when a variable is understood to be an l-value, we are interested only in the location denoted by the name and, therefore, only in the environment is specified, while when we understand it as an r-value the store is also used. For example given environment ρ, and a store σ, the effect of the command `X=Y` is to produce a new state in which the value of $\rho(\sigma(\mathtt{Y}))$ is associated with $\rho(\mathtt{X})$. Let us recall, for completeness, that a third value domain important in the language semantics is formed from *ExpressibleValues*: these are those values which can be the result of the evaluation of a complex expression.

between variables. Java (which does not have pointers) adopts the reference model for variables for all class types, but uses the traditional modifiable-variable model for primitive types (integers, reals floating point, booleans and characters).

Below, unless otherwise specified, when we talk about variables, we mean the modifiable variable.

6.3 Sequence Control Commands

Assignment is the basic command in imperative languages (and in "impure" declarative languages); it expresses the elementary computation step. The remaining commands serve to define sequence control, or rather serve to specify the order in which state modifications produced by assignments, are to be performed. These other commands can be divided into three categories:

Commands for explicit sequence control These are the sequential command and `goto`. Let us consider, in addition, the composite command, which allows us to consider a group of commands as a single one, as being in this category.

Conditional (or selection) commands These are the commands which allow the specification of alternative paths that the competition can take. They depend on the satisfaction of specific conditions.

Iterative commands These allow the repetition of a given command for a predefined number of times, or until the satisfaction of specific conditions.

Let us consider these command typologies in detail.

6.3.1 Commands for Explicit Sequence Control

Sequential Command The *sequential command*, indicated in many languages by a ";", allows us directly to specify the sequential execution of two commands. If we write:

```
C1 ; C2
```

the execution of `C2` starts immediately after `C1` terminates. In languages in which the evaluation of a command also returns a value, the value returned by the evaluation of the sequential command is that of the second argument.

Obviously we can write a sequence of commands such as:

```
C1 ; C2 ; ... ; Cn
```

with the implicit assumption that the operator ";" associates to the left.

Composite Command In modern imperative languages, as we have already seen in Chap. 4, it is possible to group a sequence of commands into a *composite command* using appropriate delimiters such as those used by Algol:

```
begin
...
end
```

or those in C:

```
{
...
}
```

Imperative and Declarative Languages

Denotable, storable and expressible values, even if they have a non-empty intersection, are conceptually distinct sets. In effect, many important differences between various languages depend on how these domains are defined. For example, functions are denotable but not expressible in Pascal, while they are expressible in Lisp and ML. A particularly important difference between various languages concerns the presence of storable value and the *Memory* semantic function which we saw in the previous box. In fact, in a rather synthetic fashion, we can classify as *imperative* those languages which have environments as well as memory functions; those languages which have only environments are *declarative*. Imperative languages, while they are high-level languages, are inspired by the physical structure of the computer. In them the concept of memory (or state) is interpreted as the set of associations between memory locations and values stored in those locations. A program, according to this paradigm, is a set of *imperative* commands and computation consists of a sequence of steps which modify the state, using as its elementary command the assignment. The terminology "imperative" here has to do with natural language: as in an imperative phrase, we say "take that apple" to express a command, so with an imperative command we can say " assign to `x` the value 1". Most programming languages normally used belong to the imperative paradigm (Fortran, Algol, Pascal, C, etc.).

Declarative languages were introduced with the aim of offering a higher level programming paradigms, close to the notations of mathematics and logic, abstracting from the characteristics of the physical machine on which the programs are executed. In declarative languages (or at least in "pure" versions of them) there are no commands to modify the state, given that there are neither modifiable variables nor a semantic memory function. Programs are formed from a set of declarations (from which the name is derived) of functions or relations which define new values. According to the elementary mechanism used to specify the characteristics of the result, declarative languages are divided into two classes: functional and logic programming languages (the latter, also called logic languages for short). In the first case, computation consists of the evaluation of functions defined by the programmer using rules of a mathematical kind (mostly composition and application). In the second form of declarative language, on the other hand, computation is based on first-order logical deduction. Let us recall that, in actuality, there exist "impure" functional and logic languages that also have imperative characteristics (in particular, they include assignment). We will encounter both functional and logic languages in two next chapters.

Such a composite command, also called a *block*, can be used in any context in which a simple command is expected and therefore, in particular, can be used inside another composite command to create a tree-like structure of arbitrary complexity.

A Quibble about ";"

Being rigorous, the ";" used to separate commands is not always a sequential command. In C, C++ and Java, for example, it is a command *terminator* more than an operator expressing concatenation. In fact, this can be easily seen when we deal with the last command in a block. In these languages, the ";" is always required, even if the command is not followed by another as in

```
{x=1;
 x=x+1;
}
```

In languages like Pascal, on the other hand, ";" is really an operator that sequentialises commands. The same example as above can be written as:

```
begin
   x:=1;
   x:=x+1
end
```

To insert a ";" without there being a command to follow it is, however, a venial programming sin which the compiler absolves by inserting an *empty command* between the ";" and the `end`. An empty command is denoted by nothing syntactically and corresponds to no action.

Goto A unique place is occupied in the panorama of sequential control commands by the `goto`. It was included in the first programming languages and continues to be included in languages. It has 2 different forms (conditional or direct). This command is directly inspired by jump instructions in assembler languages and therefore by the sequence control of the hardware machine. The execution of the command

```
goto A
```

transfers control to the point in the program at which the label `A` occurs (different languages differ in what exactly constitutes a label but these differences are not relevant to us).

Despite its apparent simplicity and naturalness, the `goto` command has been at the centre of a considerable debate since the start of the 1970s (see, for example, the famous article by Dijkstra referred to in the bibliography), and, after about 30 years of debate, we can say that it is the detractors who have beaten the supporters of this command. To clarify the sense of this debate, let us, first of all, observe that the `goto` is not essential to the expressiveness of a programming language. A theorem due to Böhm and Jacopini in fact shows that any program can be translated into an equivalent one which does not use the `goto` (the formulation of the theorem, obviously, is much more precise than our account). This result, moreover, does not come down on one side or the other. If, on the one hand, we can assert that, in principle, the `goto` is useless, on the other, it is possible to object (as has been done) that even this result shows that it is permissible to use the `goto` in programs. If just we wish to eliminate this command, in fact, it can be done using the Böhm and Jacopini transformation (which, in particular, completely destroys the structure of the reformulated program).

The nexus of the question really is not of a theoretical nature but of a pragmatic one. Using `goto`, it is easily possible to write code which soon becomes incomprehensible and which still remains incomprehensible when all `goto`s are eliminated. We can think, for example, of a program of some considerable size where we have inserted jumps between points which are some of thousands of lines of code apart. Or, we can think of a subprogram in which exits are made at different points based on some condition and the exits are performed by `goto`s. These and other arbitrary uses of this construct make the code hard to understand, and therefore hard to modify, correct and maintain; this has the obvious negative consequences in terms of cost. To all of this, we can add the fact that the `goto` with its primitive method of transferring control, does not accord well with other mechanisms present in high-level languages. What happens, for example, if we jump *inside* of a block? When and how is the activation record for this block initialised so that everything works correctly?

If `goto` were used in an extremely controlled fashion, locally to small regions of code, the majority of these disadvantages would disappear. However, the cases in which it can be useful to use this command, such as exit from loops, return from a subprograms, handling of exceptions, can, in modern programming languages, be handled by specific, more appropriate, constructs. We can therefore assert that in modern high-level languages, the `goto` is a construct whose use is disappearing. Java is the first commercial language to have completely removed it from its set of admissible commands.

Other sequence control commands If `goto` is dangerous in its general form, there are local and limited uses which are useful in given circumstances. Many languages make available limited forms of jump to confront these pragmatic necessities without having to make use of the brute force of a `goto`. Among these commands (which take on different forms in different languages), we find constructs such as `break` (for terminating the execution of a loop, of a `case`, or, in some languages, of the current block), `continue` (for terminating the current iteration in an iterative command and force the starting of the immediately following command) or `return` (to terminate the evaluation of the function, returning control to the caller, sometimes also passing a value).

Finally, a more elaborate sequence control can be implemented using exceptions. We will deal with these in a more detailed fashion in Sect. 7.3 below.

6.3.2 Conditional Commands

Conditional commands, or selection commands, express one alternative between two or more possible continuations of the computation based on appropriate logical conditions. We can divide conditional commands into two groups.

If The `if` command, originally introduced in the ALGOL60 language, is present in almost all imperative languages and also in some declarative languages, in various syntactic forms which, really, can be reduced to the form:

```
if Bexp then C1 else C2
```

where `Bexp` is a boolean expression, while `C1` and `C2` are commands. Informally, the semantics of such a command expresses an alternative in the execution of the computation, based on the evaluation of the expression `Bexp`. When this evaluation returns true, the command `C1` is executed, otherwise the command `C2` is executed. The command is often present in the form without the `else` branch:

```
if Bexp then C1
```

In this case, too, if the condition is false, the command `C1` is not executed and control passes to the command immediately after the conditional. As we saw in Chap. 2, the presence of a branching `if` as in the command

```
if Bexp1 if Bexp2 then C1 else C2
```

causes problems of ambiguity, which can be resolved using a suitable grammar which formally describes the rules adopted by the language (for example, the `else` branch belongs to the innermost `if`; this is the rule in Java and it is used in almost every language). To avoid problems of ambiguity, some languages use a " terminator" to indicate where the conditional command ends, as for example in:

```
if Bexp then C1 else C2 endif
```

Furthermore, in some cases, instead of using a list of nested `if then elses`, use is made of an `if` equipped with more branches, analogous to the following:

```
if Bexp1 then C1
   elseif Bexp2 then C2
   ...
   elseif Bexpn then Cn
   else  Cn+1
endif
```

The implementation of the conditional command poses no problems, and makes use of instructions for test and jump that are found in the underlying physical machine.[4] The evaluation of the boolean expression can use the shorter circuit technique that we saw above.

Case The command is a specialisation of the `if` command, just discussed, with more branches. In its simplest form it is written as follows:

```
case Exp  of
   label1: C1;
   label2: C2;
   ...
   labeln: Cn;
else Cn+1
```

where `Exp` is an expression whose value is of a type compatible with that of the labels `label1, ...`, `labeln`, while `C1, ...`, `Cn+1` are commands. Each label is represented by one or more constants and the constant used in different labels are different from each other. The type permitted for labels, as well as their form, varies from language to language. In most cases, a discrete type is permitted (see Sect. 8.3), including enumerations and intervals. So, for example, we can use the constants `2` and `4` to denote a label, but in some languages we can also write `2,4` to indicate either the value `2` or the value `4`, or `2 .. 4` to indicate all values between `2` and `4` (inclusive).

The meaning of this command, as stated above, is analogous to that of a multi-branch `if`. Once the expression `Exp` has been evaluated, the command which occurs in the unique branch whose label includes the value to which `Exp` evaluated is executed. The `else` branch is executed whenever there is no other branch whose label satisfies the condition stated above.

It is clear that whatever can be done using a `case` can certainly be expressed using a nested series of `if`s. Even so, many languages include some form of case in their commands, either to improve the readability of the code, or because it is possible to compile a `case` much more efficiently than a long series of nested `if`s. A `case` is, in fact, implemented in assembly language using a vector of contiguous cells called a *jump table*, in which each element of the table contains the address of the first instruction of the corresponding command in the case's branches. The use of such a table is shown in Fig. 6.3, where, for simplicity, it is assumed that the labels `label1,..`, `labeln` are the consecutive constants $0, 1, \dots, n-1$. As should be clear in the figure, the expression which appears as an argument to the `case` is evaluated first of all. The value thus obtained is then used as an offset (index) to compute the position in the jump table of the instruction which performs the jump to the chosen branch. The extension of this mechanism to the general case in which labels are sets or intervals as simple (see Exercise 3).

[4] At the assembly language level, and therefore in the language of the physical machine, there are jump operations, conditional or not, analogous to the `goto` in high-level languages.

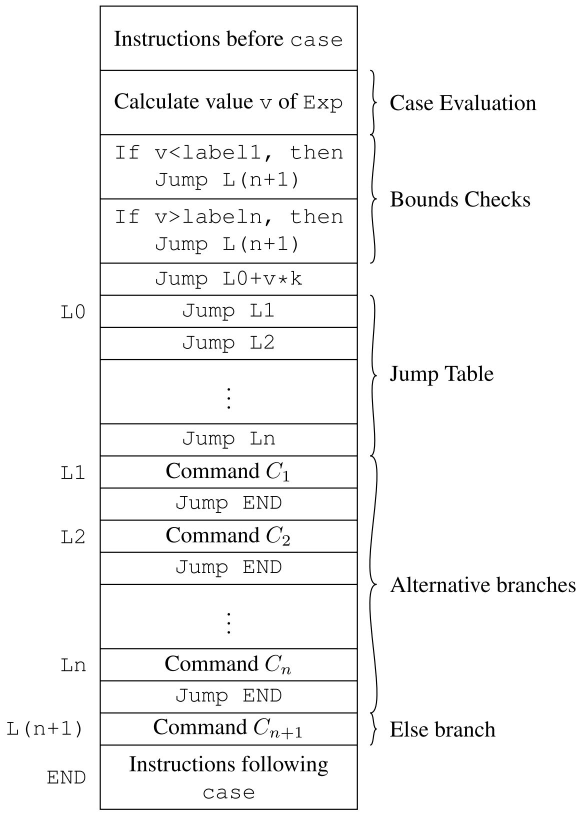

Fig. 6.3 Implementation of `case`

This implementation mechanism for `case` gives greater efficiency than a series of nested `if`s. Using a jump table, once the value of the expression is calculated, two jump instructions are required to arrive at the code of the command to execute. Using nested `if`s, on the other hand, for *n* alternative branches (in the worst case of an unbalanced `if`), it is necessary to evaluate $O(n)$ conditions and perform $O(n)$ jumps before arriving at the command of interest. The disadvantage of using a jump table is that, since it is a linear structure whose contiguous elements correspond to successive label values, it can consume a lot of space when the label values are dispersed over a fairly wide interval or when the individual labels are a type denoting a wide interval. In this case alternative methods can be used for calculating the jump address, such as sequential tests hashing or even methods based on binary search.

Different languages exhibit significant differences in their `case` commands. In C, for example, the `switch` has the following syntax (also to be found in C++ and in Java):

```
switch (Exp) body
```

where `body` can be any command that all. In general, though, the body is formed from a block in which some commands can be labelled; that is they are of the form:

```
case label : command
```

while the last command of the block is of the form:

```
default : command
```

When the expression `Exp` is evaluated and control is to be transferred to the command whose label coincides with the resulting value, if there are no labels with such a value, control passes to the command with the label `default`. If there is no `default` command, control passes to the first command following the `switch`. It can be seen that, once a the branch of the `switch` has been selected, control then flows into the immediately following branches. To obtain a construct with semantics analogous to that of the `case` we discussed above, it is necessary to insert an explicit control transfer at the end of the block, using a `break`:

```
switch (Exp){
   case label1: C1 break;
   case label2: C2 break;
   ...
   case labeln: Cn break;
   default: Cn+1 break;
}
```

It can be seen also that in a `switch`, the value returned by the evaluation of the expression might not appear in any label, in which case the entire command has no effect. Finally, lists or ranges of values are not permitted as labels. This however is no real limitation, given that lists of values can be implemented using the fact that control passes from one branch to its successor when `break` is omitted. If, for example, we write:

```
switch (Exp){
   case 1:
   case 2: C2 break;
   case 3: C3 break;
   default: C4 break;
}
```

in the case in which the value of `Exp` is `1`, given that the corresponding branch does not contain a `break` command, control passes from the `case 1` branch immediately to the `case 2` branch and therefore it is as if we had used a list of values `1,2` for the label of `C2`.

6.3.3 Iterative Commands

The commands that we have seen up to this point, excluding `goto`, only allow us to express finite computations, whose maximum length is determined statically by the length of the program text.[5] A language which had only such commands would be of highly limited expressiveness. It would certainly not be Turing complete (recall Sect. 3.3), in that it would not permit the expression of all possible algorithms (consider, for example, scanning a vector of n elements, where n is not known *a priori*).

In order to acquire the expressive power necessary to express all possible algorithms in low-level languages, jump instructions allowing the repetition of groups of instructions by jumping back to the start of the code are needed. In high-level languages, given that, as has been seen, it is desirable to avoid commands like `goto`, two basic mechanisms are employed to achieve the same effect: *structured iteration* and *recursion*. The first, which we consider in this section, is more familiar from imperative languages (and they almost always allow recursion as well). Suitable linguistic constructs (which we can regard as special versions of the jump command) allow us compactly to implement loops in which commands are repeated or *iterated*. At the linguistic level, it is possible to distinguish between *unbounded iteration* and *bounded iteration*. In bounded iteration, repetition is implemented by constructs that allow a determinate number of iterations. Unbounded iteration, on the other hand, is implemented by constructs which continue until some condition becomes true.

Recursion which we will consider in the next section, allows, instead, the expression of loops in an implicit fashion, including the possibility that a function (or procedure) can call itself, thereby repeating its own body an arbitrary number of times. The use of recursion is more common in declarative languages (in many functional and logic languages there does not, in fact, exist any iterative construct).

Unbounded iteration Unbounded iteration is logically controlled iteration. It is implemented by linguistic constructs composed of two parts: a loop *condition* (or *guard*) and a *body*, which is composed of a (possibly compound) command. When executed, the body is repeatedly executed until the guard becomes false (or true, according to the construct).

In its most common form, this type of iteration takes the form of the `while` command, originally introduced in ALGOL:`while`

```
while (Bexp) do C
```

The meaning of this command is as follows: (1) the boolean expression `Bexp` is evaluated; (2) if this evaluation returns the value *true*, execute the command `C` and return to (1); otherwise the `while` command terminates.

In some languages there are also commands that test the condition *after* execution of the command (which is therefore always executed at least once). This construct is for example present in Pascal in the following form:

[5]It can easily be seen that the maximum length of the computation is a linear function of the length of the program.

```
repeat C until Bexp
```

This is no more than an abbreviation for:

```
C;
while not Bexp do C
```

(`not Bexp` here indicates the negation of the expression `Bexp`). In C an analogous construct is `do`:

```
do C while (Bexp)
```

which corresponds to:

```
C;
while Bexp do C
```

(note that the guard is not negated as in the case of `repeat`.)

The `while` construct is simple to implement, given that it corresponds directly to a loop that is implemented on the physical machine using a conditional jump instruction. This simplicity of implementation should not deceive us about the power of this construct. Its addition to a programming language which contains only assignment and conditional commands immediately makes the language Turing complete. Our mini-language from Chap. 2 is therefore Turing complete (or: it allows the implementation of all computable functions). The same is not the case with bounded iteration which we will now turn to.

Bounded iteration Bounded iteration (sometimes also called numerically controlled iteration) is implemented by linguistic constructs that are more complex than those used for unbounded iteration; their semantics is also more elaborate. These forms are very different and not always "pure" as we will see shortly. The model that we adopt in this discussion is that of ALGOL, which was then adopted by many other languages of the same family (but *not* by C or Java).

Bounded iteration is implemented using some variant of the `for` command. Without wishing to use any specific syntax, it can be described as:

```
for I = start to end by step do
    body
```

where `I` is a variable, called the *index*, or counter, or *control variable*; `start` and `end` are two expressions (for simplicity we can assume that they are of integer type and, in general, they must be of a discreet type); `step` is a (compile-time) non-zero integer constant; *body* is the command we want to repeat. This construct, in the "pure" form we are describing, is subject to the important static semantic constraint that the control variable can not be modified (either explicitly nor implicitly) during the execution of the body.

The semantics of the bounded iteration construct can be described informally as follows (assuming that `step` is positive):

1. The expression `start` is evaluated, as is `end`. The values are frozen and stored in dedicated variables (which cannot be updated by the programmer). We denote them, respectively, as `start_save` and `end_save`.
2. `I` is initialised with the value of `start_save`.
3. If the value of `I` is strictly greater than the value of `end_save`, execution of the `for` command is terminated.
4. Execute *body* and increment `I` by the value of `step`.
5. Go to 3.

In the case in which `step` is negative, the test in step (3) determines whether `I` is strictly less than `end_save`.

It is worth emphasising the importance of step (1) above and the constraint that the control variable cannot be modified in the body. Their combined effect is to *determine* the number of times and the body will be executed *before* the loop begins execution. This number is given by the quantity, *ic* (*iteration count*), which is defined as:

$$ic = \left\lfloor \frac{\mathtt{end} - \mathtt{start} + \mathtt{step}}{\mathtt{step}} \right\rfloor$$

if *ic* is positive, otherwise it is 0. It can be seen, finally, that there is no way of producing an infinite cycle with this construct.

There are considerable differences, both syntactic and semantic, between the versions of this construct in different languages. First, not all languages require non-modifiability of the control variable and/or the freezing of the control expressions. Strictly speaking, such cases do not implement bounded iteration because they are unable to compute *ic* once and once only. It is common, though, to continue speaking of bounded iteration even when the language does not guarantee determinateness, but this is obtained on any loop, by the programmer (modifying neither directly nor indirectly the control variable and the start, end and step expressions). Also different other aspects constitute important differences between languages, of which we mention four:

Number of iterations According to the semantics which we have just given, when `step` is positive, if the value of `start` is initially (strictly) greater than the value of `end`, `body` is not executed at all. Even this is not the case in all languages, just the majority. Some languages execute the test in Step 3 after having executed `body`.

Step The requirement that `step` is a (non-zero) constant is necessary for statically determining its sign, so that the compiler can generate the appropriate code for the test in step 3. Some languages (such as Pascal and Ada) use a special syntax to indicate that `step` is negative, for example using `downto` or `reverse` in place of `to`. Other languages, such as, for example, some versions of Fortran, do not have a different syntax for the native step and their implementation of the `for` directly uses the iteration counter rather than the test of `I` and `end`. The value *ic* is computed and if this value is positive, it is used to control the loop, decrementing it by 1 until it reaches the value 0. If, on the other hand, *ic* has a negative value or is equal to 0, the loop is never repeated. It is the use of this implementation technique that suggests the name numerically controlled iteration.

Final index value The other subtle aspect concerns the value of the control variable `I` after the end of the loop. In many languages, `I` is a variable that is also visible outside of the loop. The most natural approach seems to be that of considering the value of `I` to be the last value assigned to it during execution of the `for` construct itself (in the case in which the loop terminates normally and the step is positive, the last value assigned to the index `I` is the first value greater than `end`). This approach, though, can generate type ambiguities or errors. Let us assume, for example, that `I` is declared as being of an interval type `1 .. 10` (from 1 to 10). If we use a command:

```
for I = 1 to 10 by 1 do
   body
```

The final value assigned to `I` would have to be the successor of 10, which is clearly not an admissible value. An analogous problem occurs for integer values when the calculation of `I` causes an overflow. To avoid this problem, some languages (for example Fortran IV or Pascal) consider the value of `I` to be indeterminate on termination of the loop (that is, the language definition does not specify what it should be). In other words, each implementation of these languages is allowed to behave how it wishes, with the imaginable consequence of the non-portability of programs. Other languages (for example: AlgolW, Algol 68, Ada, and, in certain circumstances, C++) cut the matter short, decreeing that the control variable is a variable that is *local* to the `for`; hence it is not visible outside the loop. In this case the header of the `for` implicitly declares the control variable with a type that is determined from that one of `start` and `end`.

Jump into a loop The last point which merits attention concerns the possibility of jumping into the middle of a `for` loop using a `goto` command. Most languages forbid such jumps for clear semantic reasons, while there are fewer restrictions on the possibility of using a `goto` for jumping out of a loop.

We have just considered a number of important aspects of the implementation of `for` loops. Particular tricks can be used by the compiler to optimise the code that is produced (for example, eliminating tests which involve constants) or by limiting overflow situations which could occur when the incrementing the index `I` (by inserting appropriate tests).

Expressiveness of bounded iteration Using bounded iteration, we can express the repetition of a command for n times, where n it is an arbitrary value not known when the program is written, but is fixed at when the iteration starts. It is clear that this is something that cannot be expressed using only conditional commands and assignment, because it is possible to repeat a command only by repeating the command in the body of the program syntactically. Given that every program has a finite length, we have a limit on the maximum number of repetitions that we can include in a specific program.

Despite this increase in expressive power, bounded iteration on its own is insufficient to make a programming language Turing complete. Think, for example, of a simple function f which can be defined as follows:

$$f(x) = \begin{cases} x & \text{if } x \text{ is even,} \\ \text{does not terminate} & \text{if } x \text{ is odd.} \end{cases}$$

Certainly such a function is computable. Every programmer would know how to implement it using a `while` or a `goto` or a recursive call so as to obtain a nonterminating computation. However such a function is not representable in a language only having assignment, sequential command, `if` and bounded iteration, given that, as can easily be verified in such a language, all programs terminate for every input. In other words, in such a language, only total functions can be defined while function f is partial.[6] In order to obtain a language that is Turing complete, it is necessary to include unbounded iteration. The complication of the informal semantics of `for` is only apparent, and the easy translation into machine language of `while` should not deceive us. In fact, from a formal viewpoint, `while` is semantically more complicated than `for`. As we saw in Chap. 2, the semantics of the `while` command is defined in terms of itself, something which, if at first sight it appears a little strange, finds its formal justification in fix point techniques which are beyond the scope of this text (and to which we will mention in the section on recursion, below). Even if we have not formally defined the semantics of `for`, the reader can convince themselves that this can be given in simpler terms (see Exercise 4). The major complication in the semantics of `while` with respect to `for` corresponds to the greater expressiveness of the first construct. It is in fact clear that every `for` command can easily be translated into a `while`.

It can now be asked why a language provides a bounded iteration construct when unbounded iteration constructs allow the same things to be done. The reply is principally of a pragmatic nature. The `for` it is a much more compact form of iteration, putting the three components of the iteration (initialisation, control and increment of the control variable) on the same line makes understanding what the loop does a lots easier; it can also prevent some common errors, such as forgetting the initialisation or increment of the control variable. The use of a `for` instead of a `while` can therefore be an important way to improve understanding and, therefore testing and maintenance of a program. There is also the implementation motive in some languages and on some architectures. A `for` a loop can often be compiled in a more efficient way (and, in particular, be optimised better) than a `while`, particularly as far as register allocation is concerned.

The for in C In C (and in its successors, among them Java), the `for` is far from being, the in a general case, a bounded iteration construct. The general version is:

[6] In reality there are other *total* functions that are not definable using only assignment, sequential composition, `if` and `for`. A famous example is the Ackerman function, for whose definition the reader is referred to texts on computability theory for example [3].

```
for (exp1; exp2; exp3)
   body
```

Its semantics is the following:

1. Evaluate `exp1`;
2. Evaluate `exp2`; if it is zero, terminate execution of the `for`;
3. Execute `body`;
4. Evaluate `exp3` and go to (2).

As can be seen, there is no way to freeze the value of the control expressions, nor is there any ban on the possibility of modifying the value of index (which, in the general case, need not even exist). It is clear how the semantics expresses the fact that in C, `for` is, when all is said and done, an abbreviation for a `while`.

Making use of the fact that, in C, a command is also an expression, we obtain the most usual form in which the `for` appears in C programs:

```
for (i = initial; i <= final; i += step){
   body
}
```

This is an abbreviation (which is very important pragmatically) of:

```
i = initial;
while (i <= final){
   body
   i += step;
}
```

For-each One of the most common iterative constructions performs the sequential scanning of all the elements of the data structure. A typical example is the following which contains a function which computes the sum of the values in an array of integers:

```
int sum(int[] A){
   int acc = 0;
   for(int i=0; i<length(A); i++)
      acc += A[i];
   return acc;
}
```

This function is full of details that the compiler knows: the first and last index of `A`, the specific check for `i` reaching its limit. The more detail that has to be added in a construct, the easier it is to make an error and much more difficult to understand at a glance what the construct does. In the case of `sum`, what we want to express is simply the application of the body to *every element of* `A`.

Hence, some languages use a special construct, called *for-each*, to perform this kind of operation. The *for-each* has the following general syntax:

```
foreach (FormalParameter : Expression) Command
```

The for-each construct expresses the application of `Command` (in which the *FormalParameter* can clearly appear) to each element of *Expression*.

Using such a construct, the operation of summing a vector can be written as:

```
int sum(int[] A){
   int acc = 0;
   foreach(int e : A)
      acc += e;
   return acc;
}
```

Here, we can read the header for the `for-each` as "for every element `e` in `A`". The vector index, together with all of the vector's limits, is hidden in a more synthetic and elegant construct.

The use of the `for-each` construct is not limited to vectors, but can also be applied to all collections over which the notion of iteration can be defined in a natural way. In addition to enumerations and sets (which we will see in Chap. 8), let us mention the particularly important case of languages which allow the user to define types which are " iterable".

Among the more common languages, Java Version 5 supports the `for-each` construct. The key word used is simply `for`[7] but the syntax is different and allows the construct to be disambiguated without problem. In Java the for-each (also called *enhanced for* in the documentation) is applicable to all subtypes of the library type `Iterable`.

6.4 Structured Programming

The rejection of the `goto` command of the 1970s was not an isolated phenomenon. The `goto`'s rejection was due to its properties, yet it was only one issue among many that contributed to a much wider debate which brought so-called *structured programming* to the fore. This can be considered as the antecedent of modern programming methodology. As the name itself suggests, it consists of a series of prescriptions aimed at allowing the development of software that has a certain structure in code and, correspondingly, in the flow of control. These prescriptions have both a methodological nature, providing precise development methods for programs, and a linguistic component, indicating appropriate typologies for the commands used (in substance, all those seen here so far, with the exception of `goto`). Let us see in more detail some salient points about structured programming and its associated linguistic implications.

[7]The for-each construct has been added on the fly when the language was already distributed and in use for some years. In a case like this, the modification of the set of reserved words is not a good design decision. Old programs which used the new keyword would stop working.

Top-down or hierarchical design of programs The program is developed by successive refinements, starting from a first (fairly abstract) specification adding successively extra detail at each step.

Code modularisation It is appropriate to group the commands which correspond to the specific functions in the algorithm that is to be implemented. To do this, all the linguistic mechanisms made available by the language are used; these range from compound commands to constructs for abstraction, such as procedures, functions, and real modules, where the language supports them.

Use of meaningful names The use of meaningful names for variables, as well as for procedures, etc., greatly simplifies the process of understanding the code and therefore eases making any changes required during maintenance. Even if this appears (and is) obvious, in practice it is too often ignored.

Extensive use of comments Comments are essential for understanding, testing, verification, correction and modification of code. A program with no comments, becomes rapidly incomprehensible, once it has reached a certain length.

Use of structured data types The use of appropriate datatypes, for example records, to group and structure information, even if it is of an heterogeneous type, eases both the design of code and its later maintenance. For example, if we can use the single variable, of type *student record*, to store the information about family name, registration number and a subscription year for a student, then the structure of the program will be much clearer than it would be if one had to use four different variables to hold the information about a single student.

Use of structured control constructs This, from a linguistic viewpoint, is the essential aspect. To implement structured programming, it is necessary to use structured control constructs, or rather constructs which, typically, have a single entry and a single exit point.

The last point is the one which interests us the most and merits extra study. The essential idea behind structured control constructs is that, by having a single entry and a single exit point, they allow structuring of the code in which the linear scanning of the program text corresponds to execution flow. If command `C2` textually a follows command `C1`, at the (unique) exit of command `C1`, when `C1` terminates, control passes to the (unique) entry point of command `C2`. Each command internally can have complex structure: branching (as in an `if`) or loops (as in `for`) with a non-linear control structure or internal jumps. The important thing is that each elementary component, externally, is visible in terms only of an entry point and an exit. This property, which is fundamental for the understanding of code, is violated in the presence of a command such as `goto` which allows jumps forward and backwards in the program. In such a case, the code can rapidly reach a state which is called "spaghetti code", where the control flow between the various program components, instead of being a simple graph with few the edges (which connect the output of a command to the input of the following one), is described by a graph in which the edges resemble a plate of spaghetti.

The control constructs seen so far, except the `goto`, are all structured and are the ones left in the modern programming languages. From a theoretical position, they allow programs for all computable functions to be written, as we have already observed. From the pragmatic point of view they are sufficient to express all types of control flow present in real applications. In particular, the constructs that were discussed at the end of Sect. 6.3.1 enable us to handle those cases in which we have to exit from a loop, a procedure, or, in some way, interrupt processing before "normal" termination occurs. All of these cases could be handled in a natural manner using a `goto`. For example, if we wished to process all the elements of a file that we have read from an external device, we could use code of the form:[8]

```
while true do{
   read(X);
   if X = end_of_file then goto end;
   elaborate(X);
}
end: ...
```

It can be observed that this use of `goto` does not violate the single entry, single exit principle because the jump only anticipates the exit, which happens at a single point in the overall construct. The structured command `break` (or its analogues) is the canonical form of this "jump to end of loop". When written in place of `goto end`, it makes the program clearer, and omits the label (the destination of the jump implicit in the `break` is the unique exit from the construct).

Let us finally recall that structured programming constituted a first reply to the demands of so-called programming in the large,[9] given that it requires the decomposition of a system of vast dimensions into different components, each of which is assigned a certain level of independence. The amount of independence depends on the abstraction mechanisms being used. For example, if procedures are used, communication between the various components can happen only through parameters. More significant answers to the needs of programming in the large cannot, though, be given solely at the linguistic level of programming languages. Software engineering has studied many methodologies for managing projects and implementing big software systems. Some of these methodologies also have linguistic implications, which, however, cannot be completely accounted for in this book. The object-oriented paradigm, together with some specification formalisms for object-oriented projects (like UML), are some of the more recent replies to issues we consider in this text.

6.5 Recursion

Recursion is another mechanism, an alternative to iteration, for obtaining Turing-equivalent programming languages. In empirical terms, a function (or procedure)

[8]Wishing to avoid `goto` and using only `while` and `if`, we find that we have to write code that is much less natural.

[9]This term denotes the implementation of large-scale software systems.

Inductive Definitions

Using an axiomatic presentation due to Giuseppe Peano, the natural numbers (non-negative integers) 0, 1, 2, 3, ..., can be defined as the least set X, satisfying the following rules:

1. $0 \in X$
2. If $n \in X$, then $n + 1 \in X$,

where we assume as primitive the concept of 0 (zero), that of number (denoted by n) and that of successor of n (written $n + 1$). This definition of the naturals clearly provides an intuitive justification of the principle of induction, a fundamental tool in mathematics and also, in some ways, in computer science. This principle can be stated as follows. To prove that a property, $P(n)$, is true for all natural numbers, it is necessary to show that the following two conditions are satisfied:

1. $P(0)$ is true;
2. For every natural, n, if $P(n)$ is true, then so is $P(n + 1)$.

In addition to the proof of properties, induction is a powerful tool for the definition of functions. In fact, it can be shown that if $g : (\mathbb{N} \times X) \to X$ is a total function, $\mathbb{N}$ denotes natural numbers and a is an element of X, then there exists a unique (total) function $f : \mathbb{N} \to X$, such that:

1. $f(0) = a$
2. $f(a + 1) = g(n, f(n))$

Such a pair of equations then provide an *inductive definition* of the function f.

What has so far been said about induction over the naturals can be generalised to arbitrary sets over which a well-founded ordering relation $<$ is defined, i.e. a relation which does not admit infinite descending chains $\cdots x_m < \cdots < x_1 < x_0$. In this case, the induction principle, called well-founded induction, can be expressed as follows. Let $<$ be a well-founded relation defined on a set A. To show that $P(a)$ has a value for every a belonging to A, it is necessary to show the following implication:

For all $a \in A$, if $P(b)$ is true for every $b < a$, then $P(a)$ is true.

that is recursive is a procedure in whose body a call to itself is included. One can have indirect recursion as well (it is best called *mutual recursion*) when a procedure P calls another procedure, Q, which, in its turn, calls P. In Chap. 5, we have already seen the example of the recursive function `fib` which compute the nth value of the Fibonacci function.

```
int fib (int n){
   if (n == 0)
      return 1;
   else
      if (n == 1)
```

```
         return 1;
      else
         return fib(n-1) + fib(n-2);
}
```

The fact that a function like `fib` can be defined in terms of itself can evoke some doubt about the nature of the function being defined. In reality, recursive definition, also called inductive definition, is fairly common in mathematics. As the box on page 153 shows in more detail, the idea is that of describing the result of the application of a function f to an argument X in terms of the application of f itself to arguments which are "smaller" than X. The domain on which f is defined must be such that it does not allow infinite chains of successively smaller elements, thus ensuring that, after a finite number of applications of the function f, we arrive at a terminal case, by the definition of which we can reconstruct the value of f applied to X. For example, recalling that the factorial of a natural number, n, is given by the product $1 \cdot 2 \cdots n-1 \cdot n$, we can inductively define the function computing the factorial as follows:

$$\begin{aligned} &\mathit{factorial}(0) = 1, \\ &\mathit{factorial}(n+1) = (n+1) \cdot \mathit{factorial}(n). \end{aligned}$$

Here n is an arbitrary natural number. In an analogous fashion, we can define the function which computes the nth term in the Fibonacci series, for which we have first provided a recursive program.

If, then, inductive definitions in mathematics and recursive functions in programming languages are similar, there is still a fundamental difference. In the case of inductive definitions, not all possible definitions of function in terms of itself will work. If for example we write:

$$\begin{aligned} &\mathit{foo}(0) = 1, \\ &\mathit{foo}(n) = \mathit{foo}(n) + 1 \quad \text{for } n > 0 \end{aligned}$$

it is clear that no total function over the naturals will satisfy this equation, so we cannot define such a function. If, on the other hand, we write:

$$\mathit{fie}(1) = \mathit{fie}(1)$$

the problem is now the opposite. Many functions satisfy these equations, so once again this does not constitute a valid definition.

On the other hand it is perfectly legitimate to write the following function in any programming language supporting recursion:

```
int foo1 (int n){
   if (n == 0)
      return 1;
   else
      return foo1(n) + 1;
}
```

And also:

```
int fie1 (int n){
   if (n == 1) return fie1(1);
}
```

These are functions which, in some case, do not terminate (when $n > 0$ for `foo1(n)` and for $n = 1$ in the case of `fie1(n)`), but from the semantic viewpoint there is no problem because, as we saw in Chap. 3, programs define partial functions.

6.5.1 Tail Recursion

In Chap. 5, we saw how, in general, the presence of recursion in a programming language makes it necessary to include dynamic memory management since it is not possible statically to determine the maximum number of instances of a single function that will be active at the same time (and, therefore, to determine the maximum number of activation records required). For example, we have seen that for the call `fib(n)` in our Fibonacci function, this number is equal to the nth element of the Fibonacci series, and obviously, n cannot be known as compilation time. If, on the other hand, we more carefully consider the nature of the recursive calls, we notice that in some cases we can avoid the allocation of new activation records for successive calls of a single function, since we can always reuse the same memory space. To understand this point let us look at two recursive functions which compute the factorial of a natural number. The first is the usual:

```
int fact (int n){
   if (n <= 1)  return 1;
   else
      return n * fact(n-1);
}
```

The activation record for a call `fib(n)` is shown in a slightly simplified form in Fig. 6.4. The field n contains the value of the actual parameter to the procedure; the field *Intermediate Result* will contain the intermediate result produced by the evaluation of `fact(n-1)`; the field *Result address*, finally, contains the address of the memory area in which the result must be returned (that is, the address the *Intermediate Result* of the caller for all calls after the first). It is important to note that the value of the *Intermediate Result* field present in the activation record of `fact(n)`, can be determined only when the reclusive call to `fact(n-1)` terminates and the value of this field, as a result of the code of `fact`, is used in the computation of `n * fact(n-1)` to obtain the value of `fact(n)`. In other words, when we have the call `fact(n)`, before it can terminate, we must know the value of `fact(n-1)`; in its turn, for the call to `fact(n-1)` to terminate we have to know the value of `fact(n-2)` and so on, recursively, right back to the terminal

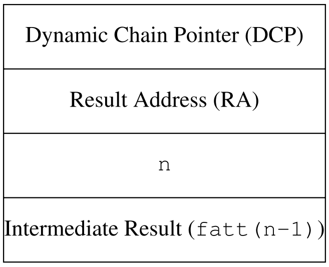

Fig. 6.4 Activation record for the `fact` function

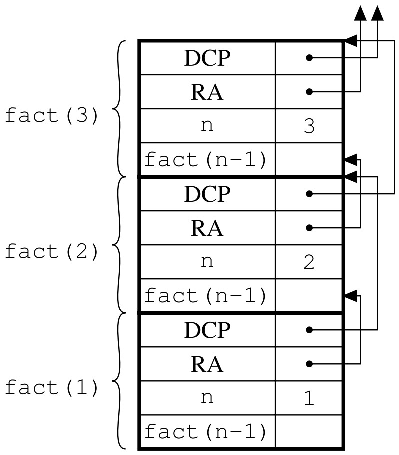

Fig. 6.5 Activation record stack and for the call `fib(3)` and the two recursive calls `fact(2)` and `fact(3)`

case `fact(1)`. Therefore all activation records for the recursive calls `fact(n)`, `fact(n-1)`,...,`fact(1)`, must reside at the same time on the stack, in distinct memory areas.

Figure 6.5 shows the stack of activation records created by the call to `fact(3)`, as well as the by the following calls to `fact(2)` and `fact(1)`. When we reach the final case, the call to `fact(1)`, terminates immediately and returns the value `1`, which, using the pointer contained in the *Result Address* of the activation record for `fact(1)`, will be returned to the *Intermediate result* field of the activation record for `fact(2)`, as is shown in Fig. 6.6. At this point, too, the call to `fact(2)` can terminate, returning the value $2 \cdot fact(1) = 2 \cdot 1$ to the call `fact(3)`, as shown in Fig. 6.7. Finally, too, the call to `fact(3)` will terminate, returning to the calling program the value $3 \cdot fact(2) = 3 \cdot 2 = 6$.

Consider now another function:

```
int factrc (int n, int res){
   if (n <= 1)
      return res;
   else
      return factrc(n-1, n * res)
}
```

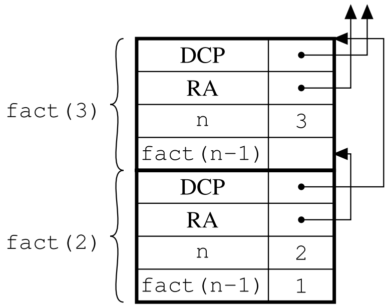

Fig. 6.6 Stack of activation records after termination of the call `fact(2)`

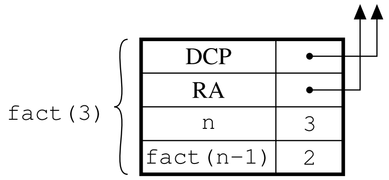

Fig. 6.7 Activation record stack after the termination of the call `fact(2)`

This, if called with `factrc(n,1)` also returns the factorial of n.

In this case, also, the initial call `factrc(n,1)` produces $n-1$ following recursive calls

```
factrc(n-1,n*1),
factrc(n-2,(n-1)*n*1),
...,
factrc(1,2*...*(n-1)*n*1).
```

However, let us now observe that for $n > 1$, the value returned by the generic call `factrc(n,res)` is *exactly the same* as the value returned by the next recursive call `factrc(n-1,n+res)` without there being any additional computation. The value finally returned from the initial call `factrc(n,1)` is therefore the same as that returned by the last recursive call `factrc(1,2*...*(n-1)*n*1)` (and is therefore $n \cdot (n-1) \cdot (n-2) \cdots 1$), without there being any requirement to "climb back up" the call chain using then the intermediate results to compute the final value, as on the other hand happens in the case of `fact`.

From what has been said, it appears clear that, once `factrc(n,res)` has recursively called `factrc(n-1,n*res)`, there is no need to continue maintaining the information present in the activation record for the call `factrc(n,res)`, given that all the information necessary to perform the calculation of the final result is passed to `factrc(n-1,n*res)`. This means that the activation record for the recursive call `factrc(n-1,n*res)` can simply reuse the memory space allocated to the activation record for `factrc(n,res)`. So the consideration is valid also for successive calls and, therefore, in short, the function `factrc` will need a single memory area to allocate a single activation record, independent of the number of recursive calls to be made. We have therefore obtained a recursive function for which the memory can be allocated statically!

Recursion of the kind illustrated by the function `factrc` is said to be *tail recursion* since the recursive call is, so to speak, the last thing that happens in the body of the procedure. After the recursive call, no other computation is performed. More generally we can give the following definition.

Definition 6.3 (Tail recursion) Let f be a function which, in its body, contains a call to a function g (different from f or equal to f). The call of g is said to be a *tail call* if the function f returns the value returned by g without having to perform any other computation. We say that the function f is tail recursive if all the recursive calls present in f are tail calls.

For example, in the function:

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

The first two recursive calls are tail calls, the third is not a tail call. Therefore the function f is not tail recursive.

Our interest in tail recursion lies in the possibility of implementing it using a single activation record and therefore using constant memory space. Our investigation of the `factrc` function, in fact, is completely general, and does not depend on the specific form of this function; it depends only on the fact that we are dealing with a tail recursive function. All of this, however, does not hold in the case in which we also consider higher-order functions (i.e. when they are functions that are passed as parameters), as we will see later in this section.

In general, it is always possible to transform a function which is not tail recursive into an equivalent one which is, by complicating the function. The idea is that all the computations which have to be made after the recursive call (and make the function non-tail recursive) should, as far as possible, be performed before the call. The part of the work which cannot be done before the recursive call (because, for example, it uses its results) is "passed" with appropriate additional parameters, to the recursive call itself. This technique is exactly the same as the one we used in the case of the tail-recursive function `factrc`, where, instead of recursively calculating the product `n*fact(n-1)` in the body of the call to `fact(n-1)`, we have added a parameter `res` which allows us to pass to the product $n \cdot (n-1) \cdot (n-2) \cdots j$ to the generic recursive call `factrc(j- 1,j* res)`. Therefore, in this case, the calculation of the factorial is performed incrementally by the successive recursive calls, in a way analogous to that performed by an *iterative* function such as the following:

```
int fact-it (int n, int res){
   res=1;
   for (i=n; i>=1; i--)
      res = res*i;
}
```

In an analogous fashion to this, we can also transform the `fib` function into a function with tail recursion `fibrc`, by addition of two additional parameters:

```
int fibrc (int n, int res1, int res2){
   if (n == 0)
      return res2;
   else
      if (n == 1)
         return res2;
      else
         return fibrc(n-1,res2,res1+res2);
}
```

The call `fibrc(n,1,1)` returns the *n*th value in the Fibonacci series. Clearly, both in the case of `fibrc` and in that of `factrc`, if we want to make the additional parameters invisible, we can encapsulate the functions inside others which have only the parameter *n*. For example the scope of the declaration of `fibrc`, we can define:

```
int fibrctrue (int n){
   return fibrc(n,1,1);
}
```

The transformation of one function into an equivalent one with tail recursion can be done automatically using a technique called *continuation passing style*, which basically consists of representing, at all points in a program, that part of the program that remain using a function called a *continuation*. If we want to convert a function into a tail-recursive one, it suffices to use a continuation to represent everything remains of the computation and passing this continuation to the recursive call. This technique however does not always produce functions which can be executed with constant memory requirement because the continuation, since it is a function, could contain the variables which will be evaluated in the environment of the caller and therefore require the caller's activation record.

6.5.2 Recursion or Iteration?

Without going into detail on theoretical results (which are, in any case, extremely important and interesting), let us recall that recursion and iteration (in its most general form) are alternative methods for achieving the same expressive power. The use of the one or the other is often due more to the predisposition of the programmer than to the nature of the problem. For the elaboration of data using rigid structures (matrices, tables, etc.), as normally happens in numerical or data processing applications, it is often easier to use iterative constructs. When, on the other hand, processing structures of a symbolic nature which naturally lend themselves to being defined in a recursive manner (for example, lists, trees, etc.), it is often more natural to use recursion.

Recursion is often considered much less efficient than iteration and therefore declarative languages are thought much less efficient than imperative ones. The argument presented above about tail recursion make us understand that recursion is not necessarily less efficient than iteration, both in terms of memory occupation and in terms of execution time. Certainly naive implementations of recursive functions, such as those often resulting from the direct translation of inductive definitions, can be fairly inefficient. This is the case, for example, for our procedure `fib(n)` which has execution time and memory occupation that are exponential in n. However, as was seen, using recursive functions that are more "astute", such as those with tail recursion, we can obtain performance similar to that of the corresponding iterative program. The function `fibrc` in fact uses a space of constant size and runs in time linear in n.

Regarding then, the distinction between imperative and declarative languages, things are more complex and will be analysed in the chapters dedicated to the functional and logic programming paradigms.

6.6 Chapter Summary

In this chapter, we have analysed the components of high-level languages relating to the control of execution flow in programs. We first considered expressions and we have analysed:

- The types of syntax most used for their description (as trees, or in prefix, infix and postfix linear form) and the related evaluation rules.
- Precedence and associativity rules required for infix notation.
- The problems generally related to the order of evaluation of the subexpressions of an expression. For the semantics of expressions to be precisely defined, this order must be precisely defined.
- Technical details on the evaluation (short-circuit, or lazy evaluation) used by some languages and how they can be considered when defining the correct value of an expression.

We then passed to commands, seeing:

- Assignment. This is the basic command in imperative languages. Even though it is fairly simple, we had to clarify the notion of variable in order to understand its semantics.
- Various commands which allow the management of control (conditionals and iteration) in a structured fashion.
- The principles of structured programming, stopping to consider age-old questions about the `goto` command.

The last section, finally, dealt with recursion, a method that stands as an alternative to iteration for expressing algorithms. We concentrated on tail recursion, a form of recursion that is particularly efficient both in space and time. This must clear up the claim that recursion is a programming method that is necessarily less efficient than iteration.

In the various boxes, we examined an historic theme that has been extremely important in the development of modern programming languages, as well as a semantic issue which precisely clarifies the difference that exists between imperative, functional and logic programs.

We still have to deal with important matters concerning control abstraction (procedures, parameters and exceptions) but this will be the subject of the next chapter.

6.7 Bibliographical Notes

Many texts provide an overview of the various constructs present in programming languages. Among these, the most complete are [7] and [8].

Two historical papers, of certain interest to those who want to know more about the `goto` question are [4] (in which Böhm and Jacopinin's theorem is proved) and Dijkstra's [5] (where the "dangerousness" of the jump command is discussed).

An interesting paper, even though not for immediate reading, which delves into themes relating to inductive definitions is [2]. For an introduction to recursion and induction that is more accessible, [9] is a very good book.

According to Abelson and Sussman [1], the term "syntactic sugar" is due to Peter Landin, a pioneer in Computer Science who made fundamental contributions in the area of programming language design.

6.8 Exercises

1. Define, in any programming language, a function, f, such that the evaluation of the expression $(a + f(b)) * (c + f(b))$ when performed from left-to-right has a result that differs from that obtained by evaluating right-to-left.
2. Show how the `if then else` construct can be used to simulate short-circuit evaluation of boolean expressions in a language which evaluates all operands before applying boolean operators.
3. Consider the following case command:

```
case Exp of
   1:    C1;
   2,3:  C2;
   4..6: C3;
   7:    C4
   else: C5
```

Provide an efficient pseudocode assembly program that corresponds to the translation of this command.

4. Define the operational semantics of the command

```
for I = start to end by step do body
```

using the techniques introduced in Chap. 3. Suggestion: using values of the expressions, `start`, `end` and `step`, the following can be computed before executing the `for`: the number, *ic*, of repetitions which must be performed (it is assumed, as already stated in the chapter, that the possible modification of `I`, `start`, `end` and `step` in the body of the `for` have no effect upon their evaluation). Once this value, *n*, has been computed, the `for` can be replaced by a sequence of *ic* commands.

5. Consider the following function:

```
int ninetyone (int x){
   if (x>100)
    return x-10;
   else
    return ninetyone(ninetyone(x+11));
   }
```

Is this tail recursive? Justify your answer.

6. The following code fragment is written in a pseudo-language which admits bounded iteration (numerically controlled) expressed using the `for` construct.

```
z=1;
for i=1 to 5+z by 1 do{
   write(i);
   z++;
}
write(z);
```

What is printed by `write`?

7. Say what the following code block prints. This code is written in a language with static scope and call by name. The definition of the language contains the following phrase: "The evaluation of the expression $E_1 \circ E_2$ where $\circ$ is any operator, consists of (i) the evaluation of E_1; (ii) then the evaluation of E_2; (iii) finally, the application of the operator $\circ$ to the two values previously obtained."

```
{int x=5;
 int P(name int m){
      int x=2;
      return m+x;
 }
 write(P(x++) + x);
}
```

References

1. H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, 1996.
2. P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, Amsterdam, 1977.
3. M. Aiello, A. Albano, G. Attardi, and U. Montanari. *Teoria della Computabilità, Logica, Teoria dei Linguaggi Formali*. ETS, Pisa, 1976 (in Italian).
4. C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966.
5. E. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.
6. B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, 1986.
7. T. Pratt and M. Zelkowitz. *Programming Languages: Design and Implementation*. Prentice-Hall, New York, 2001 (quarta edizione).
8. M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, San Mateo, 2000.
9. G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, 1993.