1.4 Chapter Summary

The chapter has introduced the concepts of abstract machine and the principle methods for implementing a programming language. In particular, we have seen:

- The *abstract machine*: an abstract formalisation for a generic executor of algorithms, formalised in terms of a specific programming language.
- The *interpreter*: an essential component of the abstract machine which characterises its behaviour, relating in operational terms the language of the abstract machine to the embedding physical world.
- The *machine language*: the language of a generic abstract machine.
- *Different language typologies*: characterised by their distance from the physical machine.
- The *implementation of a language*: in its different forms, from purely interpreted to purely compiled; the concept of *compiler* is particularly important here.
- The *concept of intermediate language*: essential in the real implementation of any language; there are some famous examples (P-code machine for Pascal and the Java Virtual Machine).
- *Hierarchies of abstract machines*: abstract machines can be hierarchically composed and many software systems can be seen in such terms.

1.5 Bibliographic Notes

The concept of abstract machine is present in many different contexts, from programming languages to operating systems, even if at times it is used in a much more informal manner than in this chapter. In some cases, it is also called a virtual machine, as for example in [5], which, however, presents an approach similar to that adopted here.

The descriptions of hardware machines that we have used can be found in any textbook on computer architecture, for example [6].

The intermediate machine was introduced in the first implementations of Pascal, for example [4]. For more recent uses of intermediate machine for Java implementations, the reader should consult some of the many texts on the JVM, for example, [3].

Finally, as far as compilation is concerned, a classic text is [1], while [2] is a more recent book with a more up-to-date treatment.

1.6 Exercises

1. Give three examples, in different contexts, of abstract machines.
2. Describe the functioning of the interpreter for a generic abstract machine.

3. Describe the differences between the interpretative and compiled implementations of a programming language, emphasising the advantages and disadvantages.
4. Assume you have available an already-implemented abstract machine, C, how could you use it to implement an abstract machine for another language, L?
5. What are the advantages in using an intermediate machine for the implementation of a language?
6. The first Pascal environments included:

   - A Pascal compiler, written in Pascal, which produced P-code (code for the intermediate machine);
   - The same compiler, translated into P-code;
   - An interpreter for P-code written in Pascal.

   To implement the Pascal language in an interpretative way on a new host machine means (manually) translating the P-code interpreter into the language on the host machine. Given such an interpretative implementation, how can one obtain a compiled implementation for the same host machine, minimising the effort required? (Hint: think about a modification to the compiler for Pascal also written in Pascal.)
7. Consider an interpreter, $\mathscr{I}^{\mathscr{L}}_{\mathscr{L}1}(X, Y)$, written in language $\mathscr{L}$, for a different language, $\mathscr{L}1$, where X is the program to be interpreted and Y is its input data. Consider a program P written in $\mathscr{L}1$. What is obtained by evaluating

   $$\mathscr{P}eval_{\mathscr{L}}(\mathscr{I}^{\mathscr{L}}_{\mathscr{L}1}, P)$$

   i.e., from the partial evaluation of $\mathscr{I}^{\mathscr{L}}_{\mathscr{L}1}$ with respect to P? (This transformation is known as Futamura's first projection.)

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, 1988.
2. A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, 1998. This text exists also for C and ML.
3. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, 2nd edition. Sun and Addison-Wesley, Cleveland, 1999.
4. S. Pemberton and M. Daniels. *Pascal Implementation: The p4 Compiler and Interpreter*. Ellis Horwood, Chichester, 1982.
5. T. Pratt and M. Zelkowitz. *Programming Languages: Design and Implementation*, 4th edition. Prentice-Hall, New York, 2001.
6. A. Tannenbaum. *Structured Computer Organization*. Prentice-Hall, New York, 1999.

Church's Thesis

The proofs of equivalence of the various programming languages (and between the various computability formalisms) are genuine theorems. Given the languages $\mathscr{L}$ and $\mathscr{L}'$, write first in $\mathscr{L}$ the interpreter for $\mathscr{L}'$ and then write in $\mathscr{L}'$ the interpreter for $\mathscr{L}$. At this point $\mathscr{L}$ and $\mathscr{L}'$ are known to be equivalent. A proof of this type has been effectively given for all existing languages, so they are therefore provably equivalent. This argument would, in reality, leave open the door to the possibility that sooner or later someone will be in a position to find an *intuitively computable* function which is not associated with a program in any existing programming language. All equivalence results proved over more than the last 70 years, however, amount to convincing evidence that this is impossible. In the mid-1930s, Alonzo Church proposed a principle (which since then has become known as Church's, or the Church-Turing, Thesis) that states exactly this impossibility. We can formulate Church's Thesis as: every intuitively computable function is computed by a Turing Machine.

In contrast to the equivalence results, Church's Thesis is not a theorem because it refers to a concept (that of intuitive computability) which is not amenable to formal reasoning. It is, rather, a philosophical principle which the computer science community assumes with considerable strength to be true so that it will not even be discredited by new computational paradigms, for example quantum computing.

a function (a sequence) which did not belong to the enumeration. Therefore the cardinality of $\mathscr{F}$ is strictly greater than that of $\mathbb{N}$.

It can be shown that $\mathscr{F}$ has the cardinality of the real numbers. This fact indicates that the set of programs (which is denumerable) is much smaller than that of all possible functions, and therefore of all possible problems.

3.5 Chapter Summary

The phenomenon of computation on which Computer Science is founded has its roots in the theory of computability which studies the formalisms in which one can express algorithms and their limits. The chapter has only presented the main result of this theory. This is a fact of the greatest importance, one that every computer scientist should know. The principal concepts which were introduced are:

- *Undecidability*: there exist many important properties of programs which cannot be determined in a mechanical fashion by an algorithm; amongst these is the halting problem.
- *Computability*: a function is computable when there exists a program which computes it. The undecidability of the halting problem assures us that there exist functions which are not computable.

- *Partiality*: the functions expressed by a program can be undefined on some arguments, corresponding to those data for which the program will fail to terminate.
- *Turing Completeness*: every general-purpose programming language computes the same set of functions as those computed by a Turing Machine.

3.6 Bibliographical Notes

The original undecidability result is in the paper by A.M. Turing [3] which ought to be necessary reading for every computer scientist with an interest in theory. More can be found on the arguments of this chapter in any good textbook on computability theory, among which, let us recommend [1], which we cited in Chap. 2, and the classic [2] which after more than 40 years continues to be one of the most authoritative references.

3.7 Exercises

1. Proof that the restricted halting problem is undecidable. That is, determine whether a program terminates when it is applied to itself. (Suggestion: if the problem were decidable, the program which decides it would have to have the same property as program K, which can be derived by contradiction in the usual fashion.)
2. Show that the problem of verifying whether a program computes a constant function is undecidable. Hint: given a generic program P, consider the program Q_P, with a single input, specified as follows:

$$Q_P(y) = \begin{cases} 1 & \text{if } P(P) \text{ terminates,} \\ \text{does not terminate} & \text{otherwise.} \end{cases}$$

   (i) Write the program Q_P;
   (ii) assume now that P is a program such that $P(P)$ terminates. What is the behaviour of Q_P, as y varies?
   (iii) what is, on other hand, the behaviour of Q_P, as y varies, if $P(P)$ does not terminate?
   (iv) from (ii) and (iii), it can be obtained that Q_P computes the constant function *one* if and only if $P(P)$ terminates;
   (v) if it were now decidable whether a program computes a constant function, the restricted halting problem would also be decidable, given that the transformation that, given P, constructs Q_P is completely general.

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

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

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

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

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

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

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

4.4 Chapter Summary

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

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

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

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

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

4.5 Bibliographical Notes

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

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

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

4.6 Exercises

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

References

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

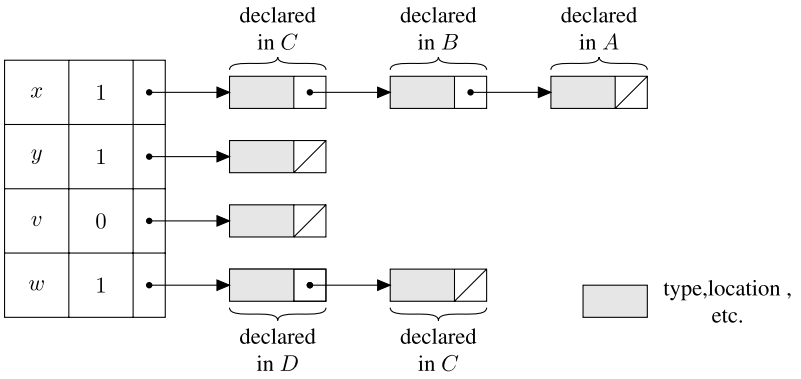
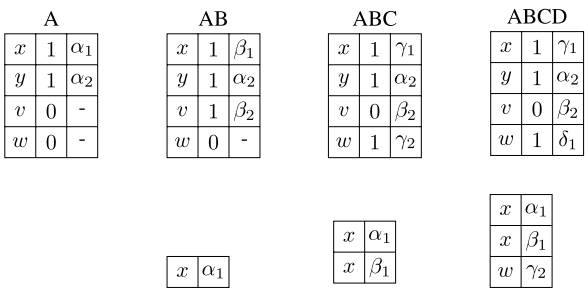


Fig. 5.19 Environment for block D in Fig. 5.16 after the call sequence A, B, C, D, with dynamic scope implemented using a CRT

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



5.6 Chapter Summary

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

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

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

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

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

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

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

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

5.7 Bibliographic Notes

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

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

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

5.8 Exercises

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

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

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

....
  }
```

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

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

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

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

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

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

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

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

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

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

References

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

(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);
}
```

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

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

7.4 Chapter Summary

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

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

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

Fig. 7.25 Static scope and exception names

Exceptions and Static Scope

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

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

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

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

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

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

7.5 Bibliographical Notes

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

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

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

7.6 Exercises

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

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

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

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

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

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

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

What is the most probable behaviour of the interpreter?

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

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

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

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

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

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

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

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

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

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

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

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

References

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

A stop and copy garbage collector can be made arbitrarily efficient, provided that there is enough memory for the two semi-spaces. In fact, the time required for a stop and copy is proportional to the number of live objects present in the heap. It is not unreasonable to assume that this quantity will be approximately constant at any moment during the execution of the program. If we increase the memory for the two semi-spaces, we will decrease the frequency with which the collector is called and, therefore, the total cost of garbage collection.

8.12 Chapter Summary

This chapter has dealt with a crucial aspect in the definition of programming languages: the organisation of data in abstract structures called data types. The main points can be summarised as follows.

- *Definition of type* as a set of values and operations and the role of types in design, implementation and execution of programs.
- *Type systems* as the set of constructs and mechanisms that regulate and define the use of types in a programming language.
- The distinction between *dynamic* and *static* type checking.
- The concept of type-safe systems, that is safe with respect to types.
- The primary *scalar types*, some of which are *discrete* types.
- The primary *composite types*, among which we have discussed *records*, *variant records* and *unions*, *arrays* and *pointers* in detail. For each of these types, we have also presented the primary storage techniques.
- The concept of *type equivalence*, distinguishing between equivalence by name and structural equivalence.
- The concept of *compatibility* and the related concepts of coercion and conversion.
- The concept of *overloading*, when a single name denotes more than one object and static disambiguation.
- The concept of *universal polymorphism*, when a single name denotes an object that belongs to many different types, finally distinguishing between parametric and subtype polymorphism.
- *Type inference*, that is mechanisms that allow the type of a complex expression to be determined from the types of its elementary types.
- Techniques for runtime checking for *dangling references*: tombstones and locks and keys.
- Techniques for *garbage collection*, that is the automatic recovery of memory, briefly presenting collectors based on reference counters, mark and sweep, mark and compact and copy.

Types are the core of a programming language. Many languages have similar constructs for sequence control but are differentiated by their type systems. It is not possible to understand the essential aspects of other programming paradigms, such as object orientation, without a deep understanding of the questions addressed in this chapter.

8.13 Bibliographic Notes

Ample treatments of programming language types can be found in [14] and the rather older [4]. Review articles which introduce the mathematical formalisms necessary for research in this area are [2, 11]. A larger treatment of the same arguments is to be found in [13]. On overloading and polymorphism in type systems, a good, clear review is [3] (which we have largely followed in this chapter).

Tombstones originally appeared in [8] (also see, by the same author, [9]). Fisher and Leblanc [5] proposed locks and keys, as well as techniques so that an abstract Pascal machine can make variant records secure.

The official definition of ALGOL68 (which is quite a read) is [15]. A more accessible introduction can be found in [12]. The definition of ML can be found in [10], while an introductory treatment of type inference is to be found in [1].

There is a large literature on garbage-collection techniques. A detailed description of a mark and sweep algorithm can be found in many algorithm books, for example [6], while [16] is a good summary of classical techniques. For a book entirely devoted to the problem, that contains pseudocode and a more-or-less complete bibliography (up to the time of publication), [7] is suggested.

8.14 Exercises

1. Consider the declaration of the multi-dimensional array:

   ```
   int A[10][10][10]
   ```

   We know that an integer can be stored in 4 bytes. The array is stored in row order, with increasing memory addresses (that is, if an element is at address i, its successor is at $i + 4$, and so on). The element `A[0][0][0]` is stored at address 0. State the address at which element `A[2][2][5]` is stored.
2. Instead of the contiguous multidimensional array allocation that we discussed in Sect. 8.4.3, some languages allow (e.g., C), or adopt (Java), a different organisation, called *row-pointer*. Let us deal with the case of a 2-dimensional array. Instead of storing rows one after the other, every row is stored separately in some region of memory (for example on the heap). Corresponding to the name of the vector, a vector of pointers is allocated. Each of the pointers points to a proper row of the array. (i) Give the formula for accessing an arbitrary element `A[i][j]` of an array allocated using this scheme. (ii) Generalise this memory technique to arrays of more than 2 dimensions. (iii) Discuss the advantages and disadvantages of this organisation in the general case. (iv) Discuss the advantages and disadvantages of this technique for the storage of 2-dimensional arrays of characters (that is, arrays of strings).
3. Consider the following (Pascal) declarations:

   ```
   type string = packed array [1..16] of char;
   ```

```
type string_pointer =  ^string;
type person = record
                name : string;
                case student: Boolean of
                   true:  (year: integer);
                   false: (socialsecno: string_pointer)
              end;
```

and assume that the variable C contains a pointer to the string "LANGUAGES". Describe the memory representation for the person record after each of the following operations:

```
var pippo : person;
pippo.student := true;
pippo.year := 223344;
pippo.student := true;
pippo.socialsecno := C;
```

4. Show that the integer type can be defined as a recursive type starting with just the value null. Then write a function that checks if two integers defined in this way are equal.
5. Given the following type definitions in a programming language which uses structural type equivalence:

```
type T1 = struct{
            int a;
            bool b;
          };
type T2 = struct{
            int a;
            bool b;
          }
type T3 = struct{
            T2 u;
            T1 v;
          }
type T4 = struct{
            T1 u;
            T2 v;
          }
```

   In the scope of the declarations T3 a and T4 b, it is claimed that the assignment a = b is permitted. Justify your answer.
6. Which type is assigned to each of the following functions using polymorphic type inference?

```
fun G(f,x){return f(f(x));}
fun H(t,x,y){if (t(x)) return x;
         else return y;}
fun K(x,y){return x;}
```

7. Using tombstones, it is necessary to invalidate a tombstone when its object is no longer meaningful. The matter is clear if the object is on the heap and is explicitly deallocated. It is less clear when the tombstone is associated with an address on the stack. In this case, indeed, it is necessary for the abstract machine to be able to determine all the tombstones that are potentially associated with an activation record. Design a possible organisation which makes this operation reasonably efficient (recall that tombstones are not allocated in activation records but in the cemetery).
8. Consider the following fragment in a pseudo-language with reference-based variables and which uses locks and keys (`C` is a class whose structure is of no importance):

```
C foo = new C();  // object OG1
C bar = new C();  // object OG2
C fie = foo;
bar = fie;
```

   Give possible values *after* the execution of the fragment for all the keys and all the locks involved.
9. Under the same assumptions as in Exercise 8, consider the following code fragment in which `free(p)` denotes the explicit deallocation of the object referred to by the pointer `p`:

```
class C { int n; C next;}
C foo = new C();  // object OG1
C bar = new C();  // object OG2
foo.next = bar;
bar.next = foo;
free(bar);
```

   For all the pointers in the fragment, give possible key values. For each object in the fragment, give possible lock values. In both cases, the values should be those *after* execution of the fragment terminates. After execution of the fragment, also execute the code `foo.n = 1; foo.next = 0;`. State what a possible result of this execution might be.
10. Consider the following fragment which is written in a pseudo-language with a reference-model for variables and a reference-counting garbage collector. If `OGG` is an arbitrary object in the heap, indicate by `OGG.cont` its (hidden) contents:

```
class C { int n; C next;}
C foo(){
   C p = new C();      // object OGG1
   p.next = new C();   // object OGG2
   C q = new C();      // object OGG3
   q.next = p.next;
   return p.next;
   }
C r = foo();
```

State what are the values of the reference counters for the three objects after execution of line 6 and then of line 9.

11. Under the same assumptions as in Exercise 10, state what the values of the reference counters are for the two objects after the execution of the following fragment. Which of the two can be returned to the free list?

```
C foo = new C();  // object OG1
C bar = new C();  // object OG2
C fie = foo;
bar = fie;
```

12. Under the same assumptions as in Exercise 10, state what the reference-counter values for the three objects after execution of the following fragment are. Which of them can be returned to the free list?

```
class C { int n; C next;}
C foo = new C();  // object OG1
bar = new C();    // object OG2
foo.next = bar;
bar = new C();    // object OG3
foo = bar;
```

13. Using your favourite programming language, define the data structures required to represent a binary tree. Then write detailed code that performs a preorder traversal of a tree without using the system stack and instead using the pointer-reversal technique.

References

1. L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987. citeseer.ist.psu.edu/cardelli88basic.html.
2. L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, 1997. citeseer.ist.psu.edu/cardelli97type.html.
3. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985. citeseer.ist.psu.edu/cardelli85understanding.html.
4. J. C. Cleaveland. *An Introduction to Data Types*. Addison-Wesley, Reading, 1986.
5. C. N. Fisher and R. J. LeBlanc. The implementation of run-time diagnostics in Pascal. *IEEE Trans. Softw. Eng.*, 6(4):313–319, 1980.
6. E. Horowitz and S. Sahni. *Fundamentals of Data Structures in Pascal*. Freeman, New York, 1994.
7. R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York, 1996.
8. D. B. Lomet. Scheme for invalidating references to freed storage. *IBM Journal of Research and Development*, 19(1):26–35, 1975.
9. D. B. Lomet. Making pointers safe in system programming languages. *IEEE Trans. Softw. Eng.*, 11(1):87–96, 1985.
10. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML—Revised*. MIT Press, Cambridge, 1997.