Programming language concepts

—*Third edition*

Carlo Ghezzi, Politecnico di Milano
Mehdi Jazayeri, Technische Universität Wien

John Wiley & Sons
New York Chichester Brisbane Toronto Singapore

ISBN 0-000-000000-0

ABCDEFGHIJ-DO-89

TABLE OF CONTENTS

1

Introduction

CHAPTER 1

This book is concerned with programming languages. Programming languages, however, do not exist in a vacuum: they are tools for writing software. A comprehensive study of programming languages must take this role into account. We begin, therefore, with a discussion of the software development process and the role of programming languages in this process. Sections 1.1. through 1.5 provide a perspective from which to view programming languages and their intended uses. From this perspective, we will weigh the merits of many of the language concepts discussed in the rest of the book.

Programming languages have been an active field of computer science for at least four decades. The languages that exist today are rooted in such historical developments, either because they evolved from previous versions, or because they derived inspiration from their predecessors. Such developments are likely to continue in the future. To appreciate this fragment of the history of science, we provide an overview of the main achievements in programming languages in Section 1.6.

Finally, Section 1.7 provides an overview of the concepts of programming languages that will be studied throughout this book. This section explains how the various concepts presented in the remaining chapters fit together.

1.1 Software development process

From the inception of an idea for a software system, until it is implemented and delivered to a customer, and even after that, the software undergoes gradual development and evolution. The software is said to have a *life cycle* composed of several phases. Each of these phases results in the development of either a part of the system or something associated with the system, such as a fragment of specification, a test plan or a users manual. In the traditional *waterfall model* of the software life cycle, the development process is a sequential combination of phases, each having well-identified starting and ending points, with clearly identifiable deliverables to the next phase. Each step may identify deficiencies in the previous one, which then must be repeated.

A sample software development process based on the waterfall model may be comprised of the following phases:

Requirement analysis and specification. The purpose of this phase is to identify and document the exact requirements for the system.These requirements are developed jointly by users and software developers. The success of a system is measured by how well the software mirrors these stated requirements, how well the requirements mirror the users' perceived needs, and how well the users' perceived needs reflect the real needs. The result of this phase is a requirements document stating what the system should do, along with users' manuals, feasibility and cost studies, performance requirements, and so on. The requirements document does not specify how the system is going to meet its requirements.

Software design and specification. Starting with the requirements document, software designers design the software system. The result of this phase is a system design specification document identifying all of the modules comprising the system and their interfaces. Separating requirements analysis from design is an instance of a fundamental "what/how" dichotomy that we encounter quite often in computer science. The general principle involves making a clear distinction between *what* the problem is and *how* to solve the problem. In this case, the requirements phase attempts to specify what the problem is. There are usually many ways that the requirements can be met. The purpose of the design phase is to specify a particular software architecture that will meet the stated requirements. The design method followed in this step can have a great impact on the quality of the resulting application; in particular, its understandability and modifiability. It can also affect the choice of the programming language to be used in system implementation.

Implementation (coding). The system is implemented to meet the design specified in the previous phase. The design specification, in this case, states the "what"; the goal of the implementation step is to choose how, among the many possible ways, the system shall be coded to meet the design specification. The result is a fully implemented and documented system.

Verification and validation. This phase assesses the quality of the implemented system, which is then delivered to the user. Note that this phase should not be concentrated at the end of the implementation step, but should occur in every phase of software development to check that intermediate deliverables of the process satisfy their objectives. For example, one should check that the design specification document is consistent with the requirements which, in turn, should match the user's needs. These checks are accomplished by answering the following two questions:

"Are we building the product right?"

"Are we building the right product?"

Two specific kinds of assessment performed during implementation are *module testing* and *integration testing*. Module testing is done by each programmer on the module he or she is working on to ensure that it meets its interface specifications. Integration testing is done on a partial aggregation of modules; it is basically aimed at uncovering intermodule inconsistencies.

Maintenance. Following delivery of the system, changes to the system may become necessary either because of detected malfunctions, or a desire to add new capabilities or to improve old ones, or changes that occurred in operational environment (e.g., the operating system of the target machine). These changes are referred to as maintenance. The importance of this phase can be seen in the fact that maintenance costs are typically at least as large as those of all the other steps combined.

Programming languages are used only in some phases of the development process. They are obviously used in the implementation phase, when algorithms and data structures are defined and coded for the modules that form the entire application. Moreover, modern higher-level languages are also used in the design phase, to describe precisely the decomposition of the entire application into modules, and the relationships among modules, before any detailed implementation takes place. We will next examine the role of the programming language in the software development process by illustrating the relationship between the programming language and other software development tools in Section 1.2 and the relationship between the programming language and design methods in Section 1.3.

1.2 Languages and software development environments

The work in any of the phases of software development may be supported by computer-aided tools. The phase currently supported best is the coding phase, with such tools as text editors, compilers, linkers, and libraries. These tools have evolved gradually, as the need for automation has been recognized. Nowadays, one can normally use an interactive editor to create a program and the file system to store it for future use. When needed, several previously created and (possibly) compiled programs may be linked to produce an executable program. A debugger is commonly used to locate faults in a program and eliminate them. These computer-aided program development tools have increased programming productivity by reducing the chances of errors.

Yet, as we have seen, software development involves much more than programming. In order to increase the productivity of software development, computer support is needed for all of its phases. By a *software development environment* we mean an integrated set of tools and techniques that aids in the development of software. The environment is used in all phases of software development: requirements, design, implementation, verification and validation, and maintenance.

An idealized scenario for the use of such an environment would be the following. A team of application and computer specialists interacting with the environment develops the system requirements. The environment keeps track of the requirements as they are being developed and updated, and guards against incompleteness or inconsistency. It also provides facilities to validate requirements against the customer's expectations, for example by providing ways to simulate or animate them. The environment ensures the currency of the documentation as changes are being made to the requirements. Following the completion of the requirements, system designers, interacting with the environment, develop an initial system design and gradually refine it, that is, they specify the needed modules and the module interfaces. Test data may also be produced at this stage. The implementers then undertake to implement the system based on the design. The environment provides support for these phases by automating some development steps, by suggesting reuse of existing design and implementation components taken from a library, by recording the relationships among all of the artifacts, so that one can trace the effect of a change in–say–the requirements document to changes in the design document and in the code. The tools provided by the software development environment to support implementation are the most familiar. They include programming language processors, such as editors, compilers, simulators, interpreters, linkers, debuggers, and others. For this ideal scenario to work, all of the tools must be compatible and integrated with tools used in the other phases. For example, the programming language must be compatible with the design methods supported by the environment at the design stage and with the design notations used to document designs. As other examples, the editor used to enter programs might be sensitive to the syntax of the language, so that syntax errors can be caught before they are even entered rather than later at compile time. A facility for test data generation might also be available for programs written in the language.

The above scenario is an ideal; it is only approximated by existing commercial support tools, known under the umbrella term of CASE (Computer Aided Software Engineering), but the trend is definitely going into the direction of better support and more complete coverage of the process.

1.3 Languages and software design methods

As mentioned earlier, the relationship between software design methods and programming languages is an important one. Some languages provide better support for some design methods than others. Older languages, such as FORTRAN, were not designed to support specific design methods. For example, the absence of suitable high-level control structures in early FORTRAN makes it difficult to systematically design algorithms in a top-down fashion. Conversely, Pascal was designed with the explicit goal of supporting top-down program development and structured programming. In both languages, the lack of constructs to define modules other than routines, makes it difficult to decompose a software system into abstract data types.

To understand the relationship between a programming language and a design method, it is important to realize that programming languages may enforce a certain programing style, often called a *programming paradigm*. For example, as we will see, Smalltalk and Eiffel are *object-oriented* languages. They enforce the development of programs based on object classes as the unit of modularization. Similarly, FORTRAN and Pascal, as originally defined, are *procedural* languages. They enforce the development of programs based on routines as the unit of modularization. Languages enforcing a specific programming paradigm can be called *paradigm-oriented*. In general, there need not be a one-to-one relationship between paradigms and programming languages. Some languages, in fact, are *paradigm-neutral* and support different paradigms. For example, C++ supports the development of procedural and object-oriented programs. The most prominent programming language paradigms are presented in the sidebar on page 20.

Design methods, in turn, guide software designers in a system's decomposition into logical components which, eventually, must be coded in a language. Different design methods have been proposed to guide software designers. For example, a procedural design method guides designers in decomposing a system into modules that realize abstract operations that may be activated by other procedural modules. An object-oriented method guides in decomposing a system into classes of objects. If the design method and the language paradigm are the same, or the language is paradigm-neutral, then the design abstractions can be directly mapped into program components. Otherwise, if the two clash, the programming effort increases. As an example, an object-oriented design method followed by implementation in FORTRAN increases the programming effort.

In general, we can state that the design method and the paradigm supported by the language should be the same. If this is the case, there is a continuum between design and implementation. Most modern high-level programming languages, in fact, can even be used as design notations. For example, a language like Ada or Eiffel can be used to document a system's decomposition into modules even at the stage where the implementation details internal to the module are still to be defined.

sidebar start 1

Here we review the most prominent programming language paradigms, with special emphasis on the unit of modularization promoted by the paradigm. Our discussion is intended to provide a roadmap that anticipates some main concepts that are studied extensively in the rest of the book.

Procedural programming. This is the conventional programming style, where programs are decomposed into computation steps that perform complex operations. Procedures and functions (collectively called routines) are used as modularization units to define such computation steps.

Functional programming. The functional style of programming is rooted in the theory of mathematical functions. It emphasizes the use of expressions and functions. The functions are the primary building blocks of the program; they may be passed freely as parameters and may be constructed and returned as result parameters of other functions.

Abstract data type programming. Abstract-data type (ADT) programming recognizes abstract data types as the unit of program modularity. CLU was the first language designed specifically to support this style of programming.

Module-based programming. Rather than emphasizing abstract-data types, module-based programming emphasizes modularization units that are groupings of entities such as variables, procedures, functions, types, etc. A program is composed of a set of such modules. Modules can be used to define precisely which services are exported to the outside world by the module. In principle, any kind of service can be provided by a module, not just the ability to generate and use abstract data. Modula-2 and Ada support this style of programming.

Object-oriented programming. The object-oriented programming style emphasizes the definition of classes of objects. Instances of classes are created by the program as needed during program execution. This style is based on the definition of hierarchies of classes and run-time selection of units to execute. Smalltalk and Eiffel are representative languages of this class. C++ and Ada 95 also support the paradigm.

Generic programming. This style emphasize the definition of generic modules that may be instantiated, either at compile-time or runtime, to create the entities—data structures, functions, and procedures—needed to form the program. This approach to programming encourages the development of high-level, generic, abstractions as units of modularity. The generic programming paradigm does not exist in isolation. It can exist jointly with object-oriented programing, as in Eiffel, with functional programming, as in ML. It also exists in languages that provide more than one paradigm, like Ada and C++.

Declarative programming. This style emphasizes the declarative description of a problem, rather than the decomposition of the problem into an algorithmic implementation. As such, programs are close to a specification. Logic languages, like PROLOG, and rule-based languages, like OPS5 and KEE, are representative of this class of languages.

sidebar end

1.4 Languages and computer architecture

Design methods influence programming languages in the sense of establishing requirements for the language to meet in order to better support software development. Computer architecture has exerted influence from the opposite direction in the sense of restraining language designs to what can be implemented efficiently on current machines. Accordingly, languages have been constrained by the ideas of Von Neumann, because most current computers are similar to the original Von Neumann architecture (Figure 1).

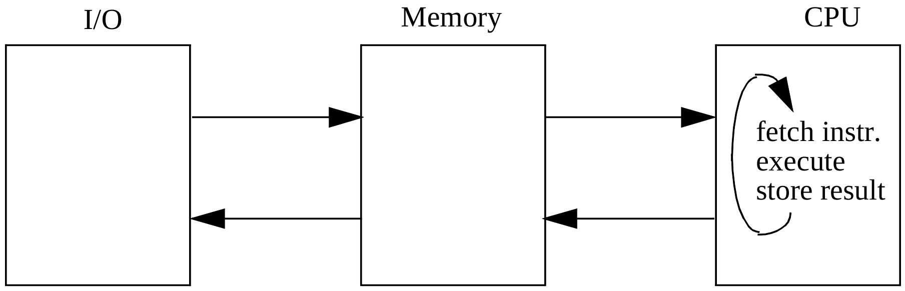


FIGURE 1. A Von Neumann computer architectur

The Von Neumann architecture, sketched in Figure 1, is based on the idea of a memory that contains data and instructions, a CPU, and an I/O unit. The CPU is responsible for taking instructions out of memory, one at a time. Machine instructions are very low-level. They require the data to be taken out of memory, manipulated via arithmetic or logic operations in the CPU, and the results copied back to some memory cells. Thus, as an instruction is executed, the *state* of the machine changes.

Conventional programming languages can be viewed as abstractions of an underlying Von Neumann architecture. For this reason, they are called Von Neumann languages. An *abstraction* of a phenomenon is a model which ignores irrelevant details and highlights the relevant aspects. Conventional programming languages keep their computation model from the underlying Von Neumann architecture, but abstract away from the details of the indivual steps of execution. Such a model consists of a sequential step-by-step execution of instructions which change the state of a computation by modifying a repository of values. Sequential step-by-step execution of language instructions reflects the sequential fetch and execution of machine instructions performed by hardware. Also, the variables of conventional programming languages, which can be modified by assignment statements, reflect the behavior of the memory cells of the computer architecture. Conventional languages based on the Von Neumann computation model are often called *imperative languages*. Other common terms are *state-based languages*, or *statement-based languages*, or simply *Von Neumann languages*.

The historical developments of imperative languages have gone through increasingly higher levels of abstractions. In the early times of computing, assembly languages were invented to provide primitive forms of abstraction, such as the ability to name operations and memory locations symbolically. Thus, instead of writing a bit string to denote the increment the contents of a memory cell by one, it is possible to write something like

```
INC DATUM
```

Many kinds of abstractions were later invented by language designers, such as procedures and functions, data types, exception handlers, classes, concurrency features, etc. As suggested by Figure 2, language developers tried to make the level of programming languages higher, to make languages easier to use by humans, but still based the concepts of the language on those of the underlying Von Neumann architecture.

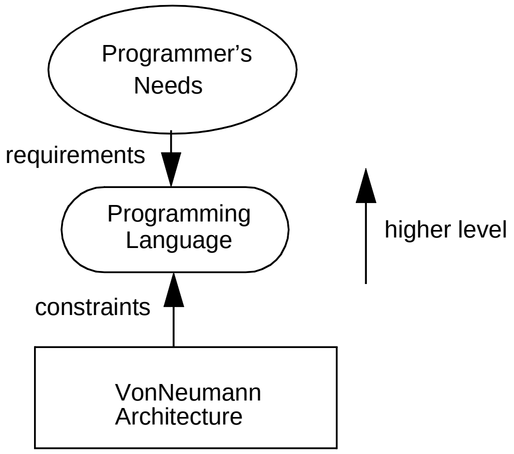


FIGURE 2. Requirements and constraints on a language

Some programming languages, namely, *functional* and *logic languages*, have abandoned the Von Neumann computation model. Both paradigms are based on mathematical foundations rather than on the technology of the underlying hardware: the theory of recursive functions and mathematical logic, respectively. The conceptual integrity of these languages, however, is in conflict with the goal of an efficient implementation. This is not unexpected, since concerns of the underlying architecture did not permeate the design of such languages in the first place. To improve efficiency, some imperative features have been introduced in most existing unconventional languages.

This book considers the prevalent languages of the last decades and the issues that have influenced their design. The primary emphasis of chapters 2 through 6 is on imperative languages. We cover the important issues in functional programming languages in Chapter 7 and logic languages in Chapter 8. Further comments and clarifications on language paradigms are provided in the sidebar on page 24.

sidebar start2

The paradigms discussed in the previous sidebar can be classified as in the hierarchy of Figure 3, according to the concepts discussed in Section 1.4.

Imperative, functional, and logic paradigms reflect the different underlying computation model of the language. The next level paradigms reflect the different organizational principles for program structuring supported by the language. As such, level 2 paradigms can apply–al least in principle–to any computation model. For example, ML provides a functional computation model and abstract data type programming. CLOS is a functional language that supports the object-oriented style. The languages supporting concurrent programming, which will be studied in Chapter 4, can still be classified under the imperative paradigm, even though the Von Neumann machine we saw in Figure 1 is purely sequential. In fact, the underlying abstract machine for the concurrent languages that will be studied in this book can be viewed as a a set of cooperating Von Neumann machine. Other kinds of parallel languages exist supporting parallelism at a much finer granularity, which do not fall under the classification shown in Figure 3.

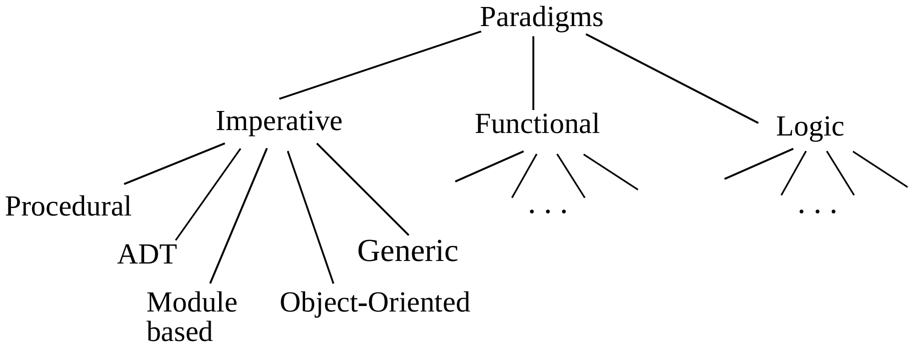


FIGURE 3. Hierarchy of paradigms

1.5 Programming language qualities

How can we define the qualities that a programming language should exhibit? In order to understand that, we should keep in mind that a programming language is a tool for the development of software. Thus, ultimately, the quality of the language must be related to the quality of the software.

Software must be reliable. Users should be able to rely on the software, i.e., the chance of failures due to faults in the program should be low. As far as possible, the system should be fault-tolerant; i.e., it should continue to provide support to the user even in the presence of infrequent or undesirable events such as hardware or software failures. The reliability requirement has gained importance as software has been called upon to accomplish increasingly complicated and often critical tasks.

Software must be maintainable. Again, as software costs have risen and increasingly complex software systems have been developed, economic considerations have reduced the possibility of throwing away existing software and developing similar applications from scratch. Existing software must be modified to meet new requirements. Also, because it is almost impossible to get the real requirements right in the first place, for such complex systems one can only hope to gradually evolve a system into the desired one.

Software must execute efficiently. Efficiency has always been a goal of any software system. This goal affects both the programming language (features that can be efficiently implemented on present-day architectures) and the choice of algorithms to be used. Although the cost of hardware continues to drop as its performance continues to increase (in terms of both speed and space), the need for efficient execution remains because computers are being applied in increasingly more demanding applications.

These three requirements–reliability, maintainability, and efficiency–can be achieved by adopting suitable methods during software development, appropriate tools in the software development environment, and by certain characteristics of the programming language. We will now discuss language issues that directly support these goals.

1.5.1 Languages and reliability

The reliability goal is promoted by several programming language qualities. A non-exhaustive list is provided hereafter. Most of them, unfortunately, are based on subjective evaluation, and are difficult to state in a precise–let alone, quantitative–way. In addition, they are not independent concepts: in some cases they are overlapping, in others they are conflicting.

Writability. It refers to the possibility of expressing a program in a way that is natural for the problem. The programmer should not be distracted by details and tricks of the language from the more important activity of problem solving. Even though it is a subjective criterion, we can agree that higher-level languages are more writable than lower-level languages (e.g., assembly or machine languages). For example, an assembly language programmer is often distracted by the addressing mechanisms needed to access certain data, such as the positioning of index registers, and so on. The easier it is to concentrate on the problem-solving activity, the less error-prone is program writing and the higher is productivity.

Readability. It should be possible to follow the logic of the program and to discover the presence of errors by examining the program. Readability is also a subjective criterion that depends a great deal on matters of taste and style. The provision of specific constructs to define new operations (via routines) and new data types, which keep the definition of such concepts separate from the rest of the program that may use them, greatly enhance readability.

Simplicity. A simple language is easy to master and allows algorithms to be expressed easily, in a way that makes the programmer self-confident. Simplicity can obviously conflict with power of the language. For example, Pascal is simpler, but less powerful than

C++.

Safety. The language should not provide features that make it possible to write harmful programs. For example, a language that does not provide **goto** statements nor pointer variables eliminates two well-known sources of danger in a program. Such features may cause subtle errors that are difficult to track during program development, and may manifest themselves unexpectedly in the delivered software. Again, features that decrease the dangers may also reduce power and flexibility.

Robustness. The language supports robustness whenever it provides the ability to deal with undesired events (arithmetic overflows, invalid input, and so on). That is, such events can be trapped and a suitable response can be programmed to respond to their occurrence. In this way, the behavior of the system becomes predictable even in anomalous situations.

1.5.2 Languages and maintainability

Programming languages should allow programs to be easily modifiable. Readability and simplicity are obviously important in this context too. Two main features that languages can provide to support modification are factoring and locality.

Factoring. This means that the language should allow programmers to factor related features into one single point. As a very simple example, if an identical operation is repeated in several points of the program, it should be possible to factor it in a routine and replace it by a routine call. In doing so, the program becomes more readable (especially if we give a meaningful name to subprograms) and more easily modifiable (a change to the fragment is localized to the routine's body). As another example, several programming languages allow constants to be given symbolic names. Choosing an appropriate name for a constant promotes readability of the program (e.g., we may use pi instead of 3.14). Moreover, a future need to change the value would necessitate a change only in the definition of the constant, rather than in every use of the constant.

Locality. This means that the effect of a language feature is restricted to a small, local portion of the entire program. Otherwise, if it extends to most of the program, the task of making the change can be exceedingly complex. For example, in abstract data type programming, the change to a data structure defined inside a class is guaranteed not affect the rest of the program as long as the operations that manipulate the data structure are invoked in the same way. Factoring and locality are strongly related concepts. In fact, factoring promotes locality, in that changes may apply only to the factored portion. Consider for example the case in which we wish to change the number of digits used to represent pi in order to improve accuracy of a geometrical computation.

1.5.3 Languages and efficiency

The need for efficiency has guided language design from the beginning. Many languages have had efficiency as a main design goal, either implicitly or explicitly. For example, FORTRAN originally was designed for a specific machine (the IBM 704). Many of FORTRAN's restrictions, such as the number of array dimensions or the form of expressions used as array indices, were based directly on what could be implemented efficiently on the IBM 704.

The issue of efficiency has changed considerably, however. Efficiency is no longer measured only by the execution speed and space. The effort required to produce a program or system initially and the effort required in maintenance can also be viewed as components of the efficiency measure. In other words, in some cases one may be more concerned with productivity of the software development process than the performance of the resulting products. Moreover, productivity concerns can span over several developments than just one. That is, one might be interested in developing software components that might be *reusable* in future similar applications. Or one might be interested in developing *portable* software (i.e., software that can be moved to different machines) to make it quickly available to different users, even if an ad hoc optimized version for each machine would be faster.

Efficiency is often a combined quality of both the language and its implementation. The language adversely affects efficiency if it disallows certain optimizations to be applied by the compiler. The implementation adversely affects efficiency if it does not take all opportunities into account in order to save space and improve speed. For example, we will see that in general a statement like

```
x = fun (y) + z + fun (y);
```

in C cannot be optimized as

```
x = 2* fun (y) + z;
```

which would cause just one call to function fun.

As different example, the language can affect efficiency by allowing multithreaded concurrent computations. An implementation adversely affects efficiency if–say–it does not reuse memory space after it is released by the program. Finally, a language that allows visibility of the underlying size of memory or access to–say–the way floating-point numbers are stored would impede portability, and thus saving in the effort of moving software to different platforms.

1.6 A brief historical perspective

This section examines briefly the developments in language design by following the evolution of ideas and concepts from a historical perspective.

The software development process originally consisted only of the implementation phase. In the early days of computing, the computer was used mainly in scientific applications. An application was programmed by one person. The problem to be solved (e.g., a differential equation) was well-understood. As a result, there was not much need for requirements analysis or design specification or even maintenance. A programming language, therefore, only needed to support one programmer, who was programming what would be by today's standards an extremely simple application. The desire to apply the computer in more and more applications led to its being used in increasingly less understood and more sophisticated environments. This, in turn, led to the need for “teams” of programmers and more disciplined approaches. The requirements and design phases, which up to then essentially were performed in one programmer's head, now required a team, with the results being communicated to other people. Because so much effort and money was being spent on the development of systems, old systems could not simply be thrown away when a new system was needed. Economic considerations forced people to enhance an existing system to meet the newly recognized needs. Also, program maintenance now became an important issue.

System reliability is another issue that has gained importance gradually, because of two major factors. One factor is that systems are being developed for users with little or no computer background; these users are not as tolerant of system failures as the system developers. The second factor is that systems are now being applied in critical areas such as chemical or nuclear plants and patient monitoring, where system failures can be disastrous. In order to ensure reliability, verification and validation became vital.

The shortcomings of programming languages have led to a great number of language design efforts. This book examines these influences on language design and assesses the extent to which the resultant languages meet their goals. Sections 1.6.1 through 1.6.6 describe the historical evolution of programming languages. Table 1 gives a genealogy of selected programming languages discussed in this book. The year we associate with each language should be taken as largely indicative: depending on the availability of the relevant information, it may mean either the year(s) of the language design, of its initial implementation, or of its first available published description.

Table 1: Genealogy of selected programming languages

Language	Year	Originator	Predecessor Language	Intended Purpose	Reference
FORTRAN	1954-57	J. Backus		Numeric computing	Glossary
ALGOL 60	1958-60	Committee	FORTRAN	Numeric computing	Naur 1963
COBOL	1959-60	Committee		Business data processing	Glossary
APL	1956-60	K. Iverson		Array processing	Iverson 1962
LISP	1956-62	J. McCarthy		Symbolic computing	Glossary
SNOBOL4	1962-66	R. Griswold		String processing	Griswold et al. 1971
PL/I	1963-64	Committee	FORTRAN ALGOL 60 COBOL	General purpose	ANSI 1976
SIMULA 67	1967	O.-J.Dahl	ALGOL 60	Simulation	Birtwistle et al. 1973
ALGOL 68	1963-68	Committee	ALGOL 60	General purpose	vanWijngaarden et al. 1976 Lindsay and van der Meulen 1977

Table 1: Genealogy of selected programming languages

Language	Year	Originator	Predecessor Language	Intended Purpose	Reference
Pascal	1971	N. Wirth	ALGOL 60	Educational and gen. purpose	Glossary
PROLOG	1972	A. Colmerauer		Artificial intelligence	Glossary
C	1974	D. Ritchie	ALGOL 68	Systems programming	Glossary
Mesa	1974	Committee	SIMULA 67	Systems programming	Geschke et al. 1977
SETL	1974	J. Schwartz		Very high level lang.	Schwartz et al. 1986
Concurrent Pascal	1975	P. Brinch Hansen	Pascal	Concurrent programming	Brinch Hansen 1977
Scheme	1975	Steele and Sussman (MIT)	LISP	Education using functional programming	Abelson and Sussman 1985
CLU	1974-77	B. Liskov	SIMULA 67	ADT programming	Liskov et al. 1981
Euclid	1977	Committee	Pascal	Verifiable programs	Lampson et al. 1977
Gypsy	1977	D. Good	Pascal	Verifiable programs	Ambler et al. 1977
Modula-2	1977	N. Wirth	Pascal	Systems programming	Glossary

Table 1: Genealogy of selected programming languages

Language	Year	Originator	Predecessor Language	Intended Purpose	Reference
Ada	1979	J. Ichbiah	Pascal SIMULA 67	General purpose Embedded systems	Glossary
Smalltalk	1971-80	A. Kay	SIMULA 67 LISP	Personal computing	Glossary
C++	1984	B. Stroustrup	C SIMULA 67	General purpose	Glossary
KEE	1984	Intellicorp	LISP	Expert systems	Kunz et al 1984
ML	1984	R. Milner	LISP	Symbolic computing	Ullman 1995
Miranda	1986	D.A.Turner	LISP	Symbolic computing	Turner 1986
Linda	1986	D. Gelernter		Parallel/ distributed programming	Ahuja et al 1986
Oberon	1987	N. Wirth	Modula-2	Systems programming	Reiser and Wirth 1992
Eiffel	1988	B. Meyer	SIMULA 67	General purpose	Meyer 1992
Modula-3	1989	Committee (Olivetti and DEC)	Mesa Modula-2	Systems programming	Cardelli et al. 1989
TCL/TK	1988	J. K. Ousterhout	OS shell languages	Rapid development, GUIs	Ousterhout 1994
Java	1995	SUN Microsystems	C++	Network computing	Glossary

1.6.1 Early high-level languages: FORTRAN, ALGOL 60, and COBOL

The first attempts towards definition of high-level languages date back to the 1950s. Language design was viewed as a challenging compromise between the users' needs for expressiveness and the machine's limited power. However, hardware was very expensive and execution efficiency concerns were the dominant design constraint.

The most important products of this historical phase were FORTRAN, ALGOL 60, and COBOL. FORTRAN and ALGOL 60 were defined as tools for solving numerical scientific problems, that is, problems involving complex computations on relatively few and simple data. COBOL was defined as a tool for solving business data-processing problems, that is, problems involving simple computations on large amounts of data (e.g., a payroll application).

These languages are among the major achievements in the whole history of computer science, because they were able to prove that the idea of a higher-level language was technically sound and economically viable. Besides that, each of these languages has brought up a number of important concepts. For example, FORTRAN introduced modularity via separately developed and compiled subprograms and possible sharing of data among modules via a global (COMMON) environment. ALGOL 60 introduced the notion of block structure and recursive procedures. COBOL introduced files and data descriptions, and a very preliminary notion of programming in quasi-natural language.

An even more convincing proof of the validity of these languages is that, apart from ALGOL 60 which did not survive but spawned into heir languages, they are still among the most widely used languages in practice. To be sure, there are other reasons for this long-term success, such as:

- the users' reluctance to move to newer languages, because of the need for compatibility with existing applications or just the fear of change.
- the fact that these languages have been evolving. For example, the present FORTRAN standard (FORTRAN 90) remains compatible with the previous standards (FORTRAN 77 and FORTRAN 66), but modernizes them and overcomes several of their major deficiencies.

1.6.2 Early schisms: LISP, APL, and SNOBOL4

As early as in the 1960s there were attempts to define programming languages whose computation model could be based on some well-characterized mathematical principles, rather than on for efficiency of implementation.

LISP is one such example. The language definition was based upon the theory of recursive functions and lambda calculus, and gave foundation to a new class of languages called functional (or applicative) languages. Pure LISP is free from the Von Neumann concepts of modifiable variables, assignment statements, **goto** statements, and so on. LISP programs are exactly like general LISP data structures, and thus the LISP interpreter can be specified in LISP in a fairly simple manner.

APL is another language that supports a functional programming style. Its very rich set of operators, especially on arrays, relieves the programmer from using lower-level iterative, element-by-element array manipulations.

SNOBOL4 is a language providing string manipulation facilities and pattern matching. The programming style it supports is highly declarative.

LISP, APL, and SNOBOL4 are heavy consumers of machine resources (time and space). All of them require highly dynamic resource management that is difficult to do efficiently on conventional machines. Yet these languages have become very successful in specialized application areas. Also, they have been adopted by groups of very devoted users. For example, LISP has become the most used language for artificial intelligence research and applications. APL has been widely used for rapid prototyping and scientific applications involving heavy usage of matrix operations. SNOBOL4 has been used successfully for text manipulations.

An important contribution of LISP and SNOBOL4 was the emphasis on symbolic computation. As we mentioned, in the early stages of computing computers were mainly used to solve numerical problems, such as systems of equations. This is why FORTRAN, ALGOL 60, and APL are mostly oriented towards numerical problem-solving. At present, however, only a small fraction of application developments are in the area of numeric computation. Major emphasis is on symbolic information processing, such as database queries and reporting, text processing, financial planning, and so on. COBOL can be seen as an initial step in this direction, because the language is more oriented towards moving and formatting data than manipulating data through complex numeric computation. It is only with LISP and SNOBOL4 that symbolic computation became the central concern of the language.

1.6.3 Putting them all together: PL/I

PL/I was designed in the mid 1960s with an ambitious goal: to integrate the most fruitful and original concepts of previous languages into a truly general purpose, universal programming language. Besides taking concepts from FORTRAN (such as separate modules), ALGOL 60 (block structure and recursive procedures), COBOL (data description facilities), and LISP (dynamic data structures), PL/I introduced less consolidated features, such as exception handling and some primitive multitasking facilities.

PL/I was probably too early. It incorporates different features, but does not really integrate them in a uniform manner. Also, newer features needed more research and experimentation before being incorporated in the language. As a result, the language is extremely large and complex. So, as time went by, it gradually disappeared.

1.6.4 The next leap forward: ALGOL 68, SIMULA 67, Pascal, and BASIC

Other languages designed in the late 1960s brought up several interesting concepts that influenced later language designs. We refer to ALGOL 68, SIMULA 67, and Pascal.

ALGOL 68 was designed as a successor to ALGOL 60. It is based on the principle of *orthogonality*: language features can be composed in a free, uniform, and non-interfering manner with predictable effects. ALGOL 68 is a good case study to see how different language concepts can interact to provide computational power. Another important concept brought up by the ALGOL 68 effort is the need for formal language specification. The ALGOL 68 Report is probably the first complete example of a formal specification for a programming language. The “purity” of ALGOL 68, the intricacies that can result from an orthogonal combination of language features, and the absence of compromises with such mundane aspects as a user-friendly syntactic notation were responsible for the early decline of ALGOL 68. The language has been used in universities and research institutions, especially in Europe, but had only a few industrial applications.

SIMULA 67 was also a successor of ALGOL 60, designed to solve discrete simulation problems. In addition to ad hoc constructs for simulation and coroutines that provide a primitive form of parallel execution, the language introduced the concept of class, a modularization mechanism that can group together a set of related routines and a data structure. Classes can be organized as hierarchies of increasing specialization. The class concept has influenced most languages designed after SIMULA 67, such as C++, CLU, Modula-2, Ada, Smalltalk, and Eiffel.

Pascal has been the most successful among these languages. Primarily conceived as a vehicle for teaching structured programming, there was a rapid expansion of interest in Pascal with the advent of low-cost personal computers. The main appeal of the language is simplicity and support to disciplined programming. The language has also undergone extensive changes and modernization, so that many Pascal dialects exist nowadays. In particular, the language has been extended with modularization and object-oriented features to support development of large programs and reuse of components.

BASIC is another language that was designed in the mid 1960s and has spawned into many, widely used, dialects. The language has a simple algebraic syntax like FORTRAN and limited control and data structures. This simplicity, and the ease and efficiency of BASIC implementations, have made the language extremely popular. The language itself does not introduce any new linguistic concepts, but was among the first available tools supporting a highly interactive, interpretive programming style. Recent improvement, like Visual BASIC, provide very high-level facilities for the rapid development of window-based interactive application.

1.6.5 C and the experiments in the 70's

In the 1970s, it became clear that the needs for supporting reliable and maintainable software imposed strong requirements on programming languages. This gave impetus to new research, experimentation, and language evaluations.

Among the most important language concepts investigated in this period were: abstract data types and visibility control to modules, strong typing and static program checking, relationship between language constructs and formal proofs of correctness, generic modules, exception handling, concurrency, and interprocess communication and synchronization. We will discuss most of these concepts in depth in the rest of this text. Among the most influential language experiments were CLU, Mesa, Concurrent Pascal, Euclid, and Gypsy.

Other languages designed in the 1970s, which survived after their experimental stage and now are used extensively, are C and Modula-2. In particular, C became very successful, partly due to the increasing availability of computers running the UNIX operating system, whose development motivated the initial design of the language. C is now among the most widely used languages, both because of its power and because of the availability of efficient implementations on a wide variety of machines.

On the non-conventional side, the family of functional languages continued to flourish, producing several LISP dialects. Among them, Scheme has been widely adopted for instructional purposes in introductory programming courses, as an alternative to conventional languages.

A major contribution in the field of nonconventional languages was provided in the early 70s by PROLOG. PROLOG was the starting point of a new language family: logic programming languages. The language had limited success at that time, but later in the early 80's gained much popularity when the so-called Fifth Generation Computer Project was launched by the Japanese government, and logic programming was chosen as the basis for the new generation of machines. PROLOG extensions were designed and implemented, such as PARLOG and Concurrent PROLOG), under the assumption that new generations of parallel machines would be designed to implement execution of logic programs efficiently. Although the revolution predicted by the project did not happen, PROLOG (and other logic languages) found their role in niche software development environments.

1.6.6 The 80's: ML, Ada, C++ and object orientation

Developments in functional programming continued in the 80s, producing such languages as Miranda and ML. The important conceptual contribution of ML was to show that programming languages can be made very powerful computationally, and yet they can preserve the ability to prove the absence of certain types of errors without executing programs.

New results were also achieved in the family of conventional languages. The desire to unify the programming languages used in embedded computer applications and the need for more reliable and maintainable software led the U.S. Department of Defense in 1978 to set down the requirements for a programming language to be used as a common language throughout the D.O.D. Because no existing language met the requirements, the D.O.D. sponsored the design of a new language. The result of this process is the Ada programming language, which can be viewed as the synthesis of state-of-the-art concepts of conventional programming languages. Ada has now evolved into the current version, Ada 95, which incorporates several amendments and improvements over the original version, in particular to support object-oriented programming.

The origins of object-oriented programming can be traced back to Simula 67. The approach, however, became popular because of the success of Smalltalk in the late 70's and, in particular, of C++. C++ succeeded in implanting object-oriented features into a successful and widely available language like C. This allowed a large population of programmers to incrementally shift from a conventional programming paradigm into an expected better one. Eiffel is another object-oriented language, which aims at supporting programming with underlying disciplined software engineering principles.

Recent advances in the Pascal and Modula-2 tradition are Modula-3 and Oberon.

1.6.7 The present

For decades the search for the ideal programming language has been like the quest for the Holy Grail of computer scientists. It is now universally accepted that this approach tries to answer the wrong question. As we mentioned earlier, programmers might be interested in different qualities, and different languages (and different implementations) may indeed provide different answers. So we now realize that the choice of the right language depends on the application. We need to learn how to live with a variety of languages, and need to be able to move from language to language when needed, as the applications change.

In the world of information systems applications that was the traditional domain of COBOL, there is an increasing number of application generators, which can generate code directly from screen forms that specify the data that should be searched in a database, or the reports that must be produced. In certain limited application domains, nonexpert programmers and end users can use such tools to develop nontrivial, practical applications, without resorting to a professional programmer. Programming tools of this kind are often called *fourth generation languages.* Other useful tools for this class of applications are personal *productivity tools* (such as spreadsheets). Production-rule based expert systems are also used in narrowly focused application domains to solve problems stated in a declarative style.

Highly interactive applications can be rapidly developed with the aid of *visual languages*, such as Visual BASIC or Visual C++. *Scripting languages*, such as TCL/TK, which specify activation patterns for existing tool fragments, are an increasingly popular support for rapid application development, which can be useful to develop prototypes.

Specific tools, languages, and environments also exist for developing expert systems, i.e., systems providing problem solving support in specific application domains, based on an explicit representation of knowledge that characterizes the domains. Examples are expert system shells, and languages such as KEE.

Finally, C++ seems to gain increasing acceptance as a general-purpose programming tool, both because it supports object-oriented programming and because it does not force abandoning more conventional approaches, as more strict approaches would. However, we do not expect the programming language field to reach a stable stage where one language will eventually take over. An important direction for new developments has started already in the area of network-centric computing. Java, a derivative of C++ supporting code mobility on the Internet, has signed the starting point of a new generation of programming languages.

1.7 A bird's eye view of programming language concepts

This chapter provides a bird's eye view of the main concepts of programming languages which will be the subjects of an in-depth investigation in all the remaining chapters. Its purpose is to show how all the various concepts that will be presented fit together in a coherent picture. Using a simple C/C++ program as an example, we look at the kinds of facilities that a programming language must support and the different ways that languages go about providing these facilities.

1.7.1 A simple program

Figure 4 shows a part of a C/C++ program that manipulates a list of phone numbers. As programmers, our inclination on encountering a program is to try to uncover what the program does and how it does it. Our purpose in this book, however, is to learn about the concepts and structure of programming languages. We are interested in the kinds of things one can do with programming languages, rather than the specifics of a given program. What are the inherent capabilities and shortcomings of different programming languages? What makes one language fundamentally different from another and what makes one language similar to another, despite apparent differences? We will use the simple program in Figure 4 to start our exploration of the structure of programming languages. Therefore, in looking at the program, we want to look at not what it does, but what kinds of linguistic facilities were used to write the program.

```
#include <iostream.h>
#include “phone.h”

extern phone_list pb;
void insert();
number lookup ();

main()
{
    int request;

    cout << “Enter 1 to insert, 2 to lookup: \n”;
    cin >> request;
    if (request == 1)
        insert();
    else if (request == 2)
        cout << lookup();
    else
        {cout << “invalid request.\n”;
        exit (1);
        }
}
```

FIGURE 4. A phone-list program

We have divided the program into three parts, separated from each other by single blank lines. The first section consists of two “#include” statements; the second part consists of three “declaration” statements; and, finally, the third part is the actual code of a function called *main* that supposedly “does the work”. We can say that the first part is used to organize the *structure* of the program, in this case in terms of the various files that constitute the program. The second part defines the *environment* in which the program will work by declaring some entities that will be used by the program in this file. These declarations may import entities defined in other files. For example, the line

```
extern phone_list pb;
```

indicates that the variable pb of type phone_list is being used in this program but has been created elsewhere. The third part deals with the actual *computation*. This is the part we most often associate with a program. It contains the program's data and algorithms. Some of the data and processing in this part may use the entities defined in the environment established in the second part. For example, in Figure 4 the routines insert and lookup are used in the main program. Another example is the output statement:

```
cout << “Enter 1 to insert, 2 to lookup: \n”;
```

which uses cout, the standard output device defined in the standard input-output library iostream.h included in the first line of the program.

Even in this short, simple program, we see that a programming language provides many different kinds of facilities. Let us look more closely at some of the major facilities and the issues involved in designing such language facilities.

1.7.2 Syntax and semantics

Any programming language specifies a set of rules for the *form* of valid programs in that language. For example, in the program of Figure 4, we see that many lines are terminated by a semicolon. We see that there are some special characters used, such as { and }. We see that every if is followed by a parenthesized expression. The *syntax* rules of the language state how to form expressions, statements, and programs that *look* right. The semantic rules of the language tell us how to build *meaningful* expressions, statements, and programs. For example, they might tell us that before using the variable request in the if-statement, we must declare that variable. They also tell us that, the declaration of a variable such as request causes storage to be reserved for the variable. On the other hand, the presence of the extern in the declaration of the variable pb indicates that the storage is reserved by some other module and not this one.

Characters are the ultimate syntactic building blocks. Every program is formed by placing characters together in some well-defined order. The syntactic rules for forming programs are rather straightforward. The semantic building blocks and rules, on the other hand, are more intricate. Indeed, most of the deep differences among the various programming languages stem from their different semantic underpinnings.

1.7.3 Semantic elements

In this section, we will look at some of the basic semantic concepts in programming languages. The idea is to examine these notions not from a programmer's point of view but from the language designer's point of view. We want to see what choices may be available to a language designer and how the designer's decisions affect the programmer.

Variables

A variable is the most pervasive concept in traditional programming languages. A variable corresponds to a region of memory which is used to hold *values* that are manipulated by the program. We refer to a variable by its *name*. The syntactic rules specify how variables may be named, for example, that they may consist of alphabetic characters. But there are many semantic issues associated with variables. A *declaration* introduces a variable by giving it a name and stating some of its semantic properties. Among the important semantic properties are:

- *scope*: what part of the program has access to the variable? For example, in the example program, the scope of the variable request extends to the end of the function called main. That is, the variable may be referred to in any part of the program from the declaration of the variable to the end of the function main. By contrast, the scope of the variable pb, is the entire file. That is, if there were other functions besides main, they could also refer to the variable pb. Usually, the location of the variable declaration determines the start of the scope of the variable.
- *type*: what kinds of values may be stored in the variable and what operations may be performed on the variable? The variable request is declared to be of type int and the variable pb is declared of type phone_list. Usually, there are a number of fundamental types defined by the language and there are some facilities for the user to define new types. Languages differ both in terms of the fundamental types and in the facilities for type definition. The fundamental types of traditional languages are dictated by the types that are supported by the hardware. Typically, as in C++, the fundamental types are integer, real, character. Pascal also has boolean types. There is a large body of work on data types that deals both with the theoretical underpinnings as well as practical implications. We will study many of these issues in detail in Chapter 3.
- *lifetime*: when is the variable created and when is it discarded? As we said, a variable represents some region of memory which is capable of holding a value. The question is when is a memory area reserved, or *allocated*, for the variable? Some possibilities are: when the program starts, when the declaration is encountered at execution time, when the unit in which the declaration occurs is entered, or there could be a statement that explicitly

> requests the allocation of storage for the variable. Indeed, C++ has all of these kinds of variables: automatic variables are allocated when the unit in which they are declared is entered and deallocated when the unit terminates; static variables live throughout the execution of the program; some variables may be created and destroyed explicitly by the programmer using the operators new and delete.

These issues will be discussed in detail in Chapter 2.

Values and references

Having defined some basic issues concerning variables, let us ponder a simple question: what is the *value* associated with a variable? Well, there are at least two answers to this question. Consider an assignment statement of the form:

```
x = y;
```

The *value* referred to by the name y is of a different kind from that referred to by the name x. We have defined a variable as a region of memory. On the right hand side of this assignment statement, we need the contents of that memory and on the left hand side we need the address of, or a *reference* to, that region. To enable us to refer to both of these kinds of values, we define two notions: an *l-value* is a value that refers to a memory location, and, therefore, may be used on the left hand side of an assignment statement; an *r-value* is a value that refers to the contents of a memory location, that is, a value that may be used on the right-hand side of an assignment statement. Referring to the assignment statement above, we need an r-value for y and an l-value for x.

In most languages, the conversions from l-values to r-values are implicit. Some languages, such as C++, also have explicit operators to do the conversions when necessary. For example, the & operator in C++ is the address-of operator, which obtains the l-value of its operand. Therefore,

```
x = &y;
```

stores the address of y into x. The & is necessary because the default rule is that on the right-hand side, the r-value is used.

Some contexts require a particular type of value. For example, the left-hand side of an assignment statement requires an l-value. Therefore:

```
3 = y; //error, left-hand side requires l-value
```

is an error because literals in C++ do not have l-values. Instead,

```
y = 3;
```

is legal since the literal 3 is an r-value.

Expressions

Expressions are syntactic constructs that allow the programmer to combine values and operations to compute new values. The language specifies syntactic as well as semantic rules for building expressions. Depending on the language, an expression may be constrained to produce a value of only one type or of different types at different times. In the program of Figure 4, we see several expressions of different types. For example, request == 1, is an expression of type boolean; “invalid request.\n” is an expression of type string.

For example, in C or C++, an assignment statement produces a value and therefore is also an expression and may be used as a constituent of another expression. Consider:

```
a = b = c + d;
```

which assigns (first) to b the value of the expression c+d and then assigns the same value to a. The language Pascal does not allow an assignment statement to be used as part of an expression.

As can be seen from this example, the order in which operations are performed in an expression may influence the value of the expression. Some languages specify the order strictly, for example right-to-left, and others leave it to the implementer to decide the order. Leaving such issues to the implementation requires the programmer to be more careful because a program that produces the correct result may not necessarily do so when compiled with a different compiler.

The major semantic issue surrounding expressions is the allowable kinds of expressions. Specifically, does the language support expressions that produce only r-values or can expressions also result in l-values (or even functions)? More on expressions will be said in Chapter 4 for conventional languages. Chapter 7 will deal with functional languages, which can also be called expression-oriented, since expressions play a central role in such languages.

1.7.4 Program organization

Programs that implement software systems and applications consist of thousands, hundreds of thousands, or even millions of lines of code. These lines together implement a particular system design that consists of many inter-related components, or *modules*. A programming language can provide mechanisms to help the programmer in managing this complexity. To some degree, the structure of the design may be reflected in the structure of the program. As we mentioned, this is straightforward whenever the design method and the programming language paradigm match.

As an example, a program in C/C++ consists of a number of *files*. By convention, a programmer may implement each design module in one file. Even more, some files may contain modules that are more generally available, referred to as libraries. In the example of Figure 4, the program includes a file called iostream.h, which provides the declarations to use the standard input-output library provided by C/C++. The language does not have any particular facilities for supporting input-output. Instead, a collection of routines make up a library that support input-output operations. Programs that want to use input/output include iostream.h. The other file included by the program is called phone.h. This file is presumably more specific to this application and contains information, such as type definitions, that are shared by different modules of the program.

Being able to break a program into a number ofindependent parts has many advantages. First, if the parts are independent, they may be implemented and validated by different people. Program debugging and maintenance is also simplified because changes may be isolated to independent modules. Second, it is more practical to store the program in several files rather than one big file. The ability to compile separate parts of the program is important in writing large applications.

In C/C++, the inclusion of files imposes an ordering relationships among the modules of a program. The main program includes some files which may in turn include other files and so on. Obviously, the included files must be written before the files that include them. This relationship imposes a hierarchy among the files that constitute the program. There are files that need no other files. These are at the lowest level of the hierarchy—level 0. At the next level are files that only include files from level 0. This file inclusion facility support the direct implementation of hierarchical designs.

Finally, if C++ is chosen, the language provides support both to procedural and object-oriented programming. The program structure can therefore match a design method based on both decomposition into abstract operations and hierarchies of abstract data types.

Similar considerations hold for Ada. Whereas the correspondence between design modules and program files in C/C++ is rather loose and by convention, in Ada this correspondence is emphasized. Each module has a specification and an implementation. Once the specification of a module is written, other modules that use this module may be written and compiled. This approach reduces the dependence among programmers in that more work may be done in parallel. Ada also supports the concept of a library where module specifications are stored. The language requires that interfaces across independently compiled modules must be checked to ensure that the called and the calling modules agree. On the other hand, the FORTRAN language also supports independently developed (procedural) modules but does not require type checking across such modules.

The program organization facilities provided by a programming language are dependent on the goals of the language. If the language is intended for writing small programs, for example for the writing of mathematical algorithms to be run on a calculator, such facilities are not crucial. If, on the other hand, the language is to be used to develop very large programs, these facilities are indispensable. Most modern languages today support at least the notion of a module for breaking up a large program into several independent parts. Where the languages differ is in the way the different modules have access to each other’s internal entities and inthe types of entities that may be imported from other modules. They also differ in the treatment of modules, e.g., whether they can be instantiated, whether they can be separately compiled, etc. These are the specific topics addressed in Chapter 5.

1.7.5 Program data and algorithms

Programming languages provide facilities for implementing algorithms. The algorithms operate on some data to produce some results. This is where programming languages differ the most from each other. The majority of programming languages, including C++, are imperative. As we can see in Figure 4, the main program consists of some variable declarations and some statements that operate on these variables. There are also input-output statements. The execution of statements modifies the values stored in the memory of the underlying machine; i.e., it modifies the state of the computation.We will deal with these in the next section. For now let us look at the issues relating to data and computation.

Data

There are many issues surrounding the idea of data. Look at the simple variable request declared in our example program. It has a type, which in this case is int. It tells us what kinds of values it may hold. Where can such a variable declaration occur in a program? Only at the beginning of a program or anywhere? When is the variable created? Does it have an initial value? Is it known to other procedures or modules of the program? How can variables be exported to other modules?

Given some elementary data items such as variables, are there mechanisms to combine them? For example, C++ provides arrays and records for building aggregate data structures. What are the kinds of components that a data structure may contain? Can a function be an element of a record? In Pascal the answer is no and in C++ the answer is yes.

Sophisticated mechanisms for data definition allow the programmer to modularize the data in the program similarly to the way that the algorithms are modularized. For example, in our program in Figure 4, we use a file phone.h to store the basic definitions concerning phone data that are used by all other modules. Object-oriented programming languages draw much of their power from the mechanisms to define and refine complex data items. Chapters 4 and 7 are devoted to these topic.

Computation

We have already seen expressions as a mechanism for computing values. Expressions are usually made up of elementary values and have a simple structure. *Control structures* are used to structure more complicated computations. For example, mechanisms such as various kinds of loops provide for repeated executions of a sequence of statements. Routine calls allow for the execution of a computation defined elsewhere in the program. Combining expressions, statements, control structures and routine calls in C++ and other conventional languages allows the programmer to write algorithms using an imperative computation paradigm.

Chapter 4 describes the programming language mechanisms used for structuring computations.

1.7.6 External environment

Programs are seldom self-contained implementations of algorithms. The data they need and the results they expect to compute are normally transferred to and from the program to the external environment. In the example of Figure 4, the user is asked to type in a request. In other cases, a program might need to access an external database; a device driver program might need to acquire the value of a particular signal.

How do programs communicate with the external environment? Some languages define specific constructs for input/output. Other languages, such as C/C++, do not provide such facilities. Instead, they rely on libraries external to the language to provide such facilities. For example, iostream.h is the header file that allows the input/output library to become accessible by the program in Figure 4 allows the program to interact with the user. The same happens for accessing an external database.

The advantage of language-supported facilities for communication with the external environment is that the programmer has a complete model of the environment and the compiler can do consistency checking. Supporting the facilities in a library makes the language simpler and allows more flexibility. For example, different libraries may be added as new devices, such as graphical ones, become available.

1.8 Bibliographic notes

Software development processes, environments and methods are covered in software engineering textbooks, such as (Ghezzi et al. 1991).

For a historical perspective on programming language developments, see (Wexelblat 1981). The Turing lectures by Backus (Backus 1978), Hoare (Hoare 1981), and Wirth (Wirth 1985) provide stimulating reflections on programming languages and their evolution. In particular, (Backus 1978) takes a strong position in favor of functional languages as opposed to the Von Neumann conventional approach.

1.9 Exercises

1. Write a short paper on the costs of programming. Discuss both the costs involved in developing and maintaining programs, and the costs involved in running programs. Discuss the role of the programming language in both.
2. List the main features of your favorite programming language that can help make programs easily maintainable. Also discuss features that hinder maintainability.

3. Can you find reasons why the optimization mentioned in Section 1.5.3 cannot be done in general for C?
4. 3Provide a succinct characterization of imperative vs. nonconventional (functional and logic languages).
5. Take one or two languages you have used and discuss the types of expressions you can write in the language.
6. Take one or two languages you have used and discuss the facilities provided for program organization.
7. Take one or two languages you have used and describe how the language supports interaction with the external environment.

1

Syntax and semantics

CHAPTER 2

A programming language is a formal notation for describing algorithms for execution by computer. Like all formal notations, a programming language has two major components: syntax and semantics. Syntax rules describe the form of any legal program. It is a set of formal rules that specify the composition of programs from letters, digits, and other characters. For example, the syntax rules may specify that each open parenthesis must match a closed parenthesis in arithmetic expressions, and that any two statements must be separated by a semicolon. The semantic rules specify “the meaning” of any syntactically valid program written in the language. Such meaning can be expressed by mapping each language construct into a domain whose semantics is known. For example, one way of describing the semantics of a language is by giving a description of each language construct in English. Such a description, of course, suffers from the informality, ambiguity, and wordiness of natural language, but it can give a reasonably intuitive view of the language. In order to provide complete formal description of language semantics, syntactically valid programs are mapped onto mathematical domains. We will provide a preliminary introduction to formal semantics in a sidebar in this chapter. A full treatment of formal semantics, however, is out of the scope of this text. Rather, we will provide a rigorous, yet informal, description of semantics by specifying the behavior of an abstract processor that executes programs written in the language. This kind of semantic characterization of a language is called operational semantics. It could be presented using a rigorous and formal notation. Instead, we will follow a more traditional and informal approach, because it is more easily and intuitively understood by computer programmers and provides a high-level view of the problems found in implementing the language.

This chapter devoted syntax and operational semantics of programming languages. It is organized as follows: In Section 2.1, we discuss how the syntax and semantics of a language can be defined. In Section 2.2 we discuss language implementation and introduce the fundamental semantic concept of binding. In Section 2.3 and Section 2.4 we discuss two important concepts of programming languages–variables and routines–and their binding properties. In Section 2.5 we define an abstract semantic processor that can be used to specify operational semantics. In Section 2.6 we discuss how the abstract processor implements the main run-time features of programming languages.

2.1 Language definition

When you read a program, how do you know if it is well formed? How do you know what it means? How does a compiler know how to translate the program? Any programming language must be defined in enough detail to enable these kinds of issues to be resolved. More specifically, a language definition should enable a person or a computer program to determine (1) whether a purported program is in fact valid, and (2) if the program is valid, what its meaning or effect is. In general, two aspects of a language-programming or natural language-must be defined: syntax and semantics.

2.1.1 Syntax

Syntax is described by a set of rules that define the form of a language: they define how sentences may be formed as sequences of basic constituents called words. Using these rules we can tell whether a sentence is legal or not. The syntax does not tell us anything about the content (or meaning) of the sentence–the semantic rules tell us that. As an example, C keywords (such as while, do, if, else,...), identifiers, numbers, operators, ... are words of the language. The C syntax tells us how to combine such words to construct well-formed statements and programs.

Words are not elementary. They are constructed out of characters belonging to an alphabet. Thus the syntax of a language is defined by two sets of rules: lexical rules and syntactic rules. Lexical rules specify the set of characters that constitute the alphabet of the language and the way such characters can be combined to form valid words.

For example, Pascal considers lowercase and uppercase characters to be identical, but C and Ada consider them to be distinct. Thus, according to the lexical rules, “Memory” and “memory” refer to the same variable in Pascal, but to distinct variables in C and Ada. The lexical rules also tell us that <> (or ¦) is a valid operator in Pascal but not in C, where the same operator is represented by !=. Ada differs from both, since “not equal” is represented as /=; delimiter <> (called “box”) stands for an undefined range of an array index.

The distinction between syntactic and lexical rules is somewhat arbitrary. They both contribute to the “external” appearance of the language. Often, we will use the terms “syntax” and “syntactic rules” in a wider sense that includes lexical components as well.

How does one define the syntax of a language? Because there are an infinite number of legal and illegal programs in any useful language, we clearly cannot enumerate them all. We need a way to define an infinite set using a finite description. FORTRAN was defined by simply stating some rules in English. ALGOL 60 was defined with a context-free grammar developed by John Backus. This method has become known as BNF or Backus Naur form (Peter Naur was the editor of the ALGOL 60 report.) BNF provides a compact and clear definition for the syntax of programming languages. We illustrate an extended version of BNF (EBNF) in the sidebar on page 53, along with the definition of a simple language. Syntax diagrams provide another way of defining syntax of programming languages. They are conceptually equivalent to BNF, but their pictorial notation is somewhat more intuitive. Syntax diagrams are also described in the sidebar.

Sidebar-start-1

EBNF is a meta-language. A meta-language is a language that is used to describe other languages. We describe EBNF first, and then we show how it can be used to describe the syntax of a simple programming language (Figure 5(a)). The symbols ::=, <, >, *, +, (,), and | are symbols of the metalanguage: they are *metasymbols*. A language is described in EBNF through a set of rules. For example, <program> ::= { <statement>* } is a rule. The symbol "::=" stands for “is defined as”. The symbol “*” stands for “an arbitrary sequence of the previous element”. Thus, the rule states that a <program> is defined as an arbitrary sequence of <statement> within brackets “{” and “}”.

The entities inside the metalanguage brackets “<”, and “>” are called nonterminals; an entity such as the “}” above is called a terminal. Terminals are what we have previously called words of the language being defined, whereas nonterminals are linguistic entities that are defined by other EBNF rules. In order to distinguish between metasymbols and terminals, Figure 5 uses the convention that terminals are written in bold. To complete our description of EBNF, the metasymbol “+” denotes one or more repetitions of the previous element (i.e., at least one element must be present, as opposed to “*”). The metasymbol “|” denotes a choice. For example, a <statement> is described in Figure 5(a) as either an <assignment>, a <conditional>, or a <loop>.

(a) Syntax rules

<program>::=**{** <statement>* **}**
<statement>::=<assignment> | <conditional> | <loop>
<assignment>::=<identifier> **=** <expr> ;
<conditional>::=**if** <expr> **{** <statement> + **}** |
        if <expr> **{** <statement> + **} else {** <statement> + **}**
<loop>::=**while** <expr> **{** <statement> + **}**
<expr> ::=<identifier> | <number> | (<expr>) | <expr> <operator> <expr>

(b) Lexical rules

<operator>::= **+** | **-** | ***** | / | **=** | ¦ | < | > | ≤ | ≥
<identifier>::= <letter> <ld>*
<ld>::= <letter> | <digit>
<number>::= <digit>+
<letter>::= **a** | **b** | **c** | . . . | **z**

FIGURE 5. EBNF definition of a simple programming language
(a) syntax rules, (b) lexical rules

The lexical rules, which describe how identifiers, numbers, and operators look like in our simple language are also described in EBNF, and shown in Figure 5(b). To do so, <operator>, <identifier>, and <number>, which are words of the language being defined, are detailed in terms of elementary symbols of the alphabet.

Figure 6 shows the syntax diagrams for the simple programming language whose EBNF has been discussed above. Nonterminals are represented by circles and terminals by boxes. The nonterminal symbol is defined with a transition diagram having one entry and one exit edge. A string of words is a valid program if it can be generated by traversing the syntax diagram from the entry to the exit edge. In this traversal, if a terminal (box) is encountered, that word must be in the string being recognized; if a nonterminal (circle) is encountered, then that nonterminal must be recognized by traversing the transition diagram for that nonterminal. When a branch in the path is encountered, any one edge may be traversed. Syntax diagrams are similar enough to EBNF to allow you to understand the rules.

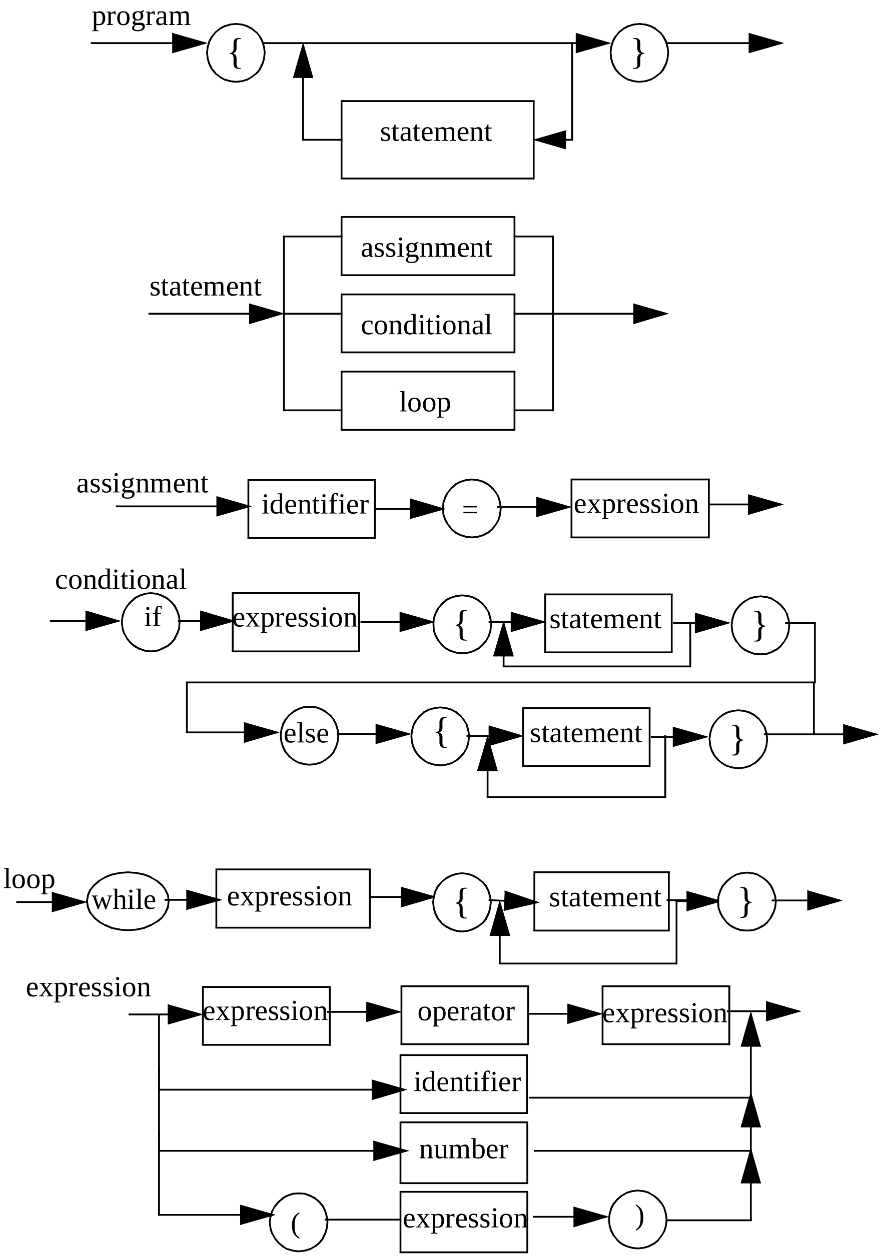


FIGURE 6. Syntax diagrams for the language described in Figure 5.

sidebar-end

In conclusion, the syntactic description of a language has two primary uses:

(a) It helps the programmer know how to write a syntactically correct program. For example, if one is unsure about the syntax of **if** statements, a look at the EBNF or syntax diagrams can quickly settle any doubts.

(b) It can be used to determine whether a program is syntactically correct. This is exactly what a compiler does. The compiler writer uses the grammar to write a syntactic analyzer (also called parser) that is capable of recognizing all valid programs. This process is now largely automated. In fact, there are programs ("compiler generators") that can use the grammar of the language as input and produce the analyzer as output. LEX and YACC are two well-known UNIX tools that generate lexical and syntax analyzers, respectively, starting from a description of the lexical and syntactic rules of the language. Several versions of these tools exist.

2.1.1.1 Abstract syntax, concrete syntax and pragmatics

Some language constructs in different programming languages have the same conceptual structure but differ in their appearance at the lexical level. For example, the C fragment

```
while (x != y) {
    . . .
};
```

and the Pascal fragment

```
while x <> y do
begin
    . . .
end
```

can both be described by simple lexical changes in the EBNF rules of Figure 5. They differ from the simple programming language of Figure 5 only in the way statements are bracketed (**begin** ... **end** vs. { ... }), the "not equal" operator (<> vs. !=), and the fact that the loop expression in C must be enclosed within parentheses. When two constructs differ only at the lexical level, we say that they follow the same *abstract syntax*, but differ at the *concrete syntax* level. That is, they have the same abstract structure and differ only in lower-level details.

Although, conceptually, concrete syntax may be irrelevant, pragmatically it may affect usability of the language and readability of programs. For example, symbol ¦ is obviously more readable than !=. As another example, the simple language of Figure 5 requires the body of while statements and the branches of conditionals to be bracketed by { and }. Other languages, such as C or Pascal, allow brackets to be omitted in the case of single statements. For example, one may write:

```
while (x != y) do x = y + 1;
```

Pragmatically, however, this may be error prone. If one more statement needs to inserted in the loop body, one should not forget to add brackets to group the statements constituting the body. Modula-2 adopts a good concrete syntax solution, by using the "end" keyword to terminate both loop and conditional statements. A similar solution is adopted by Ada. The following are Modula-2 examples:

```
if x = y then                    if x = y then                    while x = y do
    . . .                            . . .                            . . .
end                              else                             end
                                     . . .
                                 end
```

In all three fragments, the "..." part can be either a single statement or a sequence of statements separated by a semicolon.

2.1.2 Semantics

Syntax defines well-formed programs of a language. Semantics defines the meaning of syntactically correct programs in that language. For example, the semantics of C help us determine that the declaration

```
int vector [10];
```

causes ten integer elements to be reserved for a variable named vector. The first element of the vector may be referenced by vector [0]; all other elements may be referenced by an index i, $0 \leq i \leq 9$.

As another example, the semantics of C states that the instruction

```
if (a > b) max = a; else max = b;
```

means that the expression a > b must be evaluated, and depending on its value, one of the two given assignment statements is executed. Note that the syntax rules tell us how to form this statement–for example, where to put a ";"–and the semantic rules tell us what the effect of the statement is.

Actually, not all syntactically correct programs have a meaning. Thus, semantics also separates meaningful programs from syntactically correct ones. For example, according to the EBNF of the simple language described in Figure 5, one could write any expression as a condition of if and while statements. The semantics of the language might require such expressions to deliver a truth value (TRUE or FALSE, not–say–an integer value). In many cases, such rules that further constrain syntactically correct programs can be verified before a program's execution: they constitute *static semantics*, as opposed to *dynamic semantics*, which describes the effect of executing the different constructs of the programming language. In such cases, programs can be executed only if they are correct both with respect to syntax and to static semantics. In this section, we concentrate on the latter; i.e., any reference to the term "semantics" implicitly refers to "dynamic semantics".

While syntax diagrams and BNF have become standard tools for syntax description, no such tool has become widely accepted and standard for semantic description. Different formal approaches to semantic definition exist, but none is entirely satisfactory. A brief introduction to formal semantics is provided in the sidebar page 59.

In this chapter, and throughout this book, we take an operational approach to describing the semantics of programming languages. In this approach, the behavior of a simple and intuitive abstract processor is used to describe the effects of each language construct. We will describe such a machine in the next subsection. We will then describe the semantics of programming language constructs in terms of the operations of this machine.

Our machine is abstract. This means that it is not a real machine such as the Apple Macintosh PowerBook Duo 270c or the HP 9000. It is designed to show the run-time requirements of programming languages simply, rather than to execute them efficiently. It can be used as a model for language implementation in the sense that one can derive straightforward, simple implementations by applying the concepts that we discuss here. The resulting implementation, however, probably would be inefficient. To achieve efficiency, any real implementation will have to differ from the model in important ways, for example, in how data structures are arranged and accessed and in the set of machine instructions. The purpose of the model is simply to state the effects of the language, given the structure of the abstract machine. A particular implementation of the language on a given real machine is in no way obligated to implement the structure of the abstract processor used to define the semantics of the language; it is only required to implement the same effects, given the restrictions and structure of the implementation machine.

It is important to separate the semantic issues of the language from the implementation issues (we will come back to this point later). This can be done by keeping in mind which part of the description is a description (or restriction) of the machine and which is of the language.

add sidebar on java byte code???

Sidebar-start-2

A metalanguage for formal semantics must be based on well-understood and simple mathematical concepts, so that the resulting definition is rigorous and unambiguous. The ability to provide formal semantics makes language definitions independent from the implementation. The description specifies what the language does, without any reference to how this is achieved by an implementation. Furthermore, it allows comparison of different programing language features to be stated in unquestionable terms. Unfortunately, formality does not go hand-in-hand with readability. The absolute rigor of formal semantics can be useful in a reference description, but for most practical uses a rigorous–yet informal–description can suffice.

Here we briefly review two ways of formally specifying semantics: axiomatic semantics and denotational semantics. We do not go deep into the two methods, but rather we try to provide a preliminary introduction that shows the main differences between them. We base our description on the simple language described in Figure 5.

Axiomatic semantics views a program as a state machine. Programming language constructs are described by describing how their execution causes a state change. A state is described by a first-order logic predicate which defines the property of the values of program variables in that state. Thus the meaning of each construct is defined by a rule that relates the two states before and after the execution of that construct.

A predicate P that is required to hold after execution of a statement S is called a *postcondition* for S. A predicate Q such that the execution of S terminates and postcondition P holds upon termination is called a *precondition* for S and

P. For example, $y = 3$ is one possible precondition for statement

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

that leads to postcondition $x > 0$. The predicate $y \geq 0$ is also a precondition for the same statement and the same postcondition. Actually, $y \geq 0$ is the weakest precondition. A predicate W is called the *weakest precondition* for a statement S and a postcondition P, if any precondition Q for S and P implies W. Among all possible preconditions for statement S and postcondition P, W is the weakest: it specifies the fewest constraints. It is the necessary and sufficient precondition for a given statement that leads to a certain postcondition. In the example, it is easy to prove that any precondition (e.g., $y = 3$) implies the weakest precondition ($y \geq 0$). This can be stated in first-order logic as

$$y = 3 \supset y \geq 0$$

Axiomatic semantics specifies each statement of a language in terms of a function asem, called the *predicate transformer*, which yields the weakest precondition W for any statement S and any postcondition P. It also provides composition rules that allows the precondition to be evaluated for a given program and a given postcondition. Let us consider an assignment statement

```
x = expr;
```

and a postcondition P. The weakest precondition is obtained by replacing each occurrence of x in P with expression expr. We express this weakest precondition with the notation $P_{x \rightarrow expr}$. Thus[1]

$$\text{asem}\ (x = expr;,\ P) = P_{x \rightarrow expr}$$

Simple statements, such as assignment statements, can be combined into more complex actions. For example, let us consider sequences, such as

```
S1; S2;
```

If we know that

```
asem (S1;, P) = Q
```

and

```
asem (S2;, Q) = R
```

then

```
asem (S2; S1;, P) = R
```

[1] This characterization of assignments is correct for simple assignment statements (see Exercise 40).

The specification of semantics of selection is straightforward. If B is a boolean expression and L1, L2 are two statement lists, then let **if**-stat be the following statement:

```
if B then L1 else L2
```

If P is the postcondition that must be established by **if**-stat, then the weakest precondition is given by

asem (**if**-stat, P) = (B ⊃ asem (L1, P)) and (not B ⊃ asem (L2, P))

That is, function asem yields the semantics of either branch of the selection, depending on the value of the condition. For example, given the following program fragment (x, y, and max are integers)

```
if x >= y then max := x else max := y
```

and the postcondition

$(max = x \text{ and } x \geq y) \text{ or } (max = y \text{ and } y > x)$

the weakest precondition is easily proven to be true, that is, the statement satisfies the postcondition without any constraints on variables.

The specification of semantics of loops is more complex. For simplicity, let us assume that the program terminates, i.e., all loops terminate. Let P be the postcondition that must be established by

```
while B do L
```

where B is a boolean expression and L is a statement list. The problem is that we do not know how many times the body of the loop is iterated. Indeed, if we know, for example, that the number of iterations is n, the con- struct would be equivalent to the sequential composition

```
L; L; . . .; L;
```

of length n. Thus the semantics of the statement would be straightforward. Since that is unknown, we relax our requirements. Instead of providing the *weakest* precondition, which gives the exact characterization of semantics, we choose to provide just a precondition, i.e., a sufficient precondition that can be derived for a given statement and a given postcondition. The constraint on the state specified by a (non-weakest) precondition is stronger than that strictly necessary to ensure a certain postcondition to hold after execution of a statement, but nonetheless it provides a specification of what the loop does. Such a precondition Q for a while statement and a postcondition P, must be such that

- the loop terminates, and
- at loop exit, P holds.

Predicate Q can be written as Q = T and R, where T implies termination of the loop and R implies the truth of P at loop exit. Let us ignore the problem of termination and let us focus on determining R. This cannot be done mechanically, in general, but requires ingenuity from the programmer. A systematic method consists of identifying a predicate I that holds both before and after each loop iteration and such that, when the loop terminates (i.e., when the boolean expression B is false), I implies P. I is called an *invariant* predicate for the loop. Formally, I satisfies the following conditions

(i) I and B $\supset$ asem (L, I)
(ii) I and not B $\supset$ P

If we are able to identify a predicate I that satisfies both (i) and (ii), then we can take I as the desired precondition R for the loop, because P holds upon termination if R = I holds before executing the loop. In conclusion, the method of loop invariants allows us to approximate the evaluation of semantics of a while statement; the precondition is one possible valid precondition, not necessarily the weakest.

This approximation can be tolerated in practice. In fact, the main use of axiomatic semantics is proving programs correct, i.e., proving that under certain specified constraints on input data (stated as an input predicate—the precondition for the entire program), the program terminates in a final state satisfying a specified constraint on output data (stated as an output predicate).

Denotational semantics associates each language statement with a function dsem from the state of the program before the execution to the state after execution. The *state* (i.e., the values stored in the memory) is represented by a function mem from the set of program identifiers ID to values. Thus denotational semantics differs from axiomatic semantics in the way states are described (functions vs. predicates). For simplicity, we assume that values can only belong to type integer.

Let us start our analysis from arithmetic expressions and assignments[1]. For an expression expr, mem (expr) is defined as error if mem (v) is undefined for some

[1] This characterization of assignments is correct for simple assignment statements (see Exercise 41).

variable v occurring in expr. Otherwise mem (expr) is the result of evaluating expr after replacing each variable v in expr with mem (v).

If x = expr is an assignment statement and mem is the function describing the memory before executing the assignment

dsem (x := expr, mem) = error

if mem (x) is undefined for some variable x occurring in expr. Otherwise

dsem (x: = expr, mem) = mem'

where mem' (y) = mem (y) for all y ¦ x, mem' (x) = mem (expr).

As axiomatic semantics, denotational semantics is defined compositionally. That is, given the state transformation caused by each individual statement, it provides the state transformation caused by compound statements and, eventually, by the entire program.

Let us consider a statement list, like

S1; S2;

If

dsem (S1, mem) = mem1

and

dsem (S2, mem1) = mem2

then

dsem (S1; S2;, mem) = mem2

The state error propagates implicitly, i.e. dsem (S, error) = error for any kind of statement S.

Let **if** B **then** L1 **else** L2; be a conditional statement, where B is a boolean expression, L1 and L2 are two statement lists. Semantics can be defined as follows:

dsem (**if** B **then** L1 **else** L2, mem) = U

where U = dsem (L1, mem), if the mem (B) = true; otherwise U = dsem (L2, mem).

Finally, let us consider a statement like **while** B **do** L.

dsem (while B do L, mem) = mem

if mem (B) = false (i.e., if the loop is not entered, there is no change of memory); Otherwise

dsem (while B do L, mem) = dsem (while B do L, dsem (L, mem))

if mem (B) = true (i.e., a loop iteration is performed).

By applying the semantic rules provided above, given a simple program described by the language of Figure 5, it is possible to compute the value of function mem for the entire program.

sidebar-end

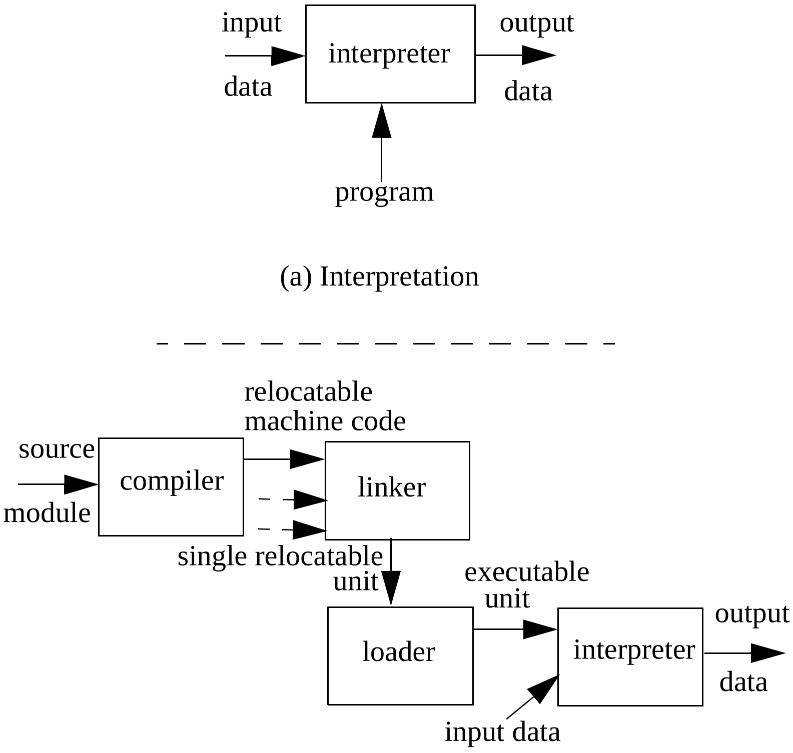
2.2 Language processing

Although in theory it is possible to build special-purpose computers to execute directly programs written in any particular language, present-day computers directly execute only a very low-level language, the machine language. Machine languages are designed on the basis of speed of execution, cost of realization, and flexibility in building new software layers upon them. On the other hand, programming languages often are designed on the basis of the ease and reliability of programming. A basic problem, then, is how a higher-level language eventually can be executed on a computer whose machine language is very different and at a much lower level.

There are generally two extreme alternatives for an implementation: interpretation and translation.

2.2.1 Interpretation

In this solution, the actions implied by the constructs of the language are executed directly (see Figure 7). Usually, for each possible action there exists a subprogram–written in machine language–to execute the action. Thus, interpretation of a program is accomplished by calling subprograms in the appropriate sequence.



(a) Interpretation

(b) Translation (+ interpretation)

FIGURE 7. Language processing by interpretation (a) and translation (b)

More precisely, an interpreter is a program that repeatedly executes the following sequence.

```
Get the next statement;
Determine the actions to be executed;
Perform the actions;
```

This sequence is very similar to the pattern of actions carried out by a traditional computer, that is:

8. Fetch the next instruction (i.e., the instruction whose address is specified by the instruction pointer).
9. Advance the instruction pointer (i.e., set the address of the instruction to be fetched next).
10. Decode the fetched instruction.
11. Execute the instruction.

This similarity shows that interpretation can be viewed as a simulation, on a host computer, of a special-purpose machine whose machine language is the higher level language.

2.2.2 Translation

In this solution, programs written in a high-level language are translated into an equivalent machine-language version before being executed. This translation is often performed in several steps (see Figure 7). Program modules might first be separately translated into relocatable machine code; modules of relocatable code are linked together into a single relocatable unit; finally, the entire program is loaded into the computer's memory as executable machine code. The translators used in each of these steps have specialized names: compiler, linker (or linkage editor), and loader, respectively.

In some cases, the machine on which the translation is performed (the host machine) is different from the machine that is to run the translated code (the target machine). This kind of translation is called cross-translation. Cross-translators offer the only viable solution when the target machine is a special-purpose processor rather han a general-purpose one that can support a translator.

Pure interpretation and pure translation are two ends of a continuous spectrum. In practice, many languages are implemented by a combination of the two techniques. A program may be translated into an intermediate code that is then interpreted. The intermediate code might be simply a formatted representation of the original program, with irrelevant information (e.g., comments and spaces) removed and the components of each statement stored in a fixed format to simplify the subsequent decoding of instructions. In this case, the solution is basically interpretive. Alternatively, the intermediate code might be the (low-level) machine code for a virtual machine that is to be later interpreted by software. This solution, which relies more heavily on translation, can be adopted for generating portable code, that is, code that is more easily transferable to different machines than machine language code. For example, for portability purposes, one of the best known initial implementations of a Pascal compiler was written in Pascal and generated an intermediate code, called Pcode. The availability of a portable implementation of the language contributed to the rapid diffusion of Pascal in many educational environments. More recently, with the widespread use of Internet, code portability became a primary concern for network application developers. A number of language efforts have recently been undertaken with the goal of supporting code mobility over a network. Language Java is perhaps the best known and most promising example. Java is first translated to an intermediate code, called Java bytecode, which is interpreted in the client machine.

In a purely interpretive solution, executing a statement may require a fairly complicated decoding process to determine the operations to be executed and their operands. In most cases, this process is identical each time the statement is encountered. Consequently, if the statement appears in a frequently-executed part of a program (e.g., an inner loop), the speed of execution is strongly affected by this decoding process. On the other hand, pure translation generates machine code for each high-level statement. In so doing, the translator decodes each high-level statement only once. Frequently-used parts are then decoded many times in their machine language representation; because this is done efficiently by hardware, pure translation can save processing time over pure interpretation. On the other hand, pure interpretation may save storage. In pure translation, each high-level language statement may expand into tens or hundreds of machine instructions. In a purely interpretive solution, high-level statements are left in the original form and the instructions necessary to execute them are stored in a subprogram of the interpreter. The storage saving is evident if the program is large and uses most of the language's statements. On the other hand, if all of the interpreter's subprograms are kept in main memory during execution, the interpreter may waste space for small programs that use only a few of the language's statements.

Compilers and interpreters differ in the way they can report on run-time errors. Typically, with compilation, any reference to the source code is lost in the generated object code. If an error is generated at run-time, it may be impossible to relate it to the source language construct being executed. This is why run-time error messages are often obscure and almost meaningless to the programmer. On the opposite, the interpreter processes source statements, and can relate a run-time error to the source statement being executed. For these reasons, certain programming environments contain both an interpreter and a compiler for a given programming language. The interpreter is used while the program is being developed, because of its improved diagnostic facilities. The compiler is then used to generate efficient code, after the program has been fully validated.

Macro processing is a special kind of translation that may occur as the first step in the translation of a program. A macro is a named source text fragment, called the macro body. Through macro processing, macro names in a text are replaced by the corresponding bodies. In C, one can write macros, handled by a preprocessor, which generates source C code through macro expansion. For example, one can use a macro to provide a symbolic name for a constant value, as in the following fragment.

```
#define upper_limit 100
. . .
sum = 0;
for (index = 0; index < upper_lmit; index++)
{
    sum += a [index];
}
```

2.2.3 The concept of binding

Programs deal with entities, such as variables, routines, statements, and so on. Program entities have certain properties called attributes. For example, a variable has a name, a type, a storage area where its value is stored; a routine has a name, formal parameters of a certain type, certain parameter-passing conventions; a statement has associated actions. Attributes must be specified before an entity is elaborated. Specifying the exact nature of an attribute is known as *binding*. For each entity, attribute information is contained in a repository called a *descriptor*.

Binding is a central concept in the definition of programming language semantics. Programming languages differ in the number of entities with which they can deal, in the number of attributes to be bound to entities, in the time at which such bindings occur (*binding time*), and in the *stability* of the binding (i.e., whether an established binding is fixed or modifiable). A binding that cannot be modified is called *static*. A modifiable binding is called *dynamic*.

Some attributes may be bound at language definition time, others at program translation time (or compile time), and others at program execution time (or run time). The following is a (nonexhaustive) list of binding examples:

- Language definition time binding. In most languages (including FORTRAN, Ada, C, and C++) the type "integer" is bound at language definition time to its well-known mathematical counterpart, i.e., to a set of algebraic operations that produce and manipulate integers;

- Language implementation time binding. In most languages (including FORTRAN, Ada, C, and C++) a set of values is bound to the integer type at language implementation time. That is, the language definition states that type "integer" must be supported and the language implementation binds it to a memory representation, which–in turn–determines the set of values that are contained in the type.
- Compile time (or translation time) binding. Pascal provides a predefined definition of type integer, but allows the programmer to redefine it. Thus type integer is bound a representation at language implementation time, but the binding can be modified at translation time.
- Execution time (or run time) binding. In most programming languages variables are bound to a value at execution time, and the binding can be modified repeatedly during execution.

In the first two examples, the binding is established before run time and cannot be changed thereafter. This kind of binding regime is often called *static*. The term static denotes both the binding time (which occurs before the program is executed) and the stability (the binding is fixed). Conversely, a binding established at run time is usually modifiable during execution. The fourth example illustrates this case. This kind of binding regime is often called *dynamic*. There are cases, however, where the binding is established at run time, and cannot be changed after being established. An example is a language providing (read only) constant variables that are initialized with an expression to be evaluated at run time.

The concepts of binding, binding time, and stability help clarify many semantic aspects of programming languages. In the next section we will use these concepts to illustrate the notion of a variable.

2.3 Variables

Conventional computers are based on the notion of a main memory consisting of elementary cells, each of which is identified by an address. The contents of a cell is an encoded representation of a value. A value is a mathematical abstraction; its encoded representation in a memory cell can be read and (usually) modified during execution. Modification implies replacing one encoding with a new encoding. With a few exceptions, programming languages can be viewed as abstractions, at different levels, of the behavior of such conventional computers. In particular, they introduce the notion of variables as an abstraction of the notion of memory cells. the variable name as an abstraction of the address. and the notion of assignment statements as an abstraction of the destructive modification of a cell.

In most of this and the following chapters we basically will restrict our considerations to these conventional, “assignment-based” programming languages. Alternative languages that support functional and declarative styles of programming will be discussed in Chapters 7 and 8.

Formally, a variable is a 5-tuple <name, scope, type, l_value, r_value>, where

- name is a string of characters used by program statements to denote the variable;
- scope is the range of program instructions over which the name is known;
- type is the variable’s type;
- l_value is the memory location associated with the variable;
- r_value is the encoded value stored in the variable’s location.

These attributes are described below, in Section 2.3.1 through Section 2.3.4, along with the different policies that can be adopted for attribute binding. Section 2.3.5 discusses the special case of references and unnamed variables, which diverge from the present model.

2.3.1 Name and scope

A variable’s *name* is usually introduced by a special statement, called *declaration* and, normally, the variable’s *scope* extends from that point until some later closing point, specified by the language. The scope of a variable is the range of program instructions over which the name is known. Program instructions can manipulate a variable through its name within its scope. We also say that a variable is visible under its name within its scope, and invisible outside it. Different programming languages adopt different rules for binding variable names to their scope.

For example, consider the following example of a C program:

```
# include <stdio.h>
main ( )
{
    int x, y;
    scanf ("%d %d", &x, &y);
        /*two decimal values are read and stored in the l_values of x and y */
    {
        /*this is a block used to swap x and y*/
        int temp;
        temp = x;
        x = y;
        y = temp;
    }
    printf ("%d %d", x, y);
}
```

The declaration int x, y; makes variables named x and y visible throughout program main. The program contains an internal block, which groups a declaration and statements. The declaration int temp; appearing in the block makes a variable named temp visible within the inner block, and invisible outside. Thus, it would be impossible to insert temp as an argument of operation printf.

In the example, if the inner block declares a new local variable named x, the outer variable named x would not be visible in it. The inner declaration masks the outer variable. The outer variable, however, continues to exist even though it is invisible. It becomes visible again when control exits the inner block.

Variables can be bound to a scope either statically or dynamically. *Static scope binding* defines the scope in terms of the lexical structure of a program, that is, each reference to a variable can be statically bound to a particular (implicit or explicit) variable declaration by examining the program text, without executing it. Static scope binding rules are adopted by most programming languages, such as C, as we saw in the previous example.

Dynamic scope binding defines the scope of a variable's name in terms of program execution. Typically, each variable declaration extends its effect over all the instructions executed thereafter, until a new declaration for a variable with the same name is encountered during execution. APL, LISP (as originally defined), and SNOBOL4 are examples of languages with dynamic scope rules.

As an example, consider the following program fragment written in a C-like notation.

```
{
    /* block A */
    int x;
    . . .
}
. . .
{
    /* block B */
    int x;
    . . .
}
. . .
{
    /* block C */
    . . .
    x = ...;
    . . .
}
```

If the language follows dynamic scoping, an execution of block A followed by block C would make variable x in the assignment in block C to refer to x declared in block A. Instead, an execution of block B followed by block C would make variable x in the assignment in block C refer to x declared in block B. Thus, name x in block C refers either to the x declared in A or the one declared in B, depending on the flow of control followed during execution.

Dynamic scope rules look quite simple and are rather easy to implement, but they have disadvantages in terms of programming discipline and efficiency of implementation. Programs are hard to read because the identity of the particular declaration to which a given variable is bound depends on the particular point of execution, and so cannot be determined statically.

2.3.2 Type

In this section we provide a preliminary introduction to types. The subject will be examined in more depth in Chapters 3 and 6. We define the type of a variable as a specification of the set of values that can be associated with the variable, together with the operations that can be legally used to create, access, and modify such values. A variable of a given type is said to be an *instance* of the type.

When the language is defined, certain type names are bound to certain classes of values and sets of operations. For example, type integer and its associated operators are bound to their mathematical counterpart. Values and operations are bound to a certain machine representation when the language is implemented. The latter binding may also restrict the set of values that can be represented, based on the storage capacity of the target machine.

In some languages, the programmer can define new types by means of type declarations. For example, in C one can write

```
typedef int vector [10];
```

This declaration establishes a binding–at translation time–between the type name vector and its implementation (i.e., an array of 10 integers, each accessible via an index in the subrange 0. .9). As a consequence of this binding, type vector inherits all the operations of the representation data structure (the array); thus, it is possible to read and modify each component of an object of type vector by indexing within the array.

There are languages that support the implementation of user-defined types (usually called *abstract data types*) by associating the new type with the set of operations that can be used on its instances; the operations are described as a set of routines in the declaration of the new type. The declaration of the new type has the following general form, expressed in C-like syntax:

```
typedef new_type_name
{
    data structure representing objects of type new_type_name;
    routines to be invoked for manipulating data objects of type new_type_name;
}
```

To provide a preview of concepts and constructs that will be discussed at length in this text, Figure 8 illustrates an example of an abstract data type (a stack of characters) implemented as a C++ class[1]. The class defines the hidden data structure (a pointer s to the first element of the stack, a pointer top to the most recently inserted character, and an integer denoting the maximum size) and five routines to be used for manipulating stack objects. Routines stack_of_char and ~stack_of_char are used to construct and destruct objects of type stack_of_char, respectively. Routine push is used to insert a new element on top of a stack object. Routine pop is used to extract an element from a stack

[1] A note for the reader who is not familiar with C or C++. Expression *top++ is evaluated as *(top++). The value of top++ is the value of top before it is incremented. Such value is used for dereferencing. Similarly, expression *--top is evaluated as *(--top). That is, top is decremented, and its new value is used for dereferencing.

object. Routine length yields the current size of a stack object.

```
class stack_of_char{
    int size;
    char* top;
    char* s;
public:
    stack_of_char (int sz) {
        top = s = new char [size =sz];
    }
    ~stack_of_char ( ) {delete [ ] s;}
    void push (char c) {*top++ = c;}
    char pop ( ) {return *--top;}
    int length ( ) {return top - s;}
};
```

FIGURE 8. User-defined type in C++

Traditional languages, such as FORTRAN, COBOL, Pascal, C, C++, Modula-2, and Ada bind variables to their type at compile time, and the binding cannot be changed during execution. This solution is called *static typing*. In these languages, the binding between a variable and its type is specified by a variable declaration. For example, in C one can write:

```
int x, y;
char c;
```

By declaring variables to belong to a given type, variables are automatically protected from the application of illegal (or nonsensical) operations. For example, in Ada the compiler can detect the application of the illegal assignment I:= not A, if I is declared to be an integer and A is a boolean. Through this check, the compiler watches for violations to static semantics concerning variables and their types. The ability to perform checks before the program is executed (*static type checking*) contributes to early error detection and enhances program reliability.

In some languages (such as FORTRAN) the first occurrence of a new variable name is also taken as an implicit declaration. The advantage of explicit declarations lies in the clarity of the programs and improved reliability, because such things as spelling errors in variable names can be caught at translation time. For example, in FORTRAN the declaration of variable ALPHA followed by a statement such as ALPA = 7.3 intended to assign a value to it, would not be detected as an error. ALPA would not be considered as an incorrect occurrence of an undeclared variable (i.e., as a misspelled, ALPHA), but as the implicit declaration of a new variable, ALPA.

Note that the issue of implicit type declarations is not a semantic one. Semantically, C and FORTRAN are equivalent with respect to the typing of variables because they both bind variables to types at translation time. FORTRAN has default rules to determine the particular binding but the time of binding and its stability are the same in the two languages. The FORTRAN rule that determines the type of a variable is quite simple. ML pushes the approach to its extreme, by allowing most types not to be stated explicitly, and yet all expressions to be type checked statically. This is achieved through a *type inference* procedure, which will be discussed in Chapter 7.

Assembly languages, LISP, APL, SNOBOL4, and Smalltalk are languages that establish a (modifiable) run-time binding between variables and their type. This binding strategy is called dynamic typing. Dynamically typed variables are also called *polymorphic variables* (literally, from ancient Greek, “multiple shape”) variables.

In most assembly languages, variables are dynamically typed. This reflects the behavior of the underlying hardware, where memory cells and registers can contain bit strings that are interpreted as values of any type. For example, the bist string stored in a cell may be added to the bit string stored in a register using integer addition. In such a case, the bit strings are interpreted as integer values. In other languages, the type of a variable depends on the value that is dynamically associated with it. For example, having assigned an integer value to a variable, such value cannot be treated as if it were–say–a string of characters. That is, the binding is still dynamic, but once a value is bound to a variable, an implicit binding with a type is also established, and the binding remains in place until a new value is assigned.

As another example, in LISP, variables are not explicitly declared; their type is implicitly determined by the value they currently hold during execution. The LISP function CAR applied to a list L yields the first element of L, which may be an atom (say, an integer) or a list of–say–strings, if L is a list of lists. If the element returned by CAR is bound to a variable, the type of such variable would be an integer in the former case, a string list in the latter. If such value is added to an integer value, the operation would be correct in the former case, but would be illegal in the latter. Moreover, suppose that the value of the variable is to be printed. The effect of the print operation depends on the type that is dynamically associated with the variable. It prints an integer in the former case; a list of strings in the latter. Such a print routine, which is applicable to arguments of more than one type, is called a *polymorphic routine*.

In general, dynamic typing prevents programs from being statically type checked: since the type is not known, it is impossible to check for type violations before executing the program. Type violations due to dynamic typing can only be checked at run time, through dynamic type checking. In order to perform dynamic type checking, information about the dynamic type of variables must be kept at run time in suitable descriptors. Such descriptors only need to exist at translation time for statically typed languages. Perhaps surprisingly, however, there are languages supporting both static type checking and polymorphic variables and routines. This will be discussed at length in Chapters 3, 6, and 7.

Programming languages that adopt dynamic typing are often implemented by interpretation. In general, in fact, there is not enough information before run time to generate object code that performs storage allocation, type checking, and binding between operation invocations and their implementation.

2.3.3 l_value

The l_value of a variable is the storage area bound to the variable during execution. The *lifetime*, or extent, of a variable is the period of time in which such binding exists. The storage area is used to hold the r_value of the variable. We will use the term *data object* (or simply, object) to denote the pair <l_value, r_value>.

The action that acquires a storage area for a variable–and thus establishes the binding–is called *memory allocation*. The lifetime extends from the point of allocation to the point in which the allocated storage is reclaimed (*memory deallocation*). In some languages, for some kinds of variables, allocation is performed before run time and storage is only reclaimed upon termination (*static allocation*). In other languages, it is performed at run time (*dynamic allocation*), either upon explicit request from the programmer via a creation statement or automatically, when the variable's declaration is encountered, and reclaimed during execution. Section 2.6 presents an extensive analysis of these issues.

In most cases, the lifetime of a program variable is a fraction of the program's execution time. It is also possible, however, to have persistent objects. A persistent object exists in the environment in which a program is executed and its lifetime has no a-priori relation with any given program's execution time. Files are an example of persistent objects. Once they are created, they can be used by different program activations, and different activations of the same program, until they are deleted through a specific command to the operating system. Similarly, persistent objects can be stored in a database, and made visible to a programming language through a specific interface. A discussion of persistent objects will be taken up in Chapter 10.

2.3.4 r_value

The r_value of a variable is the encoded value stored in the location associated with the variable (i.e., its l_value). The encoded representation is interpreted according to the variable's type. For example, a certain sequence of bits stored at a certain location would be interpreted as an integer number if the variable's type is int; it would be interpreted as a string if the type is an array of char.

l_values and r_values are the main concepts related to program execution. Program instructions access variables through their l_value and possibly modify their r_value. The terms l_value and r_value derive from the conventional form of assignment statements, such as x = y; in C. The variable appearing at the left-hand side of the assignment denotes a location (i.e., its l_value is meant).The variable appearing at the right-hand side of the assignment denotes the contents of a location (i.e., its r_value is meant). Whenever no ambiguity arises, we use the simple term "value" of a variable to denote its r_value.

The binding between a variable and the value held in its storage area is usually dynamic; the value can be modified by an assignment operation. An assignment such as b = a; causes a's r_value to be copied into the storage area referred to by b's l_value. That is, b's r_value changes. This, however, is true only for conventional imperative languages, like FORTRAN, C, Pascal, Ada, and C++. Functional and logic programming languages treat variables as their mathematical counterpart: they can be bound to a value by the evaluation process, but once the binding is established it cannot be changed during the variable's lifetime.

Some conventional languages, however, allow the binding between a variable and its value to be frozen once it is established. The resulting entity is, in every respect, a user-defined symbolic constant. For example, in C one can write

```
const float pi = 3.1415;
```

and then use pi in expressions such as

```
circumference= 2 * pi * radius;
```

Variable pi is bound to value 3.1416 and its value cannot be changed; that is, the translator reports an error if there is an assignment to pi. A similar effect can be achieved in Pascal.

Pascal and C differ in the time of binding between the const variable and its value, although binding stability is the same for both languages. In Pascal the value is provided by an expression that must be evaluated at compile time; i.e., binding time is compile time. The compiler can legally substitute the value of the constant for its symbolic name in the program. In C and Ada the value can be given as an expression involving other variables and constants: consequently, binding can only be established at run time, when the variable is created.

A subtle question concerning the binding between a (non const) variable and its value is: What is the r_value bound to the variable immediately after it is created? Some languages allow the initial value of a variable to be specified when the variable is declared. For example, in C one can write

```
int i = 0, j = 0;
```

Similarly, in Ada one would write

```
i, j: INTEGER := 0;
```

But what if no initialization is provided? There are a number of possible approaches that might be followed, but unfortunately most language definitions fail to specify which one they choose. As a result, the problem is solved differently by different implementations of the same language, and thus program behavior depends on the implementation. Moving an apparently correct program to a different platform may produce unforeseen errors or unexpected results.

One obvious and frequently adopted solution to the problem is to ignore it. In this case, the bit string found in the area of storage associated with the variable is considered its initial value. Another solution is to provide a system-defined initialization strategy: for example, integers are initialized to zero, characters to blank, and so on. Yet another solution consists of viewing an uninitialized variable as initialized with a special undefined value, and trapping any read accesses to such variables until a meaningful value is assigned to the variable. This solution, by far the cleanest, can be enforced in different ways. Its only drawback could be the cost associated with the run-time checks necessary to ensure that a meaningless value is never used in the program.

2.3.5 References and unnamed variables

Some languages allow unnamed variables that can be accessed through the r_value of another variable. Such a r_value is called a *reference* (or a *pointer*) to the variable. In turn, the reference can be the r_value of a named variable, or it can be the r_value of a referenced variable. Thus, in general, an object can be made accessible via a chain of references (called *access path*) of arbitrary length.

If A = <A_name, A_scope, A_type, A_l_value, A_r_value> is a named variable, object <A_l_value, A_r_value> is said to be directly accessible through name A_name in A_scope, with an access path of length 0. If B= <--, --, --, B_l_value, B_r_value>, where -- stands for the "don't care value", is a variable and B_l_value = A_r_value, object <B_l_value, B_r_value> is said to be accessible through name A_name in A_scope indirectly, with an access path of length 1. Similarly, one can define the concept of an object indirectly accessible through a named variable, with an access path of length i, i>1.

For example, in Pascal we can declare type PI (pointer to an integer):

```
type    PI = ^ integer;
```

We can then allocate an unnamed integer variable and have it pointed by a variable pxi of type PI:

```
new (pxi);
```

In order to access the unnamed object referenced by pxi, it is necessary to use the *dereferencing* operator ^, which can be applied to a pointer variable to obtain its r_value, i.e., the l_value of the referenced object. For example, the value of the unnamed variable can be set to zero by writing:

```
pxi^ := 0;
```

The unnamed variable can also be made accessible indirectly through a “pointer to a pointer to an integer”, as sketched below:

```
type PPI = ^PI;
var ppxi: PPI;
. . .
new (ppxi);
^ppxi := pxi;
```

Here the r_value of variable ppxi is the l_value of an unnamed variable, whose r_value is the l_value of variable x.

Other languages, like C, C++, and Ada, allow pointers to refer to named variables. For example, the following C fragment:

```
int x = 5;
int* px;
px = &x;
```

generates an integer object whose r_value is 5, directly accessible through a variable named x and indirectly accessible (with an access path of length 1) through px, declared as a pointer to integer. This is achieved by assigning to px the value of the address of x (i.e., &x). Indirect access to x is then made possible by dereferencing px. In C (and C++) the dereferencing operator is denoted by *. For example, the following C instruction

```
int y = *px;
```

assigns to y the r_value of the variable pointed at by px (i.e., 5).

Two variables are said to *share* an object if each has an access path to the object. A shared object modified via a certain access path makes the modification visible through all possible access paths. Sharing of objects is used to improve efficiency, but it can lead to programs that are hard to read, because the value of a variable can be modified even when its name is not used. In the previous C example, if one writes:

```
*px = 0;
```

the contents of the location pointed at by px becomes 0 and, because of sharing, the value of x becomes 0 too.

2.4 Routines

Programming languages allow a program to be composed of a number of units, called *routines*. The neutral term “routine” is used in this chapter in order to provide a general treatment that enlightens the important principles that are common to most programming languages, without committing to any specific feature offered by individual languages. Routines can be developed in a more or less independent fashion and can sometimes be translated separately and combined after translation. Assembly language subprograms, FORTRAN subroutines, Pascal and Ada procedures and functions, C functions are well-known examples of routines. In this chapter we will review the main syntactic and semantic features of routines, and in particular the mechanisms that control the flow of execution among routines and the bindings established when a routine is executed. Other, more general kinds of units, such as Ada packages, Modula-2 modules, and C++ classes will be described elsewhere.

In the existing programming language world, routines usually come in two forms: procedures and functions. Functions return a value; procedures do not. Some languages, e.g., C and C++, only provide functions, but procedures are easily obtained as functions returning the null value void. Figure 9 shows the example of a C function definition.

```
/* sum is a function which computes the sum
of the first n positive integers, 1 + 2 + ... + n;
parameter n is assumed to be positive */
int sum (int n)
{
    int i, s;
    s = 0;
    for (i = 1; i <= n ; ++i)
        s+= i;
    return s:
}
```

FIGURE 9. A C function definition

Like variables, routines have a name, scope, type, l_value, and r_value. A routine name is introduced in a program by a routine *declaration*. Usually the scope of such name extends from the declaration point on to some closing construct, statically or dynamically determined, depending on the language. For example, in C a function declaration extends the scope of the function till the end of the file in which the declaration occurs.

Routine activation is achieved through a *routine call*, which names the routine and specifies the parameters on which the routine operates. Since a routine is activated by call, the call statement must be in the routine's scope. Besides having their own scope, routines also define a scope for the declarations that are nested in them. Such *local* declarations are only visible within the routine. Depending on the scope rules of the language, routines can also refer to nonlocal items (e.g., variables) other than those declared locally. Nonlocal items that are potentially referenced by every unit in the program are called *global* items.

The *header* of the routine defines the routine’s name, its parameter types, and the type of the returned value (if any). In brief, the routine’s header defines the *routine type*. In the example of Figure 9, the routine’s type is:

```
routine with one int parameter and returning int
```

A routine type can be precisely defined by the concept of *signature*. The signature specifies the types of parameters and the return type. A routine fun which behaves like a function, with input parameters of types T1, T2, . . ., Tn and returning a value of type R, can be defined by the following signature:

fun: T_1 x T_2 x . . .x T_n -> R

A routine call is type correct if it conforms to the corresponding routine type. For example, the call

```
i = sum (10);     /* i is declared as an int */
```

would be correct with respect to the function definition of Figure 9. Instead, the call

```
i = sum (5.3);
```

would be incorrect.

A routine’s l_value is a reference to the memory area which stores the routine *body* (i.e., the routine’s executable statements). Activation causes execution of the routine body, which constitutes the r_value that is currently bound to the routine. In the above C example, the r_value is bound to the routine statically, at translation time. A more dynamic binding policy can be achieved by languages which support the concept of variables of type routine, to which a routine value can be assigned. Some languages support the notion of a “pointer to a routine” and provide a way of getting a routine l_value, which can be assigned (as a r_value) to a pointer. For example, the following C statement declares a pointer ps to a function with an int parameter and returning an int:

```
int (*ps) (int);
```

The following assignment

```
ps = & sum;
```

makes ps point to the l_value of the previously defined routine sum. A call may then be issued via ps as in the following example:

```
int i = (*ps) (5);   /* this invokes the r_value of the routine that is currently referred to by
ps */
```

The use of pointers to routines allows different routines to be invoked each time a pointer is dereferenced. This provides a way to achieve a dynamic binding policy, that cannot be achieved by directly calling a routine, which is statically bound to its body. Languages that exploit the distinction between routine l_value and r_value, and allow variables of type routine and pointers to routines to be defined and manipulated, treat routines in much the same way as variables: they are said to treat routines as *first-class objects*.

Some languages (like Pascal, Ada, C, and C++) distinguish between *declaration* and *definition* of a routine. A routine declaration introduces the routine's header, without specifying the body. The name is visible from the declaration point on, up to the scope end. The definition specifies both the header and the body. The distinction between declaration and definition is necessary to allow routines to call themselves in a mutual recursion scheme, as illustrated by the following fragment.

```
int A (int x, int y); // declaratiuon of afunction with two int
                      // parameters and returning an int
                      // A is visible from this point on
float B (int z) //this is a definition of a function; B is visible from this point on
{
    int w, u;
    . . .
    w = A (z, u); //calls A, which is visible at this point
    . . .
};
int A (int x, int y) //this is A’s definition
{
    float t;
    . . .
    t = B (x); //B is visible here
    . . .
}
```

A routine definition specifies a computational process. At invocation time, an instance of the process is executed for the particular values of the parameters. The representation of a routine during execution is called a *routine instance*. A routine instance is composed of a code segment and an activation record. The *code segment*, whose contents are fixed, contains the instructions of the unit. The contents of the *activation record* (also called *frame)* are changeable. The activation record contains all the information necessary to execute the routine, including, among other things, the data objects associated with the local variables of a particular routine instance. The relative position of a data object in the activation record is called its *offset*. In order to support returning the execution to the caller, the return point is saved as part of the activation record at routine invocation time.

The *referencing environment* of a routine instance U consists of U's local variables, which are bound to objects stored in U's activation record (*local environment*), and U's nonlocal variables, which are bound to objects stored in the activation records of other units (*nonlocal environment*). The modification of a data object bound to a nonlocal variable is called a *side-effect*.

Routines can often be activated *recursively*, that is, a unit can call itself either directly or indirectly through some other unit. In other words, a new activation can occur before termination of the previous. All the instances of the same unit are composed of the same code segment but different activation records. Thus, in the presence of recursion, the binding between an activation record and its code segment is necessarily dynamic. Every time a unit is activated, a binding must be established between an activation record and its code segment to form a new unit instance.

When a routine is activated, parameters may be passed from the caller to the callee. Parameter passing allows for the flow of information among program units. In most cases, only data entities may be passed. In some cases, routines may also be passed. In particular, this feature is offered by languages where routines are first-class objects.

Parameter passing and communication via nonlocal data are two different ways of achieving inter-unit information flow. Unlike communication via global environments, parameters allow for the transfer of different data at each call and provide advantages in terms of readability and modifiability.

It is necessary to distinguish between *formal parameters* (the parameters that appear in the routine's definition) and *actual parameters* (the parameters that appear in the routine's call). Most programming languages use a *positional* method for binding actual to formal parameters in routine calls. If the routine's header is

```
routine S (F1,F2, . . . Fn);
```

and the routine call is

```
call S (A1, A2,... An)
```

the positional method implies that the formal parameter Fi is to be bound to actual parameter Ai, i = 1,2,... n. In some cases the number of actual and formal parameters need not be the same. For example, in C++ formal parameters can be given a default value, which is used in case the corresponding actual parameters are not passed in the call.

For example, given the following function header:

```
int distance (int a = 0, int b =0);
```

the call distance ( ) is equivalent to distance (0, 0), and the call distance (10) is equivalent to distance (10, 0). For further comments, see Exercise 42.

Besides the positional association method, Ada allows also a named parameter association. For example, having defined a procedure with the following header

```
procedure Example (A: T1; B: T2 := B1; C: T3);
    --parameters A, B, and C are of types T1, T2, and T3, respectively;
    --a default value is specified for parameter B, given by the value of B1
```

assuming X, Y, and Z to be of types T1, T2, and T3, respectively, the following calls are legal:

```
Example (X, Y, Z);
    --this is a pure positional association
Example (X, C => Z)
    --X is bound to A positionally, B gets the default value
    --Z is bound to C in a named association
Example (C => Z, A => X, B => Y);
    --all correspondences are named here
```

We take up the issue of parameter passing in Section 2.6.6, where we give an abstract implementation model and describe the different kinds of parameter passing modes.

2.4.1 Generic routines

Routines factor a code fragment that is executed at different points of the program in a single place and assign it a name. The fragment is then executed through invocation, and customized through parameters. Often, however, similar routines must be written several times, because they differ in some detail aspects that cannot be factored through parameters. For example, if a program needs both a routine to sort arrays of integers and arrays of strings, two different routines must be written, one for each parameter type, even if the abstract algorithm chosen for the implementation of the sort operation is the same in both cases.

Generic routines, as offered by some programming languages, provide a solution to this problem. In this section we provide a view of generic routines as they appear in languages like C++ or Ada. More complex schemes will be discussed in Chapter 4 and in the case of ML functions in Chapter 8. A generic routine can be made parametric with respect to a type. In the previous example, the routine would be generic with respect to the type of the array elements. Type parameters in a generic routine, however, differ from conventional parameters, and require a different implementation scheme. A generic routine is a template from which the specific routine is generated through Instantiation, an operation that binds generic parameters to actual parameters at compile time. Such binding can be obtained via macroprocessing, which generates a new instance (i.e., an actual routine) for each type parameter. Other implementation schemes, however, are also possible.

Figure 10 shows an example of a generic swap routine in C++. Generic C++ units are called templates. More on generics, their effect on reusability of program components, and the features offered by C++ in support of these concepts will be discussed in Chapter 7.

```
template <class T>  void swap (T& a , T& b)
/* the function does not return any value; it is generic with respect to type T;
a and b refer to the the same locations as the actual parameters;
swap interchanges the two values*/
{
    T temp = a;
    a = b;
    b = temp;
}
```

FIGURE 10. A generic routine in C++

2.4.2 More on scopes: aliasing and overloading

As our discussion so far emphasized, a central issue of programming language semantics has to do with the conventions adopted for naming. In programs, names are used to denote variables and routines. The language uses special names (denoted by operators), such as + or * to denote certain predefined operations. So far, we implicitly assumed that at each point in a program a name denotes exactly one entity, based on the scope rules of the language. Since names are used to identify the corresponding entity, the assumption of unique binding between a name and an entity would make the identification unambiguous. This restriction, however, is almost never true for existing programming languages.

For example, in C one can write the following fragment:

```
int i, j, k;
float a, b, c;
...
i = j + k;
a = b + c;
```

In the example, operator + in the two instructions of the program denotes two different entities. In the first expression, it denotes integer addition; in the second, it denotes floating-point addition. Although the name is the same for the operator in the two expressions, the binding between the operator and the corresponding operation is different in the two cases, and the exact binding can be established at compile time, since the types of the operands allow for the disambiguation.

We can generalize the previous example by introducing the concept of *overloading*. A name is said to be overloaded if more than one entity is bound to the name at a given point of a program and yet the specific occurrence of the name provides enough information to allow the binding to be uniquely established. In the previous example, the types of the operands to which + is applied allows for the disambiguation.

As another example, if the second instruction of the previous fragment would be changed to

```
a = b + c + b ( );
```

the two occurrences of name b would (unambiguously) denote, respectively, variable b and routine b with no parameters and returning a float value (assuming that such routine is visible by the assignment instruction). Similarly, if another routine named b, with one int parameter and returning a float value is visible, instruction

```
a = b ( ) + c + b (i);
```

would unambiguously denote two calls to the two different routines.

Aliasing is exactly the opposite of overloading. Two names are aliases if they denote the same entity at the same program point. This concept is especially relevant in the case of variables. Two alias variables share the same data object in the same referencing environment. Thus modification of the object under one name would make the effect visible, maybe unexpectedly, under the other.

Although examples of aliasing are quite common, one should be careful since this feature may lead to error prone and difficult to read programs. An example of aliasing is shown by the following C fragment:

```
int i;
int fun (int& a);
{
    . . .
    a = a + 1;
```

```
        printf ("%d", i);
        . . .
    }
    main ( )
    {
        . . .
        x = fun (i);
        . . .
    }
```

When function f is executed, names i and a in fun denote the same data object. Thus an assignment to a would cause the value of i printed by fun to differ from the value held at the point of call.

Aliasing can easily be achieved through pointers and array elements. For example, the following assignments in C

```
    int x = 0;
    int* i = &x;
    int* j = &x;
```

would make *i, *j, and x aliases.

2.5 An abstract semantic processor

To describe the operational semantics of programming languages, we introduce a simple abstract processor, called SIMPLESEM, and we show how language constructs can be executed by sequences of operations of the abstract processor. In this section, we provide the main features of SIMPLESEM; additional details will be introduced incrementally, as additional language features are introduced.

In its basic form, SIMPLESEM consists of an *instruction pointer* (the reference to the instruction currently being executed), a *memory*, and a processor. The memory is where the instructions to be executed and the data to be manipulated are stored. For simplicity, we will assume that these two parts are stored into two separate memory sections: the *code memory* (C) and the *data memory* (D). Both C's and D's initial address is 0 (zero), and both programs and data are assumed to be stored from the initial address. The instruction pointer (ip) is always used to point to a location in C; it is initialized to 0.

We use the notation D[X] and C[X] to denote the values stored in the X-th cell of D and C, respectively. Thus X is an l_value and D[X] is the corresponding r_value. Modification of the value stored in a cell is performed by instruction set, with two parameters: the address of the cell whose contents is to be set, and the expression evaluating the new value. For example, the effect on the data memory of instruction

```
set 10, D[20]
```

is to assign the value stored at location 20 into location 10.

Input/output in SIMPLESEM is achieved quite simply by using the set instruction and referring to the special registers read and write, which provide for communication of the SIMPLESEM machine with the outside world. For example,

```
set 15, read
```

means that the value read from the input device is to be stored at location 15;

```
set write, D[50]
```

means that the value stored at location 50 is to be transferred to the output device.

We are quite liberal in the way we allow values to be combined in expressions; for example, D[15]+D[33]*D[41] would be a an acceptable expression, and

```
set 99, D[15]+D[33]*D[41]
```

would be an acceptable instruction to modify the contents of location 99.

As we mentioned, ip is SIMPLESEM's instruction pointer, which is initialized to zero at each new execution and automatically updated as each instruction is executed. The machine, in fact, operates by executing the following steps repeatedly, until it encounters a special halt instruction:

1. Get the current instruction to be executed (i.e., C[ip]);
2. Increment ip;
3. Execute the current instruction.

Notice, however, that certain programming language instructions might modify the normal sequential control flow, and this must be reflected by SIMPLESEM. In particular, we introduce the following two instructions: jump and jumpt. The former represents an unconditional jump to a certain instruction. For example,

```
jump 47
```

forces the instruction stored at address 47 of C to be the next instruction to be executed; that is, it sets ip to 47. The latter represents a conditional jump, which occurs if an expression evaluates to true. For example, in:

```
jumpt 47, D[3] > D[8]
```

the jump occurs only if the value stored in cell 3 is greater than the value stored in cell 8.

SIMPLESEM allows *indirect addressing*. For example:

```
set D[10], D[20]
```

assigns the value stored at location 20 into the cell whose address is the value stored at location 10. Thus, if value 30 is stored at location 10, the instruction modifies the contents of location 30. Indirection is also possible for jumps. For example:

```
jump D[13]
```

jumps to the instruction stored at location 88 of C, if 88 is the value stored at location 13.

SIMPLESEM, which is sketched in Figure 11, is quite simple. It is easy to understand how it works and what the effects of executing its instructions are. In other terms, we can assume that its semantics is intuitively known; it does not require further explanations that refer to other, more basic concepts. The semantics of programming languages can therefore be described by rules that specify how each construct of the language is translated into a sequence of SIMPLESEM instructions. Since SIMPLESEM is perfectly understood, the semantics of newly defined constructs becomes also known. As we will see, however, SIMPLESEM will also be enriched as new programming language concepts are introduced. This will be done in this book incrementally, as we address the semantics of new concepts.

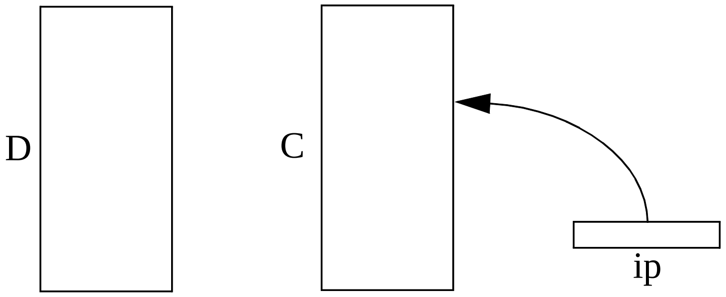


FIGURE 11. The SIMPLESEM machine

2.6 Execution-time structure

In this section we discuss how the most important concepts related to the execution-time processing of programming languages may be explained using SIMPLESEM. We will proceed gradually, from the most basic concepts to more complex structures that reflect what is provided by modern general-purpose programming languages. We will move through a hierarchy of languages that are based on variants of the C programing language. They are named C1 through C5.

Our discussion will show that languages can be classified in several categories, according to their execution-time structure.

Static languages.
Exemplified by the early versions of FORTRAN and COBOL, these languages guarantee that the memory requirements for any program can be evaluated before program execution begins. Therefore, all the needed memory can be allocated before program execution. Clearly, these languages cannot allow recursion, because recursion would require an arbitrary number of unit instances, and thus memory requirements could not be determined before execution. (As we will see later, the implementation is not required to perform the memory allocation statically. The semantics of the language, however, give the implementer the freedom to make that choice.)

Section 2.6.1 and Section 2.6.2 will discuss languages C1, C2, and its variant C2', all of which fall under the category of static languages.

Stack-based languages

Historically headed by ALGOL 60 and exemplified by the family of so-called Algol-like languages, this class is more demanding in terms of memory requirements, which cannot be computed at compile time. However, their memory usage is predictable and follows a last-in-first-out discipline: the latest allocated activation record is the next one to be deallocated. It is therefore possible to manage SIMPLESEM's D store as a stack to model the execution-time behavior of this class of languages. Notice that an implementation of these languages need not use a stack (although, most likely, it will): deallocation of discarded activation records can be avoided if store can be viewed as unbounded. In other terms, the stack is part of the semantic model we provide for the language; strictly speaking, it is not part of the semantics of the language.

Section 2.6.3 and Section 2.6.4 discuss languages C3 and C4, which fall under the category of stack-based languages.

Fully dynamic languages

These languages have un unpredictable memory usage; i.e, data are dynamically allocated only when they are needed during execution. The problem then becomes how to manage memory efficiently. In particular, how can unused memory be recognized and reallocated, if needed. To indicate that store D is not handled according to a predefined policy (like a FIFO policy for a stack memory), the term "heap" is traditionally used. This class of languages is illustrated by language C5 in Section 2.6.5.

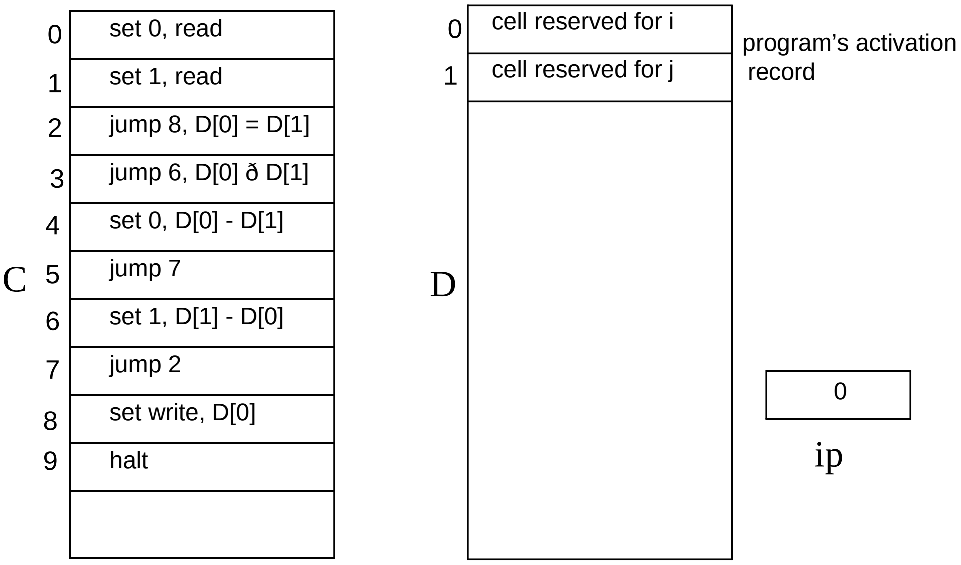
2.6.1 C1: A language with only simple statements

Let us consider a very simple programming language, called C1, which can be seen as a lexical variant of a subset of C, where we only have simple types and simple statements (there are no functions). Let us assume that the only data manipulated by the language are those whose memory requirements are known statically, such as integer and floating point values, fixed-size arrays, and structures. The entire program consists of a main routine (`main ( )`), which encloses a set of data declarations and a set of statements that manipulate these data. For simplicity, input/output is performed by invoking the operations get and print to read and write values, respectively.

```
main ( )
{
    int i, j;
    get (i, j);
    while (i != j)
        if (i > j)
            i -= j;
        else
            j -= i;
    print (i);
}
```

FIGURE 12.A C1 program

A C1 program is shown in Figure 12 and its straightforward SIMPLESEM representation before the execution starts is shown in Figure 12. The D portion shows the activation record of the main program, which contains space for all variables that appear in the program. The C portion shows the SIMPLESEM code.



```
0  set 0, read
1  set 1, read
2  jump 8, D[0] = D[1]
3  jump 6, D[0] ð D[1]
4  set 0, D[0] - D[1]
5  jump 7
6  set 1, D[1] - D[0]
7  jump 2
8  set write, D[0]
9  halt
```

FIGURE 13. Initial state of the SIMPLESEM machine for the C1 program in Figure 12

2.6.2 C2: Adding simple routines

Let us now add a new feature to C1. The resulting language, C2, allows routines to be defined in a program and allows routines to declare their own local data. A C2 program consists of a sequence of the following items:

- a (possibly empty) set of data declarations (*global data*);
- a (possibly empty) set of routine definitions and or declarations;
- a *main routine* (main ( )), which contains its local data declarations and a set of statements, that are automatically activated when the execution starts. The main routine cannot be called by other routines.

Routines may access their local data and any global data that are not redeclared internally. For simplicity, we assume that routines cannot call themselves recursively, do not have parameters, and do not return values (these restrictions will be removed later).

Figure 14 shows an example of a C2 program, whose main routine gets called initially, and causes routines beta and alpha to be called in a sequence.

```
int i = 1, j = 2, k = 3;
alpha ( )
{
    int i = 4, l = 5;
    ...
    i+=k+l;
    ...
};
beta ( )
{
    int k = 6;
    ...
    i=j+k;
    alpha ( );
    ...
};
main ( )
{
    ...
    beta ( );
    ...
}
```

FIGURE 14. A C2 program

Under the assumptions we made so far, the size of each unit’s activation record can be determined at translation time, and all activation records can be allocated before execution (*static allocation*). Thus each variable can be bound to a D memory address before execution. Static allocation is a straightforward implementation scheme which does not cause any memory allocation overhead at run time, but can waste memory space. In fact, memory is allocated for a routine even if it is never invoked. Since our purpose is to provide a semantic description, not to discuss an efficient implementation scheme, we assume static allocation. The run-time model described in Section 2.6.3 could be adapted to provide dynamic memory allocation for the C2 class of languages.

Figure 15 shows the state of the SIMPLESEM machine after instruction i += k + l of routine alpha has been executed. The first location of each activation record (offset 0) is reserved for the *return pointer*. Starting at location 1, space is reserved for the local variables. In general, for an instance of unit A, the return pointer will contain the address of the instruction that should be executed after unit A terminates. This does not apply to main, which does not return to a caller. On real computers, however, main is called by the operating system, and after termination main must return control to the operating system. Also notice that we maintain an activation record to keep global data at the low address end of store D.

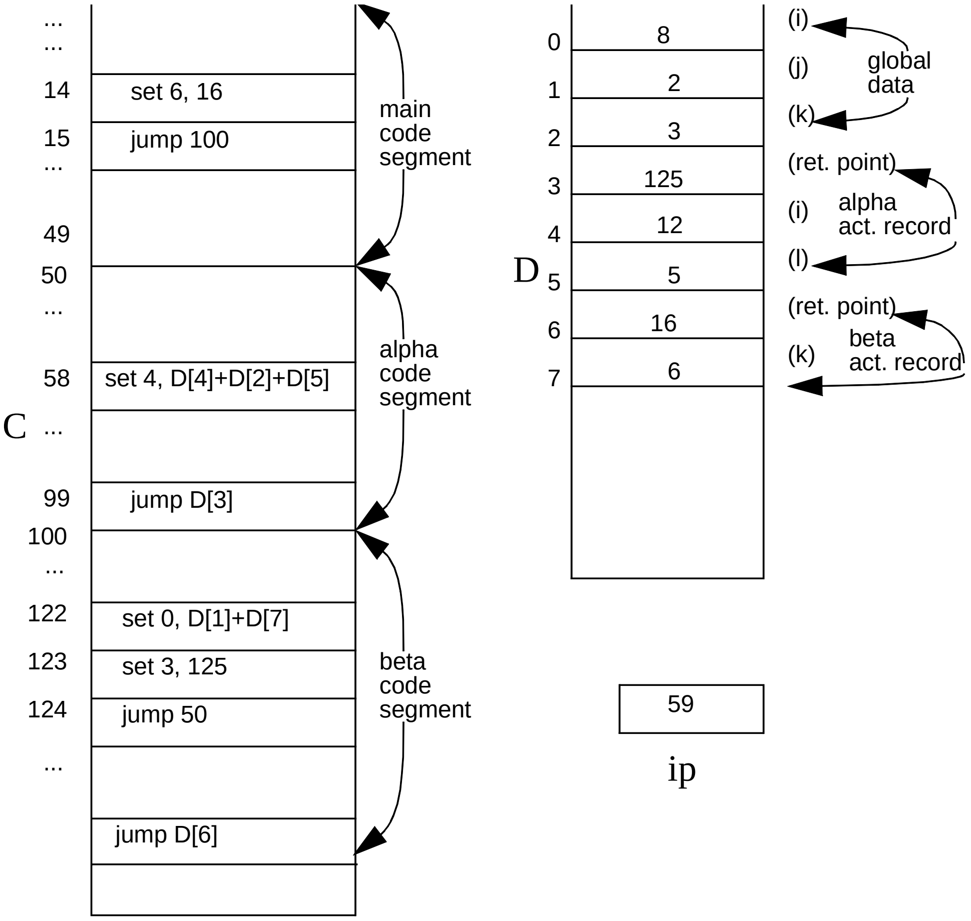


FIGURE 15. State of the SIMPLESEM executing the program of Figure 14

So far, we implicitly assumed that in C2 the main program and its routines are compiled in one monolithic step. It may be convenient instead to allow the various units to be compiled independently. This is illustrated by a variant of C2 (called C2') which allows program units to be put into separate files, and each file to be separately compiled in an arbitrary order. The file which contains the main program may also contain global data declarations, which may then be imported by other separately compiled units, which consist of single routines. If any of such routines needs to access some globally defined data, it must define them as external. Figure 16 shows the same example of Figure

14, using separate compilation.

file 1

```
int i = 1, j = 2, k = 3;
extern beta ( );
main ( )
{
    . . .
    beta ( );
    . . .
}
```

file 2

```
extern int k;
alpha ( )
{
    . . .
}
```

file 3

```
extern int i, j;
extern alpha ( );
beta ( )
{
  . . .
   alpha ( );
  . . .
}
```

FIGURE 16. Program layout for separate compilation

As in the case of C2, a SIMPLESEM implementation can reserve the first location of each activation record (except for main) for the pointer to the caller's instruction, to be executed upon return. Further consecutive locations are then reserved for local variables, which can be bound to their offset within the activation record, as each routine is independently compiled. Independent compilation, however, does not allow variables to be bound to their absolute addresses. Because of independent compilation, imported global variables cannot even be bound to their offsets in the global activation record. Similarly, routine calls cannot be bound to the starting address of the corresponding code segments.

To resolve such unresolved addresses, a *linker* is used to combine the independently translated modules into a single executable module. The linker assigns the various code segments and activation records to stores C and D and fills any missing information that the compiler was unable to evaluate.

From this discussion we see that C2 and C2' do not differ semantically. Indeed, once a linker collects all separately compiled components, C2' programs and C2 programs cannot be distinguished. Their difference is in terms of the user-support they provide for the development of large programs. C2' allows parallel development by several programmers, who might work at the same time on different units.

Independent compilation, as offered by C2', is a simplified version of the facility offered by several existing programming languages, such as FORTRAN and C.

2.6.3 C3: Supporting recursive functions

Let us add two new features to C2: the ability of routines to call themselves (*direct recursion*) or to call one another in a recursive fashion (*indirect recursion*), and the ability of routines to return values, i.e., to behave as functions. These extensions define a new language, C3, which is illustrated in Figure 17 through an example.

```
int n;
int fact ( )
{
    int loc;
    if (n > 1) {
        loc = n--;
        return loc * fact ( );
    }
    else
        return 1;
}
main ( )
{
    get (n);
    if (n >= 0)
        print (fact ( ));
    else
        print ("input error");
}
```

FIGURE 17. A C3 example

As we mentioned in Section 2.4, in order to support mutual recursion between two routines–say, A and B–the program must be written according to the following pattern:

```
A’s declaration (i.e., A’s header);
B’s definition (i.e., B’s header and body);
A’s definition;
```

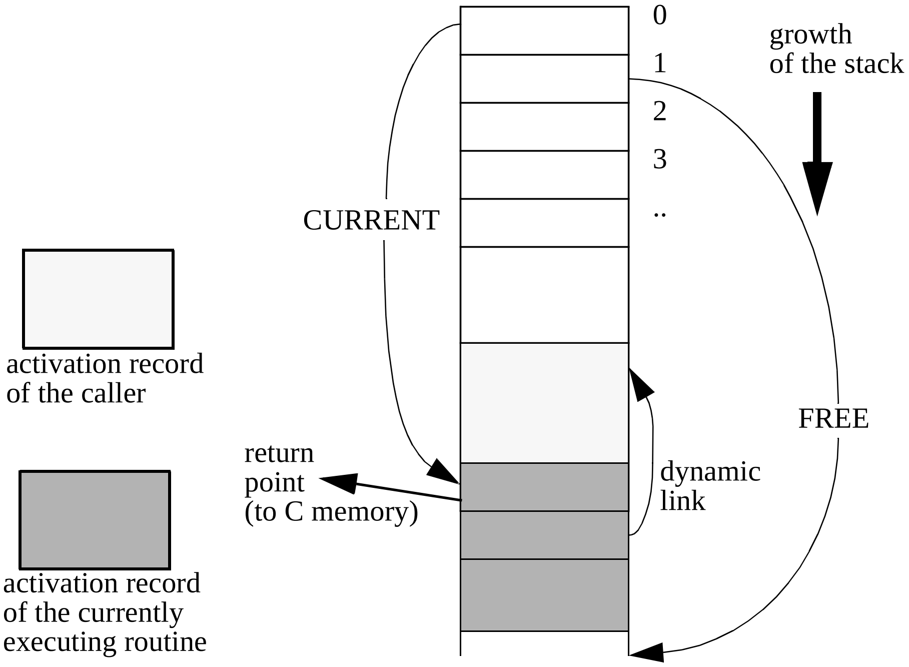
Let us first analyze the effect of the introduction of recursion. Although each unit's activation record has a known and fixed size, in C3 it is not known how many instances of any unit will be needed during execution. As an example, for the program shown in Figure 17, at a given point of execution two activations are generated for function fact if the read value of n is greater than or equal to two. All different activations have the same code segment, since the code does not change from one activation to another, but they need different activation records, storing the different values of the local environment. As for C2, the compiler can bind each variable to its offset in the corresponding activation record. However, as opposed to C2, it is not possible to perform the further binding step which transforms it into an absolute address of the D store until execution time. In fact, an activation record is allocated by the invoking function for each new invocation, and each new allocation establishes a new binding with the corresponding code segment to form a new activation of the invoked function. Consequently, the final binding step which adds the offset of a variable–known statically–to the starting address (often called *base address*) of the activation record–known dynamically–can only be performed at execution time. To make this possible, we will use the cell at address zero in D to store the base address of the activation record of the currently executing unit (we also call this value CURRENT).

When the current instance of a unit terminates, its activation record is no longer needed. In fact, no other units can access its local environment, and the semantic rule of function invocation requires a new activation record to be freshly allocated. Therefore, after a function completes its current instance, it is possible to free the space occupied by the activation record and make it available to store new activation records in the future. For example, if A calls B which then calls C, the activation records for functions are allocated in the order A, B, C. When C returns to B, C's activation record can be discarded; B's activation record is discarded next, when B returns to A. Because the activation record that is freed is the one that was most recently allocated, activation records can be allocated with a *last-in/first-out* policy on a stack-organized storage.

In order to make return from an activation possible, the following necessary information is stored in the activation record: address of the instruction to be executed *(return point*) and base address of activation record to become active upon return. In the case of C2, only the return point needed to be saved, because through it the (unique) activation record associated with the callee also becomes known. If more than one activation may exist for a given unit, this more general solution becomes necessary. Therefore we assume that the cell at offset 0 of activation records contains the return point, while the cell at offset 1 contains a pointer to the base address of the caller's activation record–this pointer is called the *dynamic link*. The chain of dynamic links originating in the currently active activation record is called the *dynamic chain*. At any time, the dynamic chain represents the dynamic sequence of unit activations.

In order to manage SIMPLESEM's D store as a stack, it is necessary to know, at run time, the address of the first free cell of D, since a new activation record is allocated from that point on. We will use D's cell at address 1 to keep this information (we call this value FREE). Finally, it is necessary to provide memory space for the value returned by the routine, if it behaves as a function. Since the routine's activation record is deallocated upon return, the returned value must be saved into the caller's activation record. That is, when a functional routine is called, the caller's activation record is extended to provide space for the return value, and the callee writes the returned value into that space (using a negative offset, since the location is in the caller's activation record) before returning[1].

Figure 18 provides an intuitive view of SIMPLESEM's D store. Activation records are allocated one on top of the previous, and the allocated memory grows from the upper part of the store (corresponding to low addresses) downwards.



[1] For further details concerning the management of return values, see Exercise 21.

FIGURE 18. Structure of the SIMPLESEM D memory implementing a stack

Since recursive routines are the main additional features of C3, we now show how the semantics of routine call and return are specified in terms of SIMPLESEM instructions.

Routine call

```
set 1, D[1] + 1             assume one cell is sufficient to hold
                            the returned value
set D[1], ip + 4            set the value of the return point in
                            the callee's
                            activation record
set D[1] + 1, D[0]          set the dynamic link of the callee's
                            activation record to point to the
                            caller's activation record
set 0, D[1]                 set CURRENT, the address of the
                            currently executed activation record
set 1, D[1] + AR            set FREE (AR is the size of the
                            callee's activation record)
jump start_addr             start_addr is an address of C where
                            the first instruction of the callee's
                            code is stored.
```

Return from routine

```
set 1, D[0]                 set FREE
set 0, D[D[0] +1]           set CURRENT
jump D[D[1]]                jump to the stored return point
```

We assume that before the execution of a C3 program starts, ip is set to point to the first instruction of main ( ) and the D memory is initialized to contain space for the global data and for the activation record of main ( ). Such activation record contains space just for local main's variables (if any); space for the address of the return instruction and for the dynamic link are not needed, since the main routine does not return to a caller. Its termination simply means that the execution terminates. The values stored in D[0] and D[1] are also assumed to be initialized before execution. D[0] is set to the address of the first location of main's activation record and D[1] must be set to the address of the first free location after main's activation record.

As an exercise, let us show how the program of Figure 17 is executed by the SIMPLESEM machine. The code stored in the C memory is the following:

```
 0  set 2, read              reads the value of n; 2 is the absolute address where
                             global variable n is stored
 1  jumpt 10, D[2] < 0       tests the value of n
 2  set 1, D[1] + 1          call to fact starts here; space for the result saved
 3  set D[1], ip + 4
 4  set D[1] + 1, D[0]
 5  set 0, D[1]
 6  set 1, D[1] + 3          3 is the size of fact’s activation record
 7  jump 12                  12 is the starting address of fact’s code
 8  set write, D[D[1] - 1]   D[1] - 1 is the address where the result of the call to fact
                             is stored
 9  jump 13                  end of the call
10  set write, “input error”
11  halt                     this is the end of the code of main
12  jumpt 23, D[2] ð 1       tests the value of n
13  set D[0] + 2, D[2]       assigns n to loc
14  set 2, D[2] - 1          decrements n
15  set 1, D[1] + 1          call to fact starts here; space for the result saved
16  set D[1], ip + 4
17  set D[1] + 1, D[0]
18  set 0, D[1]
19  set 1, D[1] + 3          3 is the size of fact’s activation record
20  jump 12                  12 is the starting address of fact’s code
21  set D[0] - 1, D[D[0] + 2] * D[D[1] - 1]     the returned value is stored in the
                             caller’s activation record
22  jump 24
23  set D[0] - 1, 1          return 1
24  set 1, D[0]              this and the next 2 instructions correspond to the return
                              from the routine
25  set 0, D[D[0] + 1]
26  jump D [D[1]]
```

Figure 19 provides two snapshots of the D memory: immediately after the first call to fact (case (a)) and at the return point from the third activation of fact when the initially read input value is 3 (case (b)). The reader is urged to try the example on paper, going through all intermediate steps of execution.

Note that the stack-based abstract implementation scheme discussed in this section also can be used for implementing C2. We discussed C2 in terms of static memory allocation, but this was simply an implementation choice. The advantage of a stack-based implementation would be that only the minimum amount of data store is allocated at any given time. The disadvantage, of course, is that a more complicated memory management scheme is needed.

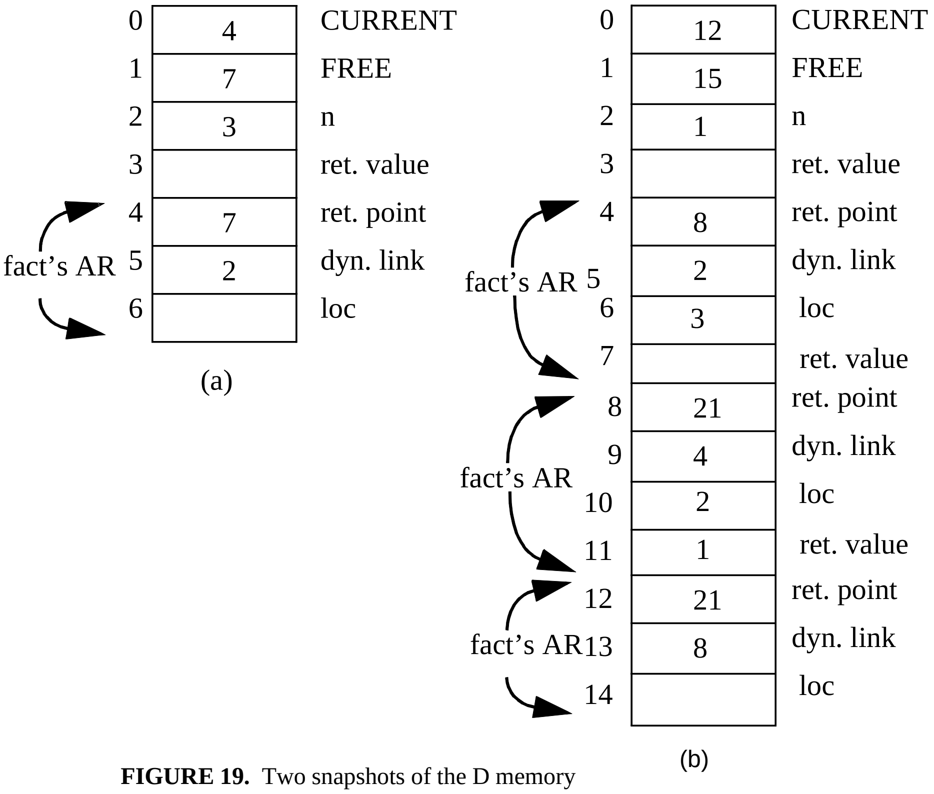


FIGURE 19. Two snapshots of the D memory

2.6.4 C4: Supporting block structure

The structuring facilities offered by C3 allow programs to be defined as a sequence of global declarations of data and routines. Routines may call themselves in a recursive fashion. In this section we discuss a new extension to our language family, which collectively define a new family C4. The family C4 contains two members: C4' and C4". C4' allows local declarations to appear within any compound statement. C4" supports the ability to nest a routine definition within another. Conventionally, the new features offered by C4' and C4" are collectively called *block structure*. Block structure is used to control the scope of variables, to define their lifetime, and to divide the program into smaller units. Any two blocks in the program may be either disjoint (i.e., they have no portion in common) or nested (i.e., one block completely encloses the other).

2.6.4.1 Nesting via compound statements

In C4', blocks have the following form of compound statement, which can appear wherever a statement can appear:

```
{<declaration_list>; <statement_list>}
```

It is easy to realize that such compound statements follow the aforementioned rule of blocks: they are either disjoint or they are nested. A compound statement defines the scope of its locally declared variables: such variables are visible within the compound, including any compound statement nested in it, provided the same name is not redeclared. An inner declaration masks an external declaration for the same name. Figure 20 shows an example of a C4' function having nested compound statements. Function f has local declarations for x, y, and w, whose scope extends from //1 to the entire function body, with the following exceptions:

- x is redeclared in //2. From that declaration until the end of the while statement the outer x is not visible;
- y is redeclared in //3. From that declaration until the end of the while statement, the outer y is not visible;
- w is redeclared in //4. From that declaration until the end of the if statement, the outer declaration is not visible.

Similarly, //2 declares variables x and z, whose visibility extends from the declaration until the end of the statement, with one exception. Since x is redeclared in //4, the outer x is masked by the inner x, which extends from the declaration until the end of the if statement.

```
int f( );
{                                   //block 1
     int x, y,w;                    //1
     while (...)
     {                              //block 2
          int x, z;                 //2
          . . .
          while (. . .)
          {                         //block 3
               int y;               //3
               . . .
          }                         //end block 3
          if (. . .)
          {                         //block 4
               int x, w;            //4
               . . .
          }                         //end block 3
     }                              //end block 2
     if (. . .)
     {                              //block 5
          int a, b, c, d;           //5
          . . .
     }                              //end block 5
}                                   //end block 1
```

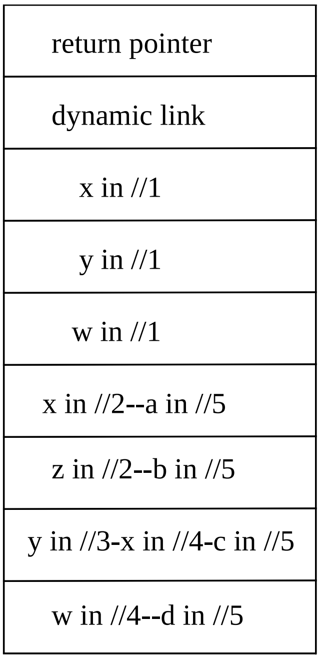
FIGURE 20. An example of nested blocks in C4'

A compound statement also defines the lifetime of locally declared data. Memory space is bound to a variable x as the block in which it is declared is entered during execution.The binding is removed when the block is exited.

In order to provide an abstract implementation of compound statements for the SIMPLESEM machine, there are two options. One consists of statically defining an activation record for a routine with nested compound statements; another consists of dynamically allocating new memory space corresponding to local data as each compound statement is entered during execution. The former scheme is simpler and more time efficient, while the latter can lead to a more space efficient implementation. We will discuss the former scheme, and leave the latter to the reader as an exercise, which can easily be solved after reading Section 2.6.4.2 (see Exercise 27).

Let us refer to the example of Figure 20. Note that the while block that declares variables x and z and the if block that declares a, b, c, and d are disjoint; similarly, the while block that declares variable y and the if block that declares x and z are disjoint. Since two disjoint blocks cannot be active at the same time, it is possible to use the same memory cells to store their local values. Thus, the activation record of function f can be defined as shown in Figure 21. The figure shows that the same cells may be used to store a and x, b and w, c and w, etc.; i.e., operator “--” denotes an overlay. The definition of overlays can be done at translation time. Having done so, the run-time behavior of C4' is exactly the same as was discussed in the case of C3.

.



return pointer

dynamic link

x in //1

y in //1

w in //1

x in //2--a in //5

z in //2--b in //5

y in //3-x in //4-c in //5

w in //4--d in //5

FIGURE 21. An activation record with overlays

A block structure can be described by *static nesting tree* (SNT), which shows how blocks are nested into one another. Each node of a SNT describes a block; descendants of a node N which represents a certain block denote the blocks that are immediately nested within the block. For example, the program of Figure 20 is described by the static nesting tree of Figure 22.

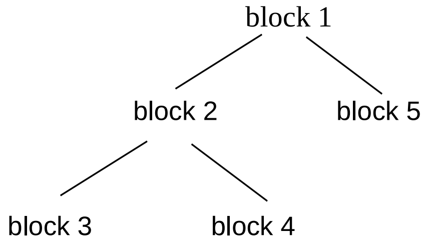


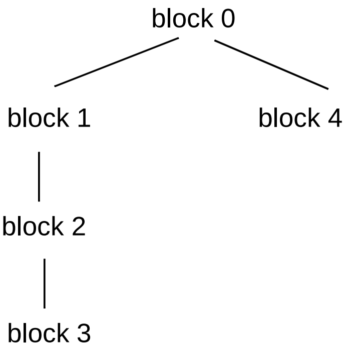
FIGURE 22. Static nesting tree for the block structure of Figure 20

2.6.4.2 Nesting via locally declared routines

As we mentioned, block structure may result from the ability to nest compound statements within unnested routines, to nest routine definitions within routines, or both. C and C++ only support the nesting of compound statements within routines. Pascal and Modula-2 allow routine nesting, but do not support nesting of compound statements. Ada allows both.

```
int x, y, z;
f1 ( )
{                      //block 1
    int t, u;  block 1 // 1                block4
    f2 ( )
    {                  //block 2
        int x, w;      // 2
        f3 ( )
        {              //block 3
            int y, w, t; // 3
            . . .
        }              //end block 3
        x = y + t + w  + z;
    }                  //end block 2
. . .
}                      //end block 1
main ( );
{                      //block 4
    int z, t;
    . . .
}                      //end block 4
```

(a)



Block 0 is introduced to represent the outermost level of the program

(b)

FIGURE 23. A C4" example (a) and its static nesting tree (b)

Let us examine how routine nesting might be incorporated into our language. The resulting variation will be called C4". As shown in Figure 23 (a), in C4" a routine may be declared within another routine. Routine f3 can only be called within f2 (e.g., it would not be visible within f1 and main). A call to f3 within f2's body would be a local call (i.e., a call to a locally declared routine). Since f3 is internal to f2, f3 can also be called within f3's body (direct recursion). Such a call would be a call to a nonlocally declared routine, since f3 is declared in the outer routine f2. Similarly, f2 can be called within f1's body (local call) and both within f2's and f3's bodies (nonlocal calls). Moreover, the data declared in //1are visible from that point until the end of f1 (i.e., //6), with one exception. If a declaration for the same name appears in internally declared routines (i.e., in //2 or in//3), the internal declarations mask the outer declaration. Also, within a routine, it is possible to access both the local variables, and nonlocal variables declared by enclosing outer routines, if they are not masked. In the example, within f3's body, it is possible to access the nonlocal variables x (declared in //2), u (declared in //1) and the global variable z.

As shown in Figure 23 (b), the concept of a static nesting tree can be defined in this case too. Block 0 is introduced to represent the outermost level of the program, which contains the declarations of variables x, y, z, and functions f1 ( ) and main ( ).

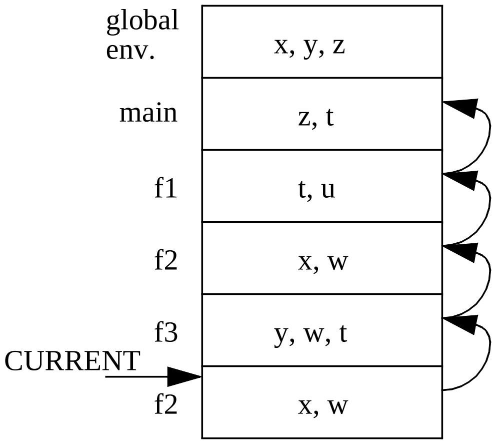


FIGURE 24. A sketch of the run-time stack (dynamic links are shown as arrowed lines)

Let us examine the effect of the following sequence of calls: main calls f1, f1 calls f2, f2 calls f3, f3 calls f2. Figure 24 shows a portion of the activation record stack corresponding to the example. The description is highly simplified, for readability purposes, but shows all the relevant information. For each activation record, we indicate the name of the corresponding routine, the dynamic link, and the names of variables whose values are kept in it. Let us suppose that the execution on the SIMPLESEM machine reaches the assignment x = y + t + w + z in f2. The translation process we discussed so far is able to bind variables x and w to offsets 2 and 3 of the topmost activation record (whose initial address is given by D[0], i.e., CURRENT); but what about variables y, t, and z? For sure, they should not be bound according to the most recently established binding for such variable names, since such binding were established by the latest activations of routines f3 (y and t) and main (z). However, the scope rules of C4" require variables y and z referenced within f2 to be the ones declared globally, and variable t to be the one declared locally in f1. In other words, the sequence of activation records stored in the stack represent the sequence of unit instances, as they are dynamically generated at execution time. But what determines the nonlocal environment are the scope rules of the language, which depend on the static nesting of routine declarations.

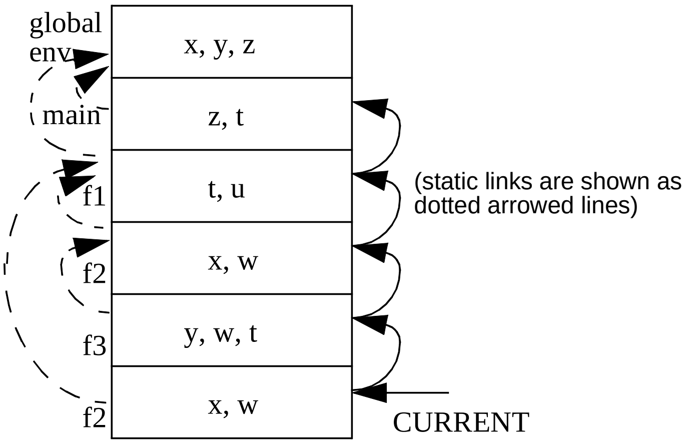


FIGURE 25. The run-time stack of Figure 24 with static links

One way to make access to nonlocal variables possible is for each activation record to contain a pointer (*static link*) up the stack to the activation record of the unit that statically encloses it in the program text. We will use the location of the activation record at offset 2 to store the value of the static link. Figure 25 shows the static links for the example of Figure 24. The sequence of static links that can be followed from the active activation record is called the *static chain*. Referencing nonlocal variables can be explained intuitively as a search that traverses the static chain. To find the correct binding between a variable and a stack location, the static chain is searched until a binding is found. In our example, the reference to t is bound to a stack location within f1's activation record, whereas references to y and z are bound to a stack location within the global environment–as indeed it should be. Notice that in this scheme, the global environment is accessed in the same uniform way as any other nonlocal environment. In such a case, the value of the static link for main's activation record is assumed to be set automatically before execution. In order to use the cell at offset 2 of main's activation record to hold the value of the static link, as we do for any other activation record, the cells at offsets 0 and 1 are kept unused. Alternatively, access to the global environment can be treated as a special case, by using absolute addresses.

In practice, searching along the static chain, which would entail considerable run-time overhead, is never necessary. A more efficient solution is based on the fact that the activation record containing a variable named in a unit U is always a fixed distance from U's activation record along the static chain. If the variable is local, the distance is obviously zero; if it is a variable declared in the immediately enclosing unit, the distance is one; if it is a variable declared in the next enclosing unit, the distance is 2, and so on. In general, for each reference to a variable, we can evaluate a *distance attribute* between that reference and the corresponding declaration. This distance attribute can be evaluated and bound to the variable at translation time. Consequently, each reference may be statically bound to a pair (distance, offset) within the activation record.

Based on the pair (distance, offset), it is possible to define the following addressing scheme for SIMPLESEM. If d is the value of the distance, starting from the address of the current activation record (CURRENT, the value stored in D[0]), we traverse d steps along the static chain. The value of the offset is then added to the address so found, and the result is the actual run-time address to the nonlocal data object. We can define this formally in terms of a recursive function fp (d), which can then be easily translated into SIMPLESEM. Function fp (d), which stands for the frame pointer–a pointer to an activation record–that is d static links away from the active activation record, can be defined as:

```
fp(d) = if d=0 then D[0] else D [fp (d-1)+2]
```

For example, fp (0) is simply D[0], i.e., the address of the current (topmost) activation record; and fp(1) is D[D[0]+ 2].

Using fp, access to a variable x, with <distance, offset> pair <d, o>, is provided by the following address:

```
D[fp(d)+ o]
```

The semantics of function call defined in Section 2.6.3 needs to be modified in the case of C4", in order to take into account the installation of static links in activation records. This can be done in the following way. First, notice that, as we did for variables, one can define the concept of distance between a routine call and the corresponding declaration. Thus, if f calls a local routine f1, then the distance between the call and the declaration is 0. If f contains a call to a function declared in the block enclosing f, the distance is 1. This, for example, would be the case if f calls itself recursively. If f is local to function g and f contains a call to a function h declared in the block enclosing g, the distance between the call and the declaration is 2, and so on. Therefore, the static link to install for activation record of the callee, if the callee is declared at distance d, should point to the activation record that is d steps along the static chain originating from the caller's activation record.

In conclusion, the semantics of routine call can be defined by the following SIMPLESEM code:

Routine call

```
set 1, D[1] + 1           set space for the result of the
                          function call (assume 1 cell needed)
set D[1], ip + 5          set the value of the return point in
                          the callee’s activation record
set D[1] + 1, D[0]        set the dynamic link of the callee’s
                          activation record to point to the
                          caller’s activation record
set D[1] + 2, fp(d)       set the static link of the callee’s
                          activation record
set 0, D[1]               set CURRENT, the address of the
                          currently executed
                          activation record
```

```
set 1, D[1] + AR          set FREE (AR is the size of the
                          callee’s activation record)
jump start_addr           start_addr is an address of memory C
                          where the first instruction of the
                          callee’s code is stored.
```

2.6.5 C5: Towards more dynamic behaviors

So far we assumed that the data storage requirements of each unit are known at compile time, so that the required amount of memory can be reserved when the unit is allocated. Furthermore, the mapping of variables to storage within the activation record can be performed at compile time; i.e., each variable is bound to its offset statically. In this section we discuss language features that invalidate this assumption, and we show how to define semantics of such features.

2.6.5.1 Activation records whose size becomes known at unit activation

Let us first introduce language C5', by relaxing the assumption that the size of all variables is known at compile time. Such is the case for dynamic arrays, that is, arrays whose bounds become known at execution time, when the unit (routine or compound statement) in which the array is declared is activated.

For example, in the Ada programming language, it is possible to define the following type:

```
type VECTOR is array (INTEGER range <>); --defines arrays with unconstrained index
```

and declare the following variables:

```
A: VECTOR (1. .N);
B: VECTOR (1. .M); --N and M must be bound to some integer value when these two dec-
larations are processed at execution time
```

The abstract implementation that defines the semantics for this case is rather straightforward. At translation time, storage can be reserved in the activation record for the descriptors of the dynamic arrays. The descriptor includes one cell in which we store a pointer to the storage area for the dynamic array and one cell for each of the lower and upper bounds of each array dimension. As the number of dimensions of the array is known at translation time, the size of the descriptor is known statically. All accesses to a dynamic array are translated as indirect references through the pointer in the descriptor, whose offset is determined statically.

At run time, the activation record is allocated in several stages.

1. The storage required for data whose size is known statically and for descriptors of dynamic arrays are allocated.
2. When the declaration of a dynamic array is encountered, the dimension entries in the descriptors are entered, the actual size of the array is evaluated, and the activation record is extended (that is, FREE is increased) to include space for the variable. (This expansion is possible because, being the active unit, the activation record is on top of the stack.)
3. The pointer in the descriptor is set to point to the area just allocated.

In the previous example, let us suppose that the descriptor allocated when variable A is declared is at offset m. The cell at offset m will point at run time to the starting address of A; the cells at offsets m+1 and m+2 will contain the lower and upper bounds, respectively, of A's index. The run-time actions corresponding to entry into the unit where A's declaration appears will update the value of D[1] (i.e., FREE) to allocate space for A, based on the known value of N, and will set the values of the descriptor at offsets m, m+1, and m+2.

Any access to elements of A are translated to indirect references. Assuming each integer occupies one location of D and supposing that I is a local variable stored at offset s, instruction A [I] = 0 would be translated into SIMPLESEM as:

$$\text{set } [\underbrace{D[D[0] + m]}_{\text{denotes the base address of A}} + \underbrace{D[D[0] + s]}_{\text{denotes the value of I}}],\ 0$$

2.6.5.2 Fully dynamic data allocation

Now let us consider another language variation, called C5", in which data can be allocated explicitly, through an executable allocation instruction. In most existing languages, this is achieved by defining pointers to data, and by providing statements that allocate such data in a fully dynamic fashion.

For example, in C++ we can define the following type for nodes of a binary tree:

```
struct node {
    int info;
    node* left;
    node* right:
};
```

The following instruction, which may appear in some code fragment:

```
node* n = new node;
```

explicitly allocates a structure with the three fields info, left, and right, and makes it accessible via the pointer n.

According to this allocation scheme, data are allocated explicitly as they are needed. We cannot allocate such data on a stack, as do automatically allocated data. For example, suppose that a function append_left is called to generate a new node and make its accessible through field left of node pointed by n. Also, suppose that n is visible by append_left as a nonlocal variable. If the node allocated by append_left would be allocated on the stack, it would be lost when append_left returns. The semantics of these dynamically allocated data, instead, is that their lifetime does not depend on the unit in which their allocation statement appears, but lasts as long as they are accessible, i.e.,they are referred to by some existing pointer variables, either directly or indirectly.

An abstract implementation of this concept using SIMPLESEM can be very simple, and consists of allocating dynamic data in D starting from the high-address end. This area of D is also called the *heap*. New data are allocated in the heap as the allocation instructions are executed, and we can assume that the size of the SIMPLESEM D store is sufficient to hold all data that are dynamically allocated via the new instruction. Figure 26 gives an overall view of how memory D is handled, in order to support both a stack and a heap. We will return to the practical issue of actually implementing dynamic allocation in a memory-efficient way in Chapter 3. From a semantic viewpoint, this simple implementation scheme can be sufficient.

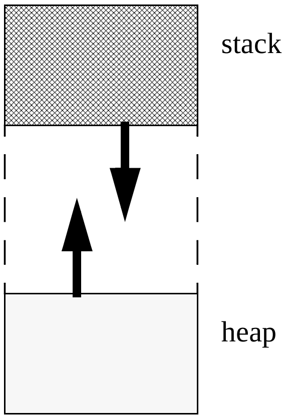


FIGURE 26. Management of the D memory

Sidebar start The Structure of Dynamic Languages

The term “dynamic languages” implies many things. In general, it refers to those languages that adopt dynamic rather than static rules. For example, APL, SNOBOL4 and several LISP variants use dynamic typing and dynamic scope rules. In principle, of course, a language designer can make these choices independently of one another. For example, one can have dynamic type rules but static scope rules. In practice, however, dynamic properties are often adopted together.[1]

In this sidebar, we will examine how the adoption of dynamic rules changes the semantics of the language in terms of run-time requirements. In general, a dynamic property implies that the corresponding bindings are carried out at run time and cannot be done at translation time. We will examine dynamic typing and dynamic scoping.

In a language that uses *dynamic typing*, the type of a variable and therefore the methods of access and the allowable operations cannot be determined at translation time. In Section 2.6.5, we saw that we need to keep a run-time descriptor for dynamic arrays variables, because we cannot determine the size of or starting address of such variables at translation time. In that case, the

[1] We already mentioned that there are dynamically typed languages (like ML and Eiffel) that support static type checking.

descriptor has to contain the information that cannot be computed at translation time, namely, the starting address and the array bounds. It was possible to keep the descriptor in the activation record because the size of the descriptor was fixed and known at translation time. In the case of dynamically typed variables, we also need to maintain the type of the variable in the descriptor. If the type of a variable may change at run time, then the size and contents of its descriptor may also change. For example, if a variable changes from a two-dimensional array to a three-dimensional array, then the descriptor needs to grow to contain the values of the bounds for the new dimension. This is in contrast to the descriptors of dynamic arrays whose contents were fixed at unit activation time. Every access to a dynamic variable must be preceded by a run-time check on the type of the variable, followed by appropriate address computation, depending on the current type of the variable.

What is maintained for each variable in the activation record for a unit? Since not only the variable's size may change during program execution, but so may the size of its descriptor, descriptors must be kept in the heap. For each variable, we maintain a pointer in the activation record that points to the variable's descriptor in the heap which, in turn, may contain a pointer to the object itself in the heap.

In order to discuss the effect of dynamic scope rules, let us consider the example program of Figure 27. The program is written using a C-like syntax, but it will be interpreted according to an APL-like semantics, i.e., according to dynamic scope rules.

```
sub2 ( )
{
     declare x;
     . . .
     . . . x  . . .;
     . . . y . . .;
     . . .
}
sub1 ( )
{
     declare y;
     . . .
     . . . x  . . .;
     . . . y . . .;
     sub2 ( );
     . . .
}
main ( )
{
     declare  x, y, z;
     z = 0;
     x = 5;
     y = 7;
     sub1;
     sub2;
     . . .
}
```

FIGURE 27. An example of a dynamically scoped language

A program consists of a number of routines and a main program. Each routine declares its local variables (y in the case of sub1, x in the case of sub2, x, y, z in the case of main). Any access to a variable that is not locally declared is implicitly assumed to be an access to a nonlocal variable. A variable declaration does not specify the variable's type: it simply introduces a new name. Routine names are considered as global identifiers.

Since scope rules are dynamic, the scope of a name is totally dependent on the run-time call chain (i.e., on the dynamic chain), rather than the static structure of the program. In the example shown in Figure 27 consider the point when the call to sub1 is issued in main. The nonlocal references to x and z within the activation of sub1 are bound to the global x and z defined by main. When function sub2 is activated from sub1, the nonlocal reference to y is bound to the most recent definition of y, that is, to the data object associated to y in sub1's activation record. Return from routines sub2 and then sub1 causes deallocation of the corresponding activation records and then execution of the call to sub2 from main. In this new activation, the nonlocal reference to y from sub2, which was previously bound to y in sub1, is now bound to the global y defined in main.

An abstract implementation mechanism to reference nonlocal data can be quite simple. Activation records can be allocated on a stack and joined together by dynamic links, as we saw in the case of conventional languages. Each entry of the activation record explicitly records the name of the variable and contains a pointer to a heap area, where the value can be stored. Allocation on a heap is necessary because the amount of storage required by each variable can vary dynamically. For each variable–say, V–the stack is searched by following the dynamic chain. The first association found for V in an activation record is the proper one. Figure 28 illustrates the stack for the program of Figure $20 when sub2 is called by sub1, which is, in turn, called by main.

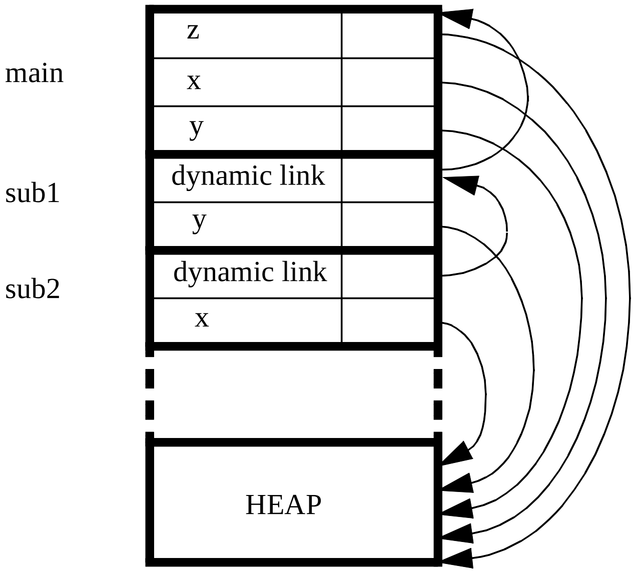


FIGURE 28. A view of the run-time memory for the program of Figure 27

Although simple, this accessing mechanism is inefficient. Another approach is to maintain a table of currently active nonlocal references. Instead of searching along the dynamic chain, a single lookup in this table is sufficient. We will not discuss this solution any further but the reader should note that this technique speeds up referencing nonlocal variables at the expense of more elaborate actions to be executed at subprogram entry and exit. These additional actions are necessary to update the table of active nonlocal references.

Sidebar end

2.6.6 Parameter passing

So far we assumed that routines do not have parameters: we only assumed that they can return a value (see Section 2.6.3). We will now remove this limitation by discussing how parameter passing may be abstractly implemented on SIMPLESEM. We first address the issue of data parameters, and then we will analyze routine parameters.

2.6.6.1 Data parameters

There are different conventions for passing data parameters to routines. The adopted convention is either predefined by the language, and therefore it is part of the language semantics, or can be chosen by the programmer from several options. In either case, it is important to know which convention is adopted, because the choice made affects the meaning of programs. The same program may in fact produce different results under different data parameter passing conventions. Three conventions for data parameters are discussed below: call by reference, call by copy, and call by name. Each of them is first introduced informally, and then defined precisely in terms of SIMPLESEM actions.

Call by Reference (or by Sharing)

The calling unit passes to the called unit the address of the actual parameter (which is in the calling unit's referencing environment). A reference to the corresponding formal parameter in the called unit is treated as a reference to the location whose address is so passed. The effect of call by reference is intuitively described in Figure 29. If the formal parameter is assigned a value, the corresponding actual parameter changes value. Thus, a variable that is transmitted as an actual parameter is shared, that is, directly modifiable by the subprogram. If an actual parameter is anything other than a variable, for example, an expression or a constant, the subprogram receives the address within the calling unit's activation record of a temporary location that contains the value of the actual parameter. Some languages treat this situation as an error.

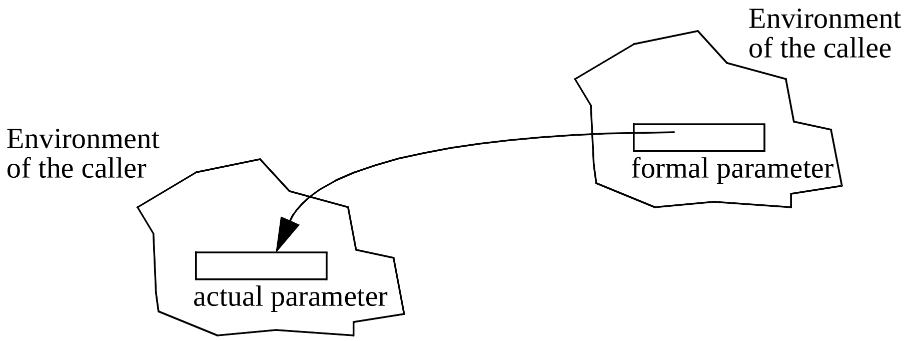


FIGURE 29.A view of call by reference

Suppose that call by reference is being added to C4. In order to define a SIMPLESEM implementation that specifies semantics precisely, we need to extend the actions described in Section 2.6.3. The callee's activation record must contain one cell for each parameter. At procedure call the caller must initialize the contents of the cell to contain the address of the corresponding actual parameter. If the parameter cell is at offset off and the actual parameter, which is bound to the pair (d, o), is not itself a by-reference parameter, the following action must be added for each parameter:

```
set D[0] + off, fp(d) + o
```

If the actual parameter itself is a by-reference parameter, the SIMPLESEM action should be:

```
set D[0] + off, D[ fp (d) + o]
```

When the routine body is executed, parameter reference is performed via indirect addressing. Thus, if x is a formal parameter and off is its offset, instruction

```
 x = 0;
```

is translated as

```
set D[D[0] + off], 0
```

Call by copy

In call by copy–unlike in call by reference–formal parameters do not share storage with actual parameters; rather, they act as local variables. Thus, call by copy protects the calling unit from intentional or inadvertent modifications of actual parameters. It is possible further to classify call by copy into three modes, according to the way local variables corresponding to formal parameters are initialized and the way their values ultimately affect the actual parameters. These three modes are call by value, by result, and by value-result.

In *call by value*, the calling unit evaluates the actual parameters, and these values are used to initialize the corresponding formal parameters, which act as local variables in the called unit. Call by value does not allow any flow of information back to the caller, since assignments to formal parameters (if permitted) do not affect the calling unit.

In *call by result*, local variables corresponding to formal parameters are not set at subprogram call, but their value, at termination, is copied back into the corresponding actual parameter's location within the environment of the caller. Call by result does not permit any flow of information from the caller to the callee.

In *call by value-result*, local variables denoting formal parameters are both initialized at subprogram call (as in call by value) and delivered upon termination (as in call by result). Information thus flows from the caller to the callee (at the point of call) and from the callee to the caller (at the return point).

A description of the semantics of call by value in terms of a SIPLESEM implementation is trivial. The callee's activation record must contain space for by-value parameters, as normal local data. The difference here is that the call must provide for initialization of such data. We leave this and the other two cases of call by copy as exercises for the reader.

One might wonder whether call by reference and call by value-result are equivalent. If this were the case, implementing parameter passing in one mode would be equivalent to implementing it in the other. It can be shown, however, that call by reference may produce a different result from call by value-result in the following cases:

- two formal parameters become aliases (i.e., the two different names denote the same object);

- a formal parameter and a nonlocal variable which is visible both by the caller and by the callee become aliases.

We will provide two examples to motivate these statements. The first case may happen if–say– a [i] and a [j] are two integer actual parameters corresponding to the formal parameters x and y, and i happens to be equal to j at the point of call. In such a case, the effect of call by reference is that x and y would be aliased, since they would refer to the same array element. If the routine contains the following statements:

```
x = 0;
y ++;
```

the result of the call is that the array element of index i (and j) is set to 1. In the case of call by value-result, let a [i] be 10 at the point of call. The call would initialize x and y to the 10. Then x becomes 0 and y becomes 11, due to the above assignment statements. Finally, upon return, first 0 is copied back into a [i] and then 11 is copied back into the same cell, if copies are performed in this order. As a result, the array element is set to 11.

As an example of the second case, suppose that a routine is called with one integer actual parameter a which corresponds to the formal parameter x. Let a be visible by the routine as a nonlocal variable. Suppose that the routine contains the following statements:

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

In the case of call by reference, the effect of the call is that a is set to 2. In the case of call by value-result, if a's value is 10 at the call point, the value becomes 11 upon return.

Call by name

As in call by reference, a formal parameter, rather than being a local variable of the subprogram, denotes a location in the environment of the caller. Unlike with call by reference, however, the formal parameter is not bound to a location at the point of call; it is bound to a (possibly different) location each time it is used within the subprogram. Consequently, each assignment to a formal parameter can refer to a different location.

Basically, in call by name each occurrence of the formal parameter is replaced textually by the actual parameter. This may be achieved by a simple kind of macro processing, with one exception that will be discussed below.

This apparently simple rule can lead to unsuspected complications. For example, the following procedure, which is intended to interchange the values of a and b (a and b are by-name parameters)

```
swap (int a, b);
int temp;
{
    temp = a;
    a = b;
    b = temp;
};
```

most likely produces an unexpected result when invoked by the call

```
swap (i, a [i])
```

The replacement rule specifies that the statements to be executed are

```
temp = i;
i = a [i];
a [i] = temp;
```

If i = 3 and a [3] = 4 before the call, i = 4 and a [4] = 3 after the call (a [3] is unaffected)!

Another trap is that the actual parameter that is (conceptually) substituted into the text of the called unit belongs to the referencing environment of the caller, not to that of the callee. For example, suppose that procedure swap also counts the number of times it is called and it is embedded in the following fragment.

```
int c;
. . .
swap (int a, b);
int temp;
{
    temp = a; a = b;
    b = temp; c ++;
}

y ( );
int c, d;
{
    swap (c, d);
};
```

When swap is called by y, the replacement rule specifies that the statements to be executed are

```
temp = c;
```

```
c = d;
d = temp;
c ++;
```

However, the location bound to name c in the last statement belongs to x's activation record, whereas the location bound to the previous occurrences of c belong to y's activation record. This shows that plain macro processing does not provide a correct implementation of call by name if there is a conflict between names of nonlocals in the routine's body and names of locals at the point of call. This example also shows the possible difficulty encountered by the programmer in foreseeing the run-time binding of actual and formal parameters.

Call by name, therefore, can easily lead to programs that are hard to read. It is also unsuspectedly hard to implement. The basic implementation technique consists of replacing each reference to a formal parameter with a call to a routine (traditionally called thunk) that evaluates a reference to the actual parameter in the appropriate environment. One such thunk is created for each actual parameter. The burden of run-time calls to thunks makes call by name costly.

Due to these difficulties, call by name has mostly theoretical and historical interests, but has been abandoned by practical programming languages.

Call by reference is the standard parameter passing mode of FORTRAN. Call by name is standard in ALGOL 60, but, optionally, the programmer can specify call by value. SIMULA 67 provides call by value, call by reference, and call by name. C++, Pascal and Modula-2 allow the programmer to pass parameters either by value (default case) or by reference. C adopts call by value, but allows call by reference to be implemented quite easily via pointers. Ada defines parameter passing based on the intended use, as either in (for input parameters), out (for output parameters), or inout (for input/output parameters), rather than in terms of the implementation mechanism (by reference or by copy). If the mode is not explicitly specified, in is assumed by default. More on this will be discussed in Chapter 4.

2.6.6.2 Routine parameters

Languages supporting variables of type routine are said to treat routines as first-class objects. In particular, they allow routines to be passed as parameters. This facility is useful in some practical situations. For example, a routine S that evaluates an analytic property of a function (e.g., derivative at a given point) can be written without knowledge of the function and can be used for different functions, if the function is described by a routine that is sent to S as a parameter. As another example, if the language does not provide explicit features for exception handling (see Chapter 4), one can transmit the exception handler as a routine parameter to the unit that may raise an exception behavior.

Routine parameters behave very differently in statically and dynamically scoped languages. Here we concentrate on statically scoped languages. Hints on how to handle dynamically scoped languages are given in a sidebar.

Consider the program in Figure 30. In this program, b is called by main (line 14) with actual parameter a; inside b, the formal parameter x is called (line 15), which in this case corresponds to a. When a is called, it should execute normally just as if it had been called directly, that is, there should be no observable differences in the behavior of a routine called directly or through a formal parameter. In particular, the invocation of a must be able to access the nonlocal environment of a (in this case the global variables u and v. Note that these variables are not visible in b because they are masked by b's local variables with the same names.) This introduces a slight difficulty because our current abstract implementation scheme does not work. As we saw in Section 2.6.4, the call to a routine is translated to several instructions. In particular, it is necessary to reserve space for the activation record of the callee and to set up its static link. In the case of “call x” in b, this is impossible at translation time because we do not know what routine x is, let alone its enclosing unit. This information, in general, will only be known at run time. We can handle this situation by passing the size of the activation record and the needed static link at the point of call.

```
int u, v;
a ( )
{
     int y;
     . . .
};
b (routine x)
{
     int u, v, y;
     c ( )
     {...
          y = . . .;
          . . .
     };
     x ( );
     b (c);
     . . .
}
 main ( )
{
     b (a);
};
```

FIGURE 30. An example of routine parameters

In general, how do we know this static link to pass? From the scope rules, we know that in order for a unit x (in this case, main) to pass routine a to routine b, x must either:

(a) Have procedure a within its scope, that is, a must be nonlocally visible or local (immediately nested); or

(b) a must be a formal parameter of x, that is, some actual procedure was passed to x as a routine parameter[1].

The two cases can be handled in the following way:

Case (a): The static link to be passed is fp (d), a pointer to the activation record that is d steps along the static chain originated in the calling unit, where d is the distance between the call point where the routine parameter is passed and its declaration (recall Section 2.6.4).

[1] Case (b) cannot occur in the case where x is main, since main cannot be called by other routines.

Case (b): The static link to be passed is the one that was passed to the caller.

We leave the task of formulating these rules in terms of SIMPLESEM as an exercise for the reader.

What about calling a routine parameter? The only difference from calling a routine directly is that both the size of the callee's activation record and its static link are simply copied from the parameter area.

The program in Figure 30 shows another subtle point: when routine parameters are used in a program, nonlocal variables visible at a given point are not necessarily those of the latest allocated activation record of the unit where such variables are locally declared. For example, after the recursive call to b when c is passed (line 16), the call to x in b (line 15) will invoke c recursively. Then the assignment to y in c (line 12) will not modify the y in the latest activation record for b but in the one allocated prior to the latest one. Figure 30 shows this point.

Let us review the impact of procedural parameters. First, we ha d to extend the basic procedure call mechanism to deal with the additional semantic complexity. Procedure calls now have to deal with different cases of objects. Both the procedure call's semantic description and its implementation have increased in complexity. Contrast this with, say, adding a new arithmetic operator to a language that requires hardly any changes to our semantic description at all. We can say that the ability to pass procedures as parameters adds to the semantic power (and complexity) of a language. On the other hand, it makes the language more uniform in the way the different language constructs are handled: routines are first-class objects, and can be treated uniformly as any other objects of the language.

This is an example of a general property of languages, called *orthogonality*. This term describes the ability of a language to support any combination of basic constructs to achieve any degree of power, without restrictions and without "special cases".

sidebar-start Routine parameters in dynamically scoped languages

Routines passed as parameters sometimes cause a peculiar problem in languages with dynamic scope rules, such as early versions of LISP. If we consider the program in Figure 30 under dynamic scope rules, when routine a is called through x, references to u and v in a will be bound to the u and v in b and not to those in main. This is difficult to use and confusing since when the routine a was written, it was quite reasonable to expect access to u and v in main but because b happens to contain variables with the same names, they mask out the variables that were probably intended to be used.

Simply stated, the problem is that the nonlocal environment, and therefore the behavior of the routine, is dependent on the dynamic sequence of calls that have been made before it was activated. Consider several programmers working on different parts of the same program. A seemingly innocuous decision, what to name a variable, can change the behavior of the program entirely.

The problem, however, was discovered very early in the development of LISP and a new feature was added to the language to allow a routine to be passed along with its naming environment. If a routine is preceded by the keyword FUNCTION, the routine is passed along with its nonlocal environment at the point of call. When such a procedure is invoked, the environment information passed with the parameter is used to set up the current nonlocal environment. This is a rather complicated mechanism, but it seems to be the only reasonable way for procedural parameters to access the nonlocal environment. Of course, a different–and more radical–solution to the problem would be to change the language semantics, and adopt static scope rules for the entire language, as most modern LISPs do.

sidebar end

2.7 Bibliographic notes

In this chapter we have studied programming language semantics in an informal but systematic way, by describing the behavior of an abstract language processor. Formal approaches to the definition of semantics are also possible, as we briefly discussed. (Meyer 1991) provides a view of the theoretical foundations of programming languages and their semantics. Our view here is oriented towards language implementation, in order to allow the reader to appreciate the resources that may needed, and the costs that may be involved, in running a program. We have emphasized the important concepts of binding, binding time, and binding stability. This viewpoint is taken by other textbooks on programming languages, from a classic (Pratt 1984) to a recent one

(Ben Ari 1996). Johnston (Johnston 1971) presents the contour model, an interesting operational model that describes the concepts of binding without referring to the stack-based abstract machine. The reader who might be interested in the details of language implementation should refer to compiler textbooks, like (Waite and Goos 1984), (Aho et al. 1986), and (Fisher and LeBlanc 1988).

2.8 Exercises

1. Provide syntax diagrams for the lexical rules of the language described in Figure 5.
2. In the example of Figure 5 a semicolon is used to terminate each statement in a sequence. That is a sequence is written as{stat1; stat2; . . .; statn;}. Modify the syntax so that the semicolon would be used as a separator between consecutive statements, that is: {stat1; stat2; . . .; statn}. Pragmatically, can you comment on the differences between these two choices?
3. Modify both the EBNF and the syntax diagrams of Figure 6 to represent Modula-2 if and while statements.
4. Briefly describe scope binding for Pascal variables. Does the language adopt static or dynamic binding?
5. In Section 2.2 we say “Perhaps surprisingly, there are languages supporting both static type checking and polymorphic variables and routines”. Why should one expect, in general, static type checking to be impossible for polymorphic variables and routines?
6. Following the definition of static binding given in Section 2.3, specify the time and stability of the binding between a C (or C++) const variable and its value. How about constants in Pascal?
7. Can the l_value of a variable be accessed only when its name is visible (i.e., within scope)?
   Why? Why not?
8. What is the solution adopted by C, Ada, Modula-2, and Eiffel to the problem of uninitialized variables?
9. In general, it is not possible to check statically that the r_value of an unitialized variable will not be used. Why?
10. Does Pascal allow a named variable and a pointer share the same data object? Why? Why not?
11. C and C++ distinguish between declaration and definition for variables. Study this language feature and write a short explanation of why this can be useful.
12. C++ also allows functions to accept more parameters than are specified in the function definition. Why and how is this possible? Write a program to check this feature.
13. What is the difference between macros and routines? Explain it in terms of the concept of binding.
14. Describe if and how routines may be passed as parameters in C/C++.
15. Describe the two ways (named vs. positional) provided by Ada to associate actual and formal parameters.
16. Discuss the EQUIVALENCE statement of FORTRAN in the light of aliasing.

17. A routine may be history-sensitive if it can produce different results when activated twice with the same values as parameters and accessible nonlocal variables. Explain why a language with static memory allocation allows writing history-sensitive routines.
18. Define an algorithm that performs the linkage step for C2'.
19. Write the sequence of SIMPLESEM instructions corresponding to a functional routine call and return to take into account return values. For simplicity, you may assume that return values may be stored in a single SIMPLESEM cell.
20. Write a simple C3 program with two mutually recursive routines, describe their SIMPLESEM implementation, and show snapshots of the D memory.
21. In Section 2.6.3 we assumed function routines to return their output value in a location within the caller's activation record. A number of detail aspects were left out from our discussion. For example, we implicitly assumed the cells allocated to hold the result to be released when the caller routine returns. This, however, implies a waste of memory space if a large number of calls is performed. Also, we did not provide a systematic way of generating SIMPLESEM code for the evaluation of expressions that contain multiple function calls, as in

    ```
    a = f (x) + b + g (y, z);
    ```

    Discuss how these problems may be solved.
22. For the following C3 program fragment, describe each stage in the life of the run-time stack until routine beta is called (recursively), by alpha. In particular, show the dynamic and static links before each routine call.

```
int i = 1, j = 2, k = 3;
beta ( );
alpha ( )
{
    int i = 4, l = 5;
    ...
    i+=k+l;
    beta ( );
    ...
};
beta ( )
{
    int k = 6;
    ...
    i=j+k;
    alpha ( );
    ...
};
main ( )
{
    ...
    beta ( );
    ...
}
```

23. Based on the treatment of recursive functions, discuss dynamic allocation in the case of C2 and show how the scheme works for the example of the Section 2.6.2.

24. In our treatment of C4' using SIMPLESEM, we said “In order to provide an abstract implementation of compound statements for the SIMPLESEM machine, there are two options. The former consists of statically defining an activation record for a routine with nested compound statements; the latter consists of dynamically allocating new memory space corresponding to local data as each compound statement is entered during execution. The former scheme is simpler and more time efficient, while the latter can lead to a more space efficient actual implementation.” Write a detailed comment justifying such a statement.
25. Consider the following extension to C3. A variable local to a routine may be declared to be own. An own variable is allocated storage the first time that its enclosing routine is activated, and its storage remains allocated until program termination. Normal scope rules apply, so that the variable is known only within the unit in which it is declared. In essence, the effect of the own declaration is to extend the lifetime of the variable to cover the entire program execution. Outline an implementation model for own variables. For simplicity, you may assume that own variables can only have simple (unstructured) types.
26. Referring to the previous exercise, assume that own variables are not automatically initialized to certain default values. Show that this limits their usefulness greatly. Hint: Show, as an example, how an own variable can be used to keep track of the number of times a routine has been called.
27. Discuss a SIMPLESEM abstract implementation using dynamic allocation for C4' nested blocks.
28. Explain why the static and dynamic links have the same value for blocks.
29. Translate function fp (Section 2.6.4) into SIMPLESEM code.
30. An implementation technique for referencing the nonlocal environment in C4", which differs from the use of static links as presented in Section 2.6.4.2, is based on the use of a *display*. The display is an array of variable length that contains, at any point during program execution, pointers to the activation records of the routines that form the referencing environment–that is, exactly those pointers that would be in the static chain. Let an identifier be bound to the (distance, offset) pair <d, o>. The display is set up such that display [d] yields the address of the activation record which contains the identifier at offset o.
    - (a) Show pictorially the equivalent of Figure 25 when displays are used instead of static links. Assume that, like current and free, the display is kept in the initial portion of the data memory.
    - (b) Show the SIMPLESEM actions that are needed to update the display when a routine is called and when a routine returns. Pay special attention to routine parameters.
    - (c) Displays and static chains are two implementation alternatives for the same semantic concept. Discuss the relative advantages and disadvantages of each solution.
31. Check a guage of your choice (e.g., Pascal or C++) to see if it allows expressions to be passed by reference. Specify in what cases (if any) this is allowed and provide a concise justification of the behavior. In cases where it does not (if any), write (and run) simple programs which demonstrate the reason.
32. Provide a full description of a SIMPLESEM implementation of call by value.
33. Provide a full description of a SIMPLESEM implementation of call by result. Note that the semantics can be different if the address of the actual result parameter is evaluated at the point of call or at the return point. Why? Show an example where the effect would be different.

34. Can a constant be passed as a by-reference parameter? Check this in a language of your choice.
35. Provide a full SIMPLESEM implementation for call by value-result. Can the abstract implementation for call by reference be used as a semantic description of call by value result?
36. Consider the example program shown below. Discuss call by reference and call by value-result for swap (a[i], a[j]). What happens if i = j?

```
 swap (int x, int y);
{
    x = x +y;
    y = x - y;
    x = x - y;
}
```

37. Write a short paper on C macros, comparing them with routines. What are the binding policies adopted by the language? How do you compare parameter handling for macros with the general parameter passing mechanisms described in this chapter?
38. Precisely discuss how call by reference can be implemented in C++.
39. Study parameter passing mechanisms in Ada and write a short paper discussing them and comparing them with respect to conventional parameter passing modes.
40. Explain why the axiomatic definition of semantics of assignment statements, given in terms of function asem, is inaccurate in the presence of side-effects in the evaluation of the right-hand side expression and aliasing for the left-hand side variable.
41. Explain why the denotational definition of semantics of assignment statements, given in terms of function dsem, is inaccurate in the presence of side-effects in the evaluation of the right-hand side expression and aliasing for the left-hand side variable.
42. We observed that in C++ formal parameters can be given a default value, which is used in case the corresponding actual parameters are not passed in the call. For example, given the following function header: int distance (int a = 0, int b =0); the call distance ( ); is equivalent to distance (0, 0); and the call distance (10); is equivalent to distance (10, 0);. Explain why this language feature interacts with overloading and how this interaction is solved by C++.

1

Structuring the data

CHAPTER 3

Computer programs can be viewed as functions that are applied to values of certain input domains to produce results in some other domains. In conventional programming languages, this function is evaluated through a sequence of steps that produce intermediate data that are stored in program variables. Languages do so by providing features to describe data, the flow of computation, and the overall program organization. This chapter is on mechanisms for structuring and organizing data values; Chapter 4 is on mechanisms for structuring and organizing computations; Chapter 5 is on the mechanisms that languages provide for combining the data and computation mechanisms into a program.

Programming languages organize data through the concept of *type*. Types are used as a way to classify data according to different categories. They are more, however, than pure sets of data. Data belonging to a type also share certain semantic behaviors. A type is thus more properly defined as a set of values and a set of operations that can be used to manipulate them. For example, the type BOOLEAN of languages like Ada and Pascal consists of the values TRUE and FALSE; Boolean algebra defines operators NOT, AND, and OR for BOOLEANs. BOOLEAN values may be created, for example, as a result of the application of relational operators (<, ð, >, Š, +, ¦) among INTEGER expressions.

Programming languages usually provide a fixed, built-in set of data types, and mechanisms for structuring more complex data types starting from the elementary ones. Built-in types are discussed in Section 3.1. Constructors that allow more complex data types to be structured starting from built-in types are discussed in Section 3.2. Section 3.3 is about type systems, i.e., on the principles that underlie the organization of a collection of types. The type system adopted by a language affects the programming style enforced by the language. It may also have a profound influence on the reliability of programs, since it may help prevent errors in the use of data. Moreover, understanding the type system of a language helps us understand subtle and complicated semantic issues. Section 3.4 reviews the type system of existing programming languages. Finally, Section 3.5 is about implementation models.

3.1 Built-in types and primitive types

Any programming language is equipped with a finite set of *built-in types* (or *predefined*) types, which normally reflect the behavior of the underlying hardware. At the hardware level, values belong to the untyped domain of bit strings, which constitutes the underlying universal domain of computer data. Data belonging to such universal domain are then interpreted differently by hardware instructions, according to different types. At the *hardware level*, a type may thus be considered as a view under which data belonging to the universal type may be manipulated. As an example of a hypothetical microcomputer, the bit string "01001010" might be interpreted as integer "74" (coded in two’s complement representation) when it is the argument of the machine instruction ADD (which does integer addition). However, it would be interpreted as a bit string by the machine instruction CPL (which does bitwise complement). It might be interpreted as ASCII character "I" if printed by instruction PCH (which prints an ASCII character).

The built-in types of a programming language reflect the different views provided by typical hardware. Examples of built-in types are:

- booleans, i.e., truth values TRUE and FALSE, along with the set of operations defined by Boolean algebra;
- characters, e.g., the set of ASCII characters;
- integers, e.g., the set of 16-bit values in the range <-32768, 37767>; and
- reals, e.g., floating point numbers with given size and precision.

Let us analyze what makes built-in types a useful concept. This discussion will help us identify the properties that types in general (i.e., not only the built-in ones) should satisfy. Built-in types can be viewed as a mechanism for *classifying* the data manipulated by a program. Moreover, they are a way of *protecting* the data against forbidden, or nonsensical, maybe unintended, manipulations of the data. Data of a certain type, in fact, are only manipulable by the operations defined for the type. In more detail, the following are advantages of built-in types:

1. *Hiding of the underlying representation.* This is an advantage provided by the abstractions of higher-level languages over lower-level (machine-level) languages. The programmer does not have access to the underlying bit string that represents a value of a certain type. The programmer may change such bit string by applying operations, but the change is visible as a new value of the built-in type, not as a new bit string. Invisibility of the underlying representation has the following benefits:

    Programming style. The abstraction provided by the language increases program readability by protecting the representation of objects from undisciplined manipulation. This contrasts with the underlying conventional hardware, which does not enforce protection, but usually allows any view to be applied on any bit string. For example, a location containing an integer may be added to one containing a character, or even to a location containing an instruction.

    Modifiability. The implementation of abstractions may be changed without affecting the programs that make use of the abstractions. Consequently, portability of programs is also improved, that is, programs can be moved to machines that use different internal data representations. One must be careful, however, regarding the precision of data representation, that might change for different implementations. For example, the range of representable integer values is different for 16- and 32-bit machines.

    Programing languages provide features to read and write values of built-in types, as well as for formatting the output. Such features may be either provided by language instructions or through predefined routines. Machines perform input/output by interacting with peripheral devices in a complicated and machine-dependent way. High-level languages hide these complications and the physical resources involved in machine input/output (registers, channels, and so on).

2. *Correct use of variables can be checked at translation time*. If the type of each variable is known to the compiler, illegal operations on a variable may be caught while the program is translated. Although type checking does not prevent all possible errors to be caught, it improves our reliance on programs. For example, in Pascal or Ada, it cannot ensure that J will never be zero in some expression I/J, but it can ensure that it will never be a character.

3. *Resolution of overloaded operators can be done at translation time*. For readability purposes, operators are often overloaded. For example, + is used for both integer and real addition, * is used for both integer and real multiplication. In each program context, however, it should be clear which specific hardware operation is to be invoked, since integer and real arithmetic differ. In a statically typed language, where all variables are bound to their type at translation time, the binding between an overloaded operator and its corresponding machine operation can be established at translation time, since the types of the operands are known. This makes the implementation more efficient than in dynamically typed languages, for which it is necessary to keep track of types in run-time descriptors.

4. *Accuracy control.* In some cases, the programmer can explicitly associate a specification of the accuracy of the representation with a type. For example, FORTRAN allows the user to choose between single and double-precision floating-point numbers. In C, integers can be short int, int, or long int. Each C compiler is free to choose appropriate size for its underlying hardware, under the restriction that short int and int are at least 16 bits long, long int is at least 32 bits long, and the number of bits of short int is no more than the number of bits of int, which is no more than the number of bits of long int. In addition, it is possible to specify whether an integer is signed or unsigned. Similarly, C provides both float (for single-precision floating point numbers) and double (for double precision floating point numbers). Accuracy specification allows the programmer to direct the compiler to allocate the exact amount of storage that is needed to represent the data with the desired precision.

Some types can be called *primitive* (or *elementary*). That is, they are not built from other types. Their values are atomic, and cannot be decomposed into simpler constituents. In most cases, built-in types coincide with primitive types, but there are exceptions. For example, in Ada both Character and String are predefined. Data of type String have constituents of type Character, however. In fact, String is predefined as:

```
type String is array (Positive range <>) of Character
```

It is also possible to declare new types that are elementary. An example is given by enumeration types in Pascal, C, or Ada. For example, in Pascal one may write:

```
type color = (white, yellow, red, green, blue, black);
```

The same would be written in Ada as

```
type color is (white, yellow, red, green, blue, black);
```

Similarly, in C one would write:

```
enum color {white, yellow, red, green, blue, black};
```

In the three cases, new constants are introduced for a new type. The constants are ordered; i.e., white < yellow < . . . < black. In Pascal and Ada, the built-in successor and predecessor functions can be applied to enumerations. For example, succ (yellow) in Pascal evaluates to red. Similarly. color’pred (red) in Ada evaluates to yellow.

3.2 Data aggregates and type constructors

Programming languages allow the programmer to specify aggregations of elementary data objects and, recursively, aggregations of aggregates. They do so by providing a number of *constructors*. The resulting objects are called *compound objects*. A well-known example is the array constructor, which constructs aggregates of homogeneous-type elements. An aggregate object has a unique name. In some cases, manipulation can be done on a single elementary component at a time, each component being accessible by a suitable selection operation. In many languages, it is also possible to manipulate (e.g., assign and compare) entire aggregates.

Older programming languages, such as FORTRAN and COBOL, provided only a limited number of constructors. For example, FORTRAN only provided the array constructor; COBOL only provided the record constructor. In addition, through constructors, they simply provided a way to define a new *single* aggregate object, not a type. Later languages, such as Pascal, allowed new compound types to be defined by specifying them as aggregates of simpler types. In such a way, any number of instances of the newly defined aggregate can be defined. According to such languages, constructors can be used to define both aggregate objects and new aggregate types.

Since in this chapter we concentrate on data types, we review constructors that generate compound data. One should not ignore, however, that routines can also be seen as constructors which allow elementary instructions to be combined to form new operations. In addition, the distinction between data and routines vanishes in the case of programming languages that treat routines as first class objects, which can be assigned, passed as parameters, be members of data structures, etc.

Type constructors are discussed and exemplified in Section 3.2.1 through Section 3.2.6. Section 3.2.7 discusses how structured data values can be denoted in some languages. Section 3.2.8 will discuss how new types can be defined not only through composition of more elementary types, but also by specifying the operations to be used for their manipulation. In the discussion, we will first describe the constructors abstractly in terms of a mathematical model, and then we will show how different programming languages provide concrete constructs to represent the abstract model.

3.2.1 Cartesian product

The Cartesian product of n sets $A_1, A_2, \ldots, A_n$, denoted $A_1 \times A_2 \times \ldots \times A_n$, is a set whose elements are ordered n-tuples $(a_1, a_2, \ldots, a_n)$, where each a_k belongs to A_k. For example, regular polygons might be described by an integer–the number of edges–and a real–the length of each edge. A polygon would thus be an element in the Cartesian product integer x real.

Programming languages view elements of a Cartesian product as composed of a number of symbolically named *fields*. In the example, a polygon could be declared as composed of an integer field (no_of_edges) holding the number of edges and a real field (edge_size) holding the length of each edge.

Examples of Cartesian product constructors in programming languages are *structures* in C, C++, Algol 68 and PL/I, *records* in COBOL, Pascal, and Ada. COBOL was the first language to introduce Cartesian products, which proved to be very useful in data processing applications. For example, in a payroll transaction, employees are described by an n-tuple of attributes (such as name, address, social security number, salary, etc.), some of which–in turn–may be described by an n-tuple of attributes (e.g., an address is composed of street name, number, city, state, and zip code). Such an aggregation can be described by a record.

As an example of a Cartesian product constructor, consider the following C declaration, which defines a new type reg_polygon and two objects a_pol and b_pol;

```
struct reg_polygon {
    int no_of_edges;
    float edge_size;
};
struct reg_polygon pol_a, pol_b = {3, 3.45};
```

The two regular polygons pol_a and pol_b are initialized as two equilateral triangles whose edge is 3.45. The notation {3, 3.45} is used to implicitly define a constant value (also called a *compound value*) of type reg_polygon (the polygon with 3 edges of length 3.45).

The fields of an element of a Cartesian product are selected by specifying their name in an appropriate syntactic notation. In the C example, one may write:

```
pol_a.no_of_edges = 4;
```

to make pol_a quadrilateral. This syntactic notation for selection, which is common in programming languages, is called the *dot notation*.

3.2.2 Finite mapping

A finite mapping is a function from a finite set of values of a domain type DT onto values of a range type RT. Such function may be defined in programming languages through the use of the mechanisms provided to define routines. This would encapsulate in the routine definition the law associating values of type RT to values of type DT. This definition is called *intensional*. In addition, programming languages, provide the *array* constructor to define finite mappings as data aggregates. This definition is called *extensional*, since all the values of the function are explicitly enumerated. For example, the C declaration

```
char digits [10];
```

defines a mapping from integers in the subrange 0 to 9 to the set of characters, although it does not state which character corresponds to each element of the subrange. The following statements

```
for (i = 0; i < 10; ++i)
    digits [i] = ' ';
```

define one such correspondence, by initializing the array to all blank characters. This example also shows that an object in the range of the function is selected by *indexing*, that is, by providing the appropriate value in the domain as an index of the array. Thus the C notation digits [i] can be viewed as the application of the mapping to the argument i. Indexing with a value which is not in the domain yields an error. Some languages specify that such an error is to be trapped. Such a trap, however, may in general only occur at run time.

C arrays provide only simple types of mappings, by restricting the domain type to be an integer subrange whose lower bound is zero. Other programming languages, such as Pascal, require the domain type to be an ordered discrete type. For example, in Pascal, it is possible to declare

```
var x: array [2. .5] of integer;
```

which defines x to be an array whose domain type is the subrange 2. .5.

As another example of Pascal, having defined a type computer_manufacturer by enumeration

```
type computer_manufacturer = (ibm, dec, hp, sun, apple, compaq);
```

one may use the array type constructor to define the following new type to represent data about each computer manufacturer

```
type c_m_data = array [computer_manufacturer] of integer
```

and then the following data objects

```
var c_m_profits, c_m_employees: c_m_data;
```

For example, c_m_employees[hp] would give the number of employees of computer manufacturer hp. If only the data regarding profits are needed, one could simply define an array data aggregate instead of defining a new type, of which many instances can be generated:

```
var c_m_profits: array [computer_manufacturer] of integers;
```

Languages that allow variables to be initialized when they are declared may also provide a way to initialize array objects. For example, in C arrays may be initialized through a compound value, as shown by the following example

```
char digits [10] = {' ', ' ', ' ', ' ', ' '};
```

where {' ', ' ', ' ', ' ', ' '} is a compound value of type "array of 5 characters."

Similarly, in Ada one might write

```
X: array (INTEGER range 2. .6) of INTEGER :=  (0, 2, 0, 5, -33);
```

to define an array whose index is in the subrange 2. .5, where X(2) = 0, X(3) = 2, X(4) = 0, X(5) = 5, X(6) = -33.

It is interesting to note that Ada uses brackets "(" and ")" instead of "[" and "]" to index arrays. This makes indexing an array syntactically identical to calling a function. As a consequence, the fact that a mapping is defined extensionally or intentionally does not affect the way the mapping is used, but depends only on how the mapping is defined; that is, it hides the implementation of the mapping abstraction from the user.

Notice that an array element can–in turn–be an array. This allows multidimensional arrays to be defined. For example, the C declaration

```
int x[10][20];
```

declares an integer rectangular array of 10 rows and 20 columns.

In some languages, such as APL, Algol 68, and Ada, indexing can be used to select more than one element of the range. For example, in Ada X(3. .5) selects a subarray of the previously declared array. This operation is called *slicing*, that is, it selects a slice of the array. Slicing is not provided by C.

In a dynamically typed language like SNOBOL4, the array construct does not require that the elements of the range set be all of the same type (or, equivalently, domain and range types of mappings are unions of all types). For example, one element might be an integer, another a real, and yet a third a string. In other words, the range set can be viewed as the union of all SNOBOL4 types. The TABLE construct provided by SNOBOL4 further extends this notion, by allowing the domain type to be the union of all SNOBOL4 types. For example, the following statements create a TABLE and assign values to some of its elements:

```
T = TABLE ( )
T<'RED'> = 'WAR'
T<6> = 25
t<4.6> = 'PEACE'
```

The TABLE construct is quite powerful because it provides the capability of *associative retrieval*, such as T<'RED'>, which yields 'WAR', or T<6>, which yields 25. Such aggregates are called *associative data structures*.

The domain of a finite mapping is often defined as a finite subset of a (theoretically) infinite set. For example, an array whose index is in the range 0. .9 defines a finite mapping whose domain is a finite subset of integers. The strategy for binding the domain of a finite mapping to a specific finite subset of a given type varies according to the language. Basically, there are three possible choices:

1. *Compile-time binding*. The subset is fixed when the program is written and it is frozen at translation time. This restriction was adopted by FORTRAN, C, and Pascal.
2. *Object-creation time binding*. The subset is fixed at run time, when an instance of the variable is created. In Section 2.6.5.1 we called finite mappings of this kind *dynamic arrays*. As an example, in Ada it is possible to declare an *unconstrained array type* by using symbol <> (the *box*) which stands for unspecified range. When a variable of the unconstrained array type is declared, the bounds must be stated as expressions to be computed at run time. Once the binding is established at run time, however, it cannot be changed (i.e., the binding is static).

   ```
   type INT_VECTOR is array (INTEGER range <>) of INTEGER;
   . . .
   X: INT_VECTOR (A. .B*C);
   ```

3. *Object-manipulation time binding*. This is the most flexible and the most costly choice in terms of run-time execution. For these so-called flexible arrays, the size of the subset can vary at any time during the object's lifetime. This is typical of dynamic languages, like SNOBOL4 and APL. Of compiled languages, only Algol 68 and CLU offer such features. The 1995 proposed C++ standard library contains vectors, which are flexible C++ arrays. Since the memory space required for such data may change during execution, allocation must use the heap memory.

3.2.3 Union and discriminated union

Cartesian products defined in Section 3.2.1 allow an aggregate to be constructed through the *conjunction* of its fields. For example, we saw the example of a polygon, which was represented as an integer (the number of edges) *and* a real (the edge size). In this section we explore a constructor which allows an element (or a type) to be specified by a *disjunction* of fields.

For example, suppose we wish to define the type of a memory address for a machine providing both absolute and relative addressing. If an address is relative, it must be added to the value of some INDEX register in order to access the corresponding memory cell. Using C, we can declare

```
union address {
    short int offset;
    long unsigned int absolute;
};
```

The declaration is very similar to the case of a Cartesian product. The difference is that here fields are mutually exclusive.

Values of type address must be treated differently if they denote offsets or absolute addresses. Given a variable of type address, however, there is no automatic way of knowing what kind of value is currently associated with the variable (i.e., whether it is an absolute or a relative address). The burden of remembering which of the fields of the union is current rests on the programmer. A possible solution is to consider an address to be an element of the following type:

```
struct safe_address {
    address location;
    descriptor kind;
};
```

where descriptor is defined as an enumeration

```
enum descriptor {abs, rel};
```

A safe address is defined as composed of two fields: one holds an address, the other holds a descriptor. The descriptor field is used to keep track of the current address kind. Such a field must be updated for each assignment to the corresponding location field.

This implementation corresponds to the abstract concept of a *discriminated union*. Discriminated unions differ from unions in that elements of a discriminated union are tagged to indicate which set the value was chosen from. Given an element e belonging to the discriminated union of two sets S and T, a function tag applied to e gives either ’S’ or ’T’. Element e can therefore be manipulated according to the value returned by tag.

Type checking must be performed at run time for elements of both unions and discriminated unions. Nothing prevents programs to be written (and compiled with no error) where an element is manipulated as a member of type T while it is in fact a member of type S or vice-versa. Discriminated unions, however, are potentially safer since they allow the programmer to explicitly take the tag field into consideration before applying an operation to an element, although they cannot prevent the programmer from breaching safety by assigning the tag field a value which is inconsistent with the other fields.

There are languages that get close to properly supporting the notion of discriminated union. For example, Pascal offers *variant records* to represent discriminated unions. The following Pascal declarations define a safe address:

```
type natural = 0. .maxint;
     address_type = (absolute, offset);
     safe_address = record
                         case kind:  address_type of
                              absolute: (abs_addr: natural);
                              offset: (off_addr: integer)
                         end
```

Type natural (defined as a subrange of non-negative integers) is introduced to represent absolute addresses. Type address_type is the enumeration of the possible values of the tag. Field kind of the variant record is called the *tag field*. According to the value of the tag field kind, either field abs_addr or field off_addr can be accessed. Access to field off_addr when the value of the tag field is absolute would result in a run-time error; similarly, access to field abs_addr when the value of the tag field is offset would result in a run-time error.

While Pascal allows the concept of discriminated union to be more naturally represented than in C, it does not make the implementation safer. In Pascal, the tag and the variant parts may be accessed in the same way as ordinary components. After the tag field of a safe address representing an offset is changed to absolute, it is possible to access field abs_addr. In principle, this should result in a run-time error, because the field should be considered as uninitialized. In practice, however, most Pascal implementations do not perform such a check, for run-time efficiency reasons. Moreover, the conventional implementation of variant records consists of overlapping all variants over the same storage area. Therefore, by changing the tag field, the machine interprets the string of bits stored in this area under the different views provided by the types of each variant.

This is an insecure–although in some cases practical–use of variant records. Viewing the same storage area under different types may be useful in modeling certain practical applications. For example, a program unit that reads from an input device might view a sequence of bytes according to the type of data that is required. In general, however, this is an unsafe programming practice, and should be normally avoided.

3.2.4 Powerset

It is often useful to define variables whose value can be any subset of a set of elements of a given type T. The type of such variables is powerset (T), the set of all subsets of elements of type T. Type T is called the *base type*. For example, suppose that a language processor accepts the following set O of options

- LIST_S, to produce a listing of the source program;
- LIST_O, to produce a listing of the object program;
- OPTIMIZE, to optimize the object code;
- SAVE_S, to save the source program in a file;
- SAVE_O, to save the object program in a file;
- EXEC, to execute the object code.

A command to the processor can be any subset of O, such as

{LIST_S, LIST_O}
{LIST_S, EXEC}
{OPTIMIZE, SAVE_O, EXEC}

That is, the type of a command is powerset (O).

Variables of type powerset (T) represent sets. The operations permitted on such variables are set operations, such as union and intersection.

Although sets (and powersets) are common and basic mathematical concepts, only a few languages–notably, Pascal and Modula-2–provide them through built-in constructors and operations. Also, the set-based language SETL makes sets the very basic data structuring mechanism. For most other languages, set data structures are provided through libraries. For example, the C++ standard library provides many data structures, including sets.

3.2.5 Sequencing

A sequence consists of any number of occurrences of elements of a certain component type CT. The important property of the sequencing constructor is that the number of occurrences of the component is unspecified; it therefore allows objects of arbitrary size to be represented.

It is rather uncommon for programming languages to provide a constructor for sequencing. In most cases, this is achieved by invoking operating system primitives which access the file system. It is therefore difficult to imagine a common abstract characterization of such a constructor. Perhaps the best example is the file constructor of Pascal, which models the conventional data processing concept of a sequential file. Elements of the file can be accessed sequentially, one after the other. Modifications can be accomplished by appending a new values at the end of an existing file. Files are provided in Ada through standard libraries, which support both sequential and direct files.

Arrays and recursive list definitions (defined next) may be used to represent sequences, if they can be stored in main memory. If the size of the sequence does not change dynamically, arrays provide the best solution. If the size needs to change while the program is executing, flexible arrays or lists must be used. The C++ standard library provides a number of sequence implementations, including vector and list.

3.2.6 Recursion

Recursion is a structuring mechanism that can be used to define aggregates whose size can grow arbitrarily and whose structure can have arbitrary complexity. A recursive data type T is defined as a structure which can contain components of type T. For example, a binary tree can be defined as either empty or as a triple composed of an atomic element, a (left) binary tree, and a (right) binary tree. Formally, if we assume that nil denotes the empty (or null) tree, int_bin_tree (the set of all binary trees of integers) may be described using the union and Cartesian product of sets:

```
    int_bin_tree = {nil} U (integer x int_bin_tree x int_bin_tree)
```

As another example, a list of integers may be described recursively as

```
    int_list = {nil} U integer x int_list
```

where nil here denotes the empty list.

Conventional programming languages allow recursive data types to be implemented via pointers. Each component of the recursive type is represented by a location containing a pointer to the data object, rather than the data object itself. Thus, in the int_list example, the implementation would be a structure, where one field contains an integer and the other field points to a structure of the same type, and so on. The list itself would be identified by another location containing the pointer to the first element of the list.

The C and in Ada fragments in Figure 31 define the type of an integer list and a variable that can point to the head of a specific integer list instance.

(C)

```
struct int_list {
    int val;
    int_list* next;
};
int_list* head;
```

(Ada)

```
type INT_LIST_NODE;
type INT_LIST_REF is access INT_LIST_NODE;
type INT_LIST_NODE is
    record
        VAL: INTEGER;
        NEXT: INT_LIST_REF;
    end;
HEAD: INT_LIST_REF;
```

FIGURE 31. Declarations of list elements in C and Ada

Similar implementations of recursive types can be provided in C++, Pascal, and Modula-2.

Functional languages, as we will see in Chapter 7, provide a more abstract way of defining and manipulating recursive types, which masks the underlying pointer-based implementation. For example, in ML a list can be denoted as either [ ] (the empty list) or as [x: :xs], the list composed of the head element x and the tail list xs. In order to find an element in a list, we can write the following self-explaining high-level function:

```
fun find (el, [ ]) = false
|    find (el, [el: :els]) = true
|    find (el, [y: : ys] = find (el, ys)
```

3.2.6.1 Insecurities of pointers

Pointers are a powerful, but low-level, programming mechanism that can be used to build complex data structures. In particular, they allow recursive data structures to be defined. As any low level mechanism, however, they often allow obscure and insecure programs to be written. Just as unrestricted goto statements broaden the context from which any labelled instruction can be executed, unrestricted pointers broaden the context from which a data object may be accessed. Let us review a number of cases of insecurities that may arise and possible ways of controlling them.

1. Some languages, like Pascal or Ada, require pointers to be typed. For example, a Pascal variable p declared of type ^integer, is restricted to point to objects of type integer. This allows the compiler to type heck the correct use of pointers and objects pointed to by pointers to be type checked for correct use by the compiler. On the other hand, other languages, like PL/I, treat pointers as untyped data objects, i.e., they allow a pointer to address any memory location, no matter what the contents of that location is. In such a case, dynamic type checking should be performed to avoid manipulation of the object via nonsensical operations.
2. C requires pointers to be typed but, unlike Pascal, it also allows arithmetic operations to be applied to pointers. For example, having declared int* p (p is a pointer to objects of type int), one can write p = p + i;, where i is an int variable. This would make x refer to the memory location which is i integer objects beyond the one p is currently pointing to. It is up to the programmer to guarantee that the object pointed by x is an integer. For example, consider the following C fragment:

   ```
   int x = 10;
   float y = 3.7;
   int* p = & x; /* &x denotes the address of x; thus p points to x */
   p ++; /* makes p point to the next location, which contains a float value */
   *p += x; /* increments by 10 the value of y, interpreted as an int */
   printf ("%f", y); /* reinterprets the modified contents of cell y as a float */
   ```

   Although potentially unsafe, pointer arithmetic can be useful in practice. In fact, C pointers and arrays are closely related. The name of an array can also be used as a pointer to its first element and any operation that can be achieved by array subscripting can also be done with pointers.

   Accessing arrays via pointers is in general faster than using the more readable array notation, unless the compiler generates optimized code. It is therefore preferable when efficiency is crucial, thus trading readability for performance. As an example of using a pointer to access an array, consider the following fragment:

   ```
   int n, vect [10]; /*declares an integer vector */
   int* p = &vect[0];  /* p points to the first element of p */
   for (n = 0; n < 10; n++) /* initializes array elements to zero */
       *p++= 0;
   ```

   Incrementing a pointer may done efficiently by a single machine instruction. Thus access to an array element may be faster than using the standard code generated to access an array element indexed by an expression.
3. The r_value of a pointer is an address, i.e., an l_value of an object. If such object is not allocated, we say that the pointer is *dangling*. A dangling pointer is a serious insecurity,

because it refers to a location that is no longer allocated for holding a value of interest. Dangling pointers may arise in languages, like C, which allow the address of any variable to be obtained (via the & operator) and assigned to a pointer. In the fragment shown in Figure 32, since px is a global variable and x is deallocated when function trouble returns, px is left dangling since the object it points to no longer exists. In order to avoid this problem, languages like Algol 68 require that in an assignment the scope of the object being pointed be at least as large as that of the pointer itself. This restriction, however, can only be checked at run time. For example, consider a routine with two formal parameters: x, an integer, and px, a pointer to integers. Whether the assignment of px = &x in the routine is legal depends on the actual parameters and obviously is unknown at compile time. As usual, checking the error at run time slows down the execution of the program; not checking the error leaves dangling pointers uncaught.

More on this will be said for Ada in Section 3.4.3.

```
void trouble (int* px)
{
    int x;
    ... px = & x; ...
    return;
}
main ( )
{
    int* P;
    ... trouble (p); ...
}
```

FIGURE 32. An example of dangling pointers in C

4. To avoid the above insecurities, some languages (like Pascal, Modula-2) further restrict the use of pointers. Pointers are typed; they cannot be manipulated through arithmetic operators; there is no way to get the address of a named variable. Yet other sources of insecurity may arise in such languages because of storage deallocation. Since the amount of heap storage allocated by an executing program can become very large, it is important to provide mechanisms for releasing heap storage as it becomes unreferenced, to allow such storage to be later allocated for new heap variables. Some languages rely on *automatic storage reclamation* to make unused heap storage available as later allocation requests are issued (see Section 3.5.2.7). Other languages provide a standard operator to explicitly deallocate heap storage. For example, Pascal provides a standard routine dispose; C++ provides delete. The operator must be explicitly used by the programmer as necessary. Unfortunately, however, the programmer can request deallocation of a heap variable while there are still pointers to it, which creates dangling pointers. This error is difficult to check, and most implementations do not provide such facility.
5. Languages that allow pointers to be components of a union may cause further insecurities. For example, if we declare a variable bad of the following type trouble, bad can be assigned an integer value, which is then interpreted as a pointer to access some unpredictable location:

```
union trouble {
    int int_var;
    int* int_ref;
}
```

In the case of C, this is the same as the result of pointer arithmetic. But in a language that does not support pointer arithmetic, union types may cause pointer insecurities. For example, the same undesirable effect may occur in Pascal using variant records.

3.2.7 Compound values

Besides supporting the ability to define structured variables, some languages allow constant values of *compound* (or *composite*) objects to be denoted. For example, in C++ one can write:

```
char hello[ ] = {’h’, ’e’, ’l’, ’l, ’o’, ’\0’};
struct complex {
    float x, y;
};
complex a = {0.0, 1.1};
```

This fragment initializes array hello to the array value {’h’, ’e’, ’l’, ’l,’ ’o’, ’\0’}, i.e., the string "hello" (’\0’ is the null character denoting the end of the string). Structure a is initialized to the structure value {0.0, 1.1}.

Ada provides a rich and elaborate set of facilities to define values of compound objects. For example, the following expressions denote objects of the type INT_LIST_NODE defined in Section 3.2.6.

```
(VAL => 5, NEXT => new INT_LIST_NODE (0, null))
    --field NEXT of the object points to a child node which contains value 0
    --and has null NEXT pointer
    --the child node is defined positionally; i.e., 0 is the value of field VAL and null
    --is the value of field NEXT

(10, null);
    --this record value is described positionally
```

Array objects can also be denoted in Ada. For example, a variable Y of the following type

```
type BOOL_MATRIX is array (0. .N, 0. .M);
```

can be initialized in the following way:

```
Y := (1. .N - 1 => (0. .M => TRUE), others => FALSE);
    --all rows except for the first and the last are initialized to TRUE
    --the first and the last are initialized to FALSE
```

This is an equivalent way of initializing the array:

```
Y := (0 | M => (0. . N => FALSE), others => TRUE);
```

```
        --row 0 and M are initialized to FALSE; others are initialized
```

The ability of compound objects to be directly denoted is a nice syntactic shorthand that frees the programmer from accessing each component at a time. Moreover, it favors a sound programming practice such that every variable is initialized as it is declared.

3.2.8 User-defined types and abstract data types

Modern programming languages provide many ways of defining new types, starting from built-in types. The simplest way, mentioned in Section 3.1, consists of defining new elementary types by enumerating their values. The constructors reviewed in the previous sections go one step further, since they allow complex data structures to be composed out of the built-in types of the language. Modern languages also allow aggregates built through composition of built-in types to be named as new types. Having given a type name to an aggregate data structure, one can declare as many variables of that type as necessary by simple declarations.

For example, after the C declaration which introduces a new type name complex

```
    struct complex {
        float real_part, imaginary_part;
    }
```

any number of instance variables may be defined to hold complex values:

```
    complex a, b, c, . . .;
```

By providing appropriate type names, program readability can be improved. In addition, by factoring the definition of similar data structures in a type declaration, modifiability is also improved. A change that needs to be applied to the data structures is applied to the type, not to all variable declarations. Factorization also reduces the chance of clerical errors and improves consistency.

The ability to define a type name for a user defined data structure is only a first step in the direction of supporting data abstractions. As we mentioned in Section 3.1, the two main benefits of introducing types in a language are classification and protection. Types allow the (otherwise unstructured) world of data to be organized as a collection of different categories. Types also allow data to be protected from undesirable manipulations by specifying exactly which operations are legal for objects of a given type and by hiding the concrete representation. Of these two properties, only the former is achieved by defining a user-defined data structure as a type. What is needed is a construct that allows both a data structure and operations to be specified for user-defined types. More precisely, we need a construct to define abstract data types. An *abstract data type* is a new type for which we can define the operations to be used for manipulating instances, while the data structure that implements the type is hidden to the users. In what follows we briefly review the constructs provided by C++ and by Eiffel to define abstract data types. Further elaboration of the concepts presented here will be discussed in Chapter 5 and 6. The way abstract data types can be defined in ML is presented in Chapter 7.

3.2.8.1 Abstract data types in C++

Abstract data types can be defined in C++ through the class construct. A class encloses the definition of a new type and explicitly provides the operations that can be invoked for correct use of instances of the type. As an example, Figure 33 shows a class defining the type of the geometrical concept of point.

```
class point {
    int x, y;
public:
    point (int a, int b) { x = a; y = b; }      // initializes the coordinates of a point
    void x_move (int a) { x += a; }             // moves the point horizontally
    void y_move (int b ){ y += b; }             // moves the point vertically
    void reset ( ) { x = 0; y = 0; }            // moves the point to the origin
};
```

FIGURE 33.A C++ class defining point

A class can be viewed as an extension of structures (or records), where fields can be both data and routines. The difference is that only some fields (declared public) are accessible from outside the class. Non-public fields are hidden to the users of the class. In the example, the class construct encapsulates both the definition of the data structure defined to represent points (the two integer numbers x and y) and of the operations provided to manipulate points. The data structure which defines a geometrical point (two integer coordinates) is not directly accessible by users of the class. Rather, points can only be manipulated by the operations defined as public routines, as shown by the following fragment:

```
point p1 (1, 3);                    // instantiates p1 and initializes its value
point p2 (55, 0);                   // instantiates p2 and initializes its value
point* p3 = new point (0, 0);       // p3 points to the origin
p1.x_move (3);                      // moves p1 horizontally
p2.y_move (99);                     // moves p2 vertically
p1.reset ( );                       // positions p1 at the origin
```

The fragment shows how operations are invoked on points by means of the dot notation; that is, by writing “object_name.public_routine_name”. The only exceptions are the invocations of constructors and destructors. We discuss constructors below; destructors will be discussed in a later example.

A *constructor* is an operation that has the same name of the new type being defined (in the example, point). A constructor is automatically invoked when an object of the class is allocated. In the case of points p1 and p2, this is done automatically when the scope in which they are declared is entered. In the case of the dynamically allocated point referenced by p3, this is done when the new instruction is executed. Invocation of the constructor allocates the data structure defined by the class and initializes its value according to the constructor’s code.

A special type of constructor is a *copy constructor*. The constructor we have seen for point builds a point out of two int values. A copy constructor is able to build a point out of an existing point. The signature of the copy construtor would be:

```
point (point&)
```

The copy constructor is fundamentally a different kind of constructor because it allows us to build a new object from an existing object without knowing the components that constitute the object. That is what our first constructor does. When a parameter is passed by value to a procedure, copy construction is used to build the formal parameter from the argument. Copy construction is almost similar to assignment with the difference that on assignment, both objects exist whereas on copy construction, a new object must be created first and then a value assigned to it.

It is also possible to define *generic abstract data types*, i.e., data types that are parametric with respect to the type of components. The construct provided to support this feature is the *template*. As an example, the C++ template in Figure 34 implements an abstract data type stack which is parametric with respect to the type of elements that it can store and manage according to a last-in first-out policy. The figure also describes a fragment that defines data objects of instantiated generic types:

```
template<class T> class Stack{
     int size;
     T* top;
     T* s;
public:
     Stack (int sz)            {top = s = new T [size = sz];}
     ~Stack ( )                {delete [ ] s;}    //destructor
     void push (T el)          {*top++ = el;}
     T pop ( )                 {return *--top;}
     int length ( )            {return top - s;}
};

void foo ( ) {
     Stack<int> int_st (30);
     Stack<item> item_st (100);
     . . .
     int_st.push (9);
     . . .
}
```

FIGURE 34.A C++ generic abstract data type and its instantiation

The template also shows an example of a destructor. A destructor is recognized by having the name of the class, prefixed by ~ (which stands for “the complement of the constructor”). The purpose of a destructor is to perform a cleanup after the last use of an object. In the example, the cleanup deallocates the array used to store the stack. It is called automatically for automatic objects (i.e., objects allocated in the runtime stack) upon exit from the scope in which the objects are declared. It must be called explicitly for dynamic objects allocated in the heap in order to free the memory when the object becomes inaccessible. This operation, as we already mentioned, may generate dangling references if the object being released is still referenced. If no constructors and/or destructors are explicitly specified for a class, the language provides for implicit construction/destruction which depends on the types of the encapsulated data.

3.2.8.2 Abstract data types in Eiffel

Eiffel provides a class construct to implement abstract data types. Figure 35 describes the abstract data type POINT in Eiffel.

Another class can become a client of POINT by declaring references to objects of type POINT:

```
p1, p2: POINT;
```

Objects can be created and bound to such references, and then manipulated according to the type’s operations:

```
p1.make_point (4, 7);
p2.make_point (55, 0);
p1.move_x (3);
p2.move_y (99);
p1.reset ( );
```

```
class POINT export
    x_move, y_move, reset
creation
    make_point
feature
    x, y: INTEGER;
    x_move (a: INTEGER) is
            -- moves the point horizontally
    do
        x := x + a
    end; --x_move
    y_move (b: INTEGER) is
            -- moves the point vertically
    do
        y := x + b
    end; --y_move
    reset is
            -- moves the point to the origin
    do
        x = 0;
        y = 0
    end; -- reset
    make_point (a, b: INTEGER) is
            -- sets the initial coordinates of the point
    do
        x := a;
        y := b
    end -- make_point
end; -- POINT
```

FIGURE 35. An Eiffel class defining point

C++ instances of an abstract data type can be either stack objects or heap objects. That is, they can be associated both with automatic variables or be dynamically allocated and referred to by pointers. In the example in Figure 33, the objects associated with variables p1 and p2 are (automatically) allocated on the stack; the objects to which p3 points is dynamically allocated on the heap. In Eiffel, all objects (except for built-in elementary values like integers) are implicitly allocated on the heap and made accessible via pointers. In the example of Figure 35, p1 and p2 are in fact pointers to objects, which are allocated (and initialized) by the invocation of the creation operation.

The Eiffel make_point is analogous to the C++ constructor but must be called explicitly to create the object. The C++ concept of copy construction—creating a new object from an existing like object—is not associated with each object. Rather, the langauge provides a function named **clone** which can be called with an object of any type to create a new object which is a copy of the original object.

The Eiffel language assumes a set of principles that should guide programmers in a disciplined and methodical development of programs. It is possible to associate a class with an *invariant property*, i.e., a predicate that characterizes all possible correct instances of the type. For example, consider a variant NON_AXIAL_INT_POINT of class POINT which describes the set of points with integer coordinates that do not belong to the axes x and y. The x- and y-coordinates of the elements of class NON_AXIAL_INT_POINT cannot be zero; that is, the invariant property for such class is written as:

```
x * y /= 0
```

To ensure that the invariant is satisfied, suitable constraints must apply to the exported routines of the class. This is stated in Eiffel by defining two predicates: a *precondition* and a *postcondition*. These two predicates characterize the states in which the routine can start and should end its execution. A class is said to be consistent if it satisfies the following conditions:

1. for every creation routine, if its precondition holds prior to execution, the invariant holds upon termination
2. for every exported routine, if the precondition and the invariant hold prior to execution, the postcondition and the invariant hold upon termination.

If these two rules are satisfied, by simple induction one can prove that the invariant will always be true for all reachable object states.

```
class NON_AXIAL_POINT export
    x_move, y_move
creation
    make_point
feature
    x, y: INTEGER;
    x_move (a: INTEGER) is
            -- moves the point horizontally
    require
        x + a /= 0
    do
        x := x + a
    ensure
        x /= 0
    end; --x_move
    y_move (b: INTEGER) is
            -- moves the point vertically
    require
        y + b /= 0
    do
        y := x + b
    ensure
        y /= 0
    end; --y_move
    make_point (a, b: INTEGER) is
            -- sets the initial coordinates of the point
    require
        a * b /= 0
    do
        x := a;
        y := b
    end -- make_point
invariant
    x * y /= 0
end; -- NON_AXIAL_POINT
```

FIGURE 36. An Eiffel class defining a point that may not lie on the axes x and y

Class NON_AXIAL_INT_POINT is presented in Figure 36. The reader should be able to verify manually that the above conditions 1. and 2. for class consistency are verified.

Eiffel does not prescribe that facilities be provided by the language implementation to check that all classes are consistent. It does not even force programmers to provide preconditions, postconditions, and invariants: assertions are optional, although their use is good programming practice. If they are present, an Eiffel implementation can check such properties at runtime. This is an effective way of debugging Eiffel programs. As we will see in Chapter 4, it also supports systematic programmed ways of error handling.

Eiffel supports the implementation of generic abstract data types, via *generic classes*. As an example, Figure 37 shows an implementation of a generic stack abstract data type in Eiffel. The definition of preconditions, postconditions, and invariants are left to the reader as an exercise.

```
class STACK [T] export
    push, pop, length
creation
    make_stack
feature
    store: ARRAY [T];
    length: INTEGER;

make_stack (n: INTEGER) is
    do  store.make (1, n); --this operation allocates an array with bounds 1, n
        length := 0;
    end; --make_stack

push (x: T) is
    do  length := length + 1;
        put (x, length); --element x is stored at index length of the array
    end; --push

pop: T is
    do  Result := store@ (length);
        -- the element in the array whose index is length is copied in the
        -- predefined object Result, which contains the value returned by the
        -- function
        length := length - 1;
    end; --pop
end --class STACK
```

FIGURE 37. An Eiffel abstract data type definition

3.3 Type systems

Types are a fundamental semantic concept of programming languages. Moreover, programming languages differ in the way types are defined and behave, and typing issues are often quite subtle. Having discussed type concepts informally in different languages so far, we now review the foundations for a theory of types. The goal is to help the reader understand the *type system* adopted by a language, defined as the set of rules used by the language to structure and organize its collection of types. Understanding the type system adopted by a language is perhaps the major step in understanding the language's semantics.

Our treatment in this section is rather abstract, and does not refer to any specific programming language features. The only assumption made is that a type is defined as a set of values and a set of operations that can be applied to such values. As usual, since values in our context are stored somewhere in the memory of a computer, we use the term *object* (or *data object*) to denote both the storage and the stored value. The operations defined for a type are the only way of manipulating its instance objects: they protect data objects from any illegal uses. Any attempt to manipulate objects with illegal operations is a *type error*. A program is said to be *type safe* (or *type secure*) if all operations in the program are guaranteed to always apply to data of the correct type, i.e., no type errors will ever occur.

3.3.1 Static versus dynamic program checking

Before focusing our discussion on type errors, a brief digression is necessary to discuss more generally the kinds of errors that may occur in a program, the different times at which such errors can be checked, and the effect of checking times on the quality of the resulting programs.

Errors can be classified in two categories: language errors and application errors. *Language errors* are syntactic and semantic errors in the use of the programming language. *Application errors* are deviations of the program behavior with respect to specifications (assuming specifications capture the required behavior correctly). The programming language should facilitate both kinds of errors to be identified and removed. Ideally, it should help prevent them from being introduced in the program. In general, programs that are readable and well structured are less error prone and easier to check. Hereafter we concentrate on language errors. A discussion of application errors is out of the scope of this book: software design methods address application errors. Therefore, here the term “error” implicitly refers to “language error”.

Error checking can be accomplished in different ways, that can be classified in two broad categories: static and dynamic. *Dynamic checking* requires the program to be executed on sample input data. *Static checking* does not. In general, if a check can be performed statically, it is preferable to do so instead of delaying the check to run-time for two main reasons. First, potential errors are detected at run time only if one can provide input data that cause the error to be revealed. For example, a type error might exist in a portion of the program that is not executed by the given input data. Second, dynamic checking slows down program execution.

Static checking is often called compile-time (or translation-time) checking. Actually, the term “compile-time checking” may not be an accurate synonym of “static checking”, since programs may be subject to separate compilation and some static checks might occur at link time. For example, the possible mismatch between a routine called by one module and defined in another might be checked at link time. Conventional linkers, unfortunately, seldom perform such checks. For simplicity, we will continue to use the terms static checking and compile-time (or translation-time) checking interchangeably.

Static checking, though preferable to dynamic checking, does not uncover all language errors. Some errors only manifest themselves at run time. For example, if div is the operator for integer division, the compiler might check that both operands are integer. However, the program would be erroneous if the value of the divisor is zero. This possibility, in general, cannot be checked by the compiler.

3.3.2 Strong typing and type checking

The type system of a language was defined as the set of rules to be followed to define and manipulate program data. Such rules constrain the set of legal programs that can be written in a language. The goal of a type system is to prevent the writing of type unsafe programs as much as possible. A type system is said to be *strong* if it guarantees type safety; i.e., programs written by following the restrictions of the type system are guaranteed not to generate type errors. A language with a strong type system is said to be a *strongly typed language*. If a language is strongly typed, the absence of type errors from programs can be guaranteed by the compiler. A type system is said to be *weak* if it is not strong. Similarly, a *weakly typed language* is a language that is not strongly typed.

In Chapter 3 we introduced the concept of a statically typed language. Such languages are said to obey to a *static type system*. Precisely, such a type system requires that the type of every expressions be known at compile time. An example of a static type system can be achieved by requiring that

1. only built-in types can be used;
2. all variables are declared with an associated type;
3. all operations are specified by stating the types of the required operands and the type of the result.

A statically typed language is a strongly typed language, but there are strongly typed languages that are not statically typed. For example, we will show in Chapters 6 and 7 examples of languages where the binding between a variable and its type cannot be established at compile time, and yet the rules of the type system guarantee type safety; i.e., they guarantee that correctly compiled programs will execute without generating type errors.

In general, we may observe that many strong type systems exist. Since all of them guarantee type safety, how should a language designer choose a type system when defining a new programming language? There are two conflicting requirements to be accommodated in such a design decision: the size of the set of legal programs and the efficiency of the type checking procedure performed by the compiler. Since a type system restricts the set of programs that can be written, we might come out with rules that allow only very simple programs. In principle, a type system which restricts the set of legal programs to the empty set is a strong type system. It is also trivial to check. But it is obviously useless. The previous example of static typing allows for a simple checking procedure, but it is overly restrictive. Dynamic typing, as we will demonstrate in Chapters 7 and 8, is a very powerful programming facility that can be combined with strong typing. In such a case, however, the is required to perform a complex type checking procedure.

3.3.3 Type compatibility

A strict type system might require operations that expect an operand of a type T to be invoked legally only with a parameter of type T. Languages, however, often allow more flexibility, by defining when an operand of another type–say Q–is also acceptable without violating type safety. In such a case, we say that the language defines whether, in the context of a given operation, type Q is *compatible* with type T. Type compatibility is also sometimes called *conformance* or *equivalence*. When compatibility is defined precisely by the type system, a type checking procedure can verify that all operations are always invoked correctly, i.e., the types of the operands are compatible with the types expected by the operation. Thus a language defining a notion of type compatibility can still have a strong type system.

Figure 38 shows a sample program fragment written in a hypothetical programming language.

```
struct s1{
     int y;
     int w;
};
struct s2{
     int y;
     int w;
};
struct s3 {
     int y;
};
s3 func (s1 z)
{
. . .
};
. . .
s1 a, x;
s2 b;
s3 c;
int d;
. . .
a = b; --(1)
x = a; --(2)
c = func (b); --(3)
d = func (a); --(4)
```

FIGURE 38. A sample program

The strict conformance rule where a type name is only compatible with itself is called *name compatibility*. Under name compatibility, in the above example, instruction (2) is type correct, since a and x have the same type name. Instruction (1) contains a type error, because a and b have different types. Similarly, instructions (3) and (4) contain type errors. In (3) the function is called with an argument of incompatible type; in (4) the value returned by the function is assigned to a variable of an incompatible type.

Structural compatibility is another possible conformance rule that languages may adopt. Type T1 is structurally compatible with type T2 if they have the same structure. This can be defined recursively as follows:

- T1 is name compatible with T2; or
- T1 and T2 are defined by applying the same type constructor to structurally compatible corresponding type components.

According to structural equivalence, instructions (1), (2), and (3) are type correct. Instruction (4) contains a type error, since type s3 is not compatible with int. Note that the definition we gave does not clearly state what happens with the field names of Cartesian products (i.e., whether they are ignored in the check or they are required to coincide and whether structurally compatible fields are required to occur in the same order or not). For simplicity, we assume that they are required to coincide and to occur in the same order. In such a case, if we rename the fields of s2 as y1 and w1, or permute their occurrence, s2 would no longer be compatible with s1.

Name compatibility is easier to implement than structural compatibility, which requires a recursive traversal of a data structure. Name compatibility is also much stronger than structural compatibility. Actually, structural compatibility goes to the extreme where type names are totally ignored in the check. Structural compatibility makes the classification of data objects implied by types exceedingly coarse.

For example, having defined the following two types:

```
struct complex {
    float a;
    float b;
};
struct point {
    float a;
    float b;
};
```

the programmer can instantiate variables to represent–say–points on a plane and values of a.c. voltage. The type system allows to use them interchangeably, although most likely the programmer has chosen two different type names in order to keep the different sets of objects separate. In conclusion, name compatibility is often preferable. It prevents two types to be considered compatible just because their structure happens to be identical by coincidence.

Often programming languages do not take much care in defining the adopted notion of type compatibility they adopt. This issue is left to be defined by the implementation. An unfortunate consequence is that different implementations may adopt different notions, and thus a program accepted by a compiler might be rejected by another. This unfortunate case happened, for example, when Pascal was originally defined, although later ISO Pascal defined type compatibility rigorously, mainly based on name compatibility. C adopts structural compatibility for all types, except structures, for which name compatibility is required.

Type compatibility in Ada is defined via name compatibility. Since the language introduces the concept of a subtype (see also Section 3.3.5), objects belonging to different subtypes of the same type are compatible. In Ada, when a variable is defined by means of a constructor, as in

```
IA: array (INTEGER range 1. .100) of INTEGER;
```

a brand new anonymous type is implicitly introduced, followed by a variable declaration:

```
type ANONYMOUS_1 is array (INTEGER range 1. .100) of INTEGER;
IA: ANONYMOUS_1;
```

Thus, if two variables IA and IB are declared:

```
IA: array (INTEGER range 1. .100) of INTEGER;
IB: array (INTEGER range 1. .100) of INTEGER;
```

the two variables are considered to have noncompatible types, since their anonymous type names would be different.

3.3.4 Type conversions

Suppose that an object of type T_1 is expected by some operation at some point of a program. Also, suppose that an object of type T_2 is available and we wish to apply the operation to such object. If T_1 and T_2 are compatible according to the type system, the application of the operation would be type correct. If they are not, one might wish to apply a type conversion from T_2 to T_1 in order to make the operation possible.

More precisely, let an operation be defined by a function fun expecting a parameter of type T_1 and evaluating a result of type R_1:

fun: T_1 -> R_1

Let x_2 be a variable of type T_2 and y_2 of type R_2. Suppose that T_1 and T_2 (R_1 and

R_2) are not compatible. How can fun be applied to x_2 and the result of the routine be assigned to y_2? This would require two conversion functions to be available, t_{21} and r_{12}, transforming objects of type T_2 into objects of type T_1 and objects of type R_1 into objects of type R_2, respectively:

t_{21}: T_2 -> T_1
r_{12}: R_1 -> R_2

Thus, the intended action can be performed by first applying t_{21} to x_2, evaluating fun with such argument, applying r_{12} to the result of the function, and finally assigning the result to y_2. That is:

(i) $y_2 = r_{12}(\text{fun } (t_{21}(x_2)))$

For some languages any required conversions are applied automatically by the compiler. Following the Algol 68 terminology, we will call such automatic conversions *coercions*. In the example, if coercions are available, the programmer might simply write

(ii) $y_2 = \text{fun } (x_2)$

and the compiler would automatically convert (ii) into (i).

In general, the kind of coercion that may occur at a given point (if any) depends on the context. For example, in C if we write

```
x = x + z;
```

where z is float and x is int, x is coerced to float to evaluate the arithmetic operator + (which stands for real addition), and the result is coerced to int for the assignment. That is, the arithmetic coercion is from int to float, but the assignment coercion is from float to int.

C provides a simple coercion system. In addition, explicit conversions can be applied in C using the *cast* construct. For example, a cast can be used to override an undesirable coercion that would otherwise be applied in a given context. For example, in the above assignment, one can force a conversion of z to int by writing

```
x = x + (int) z;
```

Such an explicit conversion in C is semantically defined by assuming that the expression to be converted is implicitly assigned to an unnamed variable of the type specified in the cast, using the coercion rules of the language.

Ada does not provide any coercions. Whenever a conversion is allowed by the language, it must be invoked explicitly. For example, if X is declared as a FLOAT variable and I is an INTEGER, assigning X to I can be accomplished by the instruction

```
I := INTEGER(X);
```

The conversion function INTEGER provided by Ada computes an integer from a floating point value by rounding to the nearest integer.

The existence of coercion rules in a language has both advantages and disadvantages. The advantage is that many desirable conversions are automatically provided by the implementation. The disadvantage is that since implicit transformations happen behind the scenes, the language becomes complicated and programs may be obscure. In addition, coercions weaken the usefulness of type checking, since they override the declared type of objects with default, context sensitive transformations. For example, Algol 68 consistently applies the principle of implicit conversions to the extreme. The type of the value required at any given point in an Algol 68 program can be determined from the context. But the way coercions interact with other constructs of the language can make programs quite hard to understand. Unexpected difficulties, in particular, arise because of the interaction between coercions and overloading of operators and routines.

3.3.5 Types and subtypes

If a type is defined as a set of values with an associated set of operations, a subtype can be defined to be a subset of those values (and, for simplicity, the same operations). In this section we explore this notion in the context of conventional languages, ignoring the ability to specify user-defined operations for subtypes. The concept of subtype will have a richer semantics in the context of object-oriented languages, as will be discussed in Chapter 6.

If ST is a subtype of T, T is also called ST’s supertype (or parent type). We assume that the operations defined for T are automatically inherited by ST. A language supporting subtypes must define:

1. a way to define subsets of a given type;
2. compatibility rules between a subtype and its supertype.

Pascal was the first programming language to introduce the concept of a subtype, as a subrange of any discrete ordinal type (i.e., integers, boolean, character, enumerations, or a subrange thereof). For example, in Pascal one may define natural numbers and digits as follows:

```
type natural = 0. .maxint;
        digit = 0. .9;
        small = -9. .9;
```

where maxint is the maximum integer value representable by an implementation.

A Pascal program can only define a subset of contiguous values of a discrete type. For example, it cannot define a subtype EVEN of all even integers or multiples of ten in the range -1000. .1000. Different subtypes of a given type are considered to be compatible among themselves and with the supertype. However, type safe operations are not guaranteed to evaluate with no error. No error arises if an object of a subtype is provided in an expression where an object of its supertype is expected. For example, if an expression requires an integer, one may provide a natural; if it expects a natural, one might provide a digit. If, however, a small is provided where a digit is expected, an error arises if the value provided is not in the range expected. That is, if an argument of type T is provided to an operation expecting an operand of type R, the expression is type safe if either R or T is a subtype of the other, or both are subtypes of another type Q. No value error will occur at run time if T is a subtype of R. In all other cases, the operation must be checked at run time and an error may arise if the value transmitted does not belong to the expected type.

Ada provides a richer notion of subtype than Pascal. A subtype of an array type can constrain its index; a subtype of a variant record type can freeze the variant; a subtype of a discrete ordinal type is a finite subset of contiguous values. Examples of Ada types and subtypes are shown in Figure 39.

```
    type Int_Vector is array (Integer range < >) of Integer;
    type Var_Rec (Tag: Boolean) is
    record X: Float;
            case Tag of
                when True => Y: Integer;
                                  Z: Real;
                when False=> U: Char;
            end case;
    end record;
    subtype Vec_100 is Int_Vector (0. .99);
        --this subtype constrains the bounds of the array to 0. .99
    subtype X_true is X (True);
        --this subtype freezes the variant where Tag = True; objects of the subtype thus
        --have fields X, Y, and Z;
    subtype SMALL is Integer range -9. .9;
--this subtype defines a small set of integers
```

FIGURE 39. Examples of Ada types and subtypes

Ada subtypes do not define new types. All values of all subtypes of a certain type T are of type T. The subtype construct can be viewed as a way to signal that certain run-time checks must be inserted by the compiler to ensure that objects of a given subtype always receive the specified restricted set of values.

3.3.6 Generic types

As we mentioned, modern languages allow parameterized (generic) abstract data types to be defined. A typical example is a stack of elements of a parameter type T, whose operations have the following signatures:

```
push: stack (T) x T -> stack (T)     --pushes an element on top of the stack
pop: stack (T) -> stack (T) x T      --extracts the topmost element from the stack
length: stack (T) -> int             --compute the length of the stack
```

An implementation of this example was illustrated in Section 3.2.8 for C++ and Eiffel. In the example, the abstract data type being defined is parametric with respect to a type, and the operations of the generic type are therefore defined as generic routines. The operations defined for the type stack(T) are supposed to work uniformly for any possible type T. However, since the type is not known, how can such routines be type checked to guarantee type safety?

A possibility is provided by languages like Ada, C++, and Eiffel, where generic types must be explicitly instantiated at compile time by binding parameter types to “real” types, that are known at compile time. This achieves static typing for each instance of each generic type, and therefore each instance is statically checked to ensure type safety. Instantiation, however, is not required in languages like ML. As we will see in Chapter 7, however, the language still ensures type safety statically.

3.3.7 Summing up: monomorphic versus polymorphic type systems

A simple strong type system can be provided by a statically typed language where every program entity (constant, variable, routine) has a specific type, defined by a declaration, and every operation requires that an operand of exactly the type that appears in the operation definition can be provided. For such a language, it is possible to verify at compile time that any occurrence of that constant, variable, or routine is type correct. Such a type system is called *monomorphic* (from ancient Greek, “single shape”): every object belongs to one and only one type. By contrast, in a *polymorphic* (“multiple shape”) programming languages every constant and every variable can belong to more than one type. Routines (e.g., functions) can accept as a formal parameter actual parameters of more than one type.

By examining closely traditional programming languages like C, Pascal, or Ada, however, we have seen in the previous secions that all deviate from strict monomorphism in one way or another. Compatibility, discussed in Section 3.3.3, is a first departure from strict monomorphism, since it allows any compatible type to be accepted where a certain type is needed. Coercion, discussed in Section 3.3.4, is also a deviation from strict monomorphism. In fact, it allows an operand of one type to be used when an object of a different type is expected. Subtyping, discussed in Section 3.3.5, provides yet another example of deviation, since an element of the subtype also belongs to the supertype. As yet another example, overloading (introduced in Section 3.3.2) allows operators, like + or *, to be applied to both integer operands and real operands.

Since polymorphic features creep in most–if not all–existing languages, a distinction between monomorphic and polymorphic languages is of no practical use. All practical languages have some degree of polymorphism. Consequently, the important questions to answer are: Can different kinds (or degrees) of polymorphism be identified? How far can we go with polymorphism, and yet retain strong typing? Understanding the possible different forms of polymorphism can help us appreciate the sometimes profound semantic differences among them. Moreover, it will help us organize concepts like coercion, subtyping, and overloading, which were examined in previous sections separately, into a coherent conceptual framework.

Polymorphism can be classified as shown in Figure 40. For the sake of simplicity and abstraction, let us discuss Figure 40 in the case of polymorphic functions, i.e., mathematical functions whose arguments (domain and range) can belong to more than one type.

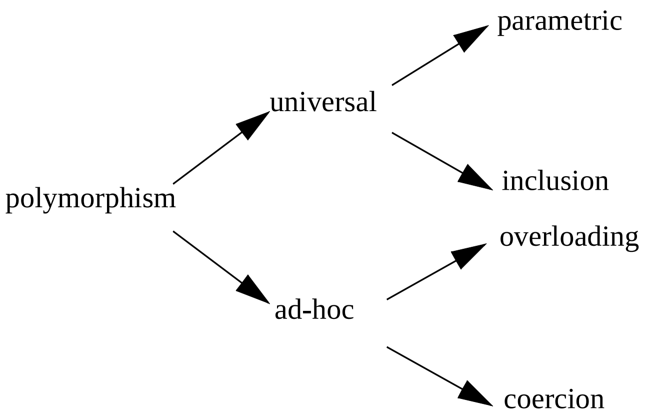


FIGURE 40. A classification of polymorhism

A first distinction is between universal polymorphism and ad-hoc polymorphism. *Ad-hoc polymorphism* does not really add to the semantics of a monomorphic language. Ad-hoc polymorphic functions work on a finite and often small set of types and may behave differently for each type. *Universal polymorphism* characterizes functions that work uniformly for an infinite set of types, all of which have some common structure. Whereas an ad-hoc polymorphic function can be viewed as a syntactic abbreviation for a small set of different monomorphic functions, a universal polymorphic function executes the same code for arguments of all admissible types.

The two major kinds of ad-hoc polymorphism are overloading and coercion.

In *overloading*, the same function name can be used in different contexts to denote different functions, and in each context the function actually denoted by a given name is uniquely determined. A *coercion* is an operation that converts the argument of a function to the type expected by the function. In such a case, polymorphism is only apparent: the function actually works for its prescribed type, although the argument of a different type may be pased to it, but it is automatically transformed to the required type prior to function evaluation. Coercions can be provided statically by code inserted by the compiler in the case of statically typed languages, or they are determined dynamically by run-time tests on type descriptors, in the case of dynamically typed languages.

Overloading and coercion can be illustrated by the C example of the arithmetic expression a + b. In C, + is an ad-hoc polymorphic function, whose behavior is different if it is applied to float values or int numbers. In the two cases, the two different machine instructions float+ (for real addition) or int+ would be needed. If the two operands are of the same type–say, float–the + operator is bound to float+; if both are bound to int, + is bound to int+. The fact that + is an overloaded operator is a purely syntactic phenomenon. Since the types of the operands are known statically, one might eliminate overloading statically by substituting the overloaded + operator with float+ or int+, respectively. If the types of the two operands are different (i.e., integer plus real or real plus integer), however, the float+ operator is invoked after converting the integer operand to real.

Figure 40 defines two kinds of universal polymorphism: parametric and inclusion polymorphism. Subtyping, discussed in Section 3.3.5, is an example of *inclusion polymorphism*. A function is indeed applicable to any type that is a subtype of a given type. This concept is applicable not only to the case of subtyping of Section 3.3.5, but also the more general concept that will be discussed in the context of object oriented languages in Chapter 6.

Parametric polymorphism is perhaps the most genuine form of universal polymorphism. In this case the polymorphic function works uniformly on a range of types that are specified as parameters. Generic routines, as in the case of ML functions, are examples of parametric polymorphic functions. In a language like Ada for which, as anticipated in Section 3.3.6, generic routines are instantiated at compile time, with full binding of type parameters to specific types, genericity is only an apparent kind of polymorphism; that is, it can be viewed as a case of ad-hoc polymorphism.

The term *dynamic polymorphism* is also frequently used to denote the case where the binding between language entities and the form they can take varies dynamically. Languages that support unrestricted forms of dynamic polymorphism cannot provide a strong type system. By providing suitable forms of inclusion and/or parametric polymorphism, however, languages can preservea strong type system. We will discuss this in Chapter 6 for object-oriented languages, which can support inclusion polymorphism, and in Chapter 7 for the functional language ML, which supports parametric polymorphism.

3.4 The type structure of existing languages

In this section we review the type structure of a number of existing programming languages. The description provides an overall hierarchical taxonomy of the features provided by each language for data structuring. For a full understanding of language semantics, such description must be complemented by a precise understanding of the rules of the type system (strong typing, type compatibility, type conversion, subtyping, genericity, and polymorphic features), according to the concepts discussed in Section 3.3. Our discussion will touch on the main features of type structures. Other comments were given in previous parts of this chapter. Moreover, the reader is urged to refer to language manuals for a discussion of all details and subtleties that cannot be addressed in this treatment.

3.4.1 Pascal

The type structure of Pascal is described in Figure 42. A different decomposition of unstructured types would be in terms of ordinal types and real types. *Ordinal types* comprise integers, booleans, characters, enumerations, and subranges. Ordinal types are characterized by a discrete set of values, each of which has (at most) a unique predecessor and a unique successor, evaluated by the built-in functions pred and succ, respectively.

Figure 42 shows how structured types can be built in Pascal. Recursive data structures are defined via pointers. Cartesian products are defined by records. Unions and discriminated unions are described by variant records. Comments on these constructs, and particularly on their possible insecurities, were given earlier. Finite mappings are described by *arrays*, whose index type must be an ordinal type. The size of an array is frozen when the array type is defined, and cannot change during execution. Pascal regards arrays with different index types as different types. For example, a1 and a2 below are different types.

```
type a1 = array [1. .50] of integer;
     a2 = array [1. .70] of integer;
```

This was a serious problem in Pascal as originally defined. Because procedures require formal parameters to have a specified type, it was not possible, for example, to write a procedure capable of sorting both arrays of type a1 and type a2. During the standardization of Pascal by ISO, a new feature was added to solve this problem. This feature, called the *conformant array*, allows the formal array parameter of a procedure to conform to the size of the actual array parameter. The actual and formal parameters are required to have the same number of indexes and the same component type. The example illustrates the use of conformant arrays.

```
procedure sort (var a: array [low. .high: integer] of CType);
var i: integer;
    more: boolean;
    temp: CType;
begin
    more := true;
    while more do begin
        more := false;
        for i := low to high - 1 do begin
            if a [i] > a [i + 1] then begin {move down element}
                temp := a [i];
                a [i] := a [i + 1];
                a [i + 1] := temp;
                more := true
            end
        end
    end
end;
```

FIGURE 41. An example of conformant arrays in Pascal

When the procedure sort is called with a one-dimensional array parameter, low and high assume the values of the lower and upper bounds of the actual parameter, respectively.

Another solution, not available in Pascal, could have been based on genericity (i.e., allowing a procedure to be generic with respect to the array bounds).

More generally, Pascal provides only limited forms of ad-hoc polymorphism. Some built-in operators, like + or succ, are overloaded. In fact, succ is applicable to operators of any ordinal type. Similarly, + can be applied to integer operands, real operands, or even sets (in which case it denotes the union operator). The language also defines cases of coercion. For example, if an integer is added to a real, the integer is coerced to a real, and the addition is performed.

As we mentioned earlier, Pascal is not a strongly typed language. For example, its original definition did not carefully define the concept of type compatibility. Moreover, subtypes are defined by the language as new types, and thus the type of an expression may in general depend on the values of the operands at run time.

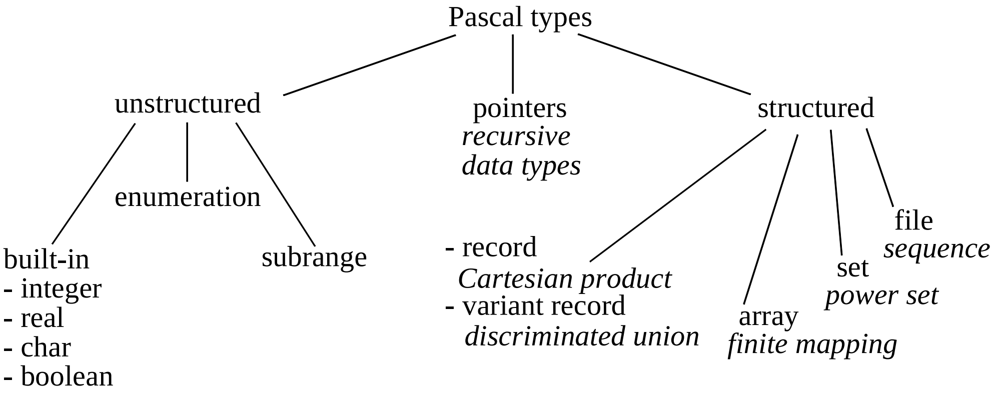


FIGURE 42. The type structure of Pascal

3.4.2 C++

The type structure of C++ is given in Figure 43. C++ distinguishes between to categories of types: *fundamental types* and *derived type*. Fundamental types are either *integral* or *floating*. Integral types comprise char, short int, int, long int, which can be used for representing integers of different sizes. Floating-point types comprise float, double, and long double. New integral types may be declared via enumerations. For example

```
enum my_small_set {low = 1, medium = 5, high = 10}
```

Arrays are declared by providing a constant expression, which defines the number of elements in the array. For example

```
float fa [15];
```

declares an array of floating-point numbers that can be indexed with a value in the range 0. .14.

C++ distinguishes between pointers and references. A reference is an alias for an object. Therefore, once a reference is bound to an object, it cannot be made to refer to a different object. For example, having declared

```
int i = 5;
int& j = i;
```

i and j denote the same object, which contains the value 5.

When the reference's lifetime starts, it must be bound to an object. Thus, for example, one cannot write the following declaration

```
int& j;
```

It is possible, however, to bind a reference to an object through parameter passing. This is actually the way C++ supports parameter passing by reference. For example, according to the following routine declaration

```
void fun (int& x, float y)
```

x represents a by-reference parameter, which is bound to its corresponding actual parameter when the routine gets called.

Pointers are quite different. A pointer is a data object whose value is the address of another object. A pointer can be made to refer a different objects during its lifetime. That is, it is possible to assign a new value to a pointer, not only to the object referenced by the pointer. Of course, it is possible to use pointers also in the case where a reference would do. In fact, references are not provided by C, but were added to the language by C++. As we mentioned, however, pointers are extremely powerful, but difficult to manage, and often dangerous to use. They should be used only when necessary. References should be used in all other cases.

Another major distinctive feature of the C++ type system is the ability to define new types through the class construct. As we discussed, this allows abstract data type implementations to be provided. If no protection is needed on the data declared in a class, classes without default access restrictions can be defined as structures, via the struct construct.

Finally, two other kinds of types can be derived in C++ by using the function and the union constructs. As we already observed, the function construct defines a new data object. It is thus possible to define pointers and references to functions, pass functions as parameters, etc. The union construct defines a structure that is capable of containing objects of different types at different times.

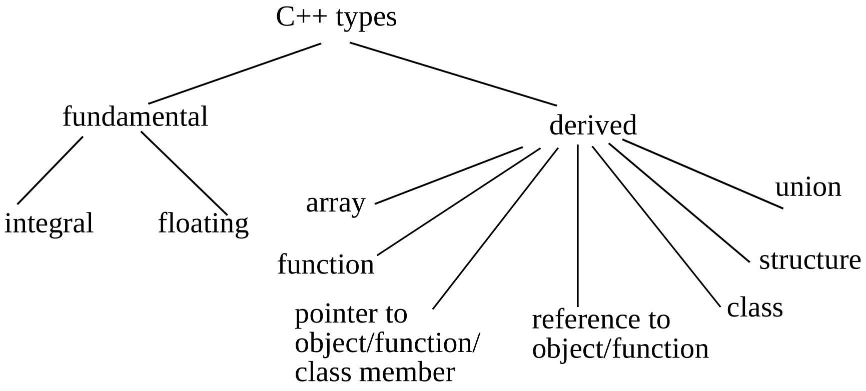


FIGURE 43. The type structure of C++

3.4.3 Ada

The type structure of Ada is described in Figure 44. Such structure is discussed and evaluated in this section, except for concurrency related types, which are discussed in Chapter 4, and tagged types, which are discussed in Chapters 5 and 6.

Unstructured (*scalar*) types can be both numeric (i.e., integers and reals) and enumerations. All scalar types are ordered, i.e., relational operators are defined on them. Enumeration types are similar to those provided by Pascal. *Integer* types comprise a set of consecutive integer values. An integer type may be either signed or modular. Any *signed* integer type is a subrange of System.Min_Int. .System.Max_Int, which denote the minimum and maximum integer representable in a given Ada implementation. A *modular* integer is an absolute value, greater than or equal to zero. The Ada language predefines a signed integer type, called Integer. Other integer types, such as Long_Integer or Short_Integer, may also be predefined by an implementation. Programmer-defined integer types may be specified as shown by the following examples:

```
type Small_Int is range -10. .10; -- range bounds may be any static expressions
type Two_Digit is mod 100; --the values range from 0 to 99;
                                        --in general, the bound must be a static expression
```

As we mentioned, Ada allows *subtypes* to be defined from given types. Subtypes do not define a new type. They conform to the notion of subtype we discussed in Section 3.3.5. Two subtypes of Integer are predefined in Ada:

```
subtype Natural is Integer range 0. .INTEGER'LAST;
subtype Positive is Integer range 1. .INTEGER'LAST;
```

Ada provides a rich and elaborate set of facilities for dealing with *real* values; only the basic aspects will be reviewed here. Real types provided by the language are just an approximation of their mathematical counterpart (universal real, in the Ada terminology). In fact, the fixed number of bits used by the implementation to represent real values makes it possible to store the exact value of only a limited subset of the universal reals. Other real numbers are approximated. Real types in Ada come in two forms: floating point and fixed point. A *floating-point* real type approximates a universal real with an error that is relative to the number's absolute value. A *fixed-point* real approximates a universal real with an error that is independent of the value being represented. The language predefines one floating-point real type, called Float. It is left to the implementation whether additional real types, such as Short_Float or Long_Float, should be provided. The programmer can define additional floating-point real types, such as:

```
type Float_1 is digits 10;
```

The digits clause specifies the minimum number of significant decimal digits required for the type. Such minimum number of digits defines the relative error bound in the approximate representation of universal reals. Given a floating point real type, attribute Digits gives the minimum number of digits associated with the type. Thus, Float_1'Digits yields 10, whereas Float'Digits yields an implementation dependent value.

Fixed-point real types provide another way of approximating universal reals, where the approximation error is independent of the value being represented. Such error bound is specified as the delta of the fixed-point real. An ordinary fixed-point real type is declared in Ada as:

```
type Fix_Pt is delta 0.01 range 0.00. .100;
```

The declaration defines both the delta and the range of values.

A decimal fixed-point type is specified by providing the delta and the number of decimal digits. For example

```
type Dec_Pt is delta 0.01 digits 3;
```

includes at least the range -99.9. .99.9

Ada’s *structured* (or *composite*) types comprise arrays and records. *Arrays* can have statically known bounds, as in Pascal. For example

```
type Month is (JAn, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
type Yearly_Pay is array (Month) of Integer;
type Summer_Pay is array (Month range Jul. .Sep) of Integer;
```

Such array types are also said to be constrained. Constraints, however, do not need to be statically defined; that is, Ada supports dynamic arrays as we show next. First, one can declare unconstrained array types by stating an unspecified range, indicated by the symbol <> (box):

```
type Some_Period_Pay is array (Month range <>) of Integer;
type Int_Vector is array (Integer range <>) of Integer;
type Bool_Matrix is array (Integer range <>, Integer range <>) of Boolean;
```

In Ada, array types are characterized by the types of the components, the number of indices, and the type of each index; the values of the bounds are not considered to be a part of the array type, and thus may be left unspecified at compile-time. The values of the bounds, however, must become known when an object is created. For example, one can declare the following variables:

```
Spring_Salary: Some_Period_Pay (Apr. .Jun);
Z: Int_Vector (-100. .100);
W: Int_Vector (20. .40);
Y: Bool_Matrix (0. .N, 0. .M);
```

Notice that the values of the bounds need not be given by a static expression. It is only required that the bounds be known at run time when the object declarations are processed.

An interesting way of instantiating the bounds of an array is by parameter passing. For example, the following function receives an object of type Int_Vector and sums its components:

```
function Sum (X: Int_Vector) return Integer;
Result: Integer := 0; --declaration with initialization
begin
```

```
    for I in X'First. .X'Last loop
        --attributes First and Last provide the lower and upper bounds of the index
        Result := Result + X (I);
    end loop;
    return Result;
end Sum;
```

The function can thus be called with array parameters of different sizes; for example

```
A := Sum (Z) + Sum (W);
```

Ada views *strings* as arrays of characters of the following predefined type:

```
type String is array (Positive range <>) of characters;
```

A line of 80 characters, initialized with all blanks, can be declared as follows:

```
Line: String (1. .80) := (1. .80 => ' ');
```

Similar to Pascal, Ada records can support both Cartesian products and (discriminated) unions. An example of a *Cartesian product* is

```
type Int_Char is
    X: Integer range 0. .100;
    Y: Character;
end record;
```

Ada provides a safe version of *discriminated unions* through variant records. For example, one may write the following Ada declarations (corresponding to the example discussed in Section 3.2.3)

```
type Address_Type is (Absolute, Offset);
type Safe_Address is record (Kind: Address_Type := Absolute)
    case Kind is
        when Absolute =>
            Abs_Addr: Natural);
        when Offset =>
            Off_Addr: Integer;
    end case;
end record;
```

Type Safe_Address has a discriminant Kind that defines the possible variants of an address. The default initial value of the discriminant is declared in the example to be Absolute. Thus an object declared as

```
X: Safe_Address;
```

is an absolute address by default. The discriminant of a variable initialized by default can be changed only by assignment of the record as a whole, not by assignment to the discriminant alone. This forbids the producing of inconsistent data objects, and makes variant records a safe representation of discriminated unions.

The discriminant of a variable can also be initialized explicitly when a variable is declared, as in the following case:

```
Y: Address (Offset);
```

In such a case, the variant for the object is frozen, and cannot be changed later. The compiler can reserve the exact amount of space required by the variant for the constrained variable. The following assignments

```
X := Y;
X := (Kind => Offset, Off_Addr => 10);
```

are legal and change the variant of variable X to Offset. The following assignment, which would change the variant for Y, is illegal

```
Y := X;
```

Access to the variant of an object whose discriminant is initialized by default, such as

```
X.Off_Addr := X.Off_Addr + 10;
```

requires a run-time check to verify that the object is accessed correctly according to its current variant. In the example, if the address is not an offset, the error exception Constraint_Error is raised.

Access types (*pointers*) are used mainly to allocate and deallocate data dynamically. As an example, the following declarations define a binary tree:

```
type Bin_Tree_Node; --incomplete type declaration
type Tree_Ref is access Bin_Tree_Node;
type Bin_Tree_Node is
    record
        Info: Character;
        Left, Right: Tree_Ref:
    end;
```

(Note that an incomplete type declaration is needed when recursive types are being defined.)

If P and Q are two pointers of type Tree_Ref, the Info component of the node referenced by P is P.Info. The node itself is P.all. Thus, assignment of the node pointed by P to the node pointed by Q is written as

```
Q.all := P.all;
```

If T is a pointer of type TREE_REF, allocation of a new node pointed by T can be accomplished as follows:

```
T := new Bin_Tree_Node;
```

The following assignment

```
T.all := (Info => 0, Left => null, Right => null);
```

initializes T to point to a node whose Left and Right pointers are null, i.e., they do not refer to any entity. The language defined value null denotes a null pointer value.

Ada also allows pointers to refer to routines. For example, the following declaration defines type Message_Routine to be any procedure with an input parameter of type String

```
type Message_Routine is access procedure (M: String);
```

If Print_This is a previously defined procedure with an input parameter M of type String, one can write

```
Give_Message: Message_Routine; --declares a pointer to a routine
. . .
Give_Message := Print_This'Access; --access yields a reference to the routine
. . .
Give_Message.all ('This Is A Serious Error');
            --invokes Print_This; ".all" (dereferencing) is optional
```

Finally, it is also possible in Ada to define pointers which refer to named objects, fields of records, or array elements. Such referenceable objects (or parts of an object) must be declared as aliased (dynamically allocated data are aliased). As the name indicates, such elements are accessible via several possible names (aliases). If an element is declared as aliased, the attribute Access can be applied to provide a pointer to the element. For example, having declared the following data

```
Structure: array (1. .10) of aliased Component;
                        --Component is a previously defined type
Mine, Yours: Component;
. . .
Mine := Structure (1)'Access; --Mine points to the first element of the array
Yours := Structure (2)'Access; --Yours points to the second element of the array
```

As we discussed in Section 3.2.6, allowing pointers to refer to named data objects (or parts thereof) can generate dangling references. This is avoided in Ada by run-time checking that an object referenced by a pointer is allocated in an activation record that is more recently allocated than the activation record of the unit in which the access type is declared. This check ensures that the object will live at least as long as the access type, which in turn ensures that the access values cannot refer to objects that do not exist.

The Ada type system is largely based on Pascal, but is richer, more complex, and more systematic. It also solves several of the problems and insecurities of the original Pascal definition. As a result, the language gets close to the goal of strong typing. Type compatibility (based on name compatibility) is explicitly specified. The notion of subtype and the necessary run-time checks are precisely defined. If a new type is to be defined from an existing type, a derived type construct is provided. For example

```
type Celsius is new Integer;
type Farenheit is new Integer;
```

define two new type; Integer is their parent type. New types inherit all properties (values, operations, and attributes) from their parent type, but they are considered as different types.

Overloading and coercion are widely available in Ada. The language also provides for inclusion polymorphism in the case of subtypes and type extensions (see Chapter 6).

Finally, Ada makes extensive use of *attributes*. Attributes are used to designate properties of data objects and types. As we saw in many examples so far, the value of an attribute is retrieved by writing the name of the entity whose attribute is being sought, followed by a ’ and the name of the attribute. Ada predefines a large number of attributes; more can be defined by an implementation.

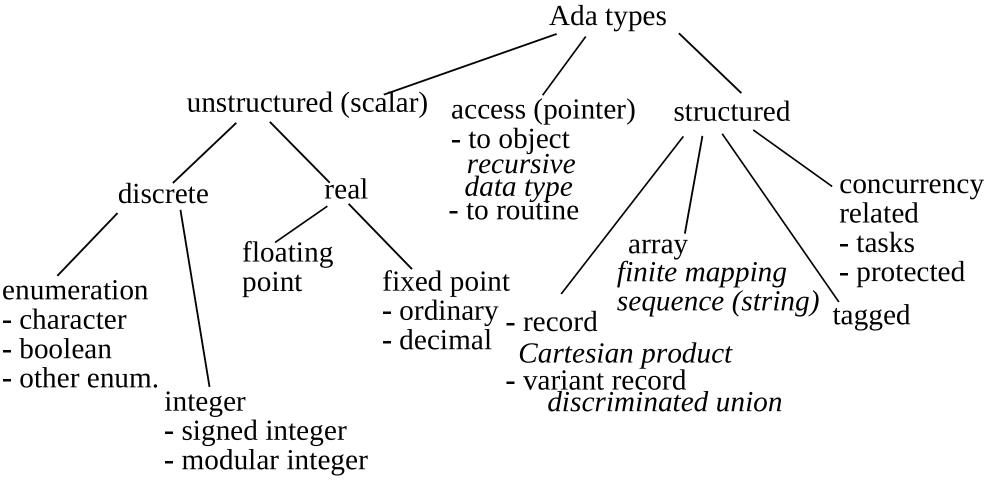


FIGURE 44. The type structure of Ada

3.5 Implementation models

This section reviews the basic implementation models for data objects. The description is language independent. It is intended to complement the conceptual model of programming language processing provided in Chapter 2, by showing how data can be represented and manipulated in a machine. It is not intended, however, to provide a detailed account of efficient techniques for representing data objects within a computer, which can be highly dependent on the hardware structure. Rather, straightforward solutions will be presented, along with some comments on alternative, more efficient representations.

Following Chapter 2, data will be represented by a pair consisting of a *descriptor* and a *data object*. Descriptors contain the most relevant attributes that are needed during the translation process. Additional attributes might be appropriate for specific purposes. Typically, descriptors are kept in a symbol table during translation, and only a subset of the attributes stored there needs to be saved at run time. Again, we will pay little attention to the physical layout of descriptors, which depends on the overall organization of the symbol table.

3.5.1 Built-in and enumerations

Integers and reals are hardware-supported on most conventional computers, which provide fixed and floating-point arithmetic. Existing computers may also provide different lengths for the supported arithmetic types. In a language like C, these are reflected by long and short prefixes. If the language provides different lengths for arithmetic types and the underlying hardware does not, an implementation is usually free to ignore the prefixes. In the case of C, it is only required that short should not be longer than int, which should not be longer than long. If we ignore the issue of different lengths for arithmetic types, for simplicity, integer and real variables may be represented as shown in Figure 45 and Figure 46.

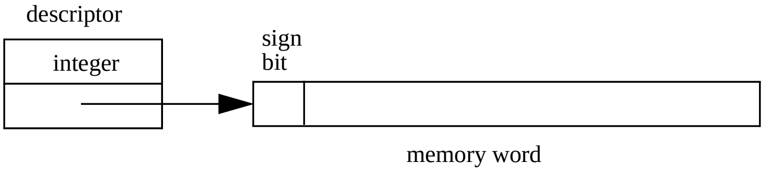


FIGURE 45. Representation of an integer variable

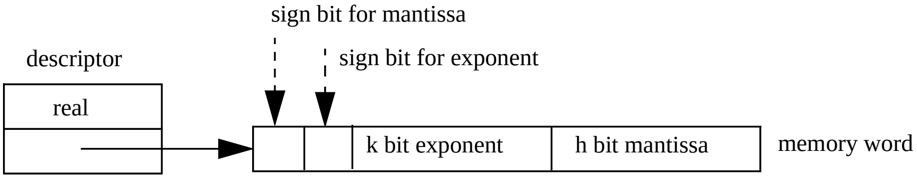


FIGURE 46. Representation of a floating-point variable

Values of an enumeration type ENUM can be mapped, one-to-one, to integer values in the range 0. .n-1, n being the cardinality of ENUM. This mapping does not introduce any possibility of mixing up values of type ENUM with values of any other enumeration type, if all run-time accesses are routed via a descriptor containing the type information. The use of the descriptor is of course not necessary for typed languages. Note that, in a language like C, the mapping of enumeration types to integers is not just part of the implementation of the type (and as such, invisible to the programmer), but it is explicitly stated in the language definition. This allows the programmer to take advantage of this knowledge, and find ways to break the protection shield provided by the type to access the representation directly.

Booleans and characters can be viewed as enumeration types, and implemented as above. To save space, characters and booleans can be stored in storage units smaller than a word (e.g., bits or bytes), which might be addressable by the hardware, or may be packaged into a single word and then retrieved by suitable word manipulation instructions that can disassemble the contents of a word. In such a case, accessing individual characters of booleans may be less efficient, but it would save memory space. Thus the choice of the implementation model depends on a trade-off between speed and space.

3.5.2 Structured types

In this section we review how to represent structured types, built via the constructors discussed in Section 3.2. Our discussion will not be dependent on the specific syntax adopted by an existing language. Rather, it will refer to a hypothetical, self-explaining, programming notation. For simplicity, we will assume that variables are declared by providing an explicit type name. For example, this means that a declaration of–say– a finite mapping X:

```
X:  float array [0. .10];
```

is a shorthand for a declaration of a type followed by a declaration of an array variable:

```
type X_type is float array [0. .10];
X: X_type;
```

Similarly, we assume that if a type declaration contains a structured component, such component is separately defined by a type declaration. For example, if a field of a Cartesian product is a finite mapping, there are two type declarations: the declaration of an array type T and the declaration of a structure, with a field of type T. As a consequence of these assumptions, each component of a structured type is either a built-in type, or it is a user-defined type.

As for built-in types, each variable is described by a descriptor and its storage layout. The descriptor contains a description of the variable’s type. In an actual implementation, for efficiency reasons, all variables of a given type might have a simplified descriptor which points to a separately stored descriptor for that type.

Section 3.5.2.1 deals with Cartesian products. Section 3.5.2.2 deals with finite mappings. Unions and discriminated unions are discussed in Section 3.5.2.3. Powersets and sequences are discussed in Sections 3.5.2.4 and 3.5.2.5. Section 3.5.2.6 discusses user-defined types through a simple class construct. Finally, Section 3.5.2.7 contains an introduction to the management of the heap memory, which is needed for dynamically allocated objects, like those defined by a recursive type, and implemented via pointers.

3.5.2.1 Cartesian product

The standard representation of a data object of a Cartesian product type is a sequential layout of components. The descriptor contains the Cartesian product type name and a set of triples (name of the selector, type of the field, reference to the data object) for each field. If the type of the field is not a built-in type, the type field points to an appropriate descriptor for the field.

Figure 47 illustrates the representation for the following case of a variable of Cartesian product type with a field which is itself of a Cartesian product type:

```
type another_type Y is struct{
    float A;
    int B;
};
type sample_type is struct {
    int X;
    another_type Y;
};
sample_type X;
```

Note that each component of the Cartesian product occupies a certain number of addressable storage units (e.g., words). In a statically typed language, if each component is guaranteed to occupy a fixed memory size, known by the compiler, the descriptor is not needed at run time, and the reference to each field can be evaluated statically by the compiler as a fixed offset with respect to the initial address of the composite object.

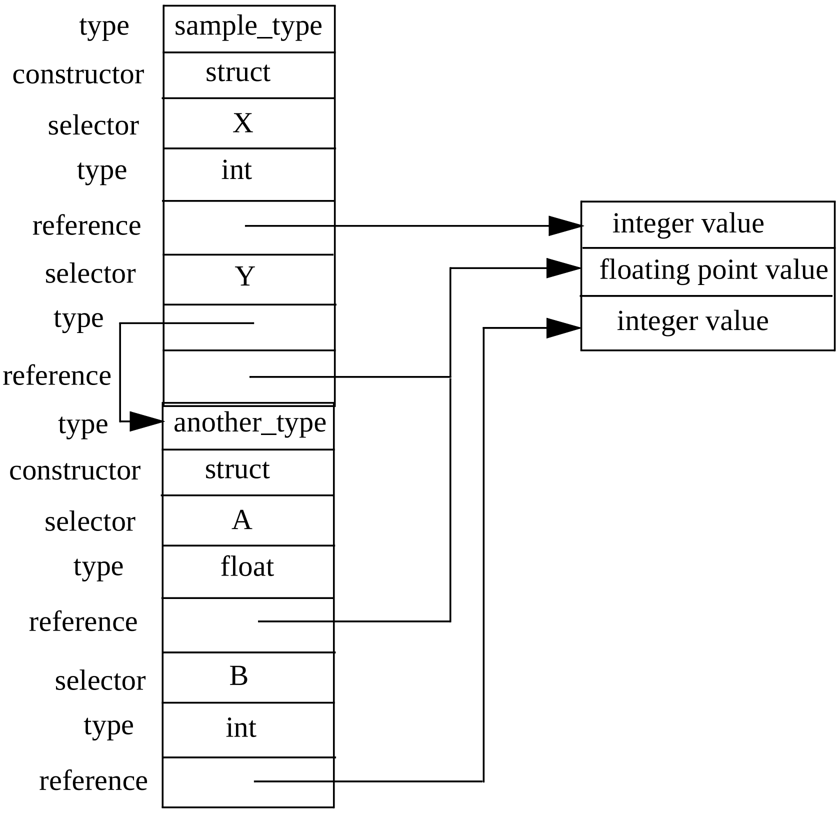


FIGURE 47. Representation of a Cartesian product

3.5.2.2 Finite mapping

A conventional representation of a finite mapping allocates a certain number of storage units (e.g., words) for each component. The descriptor contains the finite-mapping type name; the name of the domain type, with the values of the lower and upper bound; the name of the range type (or the reference to its descriptor); the reference to the first location of the data area where the data object is stored.

For example, given the declarations

```
type X_type is float array [0. .10];
X_type X;
```

the corresponding representation is given in Figure 48.

A reference to X[I] is computed as an offset from the address A_X of the first location of the array. Let the domain type be the integer subrange M. .N. Let K be the number of words occupied by each element of the array (this is known from the type of the range, but might be stored in the descriptor to avoid computing such value each time it is necessary). The offset to be evaluated for accessing A[I] is K(I - M).

In a statically typed language with arrays of statically known index bounds, the descriptor does not need to be saved at run time. The only exception are index bounds, which may be used to check at run time that the index value belongs to the stated range.

As we discussed in Chapter 2 (Section 2.6.5), in a language that supports dynamic arrays, the value of the array in the activation record is composed of two parts. The first part (often called *dope vector*) contains a reference to the data object (which, in general, can only be evaluated at run time) and the values of the bounds (to be used for index checking). The second part is the array itself, which is accessed indirectly through the dope vector.

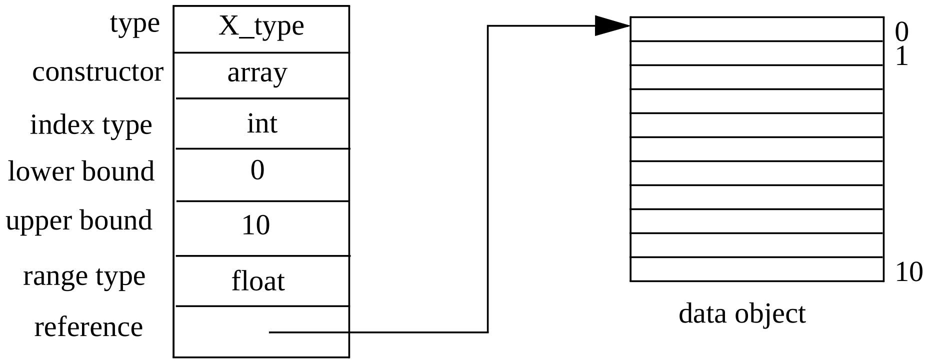


FIGURE 48. Representation of a finite mapping

3.5.2.3 Union and discriminated union

Union types do not require any special new treatment to be represented. A variable of a union type has a descriptor that is a sequence of the descriptors of the component types. Instances of the values of the component types share the same storage area.

Discriminated union types are provided by existing programming languages as extensions of the Cartesian product. For example Pascal and Ada provide discriminated unions as variant records. The variant record is a construct can be viewed as the conjunction of several fields plus a disjunction of fields, prefixed by the definition of a tag field. When the conjunction of fields is empty, we obtain a discriminated union. As an example, the reader may consider the following Pascal-like fragment. Since all variants share the same storage area, a variable Z of type Z_type can be represented as in Figure 49. Note that the various variants are accessible via a case table, according to the value of the tag field.

```
type X_type is float array [0. .10];
type Z_type = record
                          case kind:  BOOLEAN of
                             TRUE: (a: integer);
                             FALSE: (b: X_type)
             end
```

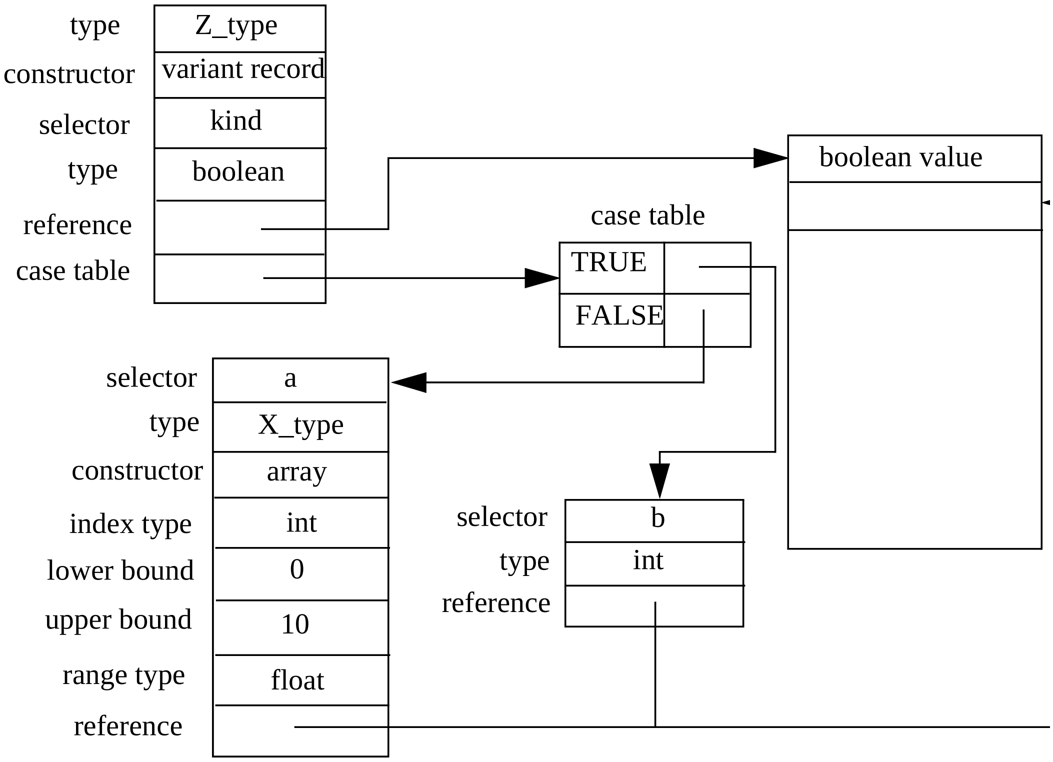


FIGURE 49. Representation of a discriminated union

3.5.2.4 Powersets

Powersets can be implemented efficiently, in terms of access time, manipulation time, and storage space, provided that a machine word has at least as many bits as there are potential members of the set. (i.e., the cardinality of the base type). The presence of the i-th element of the base type in a certain set S is denoted by a “1” as the value of the i-th bit of the word associated with S. The empty set is represented by all zeros in the word. The union between sets is easily performed by an OR between the two associated words, and the intersection by an AND. If the machine does not allow bit-level access, test for membership requires shifting the required bit into an accessible word position (e.g., the sign bit), or using a mask. The existence of such an appealing representation for powersets is responsible for the implementation-defined limits for the cardinality of representable sets, which is normally equal to the size of a memory word.

3.5.2.5 Sequences

Sequences of elements of arbitrary length on a secondary storage are represented as files. File management is ignored here, being out of scope. Strings, as supported by many languages, are just array of characters. In other languages, such as SNOBOL4 and Algol 68, strings may vary arbitrarily in length, having no programmer-specified upper bound. This kind of array with dynamically changing size must be allocated in the heap (see Chapter 2, Section 2.5.2.6).

3.5.2.6 Classes

User-defined types specified via the simple class construct introduced in Section 3.2.8 are easy to represent as extensions of structures. The differences are:

1. only public fields are visible outside the construct. Such fields must be tagged as public in the descriptor.
2. some fields may be routines. It is thus necessary to be able to store routine signatures in descriptors.

The reader can easily extend the representation scheme presented in Section 3.5.2.1 to keep these new requirements into account. Since the code of the routines encapsulated in a class is shared by all class instances, routine fields are represented as pointers to the routines.

Objects that are instances a new type defined by a class are treated as any other data object. Some languages allow the programmer to choose whether class-defined objects must be allocated on the stack or on the heap. For example, in C++ after class point is declared as in Figure 33, the following declarations are possible in some function f:

```
point x (1.3, 3.7);
point* p = new point (1.1, 0.0);
```

In the first case, x is a named variable that is allocated in f’s activation record on the stack. In the second, p is allocated in f’s activation record, while the data structure for the point is allocated on the heap. Heap management is discussed next.

3.5.2.7 Pointers and garbage collection

A pointer holds as a value the address of an object of the type to which the pointer is bound to. Pointers usually can have a special null value (e.g., void in C and C++, nil in Pascal). Such a null value can be represented by an address value that would cause a hardware-generated error trap to catch an inadvertent reference via a pointer with null value. For example, the value might be an address beyond the physical addressing space into a protected area.

Pointer variables are allocated as any other variable on the activation record stack. Data objects that are allocated via the run-time allocation primitive new are allocated in the heap. Heap management is a crucial aspect of a programming language implementation. In fact, the free storage might quickly be exhausted, unless there is some way of returning allocated storage to the free area.

The memory allocated to hold a heap object can be released for later reuse if the object is no longer accessible. Such an object is said to be *garbage*. Precisely, an object is garbage if no variable in the stack points to it and–recursively–no other garbage points to it. There are two main aspects involved in heap management. One is the recognition that some portion of allocated memory is garbage; the other is to recycle such memory for new memory allocation requests issued by the user.

Garbage recognition often relies on the programmer, who is required to notify the system that a given object became useless. For example, in order to state that the object pointed by p is garbage, in C++ one would write delete(p) and in Pascal one would write dispose(p). As we mentioned, there is no guarantee that only objects that become unreferenced be defined as garbage: the correct use of delete (or dispose) entirely relies on the programmer's responsibility. Should the programmer inadvertently mark an object as garbage, the pointer to it becomes dangling.

Another strategy is to let the run-time system take care of discovering garbage, by means of a suitable automatic *garbage collection* algorithm. Automatic garbage collection is vital for languages that make heavy use of dynamically generated variables, like LISP. In fact, garbage collection was invented for implementing the first LISP systems. Eiffel, which uniformly treats all objects as referenced by pointers, provides an automatic garbage collector. Although the subject is usually treated in courses on data structures, we provide a brief and simplified view of possible strategies for automatic garbage collection.

Garbage collection can be performed incrementally by using a *reference counting* scheme. Under such a scheme, each heap allocated object is supposed to have an extra descriptor field, to store the current number of pointers to it. Whenever a pointer variable is deallocated from the stack, the reference count of the heap object pointed by it is decreased by one. When the reference count becomes zero, the object is declared to be garbage, and the reference count of any object pointed by it is decreased by one. This method thus releases an object as soon as it is found to become unreferenced. The problem with this method, however, is that it does not work for circular heap data structures (see Figure 50). If a pointer to the head of a circular list is deallocated, the nodes of the list are not found to be garbage, because the reference count for each node is one.

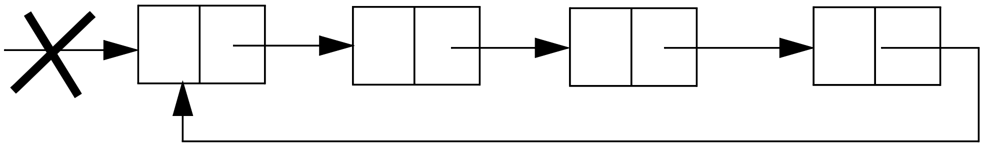


FIGURE 50. A circular heap data structure

A non-incremental strategy for automatic garbage collection consists of allocating free cells from the heap until the free space is exhausted. Only at that point the system enters a garbage collecting phase. We describe one such scheme under the simplifying assumption that:

- the heap data objects have fixed size
- it is known a-priori which fields of a heap object contain pointers to other heap data objects, and
- it is possible to find all the pointers from the stack into the heap.

The following method for garbage collection allows all reachable heap data objects to be distinguished from garbage objects. To do so, a working set of pointers T may be used. Initially, T contains the stack values which point into the heap. An element E is repeatedly extracted from T, the objects referenced by E are marked, and E is replaced by the pointers to the node(s) contained in

E, if they are not marked. When T becomes empty, all reachable heap data objects have been marked. All other objects are garbage.

A number of variations have been proposed in the literature to make this garbage collection method more efficient. Its main problem, however, is that “useful” processing time is lost when the garbage collector is invoked. In an interactive system, this may be perceived by the programmer as an unexpected slow-down of the application, which occurs at unpredictable times. In a real-time system, this can be particularly dangerous, because an urgent request for service might arrive from the environment just after the garbage collector has started its rather complex activity. Garbage collection time is distributed more uniformly over processing time by using the reference counting scheme, but unfortunately such scheme works only partially. An appealing solution, which cannot be reviewed here, is based on a parallel execution scheme, where the garbage collector and the normal language processor execute in parallel.

Having discovered which heap data objects are garbage (either by explicit notification by the programmer, or by running a garbage collector), one should decide how to recycle them by adding them to the free storage. One possibility is to link all free areas in a free list. In such a case, each block would contain at least two cells: one for the block size, and one for the pointer to the next block. It is convenient to keep the list ordered by increasing block address, so that as a block is ready to be added to the list, it can be merged with possible adjacent blocks in the list. As a new storage area is to be allocated, the free list is searched for a matching block., according to some policy.

3.6 Bibliographic notes

The systematic view of data aggregates and the classification of type constructors presented in Section 3.2 is taken from a paper of C.A.R. Hoare in (Dahl et al. 1972).

Programming language research in the 70’s emphasized the need for taming (or eliminating) insecure constructs, which can cause programs to be difficult to write and evaluate, and therefore potentially unreliable. Several works concentrated on evaluating the type structure of existing languages to find insecurities. For example, (Welsh et al. 1977) and (Tennent 1978) provide a critical assessment of the original Pascal definition. Another research direction concentrated on new language constructs that could solve the insecurities that were found in existing languages. The work on Euclid (Popek et al. 1977), which was briefly illustrated in this chapter, is a notable example of this research stream. Other language experiments emphasized the need to support program reliability through language constructs that favor modularity and information hiding. The concept of abstract data type was introduced, and languages like CLU (Liskov and Zilles 1975, Liskov and ***) were designed and implemented to support program decomposition through abstract data types. CLU, as we mentioned, is based on the early seminal work on SIMULA 67 (Dahl et al. 1970). The SIMULA 67 experience, and the language experiments made in the 70's, can be viewed as the origin of later developments both in the direction of modular languages (like Modula-2 and Ada) and object-oriented languages (like C++, Smalltalk and Eiffel). More on these concepts will be discussed in Chapters 6 and 7.

Language experiments and developments proceeded in parallel with more theoretical work which laid the semantic foundations of the concepts of type, subtype, polymorphism, strong typing, etc. We gave a cursory introduction to such work in Section 3.3. The interested reader should refer to (Cardelli and Wegner 1985) for a deeper treatment of the subject. (Cleveland 1986) is another good source for many of these concepts.

For a detailed understanding of the type systems of different languages, the reader should refer to the language manuals referenced in the Glossary. Implementation models for data objects are analyzed in the aforementioned paper by Hoare in (Dahl et al. 1972) and in most compiler textbooks, such as (Aho et al. 1986) and (Fisher and LeBlanc 1988). Garbage collection is surveyed in (Cohen 1981) and (Appel 1990). It is also briefly discussed in most textbooks on data structures, such as (Wood 1993).

3.7 Exercises

1. Ada supports discriminated unions through variant records. Write a short report describing how Ada does this, and how it differs from Pascal and C.
2. Discuss the possible strategies adopted by a programming language to bind a finite mapping to a specific finite domain (i.e., to bind an array to a specific size). Also, give examples of languages adopting each different strategy.
3. Show how a variant record type in Pascal or Ada can be used to define a binary tree of either integer or string values.

4. Write a short report illustrating how array manipulation facilities are richer in Ada than in Pascal (or C).
5. What is a dangling reference? How can it arise during execution? Why is it dangerous?
6. Consider the C++ class in Figure 3.2. What happens if the size of the array used to store the stack is exceeded by the push operation?
7. Add assertions to the Eiffel program in Figure 3.5. Discuss what happens when the length of the array is exceeded by a call to push, and what happens when pop is called for an empty stack.
8. Eiffel does not provide destructor operations (similar to C++). What does the language do when an object should be destroyed? When does this happen in Eiffel?
9. Define a C++ or an Eiffel implementation for the a generic abstract data type defining a first-in first-out queue, whose operations have the following signatures

```
enqueue: fifo (T) x T -> fifo (T) --adds an element to the queue
dequeue: fifo (T) -> fifo (T) x T --extracts an element from the queue
length: fifo (T) -> int --computes the length of the queue
```

10. Discuss the truth or falsity of the following statement, and discuss its relevance. “A program can be unsafe and yet execute without type errors for all possible input data.”
11. An index check verifies that the index of an array is in the bounds declared for the array. Can index check be performed statically? Why? Why not?
12. What is a strong type system? Why is it useful?
13. Is a static type system strong? And, conversely, is a strong type system static?
14. What kind of type compatibility does the typedef construct of C introduce?
15. In Section 3.3.4, we made the following statement “Unexpected difficulties, in particular, arise because of the interaction between coercions and overloading of operators and routines.” Provide examples that justify this statement.
16. Define monomeric and polymorphic type systems.
17. Compare genericity in Ada (or C++) and in ML. Which can be defined as an example of parametric polymorphism?
18. Check in the Pascal manual if procedures and functions can be overloaded.
19. Justify through examples the following statement on Pascal “since subtypes are defined by the language as new types, strong typing is not strictly enforced by the language”.
20. In C++, what is the difference between assigning a value to a pointer or to a reference?
21. In C++, what is the difference between taking the address (via operator &) of a pointer or of a reference?
22. Figure 5.9, which describes the C++ type system, shows the existence of pointers to class members. Study the language manual and provide an example that shows the use of this feature.
23. In C++, each class has an associated default assignment operator which does a memberwise copy of the source to the target. But when a new object of class T is declared this way:

```
T x = y;
```

    the copy constructor of x is used to construct x out of y. Why can the assignment operation not be used? What is the difference between assignment and copy construction?

24. In Eiffel, each object has a feature called copy which is used for assignment to the object. For example, a.copy(b) assigns the value of object b
25. Write a short report comparing variant records in Pascal, C++, and Ada.

26. Discuss how storage is allocated for class instances in C++ and in Eiffel.
27. Justify or confute the following statement: “Ada attributes support program portability”.
28. How can the following union variable X be represented?

```
type X_type is union Y_type, W_type;
type Y_type is float array [0. .10];
type W_type is struct{
    float A;
    int B;
};
X_type X;
```

29. Instead of having only one free list for unused heap storage areas, one could keep several free lists, one for each object type. Discuss advantages and disadvantages of this alternative implementation.
30. Write a recursive algorithm to do the marking of all reachable heap objects.
31. Write a short report on possible different policies for extracting a block from the heap free list as a new storage area needs to be allocated. In order to survey the possible solutions, you may refer to books on data structures.
32. The simplest possible reaction of the run-time system to a statement like dispose (in Pascal) is to ignore it. That is, storage is not freed for later reuse. How would you design an experiment to check what a language implementation actually does? Perform the experiment on an available implementation of Pascal
33. Referring to the implementation schemes discussed in Section 3.5, write an abstract algorithm to perform structural type compatibility. Hint: Be careful not to cause the algorithm to loop forever.

Structuring the computation

CHAPTER 4

This chapter is devoted to a detailed analysis of how computations are structured in a programming languages in terms of the flow of control among the different components of a program. We start in Section 4.1 with a discussion of the elementary constituents of any program: expressions (which play a fundamental role in all programming languages, including functional and logic) and statements (which are typical of conventional statement-based languages). Our discussion will then be based primarily on conventional programming languages. We will first analyze *statement-level control structures* (Section 4.2), which describe how individual statements can be organized into various patterns of control flow to construct program units.

Programs are often decomposed into units. For example, routines provide a way of hierarchically decomposing a program into units representing new complex operations. Once program units are constructed, it becomes necessary to structure the flow of computation among such units. Different kinds of *unit-level control structures* are discussed in Section 4.3 through Section 4.8. The simplest kind of unit-level control regime is the *routine call* and *return* (Section 4.3). Another kind of control regime is *exception handling* (Section 4.4), which supports the ability to deal with anomalous situations which may arise even asynchronously, as the program is being executed. Features supporting advanced control regimes are then introduced in Section 4.5 (*pattern matching*, which supports case-based analysis), Section 4.6 (*nondeterminism* and *backtracking*), and Section 4.7 (*event-driven* control structures). Such features are quite common in nonprocedural languages, like ML and PROLOG, and will in fact be taken up in the presentation of such languages in Chapters 7 and 8. Finally, Section 4.8 provides an introduction to the control structures needed for concurrent programming, where units execute largely independently.

4.1 Expressions and statements

Expressions define how a value can be obtained by combining other values through operators. The values from which expresions are evaluated are either denoted by a literal, as in the case of the real value 57.73, or they are the r_value of a variable.

Operators appearing in an expression denote mathematical functions. They are characterized by their *aritiy* (i.e., number of operands) and are invoked using the function's signature. A unary operator is applied to only one operand. A binary operator is applied to two operands. In general, a n-ary operator is applied to n operands. For example, '-' can be used as a unary operator to transform–say–the value of a positive expression into a negative value. In general, however, it is used as a binary operator to subtract the value of one expression from the value of another expression. Functional routine invocations can be viewed as n-ary operators, where n is the number of parameters.

Regarding the operator's notation, one can distinguish between infix, prefix, and postfix. *Infix notation* is the most common notation for binary operators: the operator is written between its two operands, as in x + y. Postfix and prefix notations are common especially for non-binary operators. In *prefix notation*, the operator appears first, and then the operands follow. This is the conventional form of function invocation, where the function name denotes the operator. In *postfix notation* the operands are followed by the corresponding operator. Assuming that the arity of each operator is fixed and known, expressions in prefix and postfix forms may be written without resorting to parentheses to specify subexpressions that are to be evaluated first. For example, the infix expression

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

can be written in prefix form as

```
* a + b c
```

and in postfix form as

```
a b c + *
```

In C, the increment and decrement unary operators ++ and -- can be written both in prefix and in postfix notation. The semantics of the two forms, however, is different; that is, they denote two distinct operators. Both expressions ++k and k++ have the side effect that the stored value of k is incremented by one. In the former case, the value of the expression is the value of k incremented by one (i.e., first, the stored value of k is incremented, and then the value of k is provided as the value of the expression). In the latter case, the value of the expression is the value of k before being incremented.

Infix notation is the most natural one to use for binary operators, since it allows programs to be written as conventional mathematical expressions. Although the programmer may use parentheses to explicitly group subexpressions that must be evaluated first, programming languages complicate matters by introducing their own conventions for operator associativity and precedence. Indeed, this is done to facilitate the programmer's task of writing expressions by reducing redundancy, but often this can generate confusion and make expressions less understandable, especially when switching languages. For example, the convention adopted by most languages is such that

```
a + b * c
```

is interpreted implicitly as

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

i.e., multiplication has precedence over binary addition (as in standard mathematics). However, consider the Pascal expression

```
a = b < c
```

and the C expression

```
a == b < c
```

In Pascal, operators < and = have the same precedence, and the language specifies that application of operators with the same precedence proceeds left to right. The meaning of the above expression is that the result of the equality test (a=b), which is a boolean value, is compared with the value of c (which must be a boolean variable). In Pascal, FALSE is assumed to be less than TRUE, so the expression yields TRUE only if a is not equal to b, and c is TRUE; it yelds FALSE in all other cases. For example, if a, b and c are all FALSE, the expression yields FALSE.

In C, operator "less than" (<) has higher precedence than "equal" (==). Thus, first b < c is evaluated. Such partial result is then compared for equality with the value of a. For example, assuming a = b = c = false (represented in C as zero), the evaluation of the expression yields 1, which in C stands for true.

Some languages, like C++ and Ada, allow operators to be programmer defined. For example, having defined a new type Set, one can define the operators + for set union and - for set difference. The ability of providing programmer-defined operators, as any other feature that is based on overloading, can in some cases make programs easier to read, and in other cases harder. Readability is improved since the programmer is allowed to use familiar standard operators and the infix notation also for newly defined types. The effect of this feature, however, is such that several actions happen behind the scenes when the program is processed. This is good whenever what happens behind the scenes matches the programmer's intuition; it is bad whenever the effects are obscure or counterintuitive to the programmer.

Some programming languages support the ability of writing conditional expressions, i.e., expressions that are composed of subexpressions, of which only one is to be evaluated, depending on the value of a condition. For example, in C one can write

```
(a > b) ? a : b
```

which would be written in a perhaps more conventionally understandable form in ML as

```
if a > b then a else b
```

to yield the maximum of the values of a and b.

ML allows for more general conditional expressions to be written using the "case" constructor, as shown by the following simple example.

```
case x of
     1 => f1 (y)
|    2 => f2 (y)
|    _ => g (y)
```

In the example, the value yielded by the expression is f1 (y) if x = 1, f2 (y) if x = 2, g (y) otherwise.

Functional programming languages are based heavily on expressions. In such languages, a program is itself an expression, defined by a function applied to operands, which may themselves be defined by functions applied to operands. Conventional languages, instead, make the values of expressions visible as a modification of the computation's state, through assignment of expressions to variables. An *assignment statement*, like x = y + z in C, changes the state by associating a new r_value with x, computed as y + z. To evaluate the expression, the r_values of variables y and z are used. The result of the expression (an r_value) is then assigned to a memory location by using the l_value of x. Since the assignment changes the state of the computation, the statement that executes next operates in the new state. Often, the next statement to be executed is the one that textually follows the one that just completed its execution. This is the case of a sequence of statements, which is represented in C as

```
statement_1;
statement_2;
. . .
statement_n;
```

The sequence can be made into a compound statement by enclosing it between the pair of brackets { and }. In other languages, like Pascal and Ada, the keywords begin and end are used instead of brackets.

In many conventional programming languages, like Pascal, the distinction between assignment statements and expressions is sharp. In others, like C, an assignment statement is actually an expression with a side-effect. The value returned by an assignment statement is the one that is stored in the left operand of the assignment operator "=". A typical example is given by the following loop which reads successive input characters until the end of file is encountered:

```
while ((c = getchar ( )) != EOF)
    /* assigns the character read to c and yields the read value, which is compared to the
end of file symbol */
    . . .
```

Furthermore, in C the assignment operator associates from right to left. That is, the statement

```
a = b = c = 0;
```

is interpreted as

```
a = (b = (c = 0))
```

Many programing languages, like Pascal, require the left-hand side of an assignment operator to be a simple denotation for an l_value. For example, it can be a variable name, or an array element, or the cell pointed by some variable. More generally, other languages, like C, allow any expression yielding a modifiable l_value to appear on the left-hand side. Thus, it is possible to write the following kind of statement

```
( p > q) ? p* : q* = 0;
```

which sets to zero the element pointed by the maximum of p and q.

As another example, one can write

```
*p++ = *q++;
```

The right-hand side expression yields the value pointed by q. The left-hand side is an expression which provides the r_value of p, which is the a reference, i.e., an l_value. So the overall effect is that the value of the object pointed by q is copied into the object pointed by p. Both pointers are also incremented as a side effect. Since the above assignment is an expression, the value of the expression is that of the object pointed by q. For example, the following concise piece of code copies a sequence of integers terminated by zero pointed by p into a sequence pointed by q.

```
while ((*p++ = *q++) != 0) { };
```

Sequences, as shown before, are the simplest form of compound statements. Often, the syntax of the language requires each statement in a sequence to be saparated from the next by a semicolon. For example, in Pascal a sequence can be written as:

```
begin
    stat_1;
    stat_2;
     . .
    stat_n
end
```

Other languages, instead, require each statement to be terminated by a semicolon, and therefore do not need any special separator symbol. For example, in C we would write

```
{
    stat_1;
    stat_2:
. . .
```

```
        stat_n;
    }
```

Although the choice between the two syntactic forms has no deep implications, pragmatically the latter can be more convenient, because one does not need to distinguish between the last statement of a sequence (which does not require the separator, and any other statements, since all are terminated by a semicolon.

Programming languages provide other kinds of compound statements in addition to sequences. We will survey them in Section 5.2. In the rest of this chapter, we implicitly concentrate on conventional languages, unless explicitly stated otherwise. Functional languages, which are not based on computations defined by successive state changes, will be studied in Chapter 7.

4.2 Conditional execution and iteration

Conditional execution of different statements can be specified in most languages by the if statement. Languages differ in several important syntactic details concerning the way such a construct is offered. Semantically, however, they are all alike. Syntactic details are not irrelevant: as we mentioned in Section 3.1.1, the syntactic appearance of a program may contribute to its readability, ease of change and, ultimately, to its reliability.

Let us start with the example of the if statement as originally provided by Algol 60. Two forms are possible, as shown by the following examples:

```
    if i = 0                          if i = 0
        then i := j;                      then i := j
                                          else begin   i := i + 1;
                                                       j := j - 1
                                          end
```

In the first case, no alternative is specified for the case i ¦ 0, and thus nothing happens if i ¦ 0. In the latter, two alternatives are present. Since the case where i ¦ 0 is described by a sequence, it must be made into a compound statement by bracketing it between begin and end.

The selection statement of Algol 60 raises a well-known ambiguity problem, illustrated by the following example

```
    if x > 0 then if x < 10 then x := 0 else x := 1000
```

It is unclear if the else alternative is part of the innermost conditional (**if** x < 10

. . .) or the outermost conditional (if x > 0 . . .). The execution of the above statement with x = 15 would assign 1000 to x under one interpretation, but leave it unchanged under the other. To eliminate ambiguity, Algol 60 requires an unconditional statement in the then branch of an if statement. Thus the above statement must be replaced by either

```
if x > 0 then begin if x < 10 then x := 0 else x := 1000 end
```

or

```
if x > 0 then begin if x < 10 then x := 0 end else x := 1000
```

The same problem is solved in C and Pascal by automatically matching an else branch to the closest conditional without an else. Even though this rule removes ambiguity, however, nested if statements are difficult to read, especially if the program is written without careful indentation (as shown above).

A syntactic variation that avoids this problem is adopted by Algol 68, Ada, and Modula-2, which use a special keyword as an enclosing final bracket of the if statement (fi in the case of Algol 68, end if in the case of Ada, end in the case of Modula-2). Thus, the above examples would be coded in Modula-2 as

```
if i = 0
    then i := j
    else   i := i + 1;
           j := j - 1
end
```

and

```
if x > 0 then  if x < 10 then x := 0 else x := 1000 end end
```

or

```
if x > 0 then if x < 10 then x := 0 end else x := 1000 end
```

depending on the desired interpretation.

Choosing among more than two alternatives using only if-then-else statements may lead to awkward constructions, such as

```
if a
    then S1
    else
        if b
          then S2
          else
```

```
                if c
                    then S3
                    else S4
                end
        end
end
```

To solve this syntactic inconvenience, Modula-2 has an else-if construct that also serves as an end bracket for the previous if. Thus the above fragment may be written as

```
if a
    then S1
else if b
    then S2
else if c
    then S3
else S4
end
```

C, Algol 68, and Ada provide similar abbreviations.

Most languages also provide an ad-hoc construct to express multiple-choice selection. For example, C++ provides the switch construct, illustrated by the following fragment:

```
switch (operator) {
    case ’+’:
        result = operand1 + operand2;
        break;
    case ’*’:
        result = operand1 * operand2;
        break;
    case ’-’:
        result = operand1 - operand2;
        break;
    case ’/’:
        result = operand1 / operand2;
        break;
    default:
        break; --do nothing
};
```

Each branch is labelled by one (or more) constant values. Based on the value of the switch expression, the branch labelled by the same value is selected. If the value of the switch expression does not match any of the labels, the (optional) default branch is executed. If the default branch is not present, no action takes place. The order in which the branches appear in the text is immaterial. In the above example, an explicit break statement is used to terminate each branch; otherwise execution would fall into the next branch.

The same example may be written in Ada as

```
case OPERATOR is
    when '+' =>  result = operand1 + operand2;
    when '*' =>  result = operand1 * operand2;
    when '-' =>  result = operand1 - operand2;
    when '/' =>  result = operand1 / operand2;
    when others => null;
end case
```

In Ada, after the selected branch is executed, the entire case statement terminates.

Iteration allows a number of actions to be executed repeatedly. Most programming languages provide different kinds of *loop constructs* to define iteration of actions (called the *loop body*). Often, they distinguish between loops where the number of repetitions is known at the start of the loop, and loops where the body is executed repeatedly as long as a condition is met. The former kind of loop is usually called a *for loop*; the latter is often called the *while loop*.

For-loops are named after a common statement provided by languages of the Algol family. For statements define a *control variable* which assumes all values of a given predefined sequence, one after the other. For each value, the loop body is executed.

Pascal allows iterations where control variables can be of any ordinal type: integer, boolean, character, enumeration, or subranges of them. A loop has the following general appearance:

```
for loop_ctr_var := lower_bound to upper_bound do statement
```

A control variable assumes all of its values from the lower to the upper bound. The language prescribes that the control variable and its lower and upper bounds must not be altered in the loop. The value of the control variable is also assumed to be undefined outside the loop.

As an example, consider the following fragment:

```
type day = (sun, mon, tue, wed, thu, fri, sat);
var week_day: day;
```

```
        . . .
        for week_day := mon to fri do . . .
```

As another example, let us consider how for-loops can be written in C++, by examining the following fragment, where the loop body is executed for all values of i from 0 to 9

```
    for (int i = 0; i < 10; i++) {. . .}
```

The statement is clearly composed of three parts: an initialization and two expressions. The initialization provides the initial state for the loop execution. The first of the two expressions specifies a test, made before each iteration, which causes the loop to be exited if the expression becomes zero (i.e., false). The second specifies the incrementing that is performed after each iteration. In the example, the statement also declares a variable i. Such variable's scope extends to the end of the block enclosing the for statement.

In C++, either or both of the expressions in a for loop can be omited. This is used to write an endless loop, as

```
    for ( ; ; ) { . . .}
```

While loops are also named after a common statement provided by languages of the Algol family. A while loop describes any number of iterations of the loop body, including zero. They have the following general form

```
    while condition do statement
```

For example, the following Pascal fragment describes the evaluation of the greatest common divisor of two variables a and b using Euclid's algorithm

```
    while a ¦ b do
        begin
            if a > b then
                a := a - b
            else
                b := b - a
        end
```

The end condition (a ¦ b) is evaluated before executing the body of the loop. The loop is exited if a is equal to b, since in such case a is the greatest common divisor. Therefore, the program works also when it is executed in the special case where the initial values of the two variables are equal.

In C++, while statements are similar. The general form is:

```
    while (expression) statement
```

Another way to write an endless loop in C++ is therefore

```
while (1) {. . .}
```

Often languages provide another similar kind of loop, where the loop control variable is checked at the end of the body. In Pascal, the construct has the following general form

```
repeat
    statement
until condition
```

In a Pascal repeat loop, the body is iterated as long as the condition evaluates to false. When it becomes true, the loop is exited.

C++ provides a do-while statement which behaves in a similar way:

```
do statement while (expression);
```

In this case the statement is executed repeatedly until the value of the expression becomes zero (i.e., the condition is false).

Ada has only one general loop structure, with the following form

```
iteration_specification loop
    loop_body
end loop
```

where iteration_specification is either

```
while condition
```

or

```
for counting_var in discrete_range
```

or

```
for counting_var in reverse discrete_range
```

An example is provided by the following fragment:

```
for K in Index_Range while A (K) /= 0 do
    B (K) := B (K) / A (K);
```

Endless loops are easy to write, since iteration_specification is optional. In addition, loops can be terminated by an unconditional exit statement

```
exit;
```

or by a conditional exit statement

```
    exit when condition
```

If the loop is nested within other loops, it is possible to exit an inner loop and any number of enclosing loops.

```
    Main_Loop:
        loop
            . . .
            loop
                . . .
                exit Main_Loop when A = 0;
                . . .
            end loop;
            . . .
    end loop Main_Loop;
    -- after the exit statement execution continues here
```

In the example, control is transferred to the statement following the end of Main_Loop when A is found to be equal to zero in the inner loop. The exit statement is used to specify a premature termination of a loop.

C++ provides a break statement, which causes termination of the smallest enclosing loop and passes control to the statement following the terminated statement, if any. It also provides a continue statement, which causes the termination of the current iteration of a loop and continuation from the next iteration (if there is one). A continue statement can appear in any kind of loop (for loop, and both kinds of while loops).

In some cases, it is useful to allow the programmer to define a mechanism to step through the elements of a given collection. To do so, a programming language might provide support for user-defined control structures, in much the same way as it provides support for user-defined types and operations. For example, having defined a set, the programmer might need to sequence through all elements in the set. User-defined control structures which sequence through elements of user-defined collections are sometimes called *iterators*. Languages providing constructs for the implementation of abstract data types easily allow iterators to be defined. For example, in C++ let the generic "collection of elements of type T" be defined by a template. To define an iterator, we can design three operations that are exported by the template: start ( ), which initializes the loop by positioning a cursor on the first element of the collection (if any), more ( ), which yields true if there are elements left to examine in the collection, and next ( ), which yields the current element and positions the cursor on the next element of the collection (if any). A typical iteration on an instantiated collection X of elements of type T would be

```
T y;
. . . ;
X.start ( );
while (X . more ( )) {
    y = X . next ( );
    . . . // manipulate y
};
```

This solution works for user-defined types, provided they define operations start, more, and next. It does not work for collections defined by built-in constructors (such as arrays), for which these operations are not defined. In Chapter 5, we will see a more general way of defining iterators which work for any kinds of collections.

4.3 Routines

Routines are a program decomposition mechanism which allows programs to be broken into several units. Routine calls are control structures that govern the flow of control among program units. The relationships among routines defined by calls are asymmetric: the caller transfers control to the callee by naming it explicitly. The callee transfers control back to the caller without naming it. The unit to which control is transfered when a routine R terminates is always the one that was executing immediately before R. Routines are used to define abstract operations. Most modern languages allow such abstract operations to be defined recursively. Moreover, many such languages allow generic operations to be defined.

Chapter 2 presented the basic runtime modeling issues of routine activation, return, and parameter passing. In this section we review how routines can be written in different languages and what style issues arise in properly structuring programs.

Most languages distinguish between two kinds of routines: procedures and functions. A *procedure* does not return a value: it is an abstract command which is called to cause some desired state change. The state may change because the value of some parameters transmitted to the procedure gets modified, or because some nonlocal variables are updated by the procedure, or because some actions are performed on the external environment (e.g., reading or writing). A *function* corresponds to its mathematical counterpart: its activation is supposed to return a value, which depends on the value of the transmitted parameters.

Pascal provides both procedures and functions. It allows formal parameters to be either by value or by reference. It also allows procedures and functions to be parameters, as shown by the following example of a procedure header:

```
procedure example (var x: T; y: Q; function f (z: R): integer);
```

In the example, x is a by-reference parameter of type T; y is a by-value parameter of type Q; f is a function parameter which takes one by-value parameter z of type R and returns an integer.

Ada provides both procedures and functions. Parameter passing mode is specified in the header of an Ada routine as either in, out, or in out. If the mode is not specified, in is assumed by default. A formal in parameter is a constant which only permits reading of the value of the corresponding actual parameter. A formal in out parameter is a variable and permits both reading and updating of the value of the associated actual parameter. A formal out parameter is a variable and permits updating of the value of the associated actual parameter. In the implementation, parameters are passed either by copy or by reference. Except for cases that are explicitly stated in the language standard, it is left to the implementation to choose whether a parameter should be passed by reference or by copy. As we discussed in Section 3.6.6, in the presence of aliasing, the two implementations may produce different results. In such a case, Ada defines the program to be erroneous; but, unfortunately, the error can only be discovered at run time.

In C all routines are functional, i.e., they return a value, unless the return type is void, which states explicitly that no value is returned. Parameters can only be passed by value. It is possible, however, to achive the effect of call by reference through the use of pointers. For example, the following routine

```
void proc (int* x, int y);
{
    *x = *x + y;
}
```

increments the object referenced by x by the value of y. If we call proc as follows

```
proc (&a, b);  /*  &a means the address of a */
```

x is initialized to point to a, and the routine increments a by the value of b.

C++ introduced a way of directly specifying call by reference. This frees the programmer from the lower level use of pointers to simulate call by reference. The previous example would be written in C++ as follows.

```
void proc (int& x, int y);
{
    x = x + y;
}

proc (a, b); -- no address operator is needed in the call
```

While Pascal only allows routines to be passed as parameters, C++ and Ada get closer to treating routines as first-class objects. For example, they provide pointers to routines, and allow pointers to be bound dynamically to different routines at run time.

4.3.0.1 Style issues: side effects and aliasing

In Chapter 3 we defined side effects as modifications of the nonlocal environment. Side effects are used principally to provide a method of communication among program units. Communication can be established through nonlocal variables. However, if the set of nonlocal variables used for this purpose is large and each unit has unrestricted access to the set of nonlocal variables, the program becomes difficult to read, understand, and modify. Each unit can potentially reference and update every variable in the nonlocal environment, perhaps in ways not intended for the variable. The problem is that once a global variable is used for communication, it is difficult to distinguish between desired and undesired side effects. For example, if unit u1 calls u2 and u2 inadvertently modifies a nonlocal variable x used for communication between units u3 and u4, the invocation of u2 produces an undesired side effect. Such errors are difficult to find and remove, because the symptoms are not easily traced to the cause of the error. (Note that a simple typing error could lead to this problem.) Another difficulty is that examination of the call instruction alone does not reveal the variables that can be affected by the call. This reduces the readability of programs because, in general, the entire program must be scanned to understand the effect of a call.

Communication via unrestricted access to nonlocal variables is particularly dangerous when the program is large and composed of several units that have been developed independently by several programmers. One way to reduce these difficulties is to use parameters as the only means of communication among units. The overhead caused by parameter passing is almost always tolerable, except for critical applications whose response times must be within severe bounds. Alternatively, it must be possible to restrict the set of nonlocal variables held in common by two units to exactly those needed for the communication between the units. Also, it can be useful to specify that a unit can only read, but not modify some variable.

Side effects also are used in passing parameters by reference. In such a case, a side effect is used to modify the actual parameter. The programmer must be careful not to produce undesired side effects on actual parameters. The same problem arises with call by name. A more substantial source of obscurity in call by name is that each assignment to the same formal parameter can affect different locations in the environment of the calling unit. Such problems do not arise in call by copy.

Languages that distinguish between functions and procedures suggest a programming style in which the use of side effects is restricted. Side effects are an acceptable programming practice for procedures. Indeed, this should be the way a procedure sends results back to the caller. Side effects, however, are unadvisable for function subprograms. In fact, function subprograms are invoked by writing the subprogram name within an expression, as in

```
    v : = x+ f (x, y) + z
```

In the presence of side effects –in Pascal, for example–the call to f might produce a change to x or y (if they are passed by reference), or even z (if z is global to the function) as a side effect. This reduces the readability of the program, since a reader expects a function to behave like a mathematical function. Also, one cannot rely on the commutativity of addition in general. In the example, if f modifies x as a side effect, the value produced for w is different if x is evaluated before or after calling f.

Besides affecting readability, side effects can prevent the compiler from generating optimized code for the evaluation of certain expressions. In the example

```
    u: = x+ z+ f (x, y) + f (x, y) + x+ z
```

the compiler cannot evaluate function f and subexpression x+ z just once.

The recognition that side effects on parameters are undesirable for functions affected the design of Ada, which allows only in formal parameters for functions.

In Chapter 2 we defined two variables to be *aliases* if they denote (*share*) the same data object during a unit activation. A modification of the data object under one variable name is automatically visible through all alias variables that share the object. An example is provided by the FORTRAN EQUIVALENCE statement. For instance, the statements

```
EQUIVALENCE (A, B)
A=5.4
```

bind the same data object to A and B and set its value to 5.4. Consequently, the statements

```
B=5.7
WRITE(6, 10)A
```

print 5.7, even though the value explicitly assigned to A was 5.4. The assignment to B affects both A and B.

As we observed in Chapter 2, aliasing may arise during the execution of a procedure when parameters are passed by reference. Consider the following C++ procedure, which is supposed to interchange the values of two integer variables without using any local variables.

```
void swap (int& x, y)
{
    x += y;
    y = x - y;
    x -=  y;
}
```

Before proceeding, examine the procedure and decide whether or not it works properly.

The answer is "generally yes"; in fact, the procedure works properly except when the two actual parameters are the same variable, as in the call

```
swap (a, a);
```

In this case, the procedure sets a to zero, because x and y become aliases and thus any assignments to x and y within the procedure affect the same location. The same problem may arise from the call

```
swap (b [ i ], b [ j ]);
```

when the index variables i and j happen to be equal.

Pointers can cause the same problems. In fact, the call

```
swap (*p, *q)
```

does not interchange the values pointed at by p and q if p and q happen to point to the same data object.

The above aliases occur because of the following two conditions.

- Formal and actual parameters share the same data objects; and
- Procedure calls have overlapping actual parameters.

Aliasing also may occur when a formal (by reference) parameter and a global variable denote the same or overlapping data objects. For example, if procedure swap is rewritten as

```
void swap (int & xr)
{
    x +=  a;
    a = x -  a;
    x -= a;
}
```

where a is a global variable, the call

```
swap (a)
```

generates an incorrect result, because of the aliasing between x and a. Aliasing does not arise if parameters are passed by value result; such parameters act as local variables within the procedure and the corresponding actual parameters become affected only at procedure exit. This is the reason of the semantic difference between call by reference and call by value-result.

The disadvantages of aliasing affect programmers, readers, and language implementers. Subprograms can become hard to understand because, occasionally, different names denote the same data object. This problem cannot be discovered by inspecting the subprogram: rather, discovery requires examining all the units that may invoke the subprogram. As a consequence of aliasing, a subprogram call may produce unexpected and incorrect results.

Aliasing also impairs the possibility of generating optimized code. For example, in the case

```
a := (x - y * z) + w;
b := (x - y * z) + u;
```

the subexpression x - y * z cannot be evaluated just once and then used in the two assignments if a is an alias for x, y, or z.

Although side effects and aliasing can cause difficulties and insecurities , programmers using conventional languages need to learn how to live with them. In fact, it is not possible to eliminate from a language all features which can cause them, such as pointers, reference parameters, global variables, and arrays. This would leave us with a very lean and impractical language indeed. Other approaches were taken in experimental languages (such as Euclid–see sidebar), but they did not become practically acceptable.

****sidebar start on Euclid

The approach taken by Euclid is to place restrictions on the use of such features as pointers, reference parameters, global variables, and arrays to rule out the possibility of aliasing. For reference parameters, the problems only arise if actual parameters are overlapping. If the actual parameters are simple variables, it is necessary to ensure that they are all distinct. Thus the procedure call

```
p(a, a)
```

is considered illegal by Euclid. Passing an array and one of its components also is prohibited. For example, the call

```
p (b [1], b)
```

to a procedure whose formal by-reference parameters are an integer x and an integer array y of indexes 1. .10 is illegal because y [1] and x are aliases. These forms of illegal aliasing can be caught at translation time.

However, the call

```
swap (b [i], b [j])
```

to the procedure swap generates aliasing only if i is equal to j. Euclid specifies that in such a case the condition i ¦ j be generated by the translator as a legality assertion. In the testing phase, legality assertions can be compiled automatically into run-time checks by using a suitable compiler option. If at run-time an assertion evaluates to false, execution is aborted and a suitable error message is produced. The main use of legality assertions, however, is in program verification. The Euclid system, in fact, includes a program verifier, and a Euclid program is considered correct only if the truth of all legality assertions is proven by the verifier.

Handling aliasing in the presence of pointers is more complex. Consider the following program fragment, written in C++ instead of Euclid for simplicity

```
T*  p, q;
p = new T;
q = p
```

The problem of aliasing between *p and *q is handled in the same way that arrays and array elements are handled, that is, *p and *q may be viewed as selectors that reference components of an implicitly defined collection of data–the set of all data objects of type T– the same way that b [i] and b [j] reference components of array b. An assignment to b [i] or b [j] is viewed as an assignment to the entire data object b, which happens to change the value stored in only one portion of b. Similarly, an assignment to *p or *q may be viewed as a modification of the set of components of type T.

This might appear to be an ingenious but tricky way of looking at the problem of aliasing for pointers. In fact, different data structures might be composed of dynamically generated components of the same type T. View-ing an assignment to *p as an assignment to the set of data objects of type T, that is, as a modification of any of such data structures, is not really helpful. To allow an extra level of checking for nonoverlapping pointers, Euclid introduces the concept of a *collection*. The programmer is required to divide all dynamic objects into separate collections and indicate which pointers can point into which collections. Each pointer can be bound to only one collection. An assignment between two pointers is legal only if the two pointers point into the same collection.

Detecting illegal aliasing between pointers caused by procedure calls is now similar to the case of arrays. In fact, a collection C and a pointer bound to C are similar to an array and a variable used as an index. Dereferencing is exactly like indexing within an array. For example, if p and q point into the same collection, and *p and *q are both passed, the nonoverlapping rule requires the test p ¦ q to be produced as a legality assertion.

Aliasing also can occur between global variables and formal parameters of a procedure. In Euclid, detection of aliasing in such cases does not require any additional work. In fact, global variables must be explicitly imported by a subprogram if they are needed, and they must be accessible in every scope from which the subprogram is called. For each imported variable, it is also necessary to indicate whether it can be read or written or both. Thus, modifiable global variables can be treated by the aliasing detection algorithm as implicit additional parameters passed by reference.

The explicit importation of global variables allows the programmer to restrict the set of variables visible within a procedure to any subset of the (non-masked) variables declared in the outer scopes. The translator thus can ensure that only visible variables are accessed in a unit and such accesses are legal, for example, that a read-only variable cannot be modified. This is an advantage over the pure ALGOL–like scope rules–especially for large programs, in which inner procedures automatically inherit all the (non-masked) variables declared in the enclosing scopes and can modify them in an uncontrolled way.

Finally, Euclid functions–as opposed to procedures–are not allowed to have by-reference parameters and can import only read variables. Thus, their execution cannot cause side effects, and they behave like mathematical functions.

An important consequence of disallowing aliasing in procedures is that passing parameters by reference is equivalent to passing them by value-result. Therefore, the choice of how to implement parameter passing can be made by the translator based exclusively on efficiency considerations.

The Euclid approach is certainly interesting, the adopted solutions are clean and favor reliable programming. Some restrictions imposed by Euclid cannot be enforced by a traditional compiler and require a program development environment that includes a program verifier. In particular, all legality assertions need to be proven by the verifier. This certainly adversely affected the practical acceptance of the language. More generally, Euclid is a good example to illustrate the tradeoffs that a language designer should achieve between freedom and flexibility, on the one side, and strict enforcement of programming discipline, on the other. Euclid goes definitely in the latter direction, whereas widely used languages like C++ go in the former.

4.4 Exceptions

Programmers often write programs under the optimistic assumption that nothing will go wrong when the program executes. Unfortunately, however, there are many reasons which may invalidate this assumption. For example, it may happen that under certain conditions an array is indexed with a value which exceeds the declared bounds. An arithmetic expression may cause a division by zero, or the square root operation may be executed with a negative argument. A request for new memory allocation issued by the run-time system might exceed the amount of storage available for the program execution. Or, finally, an embedded application might receive a message from the field which overrides a previously received message, before this message has been handled by the program.

Often programs fail unexpectedly, maybe simply displaying some obscure message, as an erroneous program state is entered. This behavior, however, is unacceptable in many cases. To improve reliability, it is necessary that such erroneous conditions can be recognized by the program, and certain actions are executed in response to the error. To do so, however, the conventional control structures we have discussed so far are simply inadequate. For example, to check that an index never exceeds the array bounds, one would need to explicitly test the value of the index before any indexing takes place, and insert appropriate response code in case the bounds are violated. Alternatively, one would like the run-time machine to be able to trap such anomalous condition, and let the response to it be programmable in the language. This would be more efficient under the assumption that bound violations are the exceptional case.

To cope with this problem, programming languages provide features for exception handling. According to the standard terminology, an *exception* denotes an undesirable, anomalous behavior which supposedly occurs rarely. The language can provide facilities to define exceptions, recognize them, and specify the response code that must be executed when the exception is raised (*exception handler*).

Exceptions have a wider meaning than merely computation errors. They refer to any kind of anomalous behavior that, intuitively and informally, corresponds to a deviation from the expected course of actions, as envisioned by the programmer. The concept of "deviation" cannot be stated in absolute and rigorous terms. It represents a design decision taken by the programmer, who decides that certain states are "normal", and "expected", while others are "anomalous". Thus, an exception does not necessarily mean that we are in the presence of a catastrophic error. It simply means that the unit being executed is unable to proceed in a manner that leads to its normal termination as specified by the programmer. For example, consider a control system which processes input messages defined by a given protocol. The normal course of actions consist of parsing the input message and performing some actions that depend on its contents. The arrival of a message which does not match the expected syntax might be considered as an exception, to be handled by an exception handler, a clearly identifiable piece of code that is separate from the rest of the program that handles the normal case.

Earlier programming languages (except PL/I) offered no special help in properly handling exceptional conditions. Most modern languages, however, provide systematic exception-handling features. With these features, the concern for anomalies may be moved out of the main line of program flow, so as not to obscure the basic algorithm.

To define exception handling, the following main decisions must be taken by a programming language designer:

4. What are the exceptions that can be handled? How can they be defined?

5. What units can raise an exception and how?

6. How and where can a handler be defined?

7. How does control flow after an exception is raised in order to reach its handler?

8. Where does control flow after an exception has been handled?

The solutions provided to such questions, which can differ from language to language, affect the semantics of exception handling, its usability, and its ease of implementation. In this section, we will analyze the solutions provided by C++, Ada, Eiffel, and ML. The exception handling facilities of PL/I and CLU are shown in sidebars.

4.4.1 Exception handling in Ada

Ada provides a set of four predefined exceptions that can be automatically trapped and raised by the underlying run-time machine:

- Constraint_Error: failure of a run-time check on a constraint, such as array index out of bounds, zero right operand of a division, etc.;
- Program_Error: failure of a run-time check on a language rule. For example, a function is required to complete normally by executing a return statement which transmits a result back to the caller. If this does not happen, the exception is raised;
- Storage_Error: failure of a run-time check on memory avaliability; for example, it may be raised by invocation of new;
- Tasking_Error: failure of a run-time check on the task system (see Section 5.8).

A program unit can declare new exceptions, such as

```
Help: exception;
```

which can be explicitly raised in their scope as

```
raise Help;
```

Once they are raised, built-in and programmer-defined exceptions behave in exactly the same way. Exception handlers can be attached to a subprogram body, a package body, or a block, after the keyword exception. For example

```
begin --this is a block with exception handlers
    ... statements ...
    exception   when Help =>handler for exception Help
                when Constraint_Error => handler for exception
                        Constraint_Error, which might be raised by a
                        division by zero
                when others => handler for any other exception that is not Help
                        nor Constraint_Error
end;
```

In the example, a list of handlers is attached to the block. The list is prefixed by the keyword **exception**, and each handler is prefixed by the keyword **when**.

If the unit that raises the exception provides a handler for it, control is transferred immediately to that handler: the actions following the point at which the exception is raised are skipped, the handler is executed, and then the program continues execution normally from the statement that follows the handler. If the currently executing unit U does not provide a handler, the unit terminates and the exception is *propagated*. The precise effect of termination and propagation depend on the kind of unit that raises the exception. If U is a block, its termination transfers control to the immediately enclosing unit, in which the exception is implicitly reraised. If U is a routine body, its termination causes the routine to return to the caller and the exception is implicitly reraised at the point of call. If U is a package body (see Chapter 5), it acts like a routine that is implicitly called when the package declaration is processed. If U is a task body (see Section 4.8) the exception is not propagated further; that is, the task terminates abnormally. If there is no handler associated with the package body, execution of the body is abandoned and execution continues in the unit that contains the package declaration, where the exception is implicitly reraised. In general, if a propagated exception is not handled at the point where it was transferred, it is further propagated, and this process might eventually lead to the end of the program. If a handler is found for an exception, after its execution the processing proceeds normally from the statement that follows the handler. Exceptions can also be explicitly reraised, via statement raise. For example, an exception handler that can only partly handle the exception might perform some recovery actions, and then might explicitly reraise the exception.

As an example, consider the program sketched in Figure 51. The figure shows the overall structure of the program, ignoring all internal details. In particular, procedures are described by showing the scope they define by using solid lines, while blocks' scopes are shown by dashed lines. Suppose that the following sequence of unit activations occurs:

- Main is activated
- block 1 is entered
- block 2 is entered
- Proc1 is called
- Proc2 is called
- block 3 is entered
- block 4 is entered

If an exception is raised at this stage, execution of block 4 is abandoned and a check is performed to see if the block provides an exception handler that can handle the exception. If a handler is found, the handler is executed and, if no further exceptions are raised, execution continues from the statements that

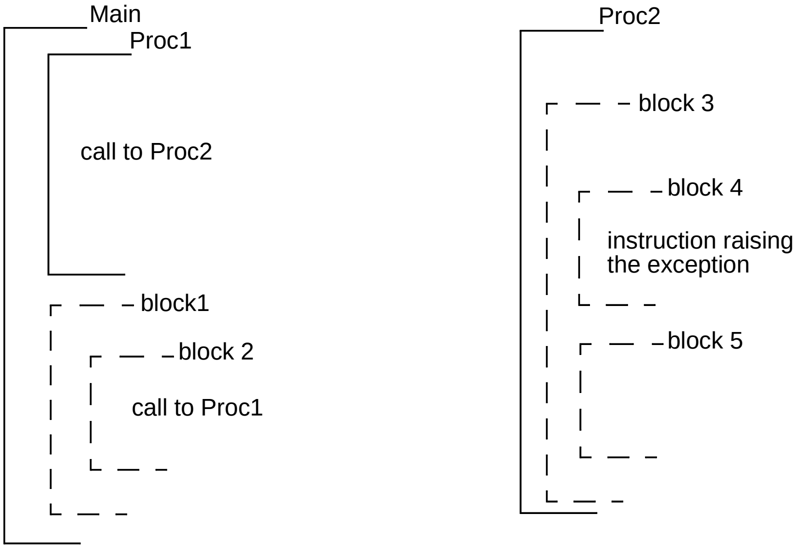


FIGURE 51. An example of an Ada program which raises an exception

follow block 4. If not, the exception is propagated to the enclosing block 3. That is, execution of block 3 is abandoned, and a check for an exception handler provided by block 3 is performed. If a handler is provided, and its execution terminates normally, procedure Proc2 returns to its caller normally. If not, the exception is propagated to the caller, and thus execution of block 2 is abandoned. If no exception handlers are provided by procedure Proc1, block 2, and block 1, eventually the Main program terminates abnormally.

To provide an abstract implementation model of exception handling, each declared exception (including built-in ones) can be bound to an internal exception name at compile time. The internal exception name should distinguish between two exceptions having the same name, but having different scopes. At run time, the binding between an exception code raised by a unit and the corresponding handler is dynamic, and follows the chain of unit activations. A possible solution consists of having in each activation record a fixed-contents *handler table,* which contains the descriptors of all the handlers that appear in the unit. (For simplicity, let us assume an implementation model of blocks which allocates a new activation record on the stack as the block is entered–see Chapter 2.) Each descriptor in the table contains

1. the internal exception name handled by the handler;
2. a pointer to the handler body.

When an exception is raised, its code is used to search for a handler in the handler table. If it is found there, control is transferred to its body. If not, the activation record is deleted from the stack, and the search is performed in the caller’s handler table using the address of the return point.

By unwinding the dynamic chain in the propagation process, an exception can be propagated outside its scope. In such a case, it can only be handled by a catch all handler (**when others** ...). The static scope rules of the language ensure that it cannot be handled by any other locally declared exception which, by coincidence, has the same name. The implementation scheme sketched above ensures this by giving them different internal exception names.

4.4.2 Exception handling in C++

Exceptions may be generated by the run-time environment (e.g., due to a division by zero) or may be explicitly raised by the program. An exception is raised by a throw instruction, which transfers an object to the corresponding handler. A handler may be attached to any piece of code (a block) which needs to be fault tolerant. To do so, the block must be prefixed by the keyword try. As an example, consider the following simple case:

```
class Help {. . . }; // objects of this class have a public attribute "kind" of type enumeration
                    // which describes the kind of help requested, and other public fields
which
                    // carry specific information about the point in the program where help
                    // is requested
class Zerodivide { }; // assume that objects of this class are generated by the run-time sys-
tem
. . .
try {
    // fault tolerant block of instructions which may raise help or zerodivide exceptions
    . . .
}
catch (Help msg) {
    // handles a Help request brought by object msg
    switch (msg.kind) {
    case MSG1:
        . . .;
    case MSG2:
        . . .;
    . . .
    }
    . . .
}
catch (Zerodivide) {
    // handles a zerodivide situation
    . . .
}
```

Suppose that the above try block contains the statement

```
throw Help (MSG1);
```

A throw expression causes the execution of the block to be abandoned, and control to be transferred to the appropriate handler. It also initializes a temporary object of the type of the operand of throw and uses the temporary to initialize the variable named in the handler. In the example, Help (MSG1) actually invokes the constructor of class Help passing a parameter which is used by the constructor to initialize field kind. The temporary object so created is used to initialize the formal parameter msg of the matching catch, and control is then transferred to the first branch (case MSG1) of the switch in the first handler attached to the block.

The above block might call routines which, in turn may raise exceptions. If one such routine raises a–say–help request and does not provide a handler for it, the routine's execution is abandoned and the exception is propagated to the point of call within the block. Execution of the block, in turn, is abandoned, and control is transferred to the handler as in the previous case. In other terms, C++, like Ada, propagates unhandled exceptions. Like Ada, a caught exception can be propagated explicitly, by simply saying throw. Also, as in Ada, after a handler is executed, execution continues from the statement that follows the one to which the matched handler is attached.

Unlike Ada, any amount of information can flow along with an exception. To raise an exception, in fact, one can throw an object, which contains data that can be used by the handler. For example, in the previous example, a help request was signalled by providing an object which contained specific information on the kind of help requested. If the data in the thrown object are not used by the handler, the catch statement can simply specify a type, without naming an object. This happens in our example for the division by zero.

C++ routines may list in their interface the exeception they may raise. This feature allows a programmer to state the intent of a routine in a precise way, by specifying both the expected normal behavior (the data it can accept and return), and its abnormal behaviors. For example

```
void foo ( ) throw (Help, Zerodivide);
```

might be the interface of a function foo which is called within the above fault tolerant block. Knowing that the used function foo may indeed raise exceptions, the client code may guard against anomalous behaviors by providing appropriate exception handling facilities, as we did.

The problem here is what happens if foo terminates by raising another exception that is not listed in its interface. This might happen, for example, because an error other than a division by zero is caught by the run-time machine (e.g., an underflow). In such a case, a special function unexpected ( ) is automatically called. Its default behavior, which could be redefined by the programmer, eventually causes abort ( ) to be called, which terminates the program execution.

The list of possible exceptions raised by a routine, however, is not required to be included in the routine interface. If no list is provided, it means that any possible exception can be propagated. Instead, if the empty list throw ( ) is provided, this means that no exception is propagated by the routine.

If an exception is repeatedly propagated and no matching handler is ever found, the special function terminate ( ) is called automatically. Its default behavior, which can be redefined by the programmer, eventually aborts the program execution.

Since the exceptions that can be raised in C++ are expressions of a given type, one can use the general facilities available to structure types (and abstract data types) to organize exceptions. For instance, one can use enumerations to structure and classify exceptions in groups. In the previous examples, if only the specific kind of needed help must be provided to handle exceptions of type Help, the following definition would suffice

```
enum Help {MSG1, MSG2, ...};
```

and the corresponding catch statement would be rewritten as

```
catch (Help msg) {
    switch (msg) {
    case MSG1:
        . . .;
    case MSG2:
        . . .;
    . . .
    }
    . . .
}
```

Other interesting ways of organizing exceptions can be achieved by organizing the corresponding classes according to subtype hierarchies, by means of subclasses (see Chapter 6).

Intuitively, an abstract implementation of the C++ mechanism can be similar to what we outlined for Ada. When an exception is raised, the dynamic chain is unwound until the appropriate handler is found. Further comments will be provided in Section 4.4.5.

4.4.3 Exception handling in Eiffel

The features provided by Eiffel to support exception handling have been strongly influenced by a set of underlying software design principles that programmers should follow. A key notion of such design principles is called the *contract*. Each software component has obligations with respect to other components, since such components may rely on it to provide their own services. Syntactically, such obligations are described by the interface of the component (a class), i.e., by the features exported to the other classes. Semantically, they are specified by the preconditions and postconditions of every exported routines and by the invariant condition. Once the program compiles correctly, syntactic obligations are guaranteed to have been verified. What may happen, however, is that semantic obligations are not fulfilled during execution. This is how exceptions may arise in Eiffel.

Thus, exceptions may arise in Eiffel because an assertion is violated (assuming that the program has been compiled under the option that sets runtime checking on). They can also arise because of anomalous states caught by the underlying abstract machine (memory exhausted, dereferencing an uninitialized pointer, ...). Finally, they can arise because a called routine fails (see below for what this means).

To respond to an exception, an exception handler (rescue clause) may be attached to any routine. There are two possible approaches to exception handling, which comply with the contract-based methodology underlying Eiffel programming. The first approach is called *organized panic*. Following this approach, the routine raising the exception fails; that is, as an exception is raised, the routine's execution is abandoned and control is transferred to the corresponding rescue clause, if any. The handler performs some clean up of the object's state and then terminates signalling failure. The clean up should leave the object in a consistent state, i.e., the invariant should be true when the handler terminates. If no rescue clause is attached to the routine, it is as if a rescue clause with an empty list of clean up statements were attached to it. Routine failure, in turn, causes an exception to be propagated to the caller. Thus, if all exceptions are handled according to organized panic, all routines eventually fail; that is, any failure causes an orderly shut down of the executing program.

As an example, consider the abstract data type NON_AXIAL_INT_POINT that was defined in Figure 4.4 and suppose that the program is compiled with the option "check assertion" on. If any of the operations is called by a client module with parameters that do not satisfy the corresponding precondition (e.g., one of the parameters of make_point is zero), control is transferred to the implicit empty rescue clause that is attached to all exported operations. This causes propagation of the failure to the object that called the operation with improper arguments. To explain the reason of the failure to the programmer, one might attach rescue clauses to the routines of the class in Figure 4.4 which print out a message describing the reason for the failure, i.e., violation of the precondition.

An alternative approach to organized panic is called *retrial*. This means that the handler can find an alternative way to fulfil the object's contract. This is achieved by a statement retry which may appear in the rescue clause and would cause re-execution of the routine's body. In such a case, if re-execution does not raise an exception, the routine does not fail and the object's contract would be fulfilled. As an example, suppose that several methods are available to solve a specific task, so that if one of them fails, another can be tried instead; the task only fails if none of the available methods succeeds. This strategy can be stated in Eiffel according to the following scheme:

```
try_several_methods is
local
    i: INTEGER;
    --it is automatically initialized to 0
do
    try_method (i);
rescue
    i := i + 1;
    if i < max_trials then
        --max_trials is a constant
        retry
    end
end
```

It is easy to verify that routine try_several_methods only fails if all possible methods fail. Otherwise, if one of the methods succeeds, the routine returns normally to its caller.

******start sidebar PL/I******

PL/I was the first language to introduce exception handling. Exceptions are called CONDITIONS in PL/I. Exception handlers are declared by ON statements:

```
ON CONDITION (exception_name) exception_handler
```

where exception_handler can be a simple statement or a block. An exception is explicitly raised by the statement

```
SIGNAL CONDITION (exception_name);
```

The language also defines a number of built-in exceptions and provides system-defined handlers for them. Built-in exceptions are automatically raised by the execution of some statements (e.g., ZERODIVIDE is raised when a divide by zero is attempted). The action performed by a system-provided handler is specified by the language. This action can be redefined, however, as with user-defined exceptions:

```
ON ZERODIVIDE BEGIN;
    . . .
END;
```

Handlers are bound to exceptions dynamically. When an ON unit is encountered during execution, a new binding takes place between an exception and a handler. Once this binding is established, it remains valid until it is overridden by the execution of another ON statement for the same exception, or until termination of the block in which the ON statement is executed. If more than one ON statement for the same exception appears in the same block, each new binding overrides the previous one. If a new ON statement for the same exception appears in an inner block, the new binding remains in force only until the inner block is terminated. When control exits a block, the bindings that existed prior to block entry are reestablished.

When an exception is raised (either automatically or by a SIGNAL statement), the handler currently bound to the exception is executed as if it were a subprogram invoked explicitly at that point. Therefore, unless otherwise specified by the handler, control subsequently will return to the point that issued the SIGNAL.

PL/I does not allow the programmer to pass any information from the point raising the exception to the exception handler. If this is necessary, the programmer must resort to global variables, which can be an unsafe programming practice. Furthermore, use of global variables is not always possible. For example, when a STRING-RANGE exception is raised, indicating an attempt to access beyond a string's bounds, there is no practical way for the exception handler to know which string is involved if two or more strings are visible in the scope.

PL/I exception-handling mechanisms can be complicated further by explicitly enabling and disabling built-in exceptions; user-defined exceptions cannot be disabled, because they must be explicitly signaled anyway. Most built-in exceptions are enabled by default and bound to the standard system-provided error handler. Enabling a previously disabled exception (or an exception that is not enabled by default) can be specified by prefixing a statement, block, or procedure with the exception name, for example

```
(ZERODIVIDE) : BEGIN
    . . .
END;
```

The scope of the prefix is static; it is the statement, block, or procedure to which it is attached. An enabled exception can be explicitly disabled by prefixing a statement, block, or procedure with NO exception_name. For example

```
(NOZERODIVIDE) : BEGIN;
    . . .
END
```

******end sidebar PL/I******

***sidebar start Exception handling in CLU

In CLU, exceptions can only be raised by procedures. That is, if a statement raises an exception, the procedure containing the statement returns abnormally by raising the exception. A procedure cannot handle an exception raised by its execution: its caller should be in charge of handling it. The exceptions that a procedure may raise are to be declared in the procedure's header. This choice is a consequence of the design method that CLU wishes to enforce. CLU views a procedure as the implementation of an abstract operation, whose meaning should be visible by other units through the operation's interface (defined by the header). The exceptions that a procedure may raise characterize the abstract behavior of the procedure, and thus should be known to the caller. Exceptions may be raised explicitly by means of a **signal** instruction. Built-in exceptions are raised automatically; for example, an exception is raised if the value of the denominator is zero in a division.

Exception handlers can be attached to statements by **except** clauses having the following syntactic form

<*statement*> **except** <*handler_list*> **end**

where <*statement*> can be any (compound) statement of the language. If the execution of a procedure invocation within <*statement*> raises an exception, control is transferred to <*handler_list*>. A<*handler_list*> has the following form

```
when <exception_list_1>: <statement_1>
. . .
when <exception_list_n>: <statement_n>
```

If the raised exception belongs to <*exception_list_i*>, then <*statement_i*> (the handler body) is executed. When the execution of the handler body is completed, control passes to the statement that follows the one to which the handler is attached. If *statement_i*> contains a call to a unit, another exception may be raised. In such a case, control flows to the **except** statement that encloses <*statement*>. If the raised exception is not named in the exception list that should handle it, it is propagated to the enclosing statements. If no handler is found within the procedure that issued the call, the procedure implicitly signals a language-defined exception **failure** and returns.

***sidebar end

4.4.4 Exception handling in ML

The functional language ML allows exceptions to be defined, raised, and handled. There are also exceptions that are predefined by the language and raised automatically by the runtime machine while the program is being executed.

As an example, the following declaration introduces an exception

```
exception Neg
```

which can be raised subsequently in the following function declaration

```
fun fact (n) =
    if n < 0 then raise Neg
    else if n = 0 then 1
    else n * fact (n - 1)
```

A call such as fact (-2) would cause the evaluation of the function to be abandoned, the exception raised and, since no handler is provided, the program to stop by writing the message "Failure: Neg".

Suppose we wish to handle the exception by returning 0 when the function is called with a negative argument. This can be done, for example, by defining the following new function

```
fun fact_0 (n) = fact (n) handle Neg => 0;
```

which uses fact as a subsidiary function. Exceptions that are not handled in a chain of function calls are implicitly propagated. That is, suppose that function fact is called by some function f which does not provide a handler for Neg;

function f, in turn is called by function g, which provides a handler for Neg, in the same way as function fact_0 does. In such a case, if the evaluation of the following expression:

```
g (f (fact (-33)))
```

results in 0.

4.4.5 A comparative evaluation

The languages we surveyed in the previous sections are good representatives of the different approaches followed by programming languages to provide exception handling. Although the field has matured in the past years and the main design decision to be faced by language designers are now basically restricted to a limited number of possible choices, still there are differences and there is no consensus on a common scheme that languages should adopt. We will compare and evaluate the different solutions adopted by existing languages by examining the questions we posed at the beginning of our discussion, that is:

1. What are the exceptions that can be handled? How can they be defined?
2. What units can raise an exception and how?
3. How and where can a handler be defined?
4. How does control flow after an exception is raised in order to reach its handler (if any)?
5. Where does control flow after an exception has been handled?

Regarding questions 1 and 2, all languages (except Eiffel) are quite similar. They all allow both built-in and programmer-defined exceptions. The main differences are whether an exception can carry information and how it can do so. In Ada (and PL/I) an exception is basically a named signal, and thus it does not allow any additional information to be passed to the handler along with it. In C++ any desirable data may be passed along with the exception[1].

Eiffel follows an original approach in that exception handling has been designed to fit a precise program development discipline. According to such discipline, an exception arises only if a routine fails because of some error. The language also explicitly and precisely defines what may cause a routine to fail. Thus, in most cases there is no need for naming exceptions, nor for providing a raise statement. All that matters is whether a failure that would

[1] Actually in Ada it is possible to pass to the handler information about the exception occurrence, and a number of predefined operations are provided to extract some limited information from the exception occurrence.

violate the object’s contract occurred in a routine[1].

Exception handlers in both Ada and C++ can be attached to any block. In Eiffel it can be attached to any routine. As an exception is raised, control is transferred to the appropriate handler. To match the raised exception with the corresponding handler, Ada and C++ unwind the run-time stack by following the dynamic chain until the relevant handler (if any) is found. In Eiffel, each routine provides its own handler (either explicitly or implicitly), and the stack is unwound only if the routine fails.

The combination of static scope rules for exception declarations, adopted by languages like Ada and C++, with dynamic binding between an exception raised by some unit and its handler cause subtlelties that can make programs hard to read. We illustrate the point in the case of C++, in order to show the reader how different language features may interfere with each other, thus making language semantics and language implementation more complex.

Consider two separate files, which contain parts of a program. File 1 contains the following definitions:

```
class A { };
void f ( ) {
    . . .
    throw A ( );
}
```

File 2 contains the following declarations and definitions

```
extern void f ( );
class A { };
void foo ( ) {
    . . .
    try {
        . . .
        f ( );
        . . .
    }
    catch (A a) {
        . . .
    }
}
```

If when f is called by foo the exception is thrown and not handled by f, propa-

[1] To provide finer control over the handling of exceptions, Eiffel also provides a Kernel Library class EXCEPTIONS, through which exceptions may also be named and raised explicitly.

gation reaches the catch point in File 2. The scope rules of the language, however, are such that the parameter of the catch and the object thrown are bound to different types, and therefore the match does not occur, and the exception is further propagated. Besides affecting understandability, ease and efficiency of the implementation are also affected. Type information, in fact, must be kept to perform the required run-time binding.

The last important point about exception handling is where control should flow after an exception is handled. There are essentially two possible solutions, which corresponds to different styles of handling exception: *termination* and *resumption*. The resumption scheme implies that the handler's code may cause control to return to the point where the exception was raised, whereas the termination scheme does not allow that. Of the languages discussed here, only PL/I fully supports the resumption scheme. Ada and C++ support termination. Although for many years the debate on termination versus resumption gave no clear indication of which approach is superior, termination has now gained wider acceptance. Practical experience with languages providing resumption has shown that the resumption mechanism is more error-prone. Furthermore, it can promote the unsafe programming practice of removing the symptom of an error without removing the cause. For example, the exception raised for an unacceptable value of an operand could be handled by arbitrarily generating an acceptable value and then resuming the computation. CLU is even stricter than the other languages in that it does not even allow the unit that raises an exception to try to handle it. Rather, the unit (a procedure) terminates abnormally and places the burden of handling the exception upon its caller. The caller expects the exception to be possibly raised, since it is listed in the unit's interface definition.

Eiffel is different from all other languages with respect to termination and resumption. Termination in Eiffel is stronger than in other languages. In fact, after control is transferred to a rescue clause which does not contain a retry, completion of the clause implies that the routine fails, and the failure is notified to the caller. In C++, on the other hand, if a catch clause terminates without raising another exception, execution continues from the statement that follows the one to which the currently completed handler is attached. Furthermore, Eiffel provides an explicit way of describing a disciplined form of resumption (retry). The retry statement provided by Eiffel does not fully correspond to the above definition of resumption, since the statement executed after the handler terminates in not the one that caused the exception. Rather, retry allows a routine that failed to be retried as a whole.

4.5 Pattern matching

Pattern matching is a high level way of stating conditions, based on which, different actions are specified to occur. Pattern matching is the most important control structure of the string manipulation programming language SNOBOL4 (see sidebar). Pattern matching is also provided by most modern functional programming languages, like ML, Miranda, SASL, and is also provided by the logical language PROLOG and by rule-based systems.

Let us start by discussing the following simple definitions of a data type and a function:

```
datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
fun day_off  (Sun) = true
    |day_off (Sat )= true
    |day_off  ( _  ) = false
```

In the example, function day_off is defined by a number of cases. Depending on the value of the parameter with which the function will be invoked, the appropriate case will be selected. Cases are checked sequentially, from the first one on. If the first two cases are not matched by the value of the parameter with which the function is called, the third alternative will be selected, since the so-called wild card "_" matches any argument.

As another example, consider the following function definition:

```
fun reverse (nil) = nil
    | reverse (head::tail) = reverse(tail) @ [head]
```

In this case, if the argument is an empty list, then reverse is defined to return the empty list. Otherwise, suppose that the argument is the list [1, 0, 5, 99, 2]. As a result of pattern matching [1, 0, 5, 99, 2] with head::tail, head is bound to 1 and tail is bound to [0, 5, 99, 2]. Thus the result of reverse is the concatenation (operator @) of reverse ([0, 5, 99, 2]) with the list [1].

As a final example, suppose that a new operation to reverse lists is to be defined, such that a (sub)list remains unchanged if its first element is zero. The following function rev would do the job:

```
fun rev(nil) = nil
    | rev(0::tail) = [0] @ tail
| rev(head::tail) = rev(tail) @ [head]
```

In this case, since pattern matching examines the various alternatives sequentially, if the function is invoked with a non-empty list whose first element is zero, the second alternative would be selected. Otherwise, for a non-empty list whose first element is not zero, the third alternative would be selected.

As the example shows, pattern matching has a twofold effect. On the one hand, it chooses the course of action based on the argument; on the other, since the pattern can be an expression with variables, it binds the variables in the pattern (if any) with the values that match. The same bound variables can then be used in the expression that defines the value of the function. Pattern matching can thus be viewed as a generalization of conventional parameter passing. The value of actual parameters is used to match the pattern appearing in the formal parameter part. Thus the case selected by pattern matching can vary from call to call.

More will be said on pattern matching for ML in Chapter 7. Chapter 8 addresses pattern matching in the case of Prolog.

***sidebar start Pattern matching in SNOBOL4

SNOBOL4 is a string-oriented language in that character strings are the most important primitive data type with many built-in operations. A *pattern* is a data structure that specifies a *set of strings*. A pattern is used in *pattern-matching statements* to examine a subject string for the presence of a pattern. For example, the statement

```
MESSAGE PAT
```

means "search the string MESSAGE for the occurrence of the pattern PAT." If, previous to this statement, we had executed these two assignment statements:

```
MESSAGE = 'THERE ARE NO ERRORS HERE.'
PAT = 'ERROR'
```

then the above pattern-matching statement will succeed. The notion of success and failure of statements is used in SNOBOL4 to control the flow of execution in a program. Each statement can specify labels of target state- ments for *success, failure,* or *unconditionally*. For example

```
MESSAGE  PAT : S (OK)  F (NOTFOUND)
```

will transfer control to the statement labelled OK if the pattern-matching succeeds and to NOTFOUND otherwise. The pattern PAT is the simplest kind of pattern we can have–simply one string. We may specify a pattern as a choice among a number of patterns:

```
SUBJECT = 'I' | 'YOU' | 'WE'
```

Now SUBJECT will match any string that contains 'I', 'YOU' or 'WE'. A pattern may be defined also as a concatenation of other patterns:

```
SENTENCE = SUBJECT  VERB  OBJECT '.'
```

The pattern SENTENCE will match any string that contains the patterns SUBJECT, VERB, OBJECT, followed by a period. We can then define patterns for SUBJECT, VERB, and OBJECT:

```
VERB = 'EAT' | 'TAKE'
OBJECT = 'FOOD' | 'THE SPOON' | 'THE CAR'
```

The set of patterns defines the grammar of a tiny and highly simplified subset of the English language. For example, the grammar can represent strings such as

```
I TAKE THE CAR
YOU EAT FOOD
```

Pattern matching can recognize the sentences that are grammatically correct. In fact, the statement

```
TEST SENTENCE
```

will succeed if a valid sentence (according to our grammar) occurs in the string TEST. This pattern will actually match a sentence anywhere in the string but SNOBOL4 provides facilities to constrain the pattern further, for example, to have one sentence and nothing more.

What we have seen so far is actually only a small sample of SNOBOL4's pattern matching power. One of the interesting features is the unevaluated expression that can be used to build *recursive patterns*. The unary operator '*' delays the evaluation of its operand. The expression *E is called an unevaluated expression. The unevaluated expression is evaluated when the interpreter encounters it as part of a pattern-matching operation. Consider the pattern PAT defined with assignment statement:

```
PAT = * PAT 'B' | 'A'
```

The value of PAT is stored as PAT 'B' | 'A', postponing the evaluation of PAT (in

*PAT) to pattern-matching time. At pattern-matching time, this pattern will match either 'A' or *PAT 'B', which at this time causes the pat- tern matcher to substitute a value for PAT. The current value of PAT is *PAT 'B' | 'A'. Therefore, PAT also will match 'AB' or *PAT 'BB'. Thus, we have a recursive definition for PAT, which causes it to match strings of the form A AB ABB ABB.. B. Now, recall from Section 3.1 that to specify the syntax of any interesting language, we need to use recursive rules. Suppose that we want to write a SNOBOL4 program to recognize arithmetic expressons as defined by the grammar we introduced in Chapter 2. The following two statements, which closely mirror the EBNF definition of arithmetic expressions, will do exactly what we want:

```
OPERATOR = '+', '-', '*', '/'
EXPRESSION = '(' *EXPRESSION ')' | *EXPRESSION OPERATOR *EXPRESSION |
IDENTIFIER
```

All these examples emphasize the declarative nature of the language. In SNOBOL4, we declare the structure pattern and leave it to the underlying implementation–the SNOBOL4 interpreter–to find a way to search for the existence of the pattern. If we were solving the same problem in a more conventional language, like C, we would spend most of our effort describing the procedures for the search. The declarative style supported by SNOBOL4 allows the language to be seen as a precursor of the paradigm provided by logic languages, which will be examined in Chapter 8.

***sidebar end

4.6 Nondeterminism and backtracking

Problem solutions can often be described via and-or decompositions into subproblems. For example, to solve problem A, one needs to solve either B, C, or D; to solve–say–C, one needs to solve E, F, and G. This can be represented as an *and/or tree* (see Figure 52). Node A, which has no incoming arcs, is called a *root node*; nodes B, D, E, F, and G, which have no exiting arcs, are called *leaf nodes*. And/or decompositions can also be described–in the hypothetical syntax of some programming language–as

```
A if B or
   C or
   D;
C if E and
   F and
   G;
```

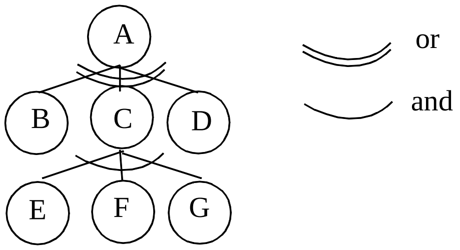


FIGURE 52. And/or tree

The solution of A is described as a *disjunction* of subproblems; the solution of C is described as a *conjunction* of subproblems. We can further assume B, D, E, F, and G to be problem solving routines, which can terminate in either a success or a failure state.

If the order in which subproblems are solved is unspecified (and irrelevant as far as the problem statement is concerned), we say that the program is *nondeterministic*. In the example, this means that the order in which B, C, or D are tried does not matter. Similarly, the way in which E, F, and G are tried does not matter. The only thing that matters is that a solution be found, if it exists, or a notification of failure is delivered to the request to solve A, if no solution exists. The latter case happens if all three subproblems in the disjunct fail, which means also that at least one of the subproblems of the conjunction failed.

An and/or problem decomposition can be viewed as a high-level design of a problem solution, which is then implemented in any progrmming language using the conventional constructs it provides. However, there are programming languages (like logic languages of the Prolog family or the string manipulation language Icon) which support this way of decomposing problems directly. Since features of this kind are very high level, a programming language incorporating them is extremely powerful. As one can imagine, however, these features are hard to implement efficiently. Abstractly, this is the theme of problem solving by exploring a large search space for solutions. Possible strategies to deal with it are described in textbooks on artificial intelligence and computer algorithms.

One solution strategy is to explore the and/or tree in *parallel*, in order to make the search for the solution time efficient. In such a case, subproblems B, D, E, F, and G would be solved in parallel. Another more classical strategy for a sequential implementation of the search is based on *backtracking*. Backtracking means that to find a solution, a choice is made at each branch of the tree. The choice can be fixed, arbitrary, or based on some heuristic knowledge of the problem being solved. If a subproblem solution fails, backtracking implies that another possible subproblem solution be tried. This ensures that the overall problem solution fails only if there is no way of solving the problem without failure. Thus, through backtracking, one needs to guarantee completeness of the search.

More on backtracking will be said in Chapter 8 in the case of logic and rule-based languages.

maybe in bibliographic remarks say that Icon combines backtracking and pattern matching.

4.7 Event-driven computations

In some cases, programs are structured conveniently as *reactive* systems, i.e., systems where certain *events* occurring in the environment cause certain program fragments to be executed. An example is provided by modern user interfaces where a number of small graphical devices (called *widgets*) are often displayed to mediate human-computer interaction. By operating on such widgets (e.g., by clicking the mouse on a push-button) the user generates an event. The event, in turn causes a certain application fragment to be executed. Execution of such a fragment may cause further screen layouts to be generated with a new context of available widgets on it. The events that can be generated in any given state are defined by the context.

The entire application can be viewed as a system which reacts to events by dispatching them to the appropriate piece of code that is responsible for handling the event. As a consequence, the application is structured as a set of fragments that are responsible for handling specific events.

This conceptual view of an application can be viewed as a way of structuring its high-level design, which would then need to be detailed by a conventional implementation. There are languages, however, that directly support this conceptual view, by providing the necessary abstractions. For example, languages like Visual Basic, Visual C++, or Tcl/Tk allow one to separately design the widgets and the code fragments, and to bind events on a widget to the fragments which respond to the event. More on such tools will be said in Chapter 9.

Another common event-driven control paradigm is the one based on so-called *triggers*. Triggers became popular in recent years, in conjunction with new developments in the field of so-called active data bases. Since there is no precise and universal definition of a trigger, we will give examples based on a hypothetical language syntax. An *active data base* consists of a conventional underlying (passive) data base and a set of *active rules* (or *triggers*) of the following form

```
on event
when condition
do action
```

When the event associated with the rule occurs, we say that the rule is *triggered*. A triggered rule is then checked to see if the condition holds. If this is the case, the rule can be executed.

As an example, the following trigger specifies that the total number of employees should be updated as a new employee record is inserted in the data base.

```
on insert in EMPLOYEE
when TRUE
do emp_number ++
```

As another example, in a database application, triggers may be used to specify some constraints that must be verified as new elements are inserted or existing elements are updated or deleted from the database. For example, a constraint might be that no employee can have a salary that is more than the average salary of managers. A trigger might watch that no insertion, update, or deletion violates the constraint; if that happens, some appropriate action would be undertaken.

A trigger-based problem solution can be viewed as a high-level design, which is then implemented in any programming language using the conventional constructs it provides. In the above example, the check that trigger conditions become true might be explicitly associated with the start and the end of each class member routine, along with the execution of the corresponding code fragment. However, there are languages where trigggers are directly available as a built-in language construct; i.e., they are implemented by the underlying run-time machine. As an example, the forthcoming SQL standard includes triggers as one of its features.

4.8 Concurrent computations

Sometimes it is convenient to structure our software as a set of concurrent units which execute in parallel. This can occur when the program is executed on a computer with multiple CPU's (multiprocessor). In such a case, if the number of processors coincides with the number of concurrent units, we say that underlying machine that executes the program provides for *physical* parallelism: each unit is in fact executed by its dedicated processor. Parallelism, however, may be simply *logical*. For example, if the underlying machine is a uniprocessor, the logical view of parallel execution may be provided by switching the CPU from one unit to another in such a way that all units appear to progress simultaneously. The switching of the execution of the uniprocessor among the various units can be performed by a software layer, implemented on top of the physical machine, which provides the programmer with a view of an abstract parallel machine where all units are executed simultaneously. Once such abstract machine is in place, one can in fact abstract away from the physical architecture of the underlying hardware, where components are actually executed. The hardware structure might be a multiprocessor, with each processor dedicated to a single unit, or it might be a multiprogrammed uniprocessor. Allowing for the possibility of different machines means that the correctness of a concurrent system cannot be based on an assumption of the speed of execution of the units. Indeed, the speed can differ greatly if every unit is executed by a dedicated processor, or if a single processor is shared by several units. Moreover, even if the architecture is known, it is difficult to design a system in such a way that its correctness depends upon the speed of execution of the units. We will return to these points in the discussion of implementation models for concurrency.

Concurrency is an important area of computer science, which is often studied in different context: machine architectures, operating systems, distributed systems, databases, etc. In this section we give an overview of how programming languages support concurrency. Concurrent programs support concurrency by allowing a number of units (called *processes*) to execute in parallel (logically or physically).

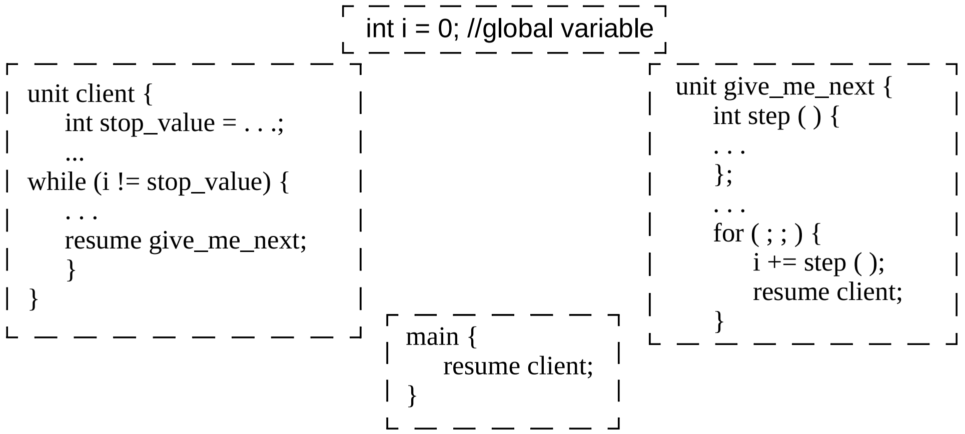
If the abstract machine that executes the program does not support concurrency, it is possible to simulate it by transferring control explicitly from one unit to another. This low-level approach is supported by*coroutines*, reviewed in the sidebar.

***Coroutine sidebar start

Coroutines are a low-level construct for describing pseudo-concurrent units. They can be used to simulate parallelism on a uniprocessor by explicitly interleaving the execution of a set of units. Therefore, they do not describe a set of concurrent units, but a particular way of sharing the processor to simulate concurrency.

Coroutines can be viewed as program units that activate one another explicitly, via a resume primitive. At any time, only one unit is executing. When a unit is executing, control may be explicitly transferred to another unit (via resume), which resumes execution at the place where it last terminated. Consequently, units activate each other explicitly in an interleaved fashion, according to a predefined pattern of behavior.

As an example, consider the the two coroutines client and give_me_next shown in Figure 53, written in a hypothetical, self-explaining programing language. Unit client repeatedly activates unit give_me_next to get the next value of a variable. Each reactivation of unit give_me_next produces a new value, which depends on the previously generated value. The two units resume one another. There is a global variable i, which is shared by client and give_me_next. Unit main, which is activated initially, resumes client.



```
int i = 0; //global variable
```

```
unit client {
     int stop_value = . . .;
     ...
while (i != stop_value) {
     . . .
     resume give_me_next;
     }
}
```

```
unit give_me_next {
     int step ( ) {
     . . .
     };
     . . .
     for ( ; ; ) {
          i += step ( );
          resume client;
     }
```

```
main {
     resume client;
}
```

FIGURE 53. An example of coroutines

An abstract implementation model of coroutines differs from the case of routines. When a coroutine A issues a resume to a coroutine B, one must save (in A's activation record) the pointer to the instruction following the in-struction resume B. Moreover, A's activation record is not deallocated. If coroutines can have nested units that may be activated recursively, each coroutine requires an activation record stack that can grow and shrink independently of the other stacks. In addition, as in the example, they may access the global environment.

***Coroutine end

To support correct interaction among processes, a language should provide suitable *synchronizition statements* (or *primitives*). We introduce this concept through an example. Suppose that a certain system contains concurrent activities of the following two kinds: producers and consumers. A *producer* produces a stream of values and places them into a suitable data structure (a *buffer* of a certain size, N). A *consumer* reads these values from the buffer in the same order as they are produced and then processes such values according to some policy. This example is a classic standard problem that exhibits many relevant issues of concurrency. An abstract description of two producer and consumer processes is shown in Figure 54. A given system might contain many such processes, and all might interact through the same buffer.

```
Process Producer

    repeat forever {
        produce an element;
        append the element to the buffer;
    }
```

```
Process Consumer

    repeat forever {
        produce an element;
        append the element to the buffer;
    }
```

FIGURE 54. Sample processes:roducer and a consumer

The two processes in Figure 54 are described by cyclic and, ideally, nonterminating program units, which cooperate to achieve the common goal of transferring data from the producer (which could be reading them from an input device) to the consumer (which could be storing them in a file). The buffering mechanism allows the two processes to proceed at their own speeds, by smoothing the effect of their variations. To guarantee the correctness of the cooperation, however, the programmer must enssure that no matter how quickly or slowly the producer and the consumer progress, there will be no attempts to write into a full buffer or to read from an empty buffer. This can be accomplished by the use of synchronization statements. In general, synchronization statements allow a process to be delayed in the execution of an operation, whenever that is necessary for correct cooperation with other concurrent units. In the example, when the buffer is full, the producer is delayed if it tries to append an element, until the consumer removes at least one element. Similarly, when the buffer is empty, the consumer is delayed if it tries to remove an item, until the producer appends at least one new element.

Another, more subtle, need for synchronization may arise when several activities can legally have access to the same buffer. For example, suppose that append and remove are implemented by the fragments in Figure 55:

```
Append

    t++;
    i = next_in ( );
    buffer [i] = x;
```

```
Remove

    t--;
    j = next_out ( );
    x = buffer [out];
```

FIGURE 55. Operations to append and remove from a buffer

where t represents the total number of elements stored in the buffer, next_in and next_out are two operations that yield the value of the buffer index where the next element can be stored and where the next element is to be read from, respectively. Let us assume that the individual statements in Figure 55 are indivisible instructions of the abstract machine, in the sense that if one such action starts to execute, it is guaranteed to finish before any other instruction execution is started. The sequences, however, cannot be assumed to be indivisible, i.e., the execution of their constituent actions may be interleaved by the underlying machine. As an example, suppose that the buffer is initially empty. A producer might start depositing into the buffer by performing the first two actions. The total number of buffered items becomes 1 and the index of the position where the item should be deposited is evaluated. Suppose that at this point another producer gets access to the buffer (since the buffer is not full). If this producer completes all three actions, the value will be deposited in the second buffer slot (since the first one was acquired by the first producer who did not complete its own deposit). At this point, a consumer might access the buffer (which is not empty, since t = 2). This is an error, however, because the consumer would read its value from the position which was assigned to the first producer, but no assignment was ever performed to such position. To avoid this error, we say that the two statement sequences must be executed in *mutual exclusion*; synchronization primitives must allow mutual exclusion to be specified.

In general, synchronization primitives may be viewed as mechanisms that constrain the order in which operations performed by different processes are executed. Let $\{P_1, P_2, \ldots, P_n\}$ be a set of concurrent processes. Each process can be assigned for execution to an abstract machine, like SIMPLESEM that was discussed in Chapter 2. Let ip_i be the value of the instruction pointer of the i-th abstract machine which executes P_i; ip_i yields the address of the instruction $C_i(ip_i)$ which is to be executed next in each process i. If the processes are logically independent, at any instant, all machines can execute $C_i(ip_i)$. Synchronization, however, may force some abstract machines to remain in idle until some condition is met that allows them to resume execution.

Besides synchronization, programming languages must provide facilities to describe *communication* among processes. Communication allows information to flow from one process to another. It is through synchronization and communication that processes cooperate in problem solving. Communication can be achieved in different ways, depending on the underlying computation model. The traditional way to achieve communication is via a *shared mem-*

ory. According to such model, all concurrent processes have access to a common set of variables. This model reflects an underlying abstract multiprocessor architecture with a common memory area where all processors can read and write. Another paradigm for communication is *message passing*. In such a case, the model reflects more closely an underlying decentralized architecture where processors are connected by a network on which messages can flow. Both paradigms, of course, can be implemented on any underlying architecture, although implementation of–say–the shared memory paradigm on a physically distributed architecture is much less natural and requires considerably more support than implementing the message passing paradigm.

The rest of this chapter is organized as follows. Section 4.8.1 illustrates how processes may be defined in programming languages, using the Ada language as a case-study. In Section 4.8.2 we review two kinds of synchronization mechanisms–semaphores and signals–and discuss communication via shared memory. We also discuss communication via message passing and the rendezvous mechanism. Finally, Section 5.8.3 discusses implementation models. Our presentation does not go into details of this last point, which goes beyond the scope of this book, and is usually discussed in textbooks on operating systems.

4.8.1 Processes

A concurrent programming language must provide constructs to define processes. Processes can belong to a type, of which several instances can be created. The language must define how and by whom a process is initiated, i.e., a new independent execution flow is spawned by an executing unit. It also need to address the issue of process termination, i.e., how can a process be terminated, what happens after a process terminates, etc.

In this section we will briefly review the main concepts and solutions provided by Ada. In Ada, processes are called *tasks*. The execution of an Ada program consists of the execution of one or more tasks, each representing a separate computation that proceeds concurrently with other tasks, with which it may interact through synchronization statements.

Tasks can be defined by a *task type*, of which many instances can be declared. It is also possible to declare a task object (shortly, a task) directly. The *declaration* of a task (type) specifies how the task (or all instances of the type) can interact with other tasks. As we will see shortly, interaction with a task can be achieved by calling one of its *entries*, which must appear in its declaration. Thus, the declaration of a task type is a declaration of an abstract data type; entries represent the operations available for interaction with task objects. In Ada, the *body* of the task (type) , which describes the implementation of the task's internal code, can be described separately from its declaration.

This is an example of a task type declaration:

```
task type SERVER is
    entry NEXT_REQUEST (NR: in REQUEST);
    entry SHUT_DOWN;
end SERVER;

task SERV_PTR is access SERVER;  --declares a pointer to a SERVER
```

These are examples of task object declarations:

```
MY_SERVER: SERVER;

task CHECKER is
    entry CHECK (in T: TEXT);
    entry END;
end CHECKER;

HIS_SERVER_PTR: SERV_PTR := new SERVER;
```

The execution of task consists of executing its body. The mechanism for *activating* a task is similar to the mechanism that allocates storage to variables. For example, consider the following fragment

```
procedure P is
    A, B: SERVER;
    HER_SERVER_PTR: SERV_PTR;
begin
    . . .
    HER_SERVER_PTR := new SERVER;
    . . .
end P;
```

Tasks A and B are activated as the block in which they are locally declared is entered at run time. The task pointed at by HER_SERVER_PTR is activated by the execution of the new operation.

The concept of task *termination* is more complex, and will not described in all its subtelties. For simplicity, let us assume that a task can terminate when it reaches the last statement of its body and (1) all of the locally declared task objects have terminated, and (2) tasks allocated by a new and referenced only by pointers local to the task have terminated.

4.8.2 Synchronization and communication

In this section we present some elementary mechanisms for process synchronization and interprocess communication: semaphores , signals and monitors, and rendezvous. Semaphores are low level synchronization mechanisms that are mainly used when interprocess communication occurs via shared variables. Monitors are higher level constructs that define abstract objects used for interprocess communication; synchronization is achieved via signals. Finally, rendevous is another mechanism that combines synchronization and communication via message passing.

4.8.2.1 Semaphores

A semaphore is a data object that can assume an integer value and can be operated on by the primitives P and V. The semaphore is initialized to a certain integer value when it is declared.

The definitions of P and V are

```
P (s):  if s>0 then s = s - 1
        else suspend current process

V (s):  if there is a process suspended on the semaphore
        then wake up process
        else s = s + 1
```

The primitives P and V are assumed to be indivisible, atomic operations; that is, only one process at a time can be executing P or V operations on the same semaphore. This must be guaranteed by the underlying implementation, which should make P and V behave like elementary machine instructions.

The semaphore has (1) an associated data structure where the descriptors of processes suspended on the semaphore are recorded, and (2) a policy for selecting one process to be woken up when required by the primitive V. Usually, the data structure is a queue served on a first-in/first-out basis. However, it is also possible to assign priorities to processes and devise more complex policies based on such priorities.

The simple producer-consumer example of Figure 54 can be solved using semaphores as shown in (as usual, we adopt an arbitrary, self-explanatory C-like notation).

```
int n = 20;
buffer buf;    // a global buffer variable, with operations append and remove which up-
date
                    // t, total number of buffered items;
semaphore  mutex = 1;        // used to guarantee mutual exclusion
     in = 0;                 // semaphore to control the reading from the buffer
     spaces =  n;            // semaphore to control the writing into the buffer
process producer {
     int i;
     for ( ; ; ) {
          produce (i);
          P (spaces);        -- wait for free spaces
          P (mutex);         -- wait for buffer availability
                             --the buffer must be used in mutual exclusion
          buffer . append (i);
          V (mutex);         -- finished accessing buffer
          V (in)             -- one more item in buffer
     };
};
process consumer {
     int j;
     for ( ; ; ) {
           P (in);           -- wait for item in buffer
          P (mutex);         -- wait for buffer availability
                             --the buffer must be used in mutual exclusion
           j = buffer.remove ( );
          V (mutex);         -- finished accessing buffer
          V (spaces)         -- one more space in buffer
     };
}
```

FIGURE 56. Producer-consumer example with semaphores

The keyword process starts the segments of code that can proceed concurrently. Three semaphores are introduced. Semaphores spaces and in are used to guarantee the logical correctness of the accesses to the buffer. In particular, spaces (number of available free positions in the buffer) suspends the producer when it tries to insert a new item into a full buffer. Similarly, in (number of items already in the buffer) suspends the consumer if it tries to remove an item from an empty buffer. Semaphore mutex is used to enforce mutual exclusion of accesses to the buffer. We can see that semaphores are used both for pure synchronization, as in mutex, to ensure that only one process may use the buffer at a time, and for a kind of communication among processes. For example, V (spaces) by the consumer communicates to the producer that it has consumed an item and that more space is now available in the buffer.

Programming with semaphores requires the programmer to associate one semaphore with each synchronization condition. Our example shows that semaphores are a simple but low-level mechanism, their use can be awkward in practice, and the resulting programs are often difficult to design and understand. Moreover, little checking can be done statically on programs that use semaphores. For example, a compiler would not be able to catch the incorrect use of a semaphore, such as one resulting from a change of V (mutex) into P (mutex) in the producer process (see Exercise 16). Catching such an error is impossible because it requires the translator to know the semantics of the program, that is, that the operations on the buffer are to be executed in mutual exclusion, and mutex is used to guarantee such mutual exclusion. Therefore, semaphores require considerable discipline on the part of the programmer. For example, one should not forget to execute a P before accessing a shared resource, or neglect to execute a V to release it.

Using semaphores for synchronization purposes other than mutual exclusion is even more awkward. In the producer-consumer example, process consumer suspends itself by executing P (spaces) when the buffer is full. It is the responsibility of some other piece of code, the consumer in this case) to provide the matching V operation. If the programmer forgets to write a V (spaces) after each consumption, the producer will become blocked forever.

Semaphores are often provided by operating systems to support systems programming. They have also been integrated into a number of existing programming languages, such as PL/I and Algol 68 (see sidebar).

***sidebar start

PL/I was the first language to allow concurrent units, called *tasks*. A procedure may be invoked as a task, in which case it executes concurrently with its caller. Tasks also can be assigned priorities. Synchronization is achieved by the use of *events*, which are binary semaphores that only can assume one of two values: '0'B and '1'B (Boolean constants 0 and 1). A P operation on a semaphore is represented by a WAIT operation on the completion of an event E: WAIT (E). A V operation is represented by signaling the completion of the event: COMPLETION (E) = '1'B. PL/I extends the notion of semaphores by allowing the WAIT operation to name several events and an integer expression e. The process will be suspended until any e events have been completed. For example, WAIT (El, E2, E3) (1) indicates the waiting for any one of the events: El, E2, or E3.

ALGOL 68 supports concurrent processes in a parallel clause whose constituent statements are elaborated concurrently. Synchronization can be provided by semaphores, which are data objects of type sema.

***Sidebar end

4.8.2.2 Monitors and signals

Concurrent Pascal introduced the signal and monitor constructs into the programming languages. *Signals* are synchronization primitives; monitors describe abstract data types in a concurrent environment. The operations that manipulate the data structure are guaranteed to be executed in mutual exclusion by the underlying implementation. Cooperation in accessing the shared data structure must be programmed explicitly by using the monitor signal primitives delay and continue.

Using the notation of Concurrent Pascal, the program in Figure 57 illustrates the use of monitors in the producer-consumer example.

```
type fifostorage =
monitor
    var contents: array [1. .n] of integer;  {buffer contents}
        tot: 0. .n; {number of items in buffer}
        in,  {position of item to be added next}
        out: 1. .n; {position of item to be removed next}
        sender, receiver: queue;
    procedure entry append (item: integer);
    begin if tot = n then delay (sender);
        contents [in] := item;
        in := (in mod n)+1;
        tot := tot + 1;
        continue (receiver)
    end;
    procedure entry remove (var item: integer);
    begin if tot = 0 then delay (receiver);
        item := contents[out];
        out := (out mod n) + 1;
        tot := tot - 1;
        continue (sender)
    end;
begin {initialization part}
    tot := 0; in := 1; out := 1
end
```

FIGURE 57. Producer-consumer example with monitor

An instance of the monitor (i.e., a buffer) can be declared as

```
var buffer: fifostorage
```

and can be created by the statement init buffer. Monitor instances are abstract objects through which interprocess communication and synchronization is coordinated.

The init statement allocates storage for the variables defined within the monitor definition (i.e., contents–the contents of the buffer, tot–the total number of buffered items, and in and out–the positions at which the next items will be appended and removed, respectively) and executes the initialization part (which sets tot to zero, and in and out to one). The monitor defines the two procedures, append and remove. They are declared with the keyword entry, which means that they are the only exported procedures that can be used to manipulate monitor instances. Cooperation between the producer and the consumer is achieved by using the synchronization primitive signals delay and continue.

The operation delay (sender) suspends the executing process (e.g., the producer) in the queue sender. The process loses its exclusive access to the monitor's data structure and its execution is delayed until another process (e.g., the consumer) executes the operation continue (sender). Similarly, with delay (receiver) a consumer process is delayed in the queue receiver if the buffer is empty, until the producer resumes it by executing the instruction continue (receiver). The execution of the continue (q) operation makes the calling process return from the monitor call and, additionally, if there are processes waiting in the queue q, one of them immediately will resume the execution of the monitor procedure that previously delayed it.

The structure of a Concurrent Pascal program that uses the above monitor to represent cooperation between a producer and a consumer is given in Figure 58.

```
const n = 20;
type fifostorage =   . . . as above . . .
type producer =
process (storage: fifostorage);
var element: integer;
begin cycle
         . . .
         storage.append (element);
         . . .
      end
end;
type consumer = process (storage: fifostorage);
var datum: integer;
begin cycle
         . . .
         storage.remove (datum);
         . . .
   end
end;
var meproducer: producer;
   youconsumer: consumer;
   buffer: fifostorage;
begin
   init buffer, meproducer (buffer), youconsumer (buffer)
end
```

FIGURE 58. Overall structure of a Concurrent Pascal program with two processes (a producer and a consumer) and one monitor (a buffer)

Processes are described in the example as nonterminating, cyclic activities (cycle...end). Two particular instances (meproducer and youconsumer) are declared as bound to an instance of the resource type fifostorage and subsequently activated as concurrent processes by the init statement.

4.8.2.3 Rendezvous

The examples given so far used shared memory for interprocess communication. A globally accessible buffer was used by producer and consumer processes, and suitable synchronization promitives were introduced to allow them to proceed safely. In the example using semaphores, synchronization and communication features were separate. Semaphores are used for synchronization; shared variables are used for communication. In the monitor example, the two issues were more intertwined, and the resulting construct is higher level. One can view the monitor construct as defined by two logical components: an abstract object which is used for communication among processes in mutual exclusion, and a signal mechanism that supports synchronization (e.g., the ability to delay and resume processes, based on some logical condition). Note that while the first component is intrinsically based on a shared memory computation paradigm, the second is not, and might be used also in a decentralized scheme for concurrent computation.

In this section we illsustrate the *rendezvous* concept introduced by the Ada programming language. The construct can be viewed as a high-level mechanism that combines synchronization and communication, where communication is based on the *message passing* conceptual paradigm. The construct, per se, can be naturally used to write software for distributed architectures, although its possible interaction with other features in Ada can make this quite difficult. Hereafter we concentrate on the basic properties of rendezvous; additional features and the interaction with other facilities provided by the language (such as scope rules and exception handling), will be ignored for the sake of simplicity.

The Ada task object in Figure 59 describes a process that handles the operations append and remove on a buffer.

```
task Buffer_Handler is   --task declaration
    entry Append (Item: in Integer);
    entry Remove (Item: out Integer);
end;
task body Buffer_Handler is --task implementation
    N: constant Integer := 20;
    Contents: array (1. .N) of Integer;
    In_Index, Out_Index: Integer range 1. .N := 1;
    Tot: Integer range 0. .N := 0;
begin loop
    select
        when Tot < N =>
            accept Append (Item: in Integer) do
                Contents (In_Index) := Item;
            end;
        In_index := (In_Index mod N)+1;
        Tot := Tot+ 1
    or
        when Tot > 0 =>
            accept Remove (Item: out Integer) do
                Item := Contents (Out_Index);
            end;
        Out_Index := (Out_Index mod N) + 1;
        Tot := Tot - 1;
    end select;
end loop;
end Buffer_Handler;
```

FIGURE 59. An Ada task that manages a buffer

The declaration of task Buffer_Handler specifies Append and Remove as entries. An entry can be viewed as a *port*, through which a task can send a message to another task, which can then accept it. The task can indicate its willingness to accept a message if it is an owner of the corresponding entry (i.e., the entry is declared in it). It does so by executing the accept statement. At this point, the sender and the receiver tasks can be viewed as meeting together (in French, they perform a *rendezvous*).

If the sender calls the entry (i.e., sends the message) before the receiver issues an accept, the sender is suspended until the rendezvous occurs. Similarly, a suspension of the receiver occurs if an accept statement is executed before the corresponding entry is called (i.e., before the message is sent). Note that a task can accept messages from more than one task; consequently, each entry potentially has a queue of tasks which sent messages to it.

The accept statement is similar to a routine. After a repetition of the header of the entry, the do. . .end part (*accept body*) specifies the statements to be executed at the rendezvous. Once a match between an entry call and the corresponding accept occurs, the sender is suspended until the accept body is executed by the called task. The accept body is the only place at which the parameters of the entry are accessible. Possible out parameters (as in the case of REMOVE) are passed back to the sender at the end of the rendezvous, that is, when the execution of the accept body is completed. Thereafter, the two tasks that met in the rendezvous can proceed in parallel.

The bodies of tasks PRODUCER and CONSUMER, which interact with BUFFER_HANDLER in the producer-consumer example, are sketched in Figure 60.

```
PRODUCER

loop
    produce a new value V;
    Buffer_Handler . Append (V);
    exit when V denotes the end of
                the stream;
end loop;
```

```
CONSUMER

loop
    Buffer_Handler . Remove ( V );
    consume V;
    exit when V denotes the end of
                the stream;
end loop
```

FIGURE 60. Sketch of the producer and consumer tasks in Ada

In the example of Figure 59, accept statements are enclosed within a select statement. The select statement specifies several alternatives, separated by or, that can be chosen in a nondeterministic fashion. The Ada selection is specified by an accept statement, possibly prefixed (as in our example) by a when condition. Execution of the select statement proceeds as follows[1].

1. The conditions of the when parts of all alternatives are evaluated. Alternatives with a true condition, or without a when part, are considered open; otherwise, they are considered closed. In the example, both alternatives are open if 0 < TOT < N.
2. An open alternative can be selected if a rendezvous is possible (i.e. an entry call already has been issued by another task). After the alternative is selected, the corresponding accept body is executed.
3. If there are open alternatives but none can be selected immediately, the task waits until a rendezvous is possible.

[1] This is a simplified view of Ada. We are ignoring several features that would complicate our presentation.

4. If there are no open alternatives, an error condition is signaled by the language-defined exception PROGRAM_ERROR.

4.8.2.4 Summing up

Semaphores, monitors, and rendezvous are all primitives for modeling concurrent systems. As we pointed out, semaphores are rather low-level mechanisms: programs are difficult to read and write, and few checks on their correct use can be done automatically. Monitors, on the other hand, are a higher-level structuring mechanism. Using monitors, a typical system structuring proceeds by identifying (1) shared resources as abstract objects with suitable access primitives (*passive entities*), and (2) processes (*active entities*) that cooperate through the use of resources. Resources are encapsulated within monitors. Mutual exclusion on the access to a shared resource is guaranteed automatically by the monitor implementation, but synchronization must be enforced by explicitly suspending and signaling processes via delay and continue statements. The distinction between active and passive entities (processes and monitors, respectively) disappears in a scheme based on rendezvous. Shared resources to be used cooperatively are represented by tasks, that is, by active components representing resource managers. A request to use a resource is represented by an entry call, i.e., by sending a message which must be accepted by the corresponding resource manager.

A system structured via monitors and processes can be re-structured via tasks and rendezvous, and vice versa; the choice between the two schemes is largely dependent on personal taste. As we mentioned, the latter scheme mirrors more directly the behavior of a concurrent system in a distributed architecture, where remote resources are actually managed by processes that behave as guardians of the resource. However, it can be somewhat awkward in case processes need to communicate via shared objects. In fact, early experience with the Ada programming language, which initially provided only rendezvous, showed that the need for additional tasks to manage shared data often led to poor performance. Therefore, Ada 95 introduced a kind of monitor construct–*protected types*–in addition to the rendezvous.

The use of an Ada protected type to implement our running example of a buffer type (Fifo_Storage) is shown in Figure 61.

```
protected type Fifo_Storage is
     entry Append (Item: in Integer);
     entry Remove (Item: out Integer);
private
     N: constant Integer := 20;
     Contents: array (1. .N) of Integer;
     In_Index, Out_Index: Integer range 1. .N := 1;
     Tot: Integer range 0. .N := 0;

protected body Fifo_Storage is
     entry  Append (Item: in Integer) when Tot < N  is
     begin
          Contents (In_Index) := Item;
          In_Index := (In_Index mod N) + 1;
          Tot := Tot + 1
     end Append;

     entry Remove (Item: out Integer) when Tot > 0 is
     begin
          Item := Contents (Out_Index);
          Out_Index : = (Out_Index mod N) + 1;
          Tot := Tot - 1;
     end Remove;
end Fifo_Storage;
```

FIGURE 61.A protected Ada type implementing a buffer

Similar to the monitor, operations defined for a protected type are executed by the underlying abstract machine in mutual exclusion. There are two kinds of possible operations: routines (i.e., procedures and functions) and entries. Entries (shown in the above example) have an associated *barrier* condition which is used for synchronization. Routines have no associated barriers. The difference with the monitor is that no explicit signals are issued. Rather, when an entry is called its barrier is evaluated; if the barrier is false then the calling process is suspended and queued. At the end of the execution of an entry (or a routine) body, all barriers which have queued tasks are re-evaluated, thus possibly allowing a suspended task whose barrier became true to be resumed. The absence of explicit signals to be exchanged for synchronization purposes makes the construct simpler to use and the corresponding abstraction easier to understand than in the case of monitors.

Ada is perhaps the best example of a programming language which provides a coherent set of well integrated features supporting concurrent programming.

Most other languages do not. Such languages often provide support for concurrent programming either via calls to low-level operating system primitives or via libraries added to language implementations.

Linda is a library for C???

To support distributed systems programming, there are libraries supporting remote procedure calls. In such a case****

task library of C++

sidebar on data concurrency???

4.8.3 Implementation models

In a concurrent system, processes either are suspended (waiting on some synchronization condition) or are potentially active, that is, there are no logical obstacles to their execution. In general, only a subset of potentially active processes can be running, unless there are as many processors as there are potentially active processes. In the common case of a uniprocessor, only one of such processes can be running at a time. It is thus customary to say that processes can be in one of the following states (see also Figure 62).

- - Waiting
- - Ready (i.e., potentially active, but presently not runnning)
- - Running

The state of a process changes from running to waiting if there is some logical condition that prevents the process from continuing its execution. That is, the proces is suspended by the execution of some synchronization statement (e.g., the buffer is full for the producer process). The state can later change from waiting to running if some other process performs a suitable synchronization statement (e.g., a consumer process signals that the buffer is not full any more).

In concurrent programming, the programmer has no direct control over the speed of execution of the processes. In particular, the user is not responsible for changing the state of a process from ready to running (operation of *selection* in Figure 62), which is done by the underlying implementation. Figure 62 shows that a process can leave the running state and enter the ready state as a consequence of the action of *preemption.*

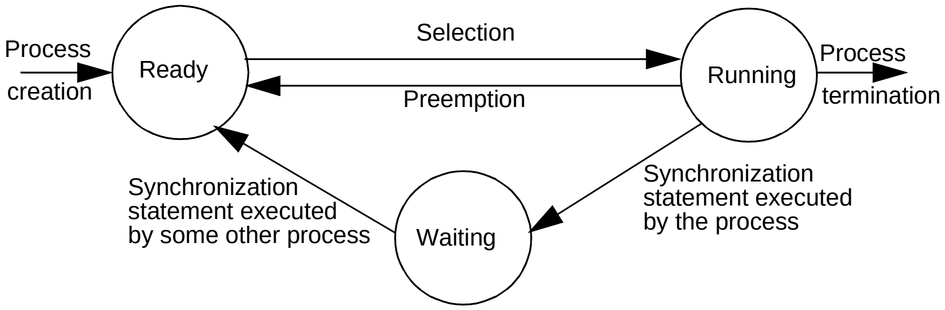


FIGURE 62. State diagram for a process

Preemption is an action performed by the underlying implementation; it forces a process to abandon its running state even if, from a logical point of view, it could safely continue to be executed. A process can be preempted either after it performs a synchronizing statement that makes another suspended process enter the ready state (e.g., a V on a semaphore) or when some other condition occurs, such as the expiration of a specified amount of time (*time slice*).

After the preemption of one process, one of the ready processes can enter the running state. This kind of implementation allows the programmer to view the system as a set of activities that proceed in parallel, even if they are all executed by the same processor. Only one process at a time can be executed by the processor, but each process runs only for a limited amount of time, after which control is given to another process. It is possible to have nonpreemptive implementation of concurrency. In this case, execution switches to another process only when the currently executing process deliberately suspends itself or requires the use of an unavailable re-source.

The portion of run-time support of a concurrent language responsible for the implementation of the state transitions shown in Figure 62, is called the *kernel*. To illustrate the basic features of a kernel, consider the case of a single processor shared by a set of processes. For the sake of simplicity, we will ignore the problems of synchronizing processes with input/output devices and concentrate our attention on the interactions among internal processes. More complete discussions of these issues are traditionally (and more properly) addressed in textbooks on operating systems. Here we provide only a glimpse of the basic problems that are relevant to understanding concurrency features of programming languages.

The information about a process needed by the kernel is represented in a *process descriptor*, one for each process. The descriptor for a process is used to store all the information needed to restore the process from a waiting or blocked state to the running state. This information (called *process status*) includes the process priority (if priorities are used) and all information required to instruct the processor about the identity and point of execution of the process–notably, the contents of the machine registers (program counter, index registers, accumulator, and so on). Saving the status of the process when the process becomes suspended and restoring the status when the process becomes running is one of the kernel's jobs.

The kernel can be viewed as an abstract data type; it hides some private data structures and provides procedures that provide the only ways to use these data structures. All of the kernel’s operations are assumed to be executed in a *noninterruptible way*; i.e., all interrupts are disabled while they are being executed. The kernel's private data structures are organized as queues of process descriptors. The descriptors of ready processes are kept by the kernel in READY_QUEUE. There is also one CONDITION_QUEUE for each condition that might suspend a process, that is, there is one queue for each semaphore and one for each object declared to be of type queue in a monitor (and for each entry in a protected Ada type). Each such queue is used to store the descriptors of all processes suspended on the semaphore or delayed in the queue. A variable RUNNING denotes the descriptor of the running process. A typical snapshot of the kernel's data structures is shown in Figure 5.4. The queues used by the kernel can be considered as instances of an abstract data type whose operations are defined by the following signatures:

```
enqueue: Queue x Descriptor -> Queue         -- inserts a descriptor into the queue
dequeue: Queue -> Queue x Descriptor         -- extracts a descriptor from the queue
empty:      Queue -> Boolean              -- true if the queue is empty; false otherwise
```

In what follows, we discuss the basic operations performed by the abstract machine to execute concurrency constructs. The notation we use is a self-explaining pseudo-code based on C++. Whenever necessary, additional comments are added to the pseudo-code.

Time slicing is implemented by a clock interrupt. Such an interrupt activates the following kernel operation Suspend-and-Select, which suspends the most recently running process into READY_QUEUE and transfers a ready process into the running state.

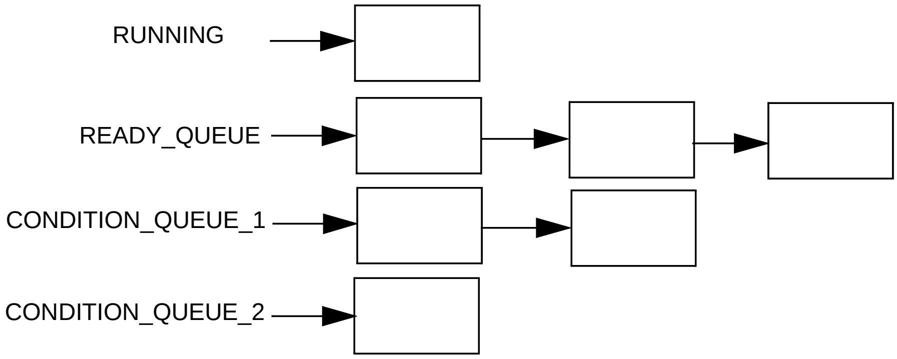


FIGURE 63. Data structures of the kernel

Operation Suspend-and-Select

```
RUNNING = process_status;
                    -- save status of running process into RUNNING
READY_QUEUE . enqueue (RUNNING);
                    -- enqueue RUNNING into READY_QUEUE
RUNNING = READY_QUEUE . dequeue ( );
                    -- move a descriptor from READY_QUEUE into RUNNING
process_status = RUNNING;
                    -- activate the new process)
```

4.8.3.1 Semaphores

If semaphores are provided by the language, primitives P and V can be implemented as calls to kernel procedures. A suspension on a condition c caused by a P operation is implemented by the following private operation of the kernel

Operation Suspend-on-Condition (c)

```
RUNNING = process_status;
CONDITION_QUEUE (c) . enqueue (RUNNING);
RUNNING = READY QUEUE . dequeue ( );
process_status = RUNNING;
```

Awakening a process waiting on condition c, caused by a V operation, is implemented by the following private operation of the kernel.

Operation Awaken

```
RUNNING = process_status;
READY_QUEUE . enqueue (RUNNING);
READY_QUEUE . enqueue (CONDITION_QUEUE (c) . dequeue ( ));
        -- move a descriptor from CONDITION_QUEUE (c) into READY_QUEUE
RUNNING = READY_QUEUE . dequeue ( );
process_status = RUNNING;
```

Note that this implementation guarantees indivisibility of primitives P and V, since we assumed that kernel operations are noninterruptible.

4.8.3.2 Monitors and signals

In the case of monitors and Ada’s protected types, a simple way to implement the required mutual exclusion consists of disabling interrupts when a monitor procedure is called and enabling them on return from the call. We assume that interrupts are enabled and disabled by a single machine instruction, and that a special machine register determines whether interrupts are enabled or disabled. This register is part of the process status and must be saved in the process descriptor when the process is suspended. For the sake of simplicity, we also assume that monitor procedures do not contain calls to other monitor procedures. When a process calls a monitor procedure, the value of the return point from the call is saved in an entry of the process descriptor. Operations delay and continue can be implemented by kernel procedures. In particular, delay is implemented by operation Suspend-on-Condition, and continue (c), where c is a condition queue, is implemented by the following operation.

```
Operation Continue (c)

   RUNNING = process_status (with interrupts enabled and program counter set to
                 the return point from the monitor call);
   READY_QUEUE . enqueue (RUNNING);
   if CONDITION_QUEUE (c) . empty then
      RUNNING = READY QUEUE . dequeue ( );
   else {
      RUNNING = CONDITION_QUEUE (c). dequeue ( );
      process_status = RUNNING
   }
```

Note that we did not state the policies for the management of queues in this abstract implementation. Queues might be handled according to a first-in-first-out policy, or one may even use sophisticated strategies which take into account waiting times and priorities. The part of the kernel responsible for choosing a policy is called the *scheduler*. In our scheme, the scheduler is a part of the implementation of the abstract data type that defines queues.

4.8.3.3 Rendezvous

In this section, we discuss some implementation issues of Ada's rendezvous mechanism. There is one queue of ready tasks (READY_QUEUE). Each entry has a descriptor that contains the following fields.

- A boolean value O describing whether the entry is open (O = true indicates that the task owning the entry is ready to accept a call to this entry).
- A reference W to a queue of descriptors of tasks whose calls to the entry are pending (waiting queue).
- A reference T to the descriptor of the task owning the entry.
- A reference I to the first instruction of the accept body (to simplify matters, we assume that no two accept statements for the same entry can appear in a select statement). This reference is significant only if the task owning the entry is ready to accept a call to the entry (that is, O = true). For simplicity, we can assume that the value of this field is a constant, statically associated with the entry.

As usual, we assume that the implementation of the synchronization statements is done by kernel operations that are noninterruptible, that is, interrupts are disabled and enabled by the kernel before and after executing such statements. The problem of passing parameters across tasks is ignored for simplicity.

Let e be an entry that is called by a task, and let DESCR (e) be e's descriptor. The implementation of a call to entry e can be done by the kernel as follows.

```
RUNNING = process_status;
(DESCR (e) . W) . enqueue (RUNNING);          -- the running process is saved
if DESCR (e).O {                         -- the entry  is open
    for all open entries oe of the task
        oe.O = false;                    -- close the entries
    RUNNING = DESCR (e) . T;             -- the task owning the entry becomes running
    RUNNING . pc = DESCR (e) . I;
                        -- pc is the field containing the value of the program counter
                           -- which is set to the value stored in field I of the entry's descriptor
else
    RUNNING = READY_QUEUE . dequeue ( );
}
```

When the end of the body of an accept statement is reached, the following kernel actions complete the rendezvous.

```
RUNNING = process_status;
READY_QUEUE .  enqueue (Descr (e) . W);
        -- move descriptor of caller referenced by field W of the entry's descriptor
        -- into READY_QUEUE
READY_QUEUE . enqueue (RUNNING);
```

```
RUNNING = READY_QUEUE . dequeue ( );
process_status = RUNNING;
```

The actions to be executed as a consequence of an accept statement for entry e (not embedded in a select statement) are

```
if (DESCR (e) . W) . empty ( ) {
    DESCR (e) . O = true;
    DESCR (e) . T = process_status;
    RUNNING = READY_QUEUE . dequeue ( )
    process_status = RUNNING;
}       -- if the waiting queue is not empty, then simply continue
        -- executing the accept body
```

To execute a select statement, a list of the open entries involved in the selection is first constructed. If this list is empty, then the exception PROGRAM_ERROR is raised. Otherwise, the following kernel actions are required.

```
if for all e in the list (DESCR (e) . W) . empty ( ) = true {
     for all e in the list {
        DESCR (e) . O = true;
        DESCR (e) . T =  process_status;
    };
    RUNNING = READY_QUEUE . dequeue ( )
    process_status = RUNNING;
else
    choose an e with( DESCR (e) . W) . empty ( ) = false
    proceed execution from instruction DESCR (e) . I
}
```

4.9 Bibliographic note

Statement-level control structures were the subject of active research in the late 60’s and early 70’s. E.W. Dijkstra was first to stress the need for discipline in programming, and the influence of unconstrained jumps (goto statements) on the production of obscure, unstructured programs (Dijkstra 1968a). Much of the subsequent research on "structured programming" was aimed at uncovering suitable control structures that could promote the writing of well-organized, readable programs. For a comprehensive retrospective view of this work, the reader may refer to (Knuth 1974).

Research on exception handling began in the early 70’s (Goodenough 1975). The main directions of investigations were design methods and language constructs to deal with exceptions. For a comprehensive survey of the field, the reader may refer to (Cristian ***). The discussion reported in the chapter on exception handling of (Stroustrup 1994) is an excellent account of the tradeoffs that must be considered by a language designer in the definition of certain programming language features. For a detailed understanding of the different choices made by different programming languages, the reader should refer to the specific bibliographic sources (see the Glossary).

Backtracking and and-or graphs are presented in most textbooks on computer algorithms, such as (Horowitz and Sahni 1978). They are also often discussed in the context of artificial intelligence methodologies (***). Event-driven control structures in the context of database applications are surveyed in (Fraternali and Tanca ***).

An extensive study of coroutines is reported by (Marlin 1980). This includes a survey of languages, a semantic description of the concept, and a discussion of programming language methodologies.

Concurrency in programming languages is often studied either as part of an operating systems course or as a separate course on concurrent programming. (Andrews 1991) and (Ben Ari 1990) provide an in-depth coverage of concurrent and distributed programing. Historically, the concept of semaphore was introduced in (Dijkstra 1968b). Monitors were introduced by (Brinch Hansen 1973) and (Hoare 1974). (Hoare 1978) is a classical in the literature on message passing,. It strongly influenced the rendezvous concept of Ada.

4.10 Exercises

1. Study the case statement of Pascal and compare it to the C++ switch statement and the Ada case statement.
2. What is the minimum possible number of iterations of the body of a Pascal while loop? How about a repeat loop?
3. It can be shown that, in principle, any program can be written using just these control structures:

- grouping statements into a block;
- if. . .then. . .else. . .
- while. . .do. . .
    - Show how other control structures (such as the case statement or a repeat loop) can be represented using the above minimal set;
    - Discuss why in practice a richer set of control structures is provided by a programming language.

4. Show how pointers to procedures (or functions) can be used in Ada to pass procedures (or functions) as parameters.
5. Ada, as originally defined, did not allow procedures or functions to be passed as parameters to a procedure (or function). Can this drawback be overcome by the use of generics? How? What are the differences with respect to passing a routine as a parameter?
6. Explain why aliasing makes the effect of implementing parameter passing by reference and by value result different. Give an example.
7. Check on the Ada manual how the language specifies what happens when the effects of passing parameters by reference and by value-result are different.
8. What are the strings matched by the following SNOBOL4 pattern?

```
OPERATOR = '+', '-'
EXPRESSION = *EXPRESSION OPERATOR *EXPRESSION | IDENTIFIER
```

9. Ada provides features to transfer specific information on the point where an exception is raised to the corresponding handler. Check on the language manual how this can be accomplished and show how these features can be used during debugging.
10. Compare the feature provided by Ada to disable exception (see the language manual) with the disabling mechanism provided by PL/I.
11. How can the exception handling facilities of C++ be used to achieve the same effect as that described by the Eiffel fragment of Section 5.4.3?

solution

```
int i
while i < n
try {
    execute method i
}
catch fail {
    i++;
    if i = n then throw
}
```

12. Suppose you have a program unit U that calls a function fun, which may raise exceptions and propagates them back to U. There are five kinds of exceptions that can be propagated: V, X, Y, Z, and W. An exception of kind V allows U to do some fixing and then re-invoke fun. An exception of kind X allows U to do some clean-up of the local state and then proceed normally. An exception of kind Y allows U to simply propagate the same exception. An exception of kind Z allows U to perform some action, and then turn the received exception into another exception which is raised. Finally, an exception of kind W forces U to terminate the entire program.
    Provide an implementation of this scheme in C++, Ada, and Eiffel.
13. Write a short assessment of exception handling in ML, according to the style of the assessment we did in Section 5.4.4.
14. Examine the manuals of a few languages of your choice to find out what happens if an exception is raised while an exception is being handled.
15. Discuss how memory is managed for coroutines, assuming a block structured language where coroutines can be declared locally inside (co)routines. Thus, the creation of a set of coroutines can be viewed as the creation of a new execution stack, one for each coroutine.

16. In the producer-consumer example implemented with semaphores in Section 4.8.3.1, suppose that V (mutex) is written incorrectly as P (mutex) in process Producer. How does the system behave?
17. When semaphores are used to implement mutual exclusion, it is possible to associate a semaphore SR with each resource R. Each access to R can then be written as

```
P (SR);
access R;
V (SR)
```

- What should the initial value of SR be?

18. Some computers provide an indivisible machine-instruction *test and set* (TS) that can be used for synchronization purposes. Let X and Y be two boolean variables. The execution of the instruction TS (X, Y) copies the value of Y into X and sets Y to false. A set of concurrent processes that must execute some instructions in mutual exclusion can use a global boolean variable PERMIT, initialized to true, and a local boolean variable X in the following way:

```
repeat TS (X, PERMIT)
until X;
instructions to be executed  in mutual exclusion;
PERMIT:= true
```

- In this case, processes do not suspend themselves; they are always executing (this is called busy waiting). Compare this solution to one based on semaphores in which P and V are implemented by the kernel.
- Describe how to implement P and V on semaphores by using the test and set primitive in a busy wait scheme.

19. Use protected types in Ada to implement semaphores.
20. Define an Ada protected type to implement a shared protected variable that can be read and written in mutual exclusion.
21. How can you define task types in Ada? What are the main differences between protected types and task types?
22. We implemented mutual exclusion of monitor procedures by disabling interrupts. An alternative solution uses a semaphore for each monitor and performs a P on the semaphore before entering a monitor procedure, and a corresponding V upon exit. Detail this implementation and compare the two solutions.
23. Show how an Ada task can be used to implement a semaphore.
24. Show how an Ada protected type can be used to implement semaphores.
25. Design an Ada package that implements the abstract data type queue that is used in the abstract implementation of concurrency in Section 5.8.3
26. Design a C++ class that implements the abstract data type queue that is used in the abstract implementation of concurrency in Section 5.8.3

Structuring the program

CHAPTER 5

The basic mechanisms described in previous chapters for structuring data (Chapter 3) and computation (Chapter 4) may be used for programming in the small. In Chapter 4, we also have seen the use of control structures for structuring large programs. This chapter deals strictly with issues of programming in the large. We describe the basic concepts for structuring large programs (encapsulation, interfaces, information hiding) and the mechanisms provided by languages to support it (packaging, separate compilation). We also consider the concept of genericity in building software component libraries. We do not go deeply into object-oriented programming, which is the subject of the next chapter.

The production of large programs—those consisting of more than several thousand lines—presents challenging problems that do not arise when developing smaller programs. The same methods and techniques that work well with small programs just don't "scale up." To stress the differences between small and large systems production, we refer to "programming in the small" and "programming in the large."

Two fundamental principles—*abstraction* and *modularity*—underlie all approaches to programming in the large. Abstraction allows us to understand and analyze the problem by concentrating on its important aspects. Modularity allows us to design and build the program from smaller pieces called modules. During problem analysis, we discover and invent abstractions that allow us to understand the problem. During program design and implementation, we try to discover a modular structure for the program. In general, if modules that implement the program correspond closely to abstractions discovered during problem analysis, the program will be easier to understand and manage. The principles of modularity and abstraction help us apply the well-known problem solving strategy known as “divide and conquer.”

The concept of a “large program” is difficult to define precisely. We certainly do not want to equate the size of a program (e.g., the number of source statements) with its complexity. Largeness relates more to the “size” and complexity of the problem being solved than to the final size of a program in terms of the number of source lines. Often, however, the size of a program is a good indication of the complexity of the problem being solved. Consider the task of building an airline reservation system. The system is expected to keep a database of flight information. Reservation agents working at remote sites may access the database at arbitrary times and in any order. They may inquire about flight information, such as time and price; make or cancel a reservation on a particular flight; update existing information, such as the local telephone number for a passenger. Certain authorized personnel can access the database to do special operations, such as adding or canceling a flight, or changing the type of the airplane assigned to a flight. Others may access the system to obtain statistical data about a particular flight or all flights.

A problem of this magnitude imposes severe restrictions on the solution strategy and the following key requirements:

- The system has to function correctly. A seemingly small error, such as assignment to the wrong pointer, may lead to losing a reservation list or interchanging two different lists and be extremely costly. To guarantee correctness of the system virtually any cost can be tolerated.
- The system is “long-lived.” The cost associated with producing such a system is so high that it is not practical to replace it with a totally new system. It is expected that the cost will be recouped only over a long period of time.
- During its lifetime, the system undergoes considerable modification. For our example, because of completely unforeseen new government regulations, changes might be required in price structure, a new type of airplane might be added, and so on. Other changes might be considered because experience with the system has uncovered new requirements. We might find it desirable to have the system find the best route automatically by trying different connections.
- Because of the magnitude of the problem, many people—tens or hundreds— are

involved in the development of the system.

The study of these problems and their solutions is outside the scope of this book: they are studied in software engineering. This chapter deals with requirements that these issues place on the programming language. Thus, this chapter is about program organization issues. Section 5.1 reviews software design methods. Design methods provide guidelines for applying divide and conquer in software design. In Section 5.2 we discuss the concepts of encapsulation, interface, separate compilation, and module libraries. These concepts provide the bases for the application of modularity in programming languages. Case studies of different languages are provided in Section 5.3.

5.1 Software design methods

To combat the complexities of programming in the large, we need a systematic design method that guides us in composing a large program out of smaller units—which we call *modules*. A good design is composed of modules that interact with one another in well-defined and controlled ways. Consequently, each module can be designed, understood, and validated independently of the other modules. Once we have achieved such a design, we need programming language facilities that help us in implementing these independent modules, their relationships, and their interactions.

The goal of software design is to find an appropriate modular decomposition of the desired system. Indeed, even though the boundaries between programming in the large and programming in the small cannot be stated rigorously, we may say that programming in the large addresses the problem of modular system decomposition, and programming in the small refers to the production of individual modules. A good modular decomposition is one that is based on modules that are as independent from each other as possible. There are many methods for achieving such modularity. A well-known approach is *information hiding* which uses the distribution of "secrets" as the basis for modular decomposition. Each module hides a particular design decision as its secret. The idea is that if design decisions have to be changed, only the module that "knows" the secret design decision needs to be modified and the other modules remain unaffected.

In Chapter 1 we discussed the importance of software design. If a design is composed of highly independent modules, it supports the requirements of large programs:

- Independent modules form the basis of work assignment to individual team members. The more independent the modules are, the more independently the team members can proceed in their work.
- The correctness of the entire system may be based on the correctness of the individual modules. The more independent the modules are, the more easily the correctness of the individual modules may be established.
- Defects in the system may be repaired and, in general, the system may be enhanced more easily because modifications may be isolated to individual modules.

5.2 Concepts in support of modularity

To summarize the discussion of the last section, the key to software design is modularization. A good module represents a useful abstraction; it interacts with other modules in well-defined and regular ways; it may be understood, designed, implemented, compiled, and enhanced with access to only the specification (not the implementation secrets) of other modules. Programming languages provide facilities for building programs in terms of constituent modules. In this chapter, we are interested in programming language concepts and facilities that help the programmer in dividing a program into subparts—modules—the relationships among those modules and the extent to which program decompositions can mirror the decomposition of the design.

We have already seen some units of program decomposition in Chapters 3 and 4. Procedures and functions are an effective way of breaking a program into two modules: one which provides a service and another which uses the service. We may say that the procedure is a server or service provider and the caller is a client. Even at this level we can see some of the differences between different types of modularization units. For example, if we provide a service as a function, then the client has to use the service in an expression. On the other hand, if we provide the service in a procedure, then the client may not use it in an expression and is forced to use a more assignment-oriented or imperative style.

Procedures and functions are units for structuring small programs, perhaps limited to a single file. Sometimes, we may want to organize a set of related functions and procedures together as a unit. For example, we saw in Chapter 3 how the class construct of C++ lets us group together a data structure and related operations. Ada and Modula-2 provide other constructs for this purpose. Before we delve into specific language facilities, we will first look at some of the underlying concepts of modularity. These concepts help motivate the need for the language facilities and help us compare the different language approaches.

5.2.1 Encapsulation

A program unit provides a service that may be used by other parts of the program, called the *clients* of the service. The unit is said to *encapsulate* the service. The purpose of encapsulation is to group together the program components that combine to provide a service and to make only the relevant aspects visible to clients. *Information hiding* is a design method that emphasizes the importance of concealing information as the basis for modularization. Encapsulation mechanisms are linguistic constructs that support the implementation of information hiding modules. Through encapsulation, a module is clearly described by two parts: the specification and the implementation. The specification describes how the services provided by the module can be accessed by clients. The implementation describes the module's internal secrets that provide the specified services.

For example, assume that a program unit implements a dictionary data structure that other units may use to store and retrieve <name, "id"> pairs. This dictionary unit makes available to its clients operations for: inserting a pair, such as <"Mehdi", 46>, retrieving elements by supplying the string component of a pair, and deleting elements by supplying the string component of a pair. The unit uses other helper routines and data structures to implement its service. The purpose of encapsulation is to ensure that the internal structure of the dictionary unit is *hidden* from the clients. By making visible to clients only those parts of the dictionary unit that they need to know, we achieve two important properties.

- The client is simplified: clients do not need to know how the unit works in order to be able to use it; and
- The service implementation is independent of clients and may be modified without affecting the clients.

Different languages provide different encapsulation facilities. For example, in C, a file is the unit of encapsulation. Typically, the entities declared at the head of a file are visible to the functions in that file and are also made available to functions in other files if those functions choose to declare them. The declaration:

```
extern int max;
```

states that the variable max to be used here, is defined—and storage for it allocated—elsewhere. Variables declared in a C function are local and known only to that function. Variables declared outside of functions are assumed to be available to other units, if they declare them using the extern specifier. But a unit may decide to hide such variables from other units by declaring them as static.

We have already seen the class construct of C++ in Chapter 3 which makes only a subset of the defined entities—those declared as public—available to clients. All other class information is hidden. Figure 64 is a C++ class that declares the dictionary service mentioned above.

```
class dict {
public:
    dict();              //to initialize a dictionary
    ~dict();             //to remove a dictionary
    void insert (char* c, int i);
    int lookup(char* c);
    remove (char* c);
private:
    struct node {
        node* next;
        char* name;
        int id};
    node* root;
};
```

FIGURE 64. Interface of a dictionary module in C++

This program declares five publicly available functions. As we know from Chapter 4, the first two functions, dict() and ~dict(), may be used to create and clean up a dictionary object, respectively. The other three functions may be used to access the dictionary object. The private part of the class defines the representation of a node of a dictionary and the root of the dictionary. This part of the declaration is not visible to the users of the class. The module encapsulates the dictionary service, both providing access and hiding unnecessary details.

As we have seen also in Chapter 3, the built-in types of a language are examples of encapsulated types. They hide the representation of the instances of those types and allow only legal operations to be performed on those instances.

5.2.2 Interface and implementation

A module encapsulates a set of entities and provides access to some of those entities. The available entities are said to be *exported* by the module. Each of the exported entities is available through an interface. The collection of the interfaces of the exported entities form the *module interface*. Clients request the services provided by a module using the module's *interface*, which describes the module's specification. The interface specifies the syntax of service requests. Some languages also support or require the specification of the interface's semantic requirements. The idea is that the interface is all that the client needs to know about the provider's unit. The implementation of the unit is hidden from the client. The separation of the interface from the implementation contributes to the independence of the client and the server from one another.

A service provider *exports* a set of entities to its clients. A client module *imports* those entities to be able to use the services of the provider module. The exported entities comprise the service provided by the module. Some languages have implicit and others explicit mechanisms for import and export of entities. Languages also differ with respect to the kinds of entities they allow to be exported. For example, some languages allow a type to be exported and others do not.

The simplest interface, one that we have already seen in Chapter 4, is a procedure or function interface. A function declaration such as:

```
int max (int& x, int& y)
```

specifies to the clients that the function max may be called by passing to it two integers; the function will return an integer result. We introduced the term *signature* to refer to these requirements on input and output for procedures and functions. Procedure signatures form the basis of type-checking across procedures. The name of the function, max, is intended to convey something about the semantics of the function, namely that the integer it will return is the maximum of the two integer input parameters. Ideally, the interface would specify the semantics and the requirements on parameters (for example that they must be positive integers). Most programming languages do not support such facilities, however, and they are left as the task of the designer to be documented in the design documents. An exception is the Eiffel language. In Chapter 3, we saw the use of preconditions and postconditions to specify such semantic requirements for procedures, functions, and classes.

In C++, where the unit of encapsulation is a class, the interface to the class consists of the interfaces of all the member functions of the class that are available to clients as well as any other entities, such as types and variables, that are made public by the unit. The public entities defined in Figure 64 constitute the interface of the dictionary unit.

Ada treats the separation of interface and implementation quite strictly. In Ada, the unit of encapsulation is a package. A package encapsulates a set of entities such as procedures, functions, variables, and types. The package interface consists of the interfaces provided by each of those entities. The Ada package supports encapsulation by requiring the interface of a package (called **package specification**) to be declared separately from the implementation of the package (called **package body**). Figure 65 shows the Ada package specification for our dictionary unit. The implementation of the dictionary package is shown in Figure 66. The package body contains all the implementation details that are hidden from the clients. This separation helps achieve both of the goals stated for encapsulation in Section 5.2.1. The package body, as can be seen in the figure, defines both the implementation of the entities defined in the package interface and the implementation of other entities internal to the module. These entities are completely hidden from the clients of the package. The package specification and the package body may appear in different files and need not even be compiled together. To write and compile a client module, only the service's package specification is necessary.

There are significant differences between the packages of Ada and classes of C++. Even from this simple example we can see a difference between the models supported by C++ and Ada. In C++, the client can declare several instances of the dictionary class. In Ada, on the other hand, a module may be declared once only and the client obtains access to only a single dictionary.

```
package Dictionary is
    procedure insert (C:String; I: Integer);
    function lookup(C:String): Integer;
    procedure remove (C: String);
end Dictionary;
```

FIGURE 65. Package specification in Ada

```
package body Dictionary is
    type node;
    type node_ptr is access node;
    type node is
        record
            name: String;
            id: Integer;
            next: node_ptr;
        end record;
    root: node_ptr;
    procedure insert (C:String; I: Integer) is
    begin
        --imlementation...
    end insert;
    function lookup(C:String): Integer is
    begin
        --imlementation...
    end lookup;
    procedure remove (C: String) is
    begin
        --imlementation...
    end remove;
begin
    root := null;
end Dictionary;
```

FIGURE 66. Package body in Ada

5.2.3 Separate and independent compilation

The idea of modularity is to enable the construction of large programs out of smaller parts that are developed independently. At the implementation level, independent development of modules implies that they may be compiled and tested individually, independently of the rest of the program. This is referred to as *independent* compilation. The term *separate* compilation is used to refer to the ability to compile units individually but subject to certain ordering constraints. For example, C supports independent compilation and Ada supports separate compilation. In Ada, as we will see later, some units may not be compiled until other units have been compiled. The ordering is imposed to allow checking of interunit references. With independent compilation, normally there is no static checking of entities imported by a module.

To illustrate this point, consider the program sketch in Figure 67, written in a hypothetical programming language. Separate compilation means that unit B, which imports routine X from unit A, must be compiled after A. This allows any call to X issued by B to be checked statically against X's definition in A. If the language allows module interfaces to be compiled separately from their bodies, only A's interface must be compiled before B; its body can be compiled at any time after its corresponding interface has been compiled.

```
Unit A                                  Unit B
    export routine X (int, int);            . . .
    . . .                                   call X (. . .);
end A                                       . . .
                                        end B
```

FIGURE 67. Sketch of a program composed of two units

Independent or separate compilation is a necessity in the development of large programs because it allows different programmers to work concurrently on different parts of the program. It is also impractical to recompile thousands of modules when only a few modules have changed. Language concepts and features are available to allow implementations to determine the fewest number of units that must be recompiled. In general, programming languages define:

- the unit of compilation: what may be compiled independently?
- the order of compilation: are compilation units required to be compiled in any particular order?
- amount of checking between separately-compiled modules: are inter-unit interactions checked for validity?

The issue of separate compilation is at the border of the language definition and its implementation. Clearly, if the language requires inter-unit checking to be performed, this implies a programming environment that is able to check module implementations against the interfaces of compilation units from which they import services, for example a type-checking linker. Interface-checking of separately compiled modules is analogous to static type-checking for programming in the small: both are aimed at the development of safe and reliable programs.

5.2.4 Libraries of modules

We have seen that C++ class and Ada's package make it possible to group related entities into a single unit. But large programs consist of hundreds or even thousands of such units. To control the complexity of dealing with the large number of entities exported by all these units, it is essential to be able to organize these units into related groups. For example, it is difficult to ensure that all the thousands of units have unique names! In general, we can always find groupings of units that are related rather closely.

A common example of a grouping of related services is a library of modules such as a library of matrix manipulation routines. A library collects together a number of related and commonly used services. Clients typically need to make use of different libraries in the same program and since libraries are written by different people, the names in different libraries may conflict. For example, a library for manipulating lists and a library for manipulating dictionaries may both export procedures named insert. Mechanisms are needed for clients to conveniently distinguish between such identically-named services. We have seen that the dot notation helps with this problem at the module level. But consider trying to use two different releases of the same library at the same time. How can you use some of the entities from one release and some from the other? Both C++ and Ada have recent additions to the language to deal with these issues. We will describe these facilities when we discuss specific languages: namespaces of C++ on page 295 and child libraries of Ada on page 302.

5.3 Language features for programming in the large

We have so far discussed the concepts of programming in the large with isolated examples from programming languages. In this section we look at some interesting ways that existing programming languages support—or do not support—the programming in the large concepts. All programming languages provide features for decomposing programs into smaller and largely autonomous units. We refer to such units as *physical* modules; we will use the term *logical* module to denote a module identified at the design stage. A logical module represents an abstraction identified at the design stage by the designer. A logical module may be implemented by one or more physical modules. The closer the relationship between the physical modules and logical modules is, the better the physical program organization reflects the logical design structure.

We will discuss the relevant aspects of each language based on the following points:

- Module encapsulation: What is the unit of modularity and encapsulation supported by the language, and how well does it support different programming paradigms?
- Separation of interface from implementation: What is the relationship among modules that form a program? What entities may be exported and imported by a module?
- Program organization and module groupings: How independently can physical modules be implemented and compiled? What are the visibility and access control mechanisms supported by the language?

We will discuss Pascal, C, C++, Ada, and ML. Pascal and C are viewed here as a representative of the class of traditional, minimalist, procedural languages. Our conclusions about them hold, with minor changes, for other members of the class such as FORTRAN. C++ is a representative of class-based languages. Ada is a representative of module-based languages, although the 1995 version of the language has enhanced its object-orientation support. ML is reviewed as a representative of functional languages. A few comments on other languages will be given in Section 5.3.6.

In general, our discussion here is not about programming paradigms. Object-oriented and functional programming support will be covered in, respectively, Chapters 6 and 7.

5.3.1 Pascal

In this section we provide an assessment of Pascal's features for programming in the large. Since many dialects and extensions of Pascal exist, here we consider the original version of the language. Most of the inconveniences discussed here have been eliminated by the enhancements provided by modern implementations.

The only features provided by Pascal for decomposing a program into modules are procedures and functions, which can be used to implement procedural abstractions. The language thus only supports procedural programming. Some later versions of the language have modified the original version of Pascal extensively by adding object-oriented programming features.

A Pascal program has the following structure.

```
program program_name (files);
declarations of constants, types, variables, procedures and functions;
begin
    statements (no declarations)
end.
```

A program consists of declarations and operations. The operations are either the built-in ones provided by the language or those declared as functions and procedures. A procedure or function itself may contain the declaration of constants, types, variables, and other procedures and functions. The organization of a Pascal program is thus a tree structure of modules (see static nesting tree in Section 2.6.4 on page 104). The tree structure represents the textual nesting of lower-level modules. Nesting is used to control the scope of names declared within modules, according to the static binding rule presented in Section 2.6.4.

To assess the structure of Pascal programs, consider the following example. Suppose that the top-down modular design of a module A identifies two modules B and C providing subsidiary procedural abstractions. Similarly, module B invokes two private procedural abstractions provided by modules D and E. Module C invokes a private procedural abstraction provided by F. Figure 68 shows a nesting structure for a program that satisfies the design constraints.

A basic problem with the solution of Figure 68 is that the structure does not enforce the restrictions on procedure invocations found at the design stage. Actually, the structure allows for the possibility of several other invocations. For example E can invoke D, B, and A; C can invoke B and A, and so on. On the other hand, the structure of Figure 68 imposes some restrictions that might become undesirable. For example, if we discover that module F needs the procedural abstraction provided by module E, the current structure is no longer adequate. Figure 69 shows a rearrangement of the program structure that is compatible with this new requirement. The problem with this new organization is that the structure no longer displays the hierarchical decomposition of abstractions. Module E appears to be a subsidiary abstraction used by A, although the only reason for its placement at that level in the tree is that both modules B and F need to refer to it. Similar problems occur for variables, constants and types. The tree structure provides indiscriminate access to variables declared in enclosing modules. In addition, if any two modules M and N need to share a variable, this variable must be declared in a module that statically encloses both M and N and thus the variable becomes accessible to any other modules enclosed by this enclosing module.

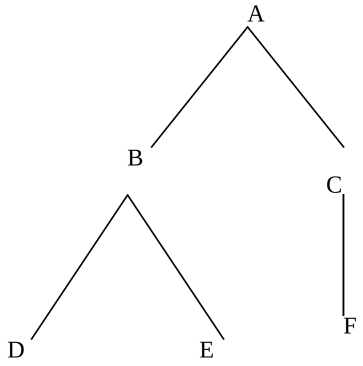


FIGURE 68. Static nesting tree of a hypothetical Pascal program

Further problems are caused by the textual layout of Pascal programs. The entire program is a single monolithic text. If the program is large, module boundaries are not immediately visible, even if the programmer uses careful conventions for indentation. A module heading can appear well before its body, because of intervening inner module declarations. Consequently, programs can be difficult to read and modify.

The problems with Pascal discussed in this section stem from block structure, and therefore hold for other ALGOL-like languages. Block structure is ade-

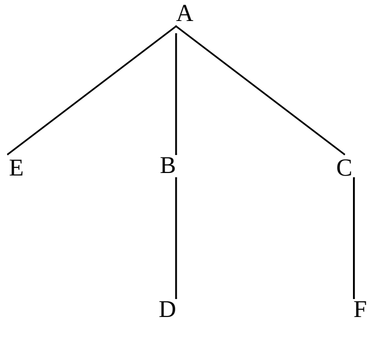


FIGURE 69.A rearrangement of the program structure of Figure 68.

quate for programming in the small because it supports stepwise refinement quite naturally. It is not so valuable for structuring large programs. The program structure resulting from nesting may interfere with the logical structure found during design. This can impair the writability, readability, and modifiability of programs.

Another important question to address is how Pascal modules can be developed independently, and how long-lived and reusable they are. These goals are achieved by applying information hiding at the design stage to obtain a clean definition of module interfaces. In addition, it is desirable to support the separate implementation of modules. It should be possible to compile and certify modules separately. Separately compiled and tested modules should be kept in a library, ready for later reuse.

The original Pascal Report does not address these issues, although most Pascal implementation provided their own solutions. Thus, a number of important questions are left unanswered by the original Report, such as

- What program entities can a separate compilation unit export?
- How is a unit interface specified?
- What amount of type checking across unit interfaces is prescribed to occur?

Different implementations have indeed adopted different solutions to these points. As a result, Pascal programs developed on different platforms may be incompatible. For example, some implementations allow outer-level procedures and functions to be compiled independently. Independently compiled unitsare assembled via a standard linker, which resolves the bindings between the entities imported by each module and the corresponding entities exported by other modules. No intermodule checkes are performed, however, to verify that, say, a call to an external procedure is inconsistent with the corresponding procedure declaration. Errors of this kind might remain uncaught, unfortunately. There are modern implementations of Pascal, such as Turbo Pascal, however, which provide safer separate-compilation facilities based on the notion of a module that encapsulates a set of constants, procedures and types.

5.3.2 C

C provides functions to decompose a program into procedural abstractions. In addition, it relies on a minimum of language features and a number of conventions to support programming in the large. These conventions are well recognized by C programmers and are even reflected in tools that have been developed to support the language. Indeed, a major portion of the programming in the large support is provided by the file-inclusion commands of the C preprocessor. Thus, even though the compiler does not provide any explicit support or checking for inter-module interaction, the combination of conventions and the preprocessor has proven in practice to be an adequate and popular way to support programming in the large.

The C unit of physical modularity is a file. A logical module is implemented in C by two physical modules (files) which we may roughly call the module’s interface and its implementation. The interface, called a “header” or an “include” file, declares all symbols exported by the module and thus available to the clients of the module. The header file contains the information necessary to satisfy the type system when the client modules are compiled. The implementation file of the module contains the private part of the module and implements the exported services. A client module needing to use the functionality of another module “includes” the header file of the provider module. A header file may declare constants, type definitions, variables, and functions. Only the prototype of the function—its signature—is given by the declaration; the function definition appears in the implementation file. Functions may not be nested. Any names defined in a file are known throughout that file and may also be known outside of that file.

The header files are used to resolve inter-module references at compile-time. At link-time, all implementation files are searched to resolve inter-module (i.e. inter-file) references. The header file is usually named with a .h extension and the implementation file is named with a .c extension. These conventions have largely overcome the lack of any explicit support for program organization.

Figure 70 shows the header and implementation files for a module providing a stack data structure. language provides no encapsulation facilities. For example, the main program in Figure 70 has complete access to the internal structure of the stacks s1 and s2. In fact, this property is used by the main program to initialize the stacks s1 and s2 to set their stack pointers (top) to 0. There are ways to implement this program to reduce this interference between client and server but all depend on the care taken by the programmer. There is no control over what is exported: by default, all entities in a file are exported. Files may be compiled separately and inter-file references are resolved at link time with no type-checking. A file may be compiled as long as all the files it includes are available.

```
/* file stack.h */
/*declarations exported to clients*/
typedef struct stack {
    int elments[100]; /* stack of 100 ints */
    int top;          /*number of elements*/
};
extern void push(stack, int);
extern int pop(stack);
/* end of file stack.h */
/*****----------------------end of file        ****/

/*file stack.c */
/*implementation of stack operations*/
#include"stack.h"
void push(stack s, int i) {
    s.elements[s.top++] = i;
};
int pop (stack s) {
    return --s.top;
};
/*****----------------------end of file        ****/

/*file main.c */
/*A client of stack*/
#include "stack.h"
void main(){
    stack s1, s2;          /*declare two stacks */
    s1.top = 0; s2.top = 0; /* initialize them */
    int i;
    push (s1, 5); /* push something on first stack */
    push (s2, 6); /* push something on second stack*/
    ...
    i = pop(s1);   /* pop first stack */
    ...
}
```

FIGURE 70. Separate files implementing and using a stack in C

The general structure of a C file is shown in Figure 72. All files have similar structure except that one of the files (only) must contain a function named main, which is called to start the program. Because functions are not allowed to be nested in C, the nesting problems of Pascal do not occur.

```
#include ...various files...
global declarations
function definitions
void main (parameters)
{
...one main function needed
...in a program
}
```

FIGURE 72. Structure of a C module

Any names defined in the outer level of a file are implicitly known globally. These include the names of all the functions defined in the file and any other entities defined outside of those functions. There are two ways to control such indiscriminate dispersion of names.

- A module wanting to use an entity that is defined externally must declare such entities as being externally defined.
- A module wanting to limit the scope of one of its defined entities to be local to itself only may declare such an entity to be static.

The following two lines import the integer variable maximum_length and hides the integer variable local_size from other modules.

```
extern int maximum_length;
static int local_size;
```

There are no explicit import/export facilities. All control over module independence relies on convention and implementer competence.

5.3.3 C++

C++ is based on C and it shares C’s reliance on conventions and unit of physical modularity as the file. As C, C++ provides functions as a decomposition construct to implement abstract operations. Nevertheless, C++’s most important enhancements to C are in the area of programming in the large. In particular, the class construct of C++ provides a unit of logical modularity that supports the implementation of information hiding modules and abstract data types. Combined with templates, classes may be used to implement generic abstract data types. The class provides encapsulation and control over interfaces. In this chapter, we review the use of classes as modules. We will examine the use of classes to support object-oriented programming in Chapter 6.

5.3.3.1 Encapsulation in C++

The unit of logical modularity in C++ is the class. A class serves several purposes including:

- A class defines a new (user-defined) data type.
- A class defines an encapsulated unit.

Entities defined by a class are either *public*—exported to clients—or *private*—hidden from clients. There are also *protected* variables which will be discussed in the next chapter.

Since a class defines a user-defined type, to use the services offered by a class, the client must create an instance of the class, called an *object*, and use that object. C++ supports the style of programming in which programmers write applications by extending the types of the language with user-defined types. Class derivation is a mechanism that supports the definition of new types based on existing types. We will examine this in more detail in the next chapter.

Classes may be nested. But as we saw in the case of Pascal, nesting may be used only for programming in the small and is of limited utility for programming in the large.

Both classes and functions may be generic, supporting a generic programming style. We will discuss generic units in Section 5.4.

5.3.3.2 Program organization

Classes define the abstractions from which the program is to be composed. The main program or a client creates instances of the classes and calls on them to perform the desired task. We saw the definition of a C++ template module implementing a generic abstract data type stack in Chapter 3. Figure 73 shows a class implementing a stack of integers[1]. The implementation separates the interface and the implementation in different files.

[1] The same problem can of course be solved by instantiating the generic class.

```
/* file stack.H */
/*declarations exported to clients*/
class stack {
public:
     stack();
     void push(int);
     int push pop();
private:
     int elments[100]; /* stack represented as array */
     int top = 0;            /*number of elements*/
};
// the implementation follows and may be in a separate file
void stack::push( int i) {
     elements[top++] = i;
};
int stack::pop (int i) {
     return elements[--top];
};
/*end of stack.H*/

/*main.c */
/*A client of stack*/
#include “stack.h”
main(){
     stack s1, s2;  /*declare two stacks */
     int i;
     s1.push (5); /* push something on first stack */
     s2.push (6); /* push something on second stack*/
     ...
     i = s1.pop();   /* pop first stack */
...  ...
}
```

FIGURE 73. Stack class in C++

Some points to observe about this program are:

In the main program, stacks are declared in the same way that variables of language-defined types are declared. The operations exported by stack, push and pop, are called in the main program by using the dot notation and accessing the desired operation of the appropriate stack objects (s1 or s2). The definitions of the operations push and pop may appear in the class body or outside of it. Finally, the compiler will try to expand the code of the member functions in-line, if possible, to avoid the overhead of a procedure call.

C++ supports the development of independent modules (but does not enforce it):

1. A class's interface and implementation may be separated and even compiled separately from each other. The implementation must include the interface definition and therefore must be compiled after the interface file exists.
2. Client modules may be compiled with access to only the interface modules of the service providers and not their implementation modules.
3. Any names defined in a class are local to the class unless explicitly declared to be public. Even so, client modules must use the class name to gain access to the names internal to the class.

5.3.3.3 Grouping of units

C++ has several mechanisms for relating classes to each other. First, classes may be nested. As we have said before, this is a programming in the small feature. Two other mechanisms, "friend" functions and namespaces, are discussed next.

Friend functions. A class in C++ defines a user-defined type. As a result, the operations it defines as public are operations on objects of that type. Some operations do not naturally belong to one object or another. For example, if we define a class for complex numbers, it may have a data part that stores the real and imaginary parts of the number, along with exported operations that let clients create and manipulate objects of type complex. But what about an addition operation that takes two complex objects to add together? Which of the two complex objects is the operation a member of? As another example, consider defining a function that multiplies a vector with a matrix. Should this function be a member of the vector class or the matrix class? To be able to implement such functions efficiently, they need to have access to the private parts of the objects they manipulate but they do not really belong to a particular object. Module-based languages such as Ada and Modula-2 allow these related entities to be packaged together in a single module. A class-based language such as C++ must adopt a different solution. In C++, a class can grant access to its private parts by declaring certain functions as its "friend". Friend functions have the same rights as member functions of the class but are otherwise normal global functions.

Figure 74 shows the definition of a complex number class. The class defines the type complex which is internally composed of two doubles, representing the real and imaginary parts of a complex number. These are hidden from clients. The class exports a method of constructing a complex number out of two doubles. Thus, the following declaration creates two complex numbers:

```
complex x(1.0, 2.0), y(2.5, 3.5);
```

The other declarations state that the operator functions to be defined later (+, -, *, and /) are friends of the class complex and thus may access the private parts of the class. They are not member functions of the class and they are not exported by the class. They are simply given preferential treatment by the class. Of course, friend functions, even though not exported, are visible to clients because they are global functions.

```
class complex {
public:
    complex(double r, double i ){re = r; im = i;}

    friend complex operator+ (complex, complex);
    friend complex operator- (complex, complex);
    friend complex operator* (complex, complex);
    friend complex operator/ (complex, complex);
private:
    double re, im;
};
```

FIGURE 74. Illustration of the use of friend declarations in C++

Defining these operators as friend functions allows the clients to naturally use these functions as binary operations such as:

```
complex c = x + y;
```

If the operation + was made a member of the class, the notation for clients would be quite awkward. For example, we might have had to write something like:

```
c.add(x)
```

in order to add the complex x to complex c.

The requirement for friend functions is a direct consequence of C++'s use of classes as user-defined types. In a language like Ada where the package is used not to define types but to group related entities, we would naturally group together type definitions for complex and its related functions in the same package. The functions automatically gain access to the private parts of the package because they are part of the package. In both cases, any changes to the representation of the data may require changes to the functions, whether they are part of a package or they are friend functions.

Namespaces. In C and in C++, the unit of global naming is a file. Any names defined at the outer level of a file are known globally by default. For example, the names of all classes defined in a library are known to any client that includes that file. What if two libraries provide two classes with the same name? How can a client use both of those classes? How can a library provider add a new service to its library and be sure that the new name of the service does not conflict with any existing uses of the clients? Since names are created by independent people, a single global name space is a serious problem in the development of large programs. The solution of C++ is to partition the global name space into a smaller groups; each group is called a *namespace*. The names defined in a namespace are independent from those in any other namespace and may be referenced by supplying the name of the namespace. This mechanisms enables library providers to provide their libraries in their own namespaces with a guarantee of independence from other library providers. Of course, it is necessary for the names of the namespaces themselves to be unique.

For example, consider the XYZ Corp. that provides a library of classes for manipulating turbine engines. It might provide its library in a namespace XYZCorp:

```
namespace XYZCorp {
    typedef turbodiesel ...;
    void start (turbodiesel);
    //...other definitions

}
```

A client wanting to use the turbodiesel definition has several options. One is to directly name the definition. The :: operator is used to qualify the namespace in which to look for the desired object.

```
XYZCorp::turbodiesel t;
```

Another option is to first create a synonym for the name so that the namespace name does not need to be repeated:

```
using XYZCorp::turbodiesel;  //creates a synonym turbodiesel
//...
turbodiesel t;
XYZCorp::start (t);
```

The final option is for a client that wants to import all the definitions from a namespace. The namespace may be opened by importing it:

```
using namespace XYZCorp; //this “opens” the namespace completely
turbodiesel t;
start (t);
```

The namespace mechanism is intended to help library providers become independent of other library providers, enable them to update their libraries without danger of interfering with client code, and even provide new releases of libraries that co-exist with older releases (each release lives in a different namespace).

The :: operator is used generally to deal with scope resolution. For example, ::x refers to x in the global environment. X::x refers to x in the scope X which may be a namespace or a class whose name, X, known in the current referencing environment.

5.3.4 Ada

Ada was designed specifically to support programming in the large. It has elaborate facilities for the support of modules, encapsulation, and interfaces. Rather than relying on convention as in C and C++, Ada makes an explicit distinction between specification and implementation of a module. A file may be compiled if the specifications of the modules it uses are available. Thus, Ada naturally supports a software development process in which module specifications are developed first and implementation of those modules may proceed independently. Ada also requires the existence of a compile-time library in which module specifications are compiled. A module may be compiled if all the module specifications it needs are already in the library. This library supports the checking of inter-module references at compile time (Section 3.4.3 on page 177).

5.3.4.1 Encapsulation in Ada

The package is Ada’s unit of modularity. An Ada module encapsulates a group of entities and thus supports module-based programming. We have already seen that the language’s explicit distinction between module specification and module body forces the programmer to separate what is expected by the module from what is hidden within the module. Additionally, Ada supports concurrent modules or tasks.

In addition to the conceptual modularity at the package level, Ada supports the separate compilation of procedures and functions as well as packages. We will see an example of this in the next section.

All units in Ada may also be generic. We will discuss generic units in Section .

5.3.4.2 Program organization

An Ada program is a linear collection of modules that can be either subprograms or packages. These modules are called units. One particular unit that implements a subprogram is the main program in the usual sense. Module declarations may be nested. Consequently, a unit can be organized as a tree structure of modules. Any abuse of nesting within a unit causes the same problems discussed for Pascal. These problems can be mitigated by the use of the subunit facility offered by the language. This facility permits the body of a module embedded in the declarative part of a unit (or subunit) to be written separately from the enclosing unit (or subunit). Instead of the entire module, only a *stub* need appear in the declarative part of the enclosing unit. The following example illustrates the concept of the subunit.

```
procedure X ( ... ) is                              --unit specification
    W: INTEGER;
    package Y is                                    --inner unit specification
        A: INTEGER;
        function B (C: INTEGER) return INTEGER;
    end Y;
    package body Y is separate;                     --this is a stub
begin       -- uses of  package Y and variable W
    ...
    ...
    ...
end X;
-----------------------------------next file--------------

separate (X)
package body Y is
    procedure Z (...) is separate;                --this is a stub
    function B (C: INTEGER) return INTEGER is
    begin       --use procedure Z
        ...
        ...
        ...
    end B;
end Y;
-----------------------------------next file--------------

separate (X.Y)
```

```
procedure Z (...) is
begin
    ...
end Z;
```

The prefix **separate** (X) specifies package body Y as a subunit of unit X. Similarly, **separate** (X.Y) specifies procedure Z as a subunit of package Y nested within X. The subunit facility not only can improve the readability of programs, but supports a useful technique in top-down programming. When writing a program at a certain level of abstraction, we may want to leave some details to be decided at a lower level. Suppose you realize that a certain procedure is required to accomplish a given task. Although calls to that procedure can be immediately useful when you want to test the execution flow, the body of the procedure can be written at a later time. For now, all you need is a *stub*. The subunit facility, however, does not overcome all the problems caused by the tree nesting structure. The textually separate subunit body is still considered to be logically located at the point at which the corresponding stub appears in the enclosing (sub)unit. It is exactly this point that determines the entities visible to the subunit. In the example, both subunits Y and Z can access variable W declared in unit X.

The interface of an Ada unit consists of the **with** statement, which lists the names of units from which entities are imported, and the *unit specification* (enclosed within a **is... end** pair), which lists the entities exported by the unit. Each logical module discovered at the design stage can be implemented as a unit. If the top-down design was done carefully, logical modules should be relatively simple. Consequently, the nesting within units should be shallow or even nonexistent. Ada does not forbid an abuse of nesting within units. Actually, the entire program could be designed as a single unit with a deeply nested tree structure. It is up to the designer and programmer to achieve a more desirable program structure.

The last program structuring issue is how the interfaces (i.e., import/export relationships) among units are specified in Ada. A unit exports all the entities specified in its specification part. It can import entities from other units if and only if the names of such units are listed in a suitable statement (**with** statement) that prefixes the unit. For example, the following unit lists unit X (a subprogram) in its **with** statement. Consequently, it is legal to use X within T’s body.

```
with X;
package T is
    C: INTEGER;
    procedure D (...);
end T;
package body T is
    ...
    ...
    ...
end T;
```

Similarly, the following procedure U can legally call procedure T.D and access variable T.C. On the other hand, unit X is not visible by U.

```
with T;
procedure U (...) is
    ...
end U;
```

5.3.4.3 Interface and implementation

We have already seen in Section that Ada strictly separates the specification and body of a package. In the previous section, we have seen how the **use** and **with** clauses are used to import services from packages. These facilities are used also to support separate compilation. Recall that separate compilation, as opposed to independent compilation, places a partial ordering on compilation units.

The set of units and subunits comprising a program can be compiled in one or more separate compilations. Each compilation translates one or more units and/or subunits. The order of compilation must satisfy the following constraints.

- A unit can be compiled only if all units mentioned in its with statement have been compiled previously.
- A subunit can be compiled only if the enclosing unit has been compiled previously.

In addition, unit specifications can be compiled separately from their bodies. A unit body must be compiled after its specification. The specification of a unit U mentioned in the **with** statement of a unit W must be compiled before W. On the other hand, U’s body may be compiled either before or after W. These constraints ensure that a unit is submitted for compilation only after the compilation of unit specifications from which it can import entities. The compiler saves in a library file the descriptors of all entities exported by units.

When a unit is submitted for compilation, the compiler uses the library file to perform the same amount of type checking on the unit whether the program is compiled in parts or as a whole.

Ada’s choice of a package as an encapsulation mechanism, together with its reliance on separate compilation, and the separation of specification and body creates an interesting issue when a package wants to export a type. This issue leads to the private type feature of Ada.

The private type. In Figure 65, we declared a dictionary module that exports procedures and functions only. When the client declares its intention to use the dictionary package, the dictionary object is allocated. The representation of the object is not known to the client. From the package body, we can see that the entries in the dictionary are actually records that contain three different fields. What if we want to export to the client a type such as dictionary_entry? This would enable the client to declare variables of type dictionary_entry. We would like to export the type but not its representation. From the language design point of view there is a conflict here. The Ada language specifies that a client may be compiled with the knowledge only of the specification of the provider module. But if the provider module is exporting a type and not its representation, the size of the type cannot be determined from the specification. Thus, when the compiler is compiling the client, it cannot determine how much memory to allocate for variables of the exported type. Ada’s solution to this problem is the **private type**. The specification must contain the representation of the type but as a private type.

If a package unit exports an encapsulated private data type, the type’s representation is hidden to the programmer but known to the compiler, thanks to the private clause appearing in the package specification. Consequently, the compiler can generate code to allocate variables for such types declared in other units submitted for compilation prior to the package body (but after its specification). When a unit is modified, it may be necessary to recompile several units. The change may potentially affect its subunits as well as all the units that name it in their **with** statements. In principle, all potentially affected units must be recompiled.

The separate compilation facility of Ada supports an incremental rather than a parallel development of programs, because units must be developed according to a partial ordering. This is not an arbitrary restriction, but a conscious design decision in support of methodical program development. A unit can be submitted for compilation only after the interfaces of all used units are frozen. Consequently, the programmer is forced to postpone the design of a unit body until these interfaces have been designed. One of the goals of separate compilation is to support production of reusable software. Certified modules can be kept in a library and later combined to form different programs. The Ada solution is deficient on this point for package units exporting encapsulated (private) data types. The visible part (the specification) of such packages must include the type's operations and a private clause that specifies the type's internal representation. This representation is not usable outside the package body; it is there only for supporting separate compilation. Logically, this information belongs in the package body, together with the procedure bodies implementing the type's operations. Besides being aesthetically unpleasant, this feature has some unfortunate consequences:

- It violates the principle of top-down design. The representation must be determined at the same time as the specification of the data type, and both appear in the same textual unit.
- It limits the power of the language to support libraries of reusable modules, unless special care is taken in the implementation. For example, a module using FIFO queues is compiled and validated with respect to a FIFO queue package providing a specific representation for FIFO queues (e.g., arrays). The module must be recompiled if one wants to reuse it in a different program in which FIFO queues are implemented by a different data structure, even though the interfaces for manipulating FIFO queues are the same in both cases.

5.3.4.4 Grouping of units

Ada has many features for supporting programming in the large. Two clauses, **use** and **with,** are used to import services from other packages. Child library units are used to group packages together in hierarchical organizations. These facilities are defined to enable safe separate compilation.

The with and use clauses. The **with** clause is used by a client to import from a provider module. For example, if we want to write a module to manipulate telephone numbers and we want to use the dictionary module specified in Figure 65, we prefix the telephone module with a **with** clause:

```
with dictionary;
package phone_list is
    ...
    --references to dictionary.insert(), etc.
    ...
    end phone_list;
```

Now, inside the phone_list package, we may refer to the exported entities of the dictionary package. These references have to be prefixed by the name of the package from which they are imported. For example, dictionary.insert(...). To gain direct visibility, and avoid the need to use the dotted name, Ada provides the **use** clause:

```
    with dictionary; use dictionary;
    package phone_list is
        ...
        --references to insert(), etc.
        ...
    end phone_list;
```

Child libraries. The Ada package groups together a set of related entities. Clients may import either selective services from a package or all the services provided by the package by using the **use** clause. The package is inadequate as a structural mechanism for grouping a collection of library modules. Here are some examples of problems that could occur:

- Suppose a client uses two different libraries, encapsulated in packages A and B. Since the client expects to make extensive use of both libraries, it uses the **use** clause to import all the library services. But if libraries A and B export entities with the same name, the client would encounter name clashes at compile time. Ada provides a renaming facility to get around this problem.
- More serious is the case where there are no name clashes. The client compiles and works properly. But suppose that a new version of library B is released with new functionality. It happens that one of the new functions introduced in B has a name identical to a name provide by A. The next time that the client code is compiled, compilation errors will show up due to name clashes. These errors would be particularly confusing because the previously working client code appears to not work even though it may not have been changed.
- In the previous case, after the release of the new version of the library B, the client code has to be recompiled even though it does not make use of the new functionality of the library B. The recompilation is necessary only to satisfy Ada’s rules on the order of compilation.

Ada 95 has addressed these problems by introducing the notion of child libraries which allow packages to be hierarchically organized. The idea is that if new functionality is added to an existing library package, the new functionality may itself be organized as new package that is a child of the original library. The child package can be implemented using the facilities of the parent package. But the clients of the original library are not affected by the introduction of a child package. The child package makes it possible to add functionality to a package without disturbing the existing clients of the package.

In general, a library developer may provide a number of packages organized as a tree. Each package other than the root package has a parent package. An existing library may be extended by adding a child library unit to one of its existing nodes. The parent library unit, nor any clients of the parent need to be recompiled. For example, if the library Root exists, we may add Root.Child without disturbing Root or clients of Root. The Root.Child may be compiled separately. It has visibility to Root and to Root's siblings.

```
package Root is
    --specification of Root library
    --...
end Root;
-----------
package Root.Child is
    --specification of a child library unit
    --...
end Root.Child;
-----------
package body Root.Child is
    --implementation of Root.Child
    --...
end Root.Child;
-----------
```

Each of the above segments may be compiled separately. The clients of Root need not be recompiled if they do not use Root.Child.

5.3.5 ML

Modularity is not only the province of imperative languages. The notion of module is important in any language that is to be used for programming in the large. For example, ML is a functional programming language with extensive support for modularity and abstraction. In Chapter 7, we will study the basics of functional programming and ML. In Chapter 7, we will see ML's support for defining new types and abstract data types, which also help in programming in the large. In this section we give a brief overview of ML's support for modules.

5.3.5.1 Encapsulation in ML

A module is a separately compilable unit. A unit may contain structures, signatures, and functors. Structures are the main building blocks; signatures are used to define interfaces for structures; functors are used to build a new structure out of an existing structure. We will discuss these more in Chapter 7. Here, we only examine the structure construct as a packaging unit.

The ML *structure* is somewhat like the Ada package, used to group together a set of entities. For example, our dictionary example package of Figure 66 may be written in ML as given in Figure 75. Recall the syntax and case analysis style of programming from Chapter 5. We will describe the details of the functions in Chapter 7. Here, we are only interested in what is exported by the structure, that is, module.

```
structure Dictionary =
struct
    exception NotFound;

    val root = nil;  (*create an empty dictionary*)

 (* insert (c, i, D) inserts pair <c,i> in dictionary D*)
    fun insert (c:string, i:int, nil) = [(c,i)]
    |    insert (c, i, (cc, ii)::cs) =
         if c=cc then (c,i)::cs
         else (cc, ii)::insert(c,i,cs);

 (* lookup (c, D) finds the value i such that pair <c,i> is in dictionary D *)
    fun lookup(c:string, nil) = raise NotFound
    |    lookup (c, (cc,ii:int)::cs) =
         if c = cc then ii
         else lookup(c,cs);
end;
```

FIGURE 75. Dictionary module in ML (types string and int are not necessary but used for explanation here)

Such a structure definition corresponds to the package body in that it gives the implementation for the entities being defined. It also has the property that all the entities are exported. This structure exports an exception, NotFound, a variable root, and two functions insert and lookup. To use the structure, a client uses the dot notation:

```
val D = Dictionary.create;  (*create an empty dictionary *)
val newD = Dictionary.insert (“Mehdi”, 46, D);  (*insert a pair*)
...
D.lookup(“Mehdi”, D); (*produces value 46*)
```

5.3.5.2 Interface and implementation

The signature of a structure definition consists of the signatures and types of all the entities defined in the structure. ML also provides a construct to define a signature independently of any structure. A signature may be viewed as a specification for a module. For example, Figure 76 gives the signature of a module that exports an exception called NotFound and a function called lookup. A signature may be used as a specification for a structure. For example, we may use the signature of Figure 76 to restrict the exported entities of the structure of Figure 75. The system will do type checking to ensure that the structure provides at least what the signature requires.

```
signature DictLookupSig = sig
    exception NotFound;

    val lookup : string * (string * int) list -> int
end
```

FIGURE 76. A signature defintion for specialized dictionary

We can use the structure and signature we have to create a new module with a restricted interface and use it accordingly:

```
structure LookupDict: DictLookupSig = Dictionary;
val L = LookupDict.create; (* not allowed, must be done by someone else using a different
interface *)
lookupDict.lookup("Mehdi", L);
lookupDict.insert("Carlo", 50, L); --error, insert not available
```

We can see that the ability to define signatures means that we can provide different interfaces to the same implementation, something not possible in Ada or C++. We can also provide different implementations to meet the same interface.

ML also supports the concept of *generic* modules or structures. The signature facility may be combined with generic structures to instantiate a structure for particular types. For example, the dictionaries that we have defined so far, both in Ada and in ML have been specific to <string, integer> pairs. In ML, we can remove the occurrences of the terms string and int from Figure 75 and have a generic dictionary. We will see this in Section 5.4.3.

5.3.6 Abstract data types, classes, and modules

We have discussed an abstract data type as a program modularization concept. Languages that provide a class construct, such as C++, support the implementation of abstract data types directly. For example, Figure 73 shows a class that implements an abstract data type stack and a client that declares instances of the stack and uses them. The name of the class is used as the type name to instantiate the objects necessary. Operations are performed directly on the instantiated objects, e.g. s1.push(...).

Module-based languages such as Ada or Modula-2, however, do not support objects directly and are operation-oriented. We use a module to implements an abstract data type by packaging the type and its operations together. But the client creates instances of the abstract data type and passes them to operations as necessary, rather than calling the operations associated with the object. That is, rather than calling s1.push(...) the client calls push(s1, ...). In object-based languages, the object is an implicit first parameter automatically passed to the operation.

In a module-based language, the need for a construct such as friend functions does not appear: we simply put all the related functions and types in the same module and they gain visibility to each other. In an object-based language, the requirement to package a single type and its operations together makes it difficult to deal with operations that do not belong clearly to a single type. We generally have to make such operations global. Java resolves this dichotomy by supporting both modules and classes. Classes defined together in the same module have visibility to one another's internal structure.

A final comparison of Ada and C++ styles concerns the export of types. In both languages, a client may instantiate a variable of a type defined by another module (or class). Given a declaration of the form s: T in a client, an important question for the compiler is how much storage to allocate for the instance of s. Even though logically this information is part of the implementation of the module that implements the type, not part of its specification, the compiler needs the information at the time it compiles the client. This is the reason, as we have seen, that Ada requires the private clause in the specification part of a package. The information in the private part is there only for the compiler. C++ also requires the same information in the private part of a class definition.

Requiring the data representation in the specification of a module means that if the representation changes, the clients will have to be recompiled. This is a serious cost in large system development. To address this problem, Modula-2 introduced the notion of *opaque* export which allows types to be exported without the details of their representation. Variables of such types are constrained to be accessible via pointers; therefore there is no need to have the equivalent of Ada's private clause in the interface. In fact, the amount of storage to be allocated in client modules for such data objects is known to be the size of a pointer. The restriction that abstract data types be accessible via pointers means that every access incurs the cost of a pointer dereference but ensures intermodule decoupling. Changing the data structure for an abstract data type does not affect client modules either from a logical or from an implementation viewpoint. The client modules do not need to be recompiled. In CLU and Eiffel, all objects are accessed through pointers and therefore there is no need to have the representation of a type in its specification.

5.4 Generic units

In this chapter, we have considered the issue of modularity as a support for developing large programs. One important approach to developing large programs is to build them from existing modules. Traditional software libraries offer examples of how such existing modules may be packaged and used. One of the criteria for evaluating the suitability of a language for programming in the large is whether it provides language mechanisms that enable the construction of independent components, the packaging together of related components, the use and combination of multiple libraries by clients, etc. We have seen the namespaces of C++ and the libraries and child libraries of Ada 95 as language mechanisms explicitly developed for the support of such packaging of related and independent software components. In this section, we concentrate on genericity as a mechanism for building individual modules that are general and thus usable in many contexts by many clients.

5.4.1 Generic data structures

Let us first consider the development of libraries of standard data structures, for example, stacks and queues. What should be the types of elements stored in these structures? Early typed languages such as Pascal and C require the designer to define one structure for each data type to be supported. This is an unsatisfactory solution for two reasons: one is that the solution only works for only the types the library designer knows about and not for any types to be defined by the user of the library; the second is that the solution forces the library designer towards code duplication. C++ templates and Ada generics allow us to avoid such code duplication and define a template data structure that is independent of the type of the element to be stored in the structure. For example, we can define a generic *pair* data structure in C++:

```
template <class T1, class T2>
class pair {
    public:
        T1 first;
        T2 second;
        pair (T1 x, T2 y) : first(x), second(y) { }
};
```

The template parameters T1 and T2 stand for any type. We may "instantiate" a particular pair by supplying concrete types for T1 and T2. For example, we may create a pair of integers or a string, integer pair or a pair of employees:

```
pair<int, int> intint(2, 1456);
pair<string, int> stringint("Mehdi", 46);
pair<employee_t, employee_t> (jack, jill);   /*pair of user-defined type employee_t*/
```

We may refer to pair as a parameterized or generic type. The template of C++ allows us to define such a parameterized type which may later be used to create concrete types such as pair<int, int>. C++'s template facility is particularly general because it uses classes as parameters and classes represent types uniformly: we may instantiate a template with either user-defined or primitive types. Eiffel supports a similar scheme for generic classes, with which, for example, we can define a class stack [T] and then instantiate an intstack from stack[integer]. In Chapter 3 we saw examples of generic stacks both in C++ and Eiffel.

5.4.2 Generic algorithms

Templates may also be used to define generic algorithms. For example, in Chapter 2, we saw the following generic function swap which interchanges the values of its two parameters:

```
template <class T>
void swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

This function may be used for any two parameters of the same type that support the “=” operation. Therefore, we can use it to swap integers, reals, and even user-defined types such as pairs. This is quite a useful facility because it gives us the possibility to write higher-level generic functions such as sort if they only use generic functions. The ability to write such generic functions is helped in C++ by the fact that generic functions do not have to be instantiated to be used.

To use a template data structure, C++ requires explicit instantiation of the structure, as we saw, for example, in pair<int,int>. For functions, on the other hand, explicit instantiation is not necessary. The compiler will infer the instance required and generate it automatically. For example, the following program fragment is valid:

```
int i, j;
char x, y;
pair<int,string> p1, p2;
...
swap(i, j); //swap integers
swap(x, y); //swap strings
swap(p1, p2); //swap pairs
```

the compiler will generate three different swap functions, for integers, strings and pairs of (int, string). To generate an appropriate function, the compiler checks at generation time that the parameters meet the expected requirements. Examination of the body of swap shows that the parameters passed must support assignment, that is, to be able to be passed and to be able to be assigned. (Exercise 22 asks you to explain why pairs meet this requirement.)

The implicit parameter requirements in C++ are made explicit in Ada generic functions. The same swap function is defined in Ada as:

```
generic
    type T is private;
procedure swap (x, y: T) is
begin
    temp: T = x;
    x = y;
    y = temp;
end swap;
```

The generic is explicitly stated to be based on a type T which is **private**. The private indication means that the type supports assignment and equality. In general, if other operations are required of the type, they have to be stated. For example, a generic max function will require its operands to support an order operations such as “>”:

```
generic
    type T is private;
    with function “<“ (x, y: T) return BOOLEAN is <>;
function max (x, y: T) return BOOLEAN is
begin
    if x<y
        then return x;
        else return y;
    end if;
end max;
```

To use the function, we have to first instantiate an instance of it:

```
function int_max is new max (INTEGER);
```

The type parameters passed at instantiation time are checked to ensure that they support the required operations. After instantiation, we have a new function that we may call:

```
m := int_max (3, 6);
```

The Ada view is that different functions are generated and used while the C++ view is that there is just one function max which is generic. It is the compiler’s job to generate as many instances as it needs to satisfy all the calls to the function. The C++ approach is more flexible and is more supportive of generic programming because generic functions are not treated any differently from nongeneric functions: you simply call them. Ada treats generic functions as a special type of function that you must instantiate before you can call.

We will examine ML’s generic functions (called polymorphic) in Chapter 7.

In summary, generic routines allow us to parameterize algorithms and achieve a higher level of generality by capturing an algorithm in a type-independent way.

5.4.3 Generic modules

Collections of data and algorithms may also be packaged together and collectively made to depend on some generic type parameter. Both C++ classes and Ada packages may be defined as generic in the types they use. We saw a generic stack class in Chapter 3.

The ML support for generic modules is particularly interesting because of the separation of structures and signatures. Recall the ML dictionary module in Section 5.3.5. The signature definition of Figure 75 can be defined in a generic way by not making any mention of specific types such as int and string. We have defined such a generic structure in Figure 77. The signature of this module is independent of specific types. Can we apply the signature of Figure 76 to this structure? That signature definition indeed matches this structure because the structure is more general than the signature requires. By applying the signature, we are restricting the view of the structure. Applying a signature to a polymorphic structure is similar to package instantiation in Ada.

```
structure Dictionary =
struct
    exception NotFound;

    val root = nil;  (*create an empty dictionary*)

   (* insert (c, i, D) inserts pair <c,i> in dictionary D*)
    fun insert (c, i, nil) = [(c,i)]
    |    insert (c, i, (cc, ii)::cs) =
         if c=cc then (c,i)::cs
         else (cc, ii)::insert(c,i,cs);

   (* lookup (c, D) finds the value i such that pair <c,i> is in dictionary D *)
    fun lookup(c, nil) = raise NotFound
    |    lookup (c, (cc,ii:int)::cs) =
         if c = cc then ii
         else lookup(c,cs);
end
```

FIGURE 77. A polymorphic dictionary module in ML

5.4.4 Higher levels of genericity

We have seen that we may define a generic algorithm that works on any type of object passed to it. For example, the max algorithm may be applied to any ordered type. This facility allows us to write one algorithm for n different data types rather than n different algorithms. It leads to great savings for writers of libraries. But consider a higher level of generality. Suppose we want to write an algorithm that works on different types of data *structures*, not just different data types. For example, we may want to write one algorithm to do a linear search in any “linear” data structure. Of course, we have to capture the notion of linearity somehow but intuitively, we want to be able to find an element in a collection regardless of whether the collection is implemented as an array, a list. The goal of the generic programming paradigm is to develop exactly these kinds of units. In Chapter 3, we saw one kind of iterator for stepping through a collection. Here we will examine a different kind of iterator.

A high level of genericity is usually associated with functional languages and we will see it in the context of ML in Chapter 7. There are no particular language facilities in Ada or C++ for this kind of programming. However, the flexibility of C++ templates, combined with overloading of operators supports a high degree of generic programming. For example, consider the following function find:

```
template<class Iter, class T>
Iter find ( Iter f, Iter l, T x)
{
    while (*f  !=  last  &&  *f  !=  x)
        ++f;
    return f;
}
```

We might think of this function as accepting two pointers into a sequence of elements. It sequences through the elements by using the ++ on the first pointer until either the value x is found or the sequence is exhausted. So, the following code fragment looks through the first half of an integer array:

```
int a[100];
int x;
int *r;
...
r= find(x, &a[0], &a[50]);
if (r == &a[5])
    // not found
...
```

Here, we have used an integer pointer as the template parameter. However, the function is quite abstract: nothing in its description constrains us to use it with pointers and arrays! It is based on an abstract object which we have called Iter (for *iterator*). We can think of an iterator as a generalization of a C++ pointer. It must support the operations: *, to return a value, ++ to step to the next position, == and != for comparison with another Iter. Certainly pointers meet these requirements. But we might imagine writing a list object that also provides an Iter type object which supports ++, *, ==, and != operations with the same semantics as those of pointers into arrays. More importantly, any time a library writer provides a new linear structure, he can also provide it with such iterators. In this way, any generic operations will be immediately usable with the library's new data structures. What we are doing is to treat operations such as !=, *, and ++ as generic operations and writing a higher level operation find in terms of them. This style of generic programming is possible in C++ and likely will be the way standard libraries are provided. The advantages of such an approach for programming in the large is the reduction of the amount of code that needs to be written because one generic unit may be customized automatically depending on the context of its use. It is a form of modularity in which we modularize based on common properties and specific properties. Object-oriented programming is another approach to achieving this same kind of modularity. That will be the subject of the next chapter.

5.5 Summary

Appropriate abstractions and proper modularization help us confront the inherent complexities of large programs. Even with appropriate modularization, however, writing all the modules from scratch is a tedious and time-consuming task. Rather than inventing new abstractions and implementing new modules for each new program, we can improve software productivity by using previously developed abstractions and modules. In this chapter, we have studied linguistic mechanisms that help in the building of modules and libraries of modules that may be used by others. But where do such useful modules come from and how can we build them? Different programming paradigms provide different answers to these questions. In the next chapter, we see how the object-oriented paradigm answers these questions.

5.6 Bibliographic notes

The problems of programming in the large and techniques for addressing them are covered in software engineering textbooks such as (Ghezzi 91). An informal but insightful and entertaining account of problems arising in the production of large software systems may be found in (Brooks 1995). The distinction between programming in the small and programming in the large was pointed out in (DeRemer and Kron 1976). The methodology of stepwise refinement is described in (Dijkstra 1972). Information hiding was introduced in (Parnas 1972a), (Parnas 1972b). Lauer and Satterthwaite (1979) describe the Mesa system and, in particular, how it supports the design of large systems. Separate compilation facilities for Pascal are described in (Jensen and Wirth 1975), (Kieburtz et al. 1978), (LeBlanc and Fisher 1979), and (Celentano et al. 1980); for SIMULA 67 in (Birtwistle et al. 1976) and (Schwartz 1978c). (Jazayeri 95) discusses the construction of use of generic components in C++. CLOS was one of the first language implementations to combine generic functions and object-oriented programming. Barnes95 is an excellent reference for Ada 95.

5.7 Exercises

1. Discuss the effect of global variables on the writability and readability of large programs.
2. Study and present the main features of FORTRAN's separate compilation.
3. Why is the ALGOL-like program structure inadequate for programming in the large?
4. Complete the implementation of the package body in Figure 66 on page 281.
5. Design the interface of an Ada module that provides a symbol table for a translator (e.g., an assembler), and show how a separately compiled procedure can access the symbol table. The data structure representing the symbol table should be hidden from the procedure, and all accesses to the symbol table should be routed through abstract operations provided by the symbol table module. Can you compile the procedure before implementing a representation for the symbol table? Why? What is wrong if you cannot?
6. Do the same as Exercise 4 but in C++. Do you run into the same problems?
7. Suppose two Ada units U1 and U2 must use the same procedure P. Can P be embedded in a single subunit? Can P be embedded in a single unit? In the latter case, what are the constraints on the order of compilation?
8. In this exercise, we compare nested packages and child libraries. In particular, answer the following questions:
   Can nested packages be compiled separately? Can child libraries be compiled separately?
   Can a package have both a child package and a nested package with the same name? Why not?
   What can you conclude about the utility of nested packages?
9. Describe the tools that an ideal program-development system should provide to support independent development of modules, system structuring from independently developed modules, and complete intermodule type checking.
10. Storage classes of C: automatic, extern, static (???)
11. Achieving visibility in C units: If a variables is declared in a function, which units have access to it? If a variable is declared outside of functions, which units have access to it? If a variable is declared as extern, where is it defined? If a variable is defined as static, which units have access to it?
12. We have seen in Chapter 2 that the scope rules of the language provide for the control of names and their visibility. Discuss the relationship between name scopes in block structured languages and the child libraries of Ada and the namespaces of C++.
13. In this chapter, we have discussed the need for a unit to give access to its private parts only to some units but not export it to all other units. Ada and C++ have two different ways of

satisfying this requirement. What is the solution provided by each language? Compare the two facilities.

14. (Perhaps for ch 3) C++ has a default assignment operation defined for classes. If the user does not define the assignment operation for a new class, the language uses a memberwise copy by default. Is this a good decision? Is memberwise copy desirable in most situations (hint: consider a stack copy)?
15. Solve the problem of Figure 73 by instantiating the generic class defined in Figure 34 on page 155.
16.

    i) Write a C++ and an Eiffel program for a generic abstract data type defining a first-in first-out queue which can contain an unbounded number of items. The operations provided by queues have the following signatures:

    enqueue: fifo (T) x T -> fifo (T) --adds an element to the queue
    dequeue: fifo (T) -> fifo (T) x T --extracts an element from the queue
    length: fifo (T) -> int --computes the length of the queue

    ii) Show how instances can be created.

    iii) Next, provide fixed-length queues, such that an exception is raised if one tries to enqueue an element in a full queue.

    iv) Show examples in which you generate instance objects that are unbounded queues and fixed-length queues, and illustrate the kinds of polymorphism that can arise with these two types.

    Besides providing the program which solves this problem, write also a short description of the rationale of your implementation.

17. Assume that I buy a software library from a vendor. The library contains the specification of an abstract object stack. I write a program in which I create an instance of type stack.

    i) Assume the stack object is written in Ada and the vendor decides to change the implementation of its push operation. Do I need to recompile my program? Assume the vendor decides to change the representation of the stack. Do I need to recompile my program? Explain your answers.

    ii) Explain the same two problems if the language used is C++.

    iii) Explain the same two problems if the language used is Eiffel.

18. Consider a generic function swap (x, y) which interchanges the values of its two arguments. Write a bubble sort in C++ that uses swap to interchange the elements. How would your solution be different if you try the same approach in Ada?
19. Without generics, if we have m data types, n algorithms, and p data types, we need to write on the order of m*n*p library components to support all possible algorithms on all possible data structures for all possible data types. Explain how the use of generics reduces the number of library components that need to be written. Assuming that we could write all the components without generics, what other deficiency remains?
20. Ada defines two kinds of types: private type and limited private type. What is the difference between these two? Is there a similar concept in C++? If not, why not? Does their absence imply a lack of functionality in C++?
21. What is the difference between overloaded functions and generic functions in C++?
22. We did not define an assignment operator for the template type pair in Section 5.4.1. Yet, in Section 5.4.2 we used swap with pairs. Swap requires the assignment operator for its operands. Is the assignment operator defined for pairs (Hint: check the C++ rule for class definitions).

23. Compare the implementation of Ada generics versus C++ templates. Does the source of a C++ template function need to be available to be able to compile the client code? Is this necessary for an Ada generic function? If there are two different calls to swap(x,y), will a C++ compiler generate two instances of swap? What about Ada?

24. Suppose we want to write a generic sort routine to sort elements of type T. We will want to use our swap routine from section 5.4.2 on page 308. A fragment of the C++ might look like this:

```
template<class T>
sort (...)
{
...swap (x, y);
...
}
```

If we were to write sort in Ada, we would have to instantiate swap first. What type should we use to instantiate swap? Explain the problem. Check the Ada definition to find a solution to this problem.

25. Consider the following generic signature in ML:

```
signature DictLookupSig = sig
    exception NotFound;

    val lookup : ‘t * (‘‘t *’ ‘’t) list -> int
end
```

Does the signature match the structure of Figure 75? Does it match the structure of Figure 76?

26. In Section 5.4.4, we saw a generic function called find. We said that this function may be applied to a list data structure if the list provides an appropriate iterator. Write a class list which provides such an iterator. That is, class list provides a type called iterator. (**give more details**)

Object-oriented languages

CHAPTER 6

In the last chapter, we discussed the problems of programming in the large and the solutions offered by different programming languages. Modularity and abstraction are the two most important principles for building large programs. In the “pure” object-oriented style of programming, the unit of modularity is an abstract data type. We have seen that classes may be used to define abstract data types. Another important principle of object-oriented programming is that you may define new classes of objects by extending or refining existing classes.

Some programming languages have been designed expressly to support this style of programming. These languages, namely Smalltalk and Eiffel, are called object-oriented programming languages. Other languages, such as C++ and Ada 95, while not exclusively object-oriented, support the paradigm through features that enable the programming of extensible abstractions. All object-oriented languages trace their roots to the language Simula 67 which introduced the concept of class and subclass in 1967. In this chapter we examine the essential programming language features for the support of object-oriented programming and look at some representative object-oriented programming languages.

The starting point for object-oriented programming is abstract data types which we have already examined in Chapters 3 and 5. We have seen, for example, that the class construct of C++ directly supports the definition of abstract data types. We may design classes Chair and Table if our application deals with such entities and then create as many instances of the specific Chairs and Tables that we need. Next comes the notion of *inheritance*. For example, rather than designing a class DiningTable and another class Desk, we might first design a class Table which captures the properties of different kinds of tables and then “derive” DiningTable and Desk as new classes that “inherit” properties of Table and add their own unique attributes. This is a linguistic issue but also requires a supporting design style. For our example, instead of considering the problem domain to consists of chairs and tables and desks, we might decide that the problem domain deals with *furniture*; some particular kinds of furniture are tables and chairs; particular kinds of tables are desks and dining tables; particular kinds of chairs are lounge chairs and sofas. Some concepts such as furniture are abstract and only exist as descriptions. Any particular piece of furniture is actually an instance of a more concrete class such as the chair class. By factoring the common properties of individual concrete objects at the abstract level, we only need to describe them once rather than many times. The individual kinds of objects such as chairs only need to be described by describing their specific features that make them unique as pieces of furniture. We say that a chair *inherits* the properties of furniture and may extend or modify these properties as necessary.

Object-oriented programming is an attractive methodology because it promises the ability to package a whole set of related concepts tied together through their inheritance relationships. It aims to enable the production of libraries of related components that are

- easy to understand by users because of the relationships among the components
- easy to extend by the use of inheritance

Many languages have been or are being extended to support object-oriented programming. The goal of this chapter is to examine the concepts underlying object-oriented programming and the implementation of these concepts in several programming languages. In Section 6.1, we introduce the basic concepts of object-oriented programming. In Section 6.2 we examine the relationship between inheritance and the type system of the language. In Section 6.3 we review the support of object orientation in C++, Eiffel, and Ada 95, and Smalltalk. Object-orientation has affected not only the implementation phase of the software process but most other phases as well. In Section 6.4 we briefly review this impact on design and analysis phases of the software process.

6.1 Concepts of object-oriented programming

There are several definitions of what object-oriented programming is. Many people refer to an object-oriented program as any program that deals with entities that may be informally called “objects.” In contrast to traditional procedural languages in which the programs consist of procedures and data structures, objects are entities that encapsulate data and related operations. For example, given a stack data structure s, in a procedural language we would call a push operation to add an element as in:

```
push (s, x);
```

Dealing with objects, as we have seen in C++, we tell the stack object to push an element onto itself, as in:

```
s.push(x);
```

We have seen that in C++ and Eiffel we can use classes to define and create objects. We call a programming language that supports the definition and use of objects *object-based*. *Object-oriented* programming languages support additional features. In particular, object-oriented programming languages are characterized by their support of four facilities:

- abstract data type definitions,
- inheritance,
- inclusion polymorphism, and
- dynamic binding of function calls.

We have already discussed abstract data types extensively. They are used in object-oriented programming to define the properties of classes of objects. Inheritance is a mechanism that allows us to define one abstract data type by *deriving* it from an existing abstract data type. The newly defined type “inherits” the properties of the parent type. Inclusion polymorphism allows a variable to refer to an object of a class or an object of any of its derived classes. Dynamic binding supports the use of polymorphic functions; the identity of a function applied to a polymorphic variable is resolved dynamically based on the type of the object referred to by the variable.

The pure terminology of object-oriented languages refers to *objects* that are *instances* of *classes*. An object contains a number of *instance variables* and supports a number of *methods*. A *message* is sent to an object to request the invocation of one of its methods. For example, s.push(5) is interpreted as sending the message push(5) to object s. The message is a request to s to invoke its method push with the parameter 5. The syntax of Smalltalk reflects this interpretation directly. In Smalltalk, the same statement would be written as: s push 5. In a dynamically-typed language such as Smalltalk, the object that receives a message first checks to see that it can perform the method requested. If not, it reports an error. Other languages, such as Eiffel and C++, allow polymorphism but restrict it to enable static type checking. Thus, such languages combine polymorphism with strong typing.

Having previewed the concepts of object-oriented programming in general, in Section 6.1.1 through 6.1.4 we provide a more detailed look at these concepts, using C++ as the example language. Following C++ terminology, we will call methods *member functions* and messages simply *function calls*. In Section 6.3, we will review more specific concepts of C++ as well as other languages.

6.1.1 Classes of objects

The first requirement for object-oriented programming is to be able to define abstract data types. We have already seen that this can be done using the class construct. As an example, recall the following definition of a stack class from Section 5.3.3.2:

```
class stack{
    public:
        void push(int) {elements[top++] = i;};
        int pop() {return elements[--top];};
    private:
        int elements[100];
        int top=0;
};
```

This class is a particular implementation of a fixed-size stack abstraction. We may use objects of this class in many different applications. As observed in Chapters 3 and 5, the class construct enables us to encapsulate the representation used for the data structure and export useful operations to be used by clients.

A client may create as many objects of the stack class as desired:

```
stack s1, s2;
s1.push(3);
s1.push(4);
s2.push(3);
if (s1.pop() == s2.pop) {...}
```

Clients may create objects of this class just as they may create variables of language-defined types. In fact, classes are user-defined types. They share most properties of language-defined types, including storage properties. For example, the above fragment creates stacks s1 and s2 as automatic variables. We may also create stacks in the free store:

```
stack* sp = new stack;
```

To access member functions (e.g., pop) the following notations denote equivalent expressions:

```
(*sp).pop();
```

and (more commonly used)

```
sp -> pop();
```

While useful, the class facility only addresses the question of how to encapsulate useful abstractions. What we have seen so far does not address the need to create new abstractions based on existing ones. Inheritance is the mechanism used for this purpose.

6.1.2 Inheritance

In the last section, we defined a stack class. Now suppose that we are writing an application in which we need a stack but we also need to know how many elements have already been pushed on the stack. What should we do? Write a new counting_stack class? Take the code of the above class and modify it? Use the same stack and keep track of the number of push operations externally in the client? All of these solutions are deficient in a programming in the large context. The first does not take advantage of work already done. The second creates two similar code modules that need to be maintained independently. Therefore, if a defect is found in the code or an optimization to the code is discovered, the changes must be applied to both copies of the code. The third alternative improperly separates the concerns of the client and the server and complicates the client code. The basic issue is that we already have a stack abstraction and the new abstraction we want should be a simple extension of it.

Inheritance is a linguistic mechanism that allows us to do just that by defining a new class which “inherits” the properties of a parent class. We may then add new properties to the child class or redefine inherited properties. The terminology in C++ is to *derive* a class from a *base* class. In this case, we want to derive a counting_stack from stack as shown below:

```
class counting_stack: public stack {
    public:
        int size(); //return number of elements on the stack
};
```

This new class simply inherits all the functions of the class stack[1]. All public member functions of stack become public member functions of counting_stack (that's what the public before stack specifies). We have also specified that there will be a new public function size() which is intended to return the number of elements stored in the stack. The class stack is called a base class or the parent class of counting_stack. The class counting_stack is said to be derived from its base class. The terms *subclass* and *superclass* are also used to refer to derived and base classes respectively.

Even this simple example demonstrates that inheritance is a fundamental concept for supporting programming in the large in that it enables us to develop modules based on existing ones without any modifications to the existing modules. This is the property that makes object-oriented programming an attractive paradigm for software engineering.

6.1.3 Polymorphism

The next feature of object-oriented programming languages is the support of polymorphism. All classes derived from the same base class may be viewed informally as specialized versions of that base class. Object-oriented languages provide polymorphic variables that may refer to objects of different classes. Object-oriented languages that adopt a strong type system limit the polymorphism of such variables: usually, a variable of class T is allowed to refer to objects of type T or any classes derived from T.

In the case of our example stack and counting_stack in the previous section, this means that a variable of type stack may also refer to an object of type counting_stack. In purely object-oriented languages such as Eiffel and Smalltalk, *all* objects are referred to through references and references may be polymorphic. In C++, only pointer, reference variables, and by reference parameters may be polymorphic. That is, a stack pointer may also point to a counting_stack object. Similarly, a formal by reference parameter expecting an actual parameter of type stack can refer to an actual parameter of type counting_stack. As an example, if we have two pointer variables declared:

[1] Actually, in C++ this cannot be done so simply. A way to implement size () is to return the value of top, but this is hidden to the subclass. As we will see in Section 6.3.1.4, this can be done, but class stack needs to be slightly changed. Other languages, like Eiffel, would require no change to be made in the parent class.

```
stack* sp = new stack;
counting_stack* csp = new counting_stack;
...
sp = csp;  //okay
...
csp = sp; //statically not sure if okay--disallowed
```

The assignment sp = csp; allows a pointer to a class to point to an object of a subclass. That is, the type of the object that is currently pointed at by sp can be of any of its derived types such as counting_stack. The assignment csp = sp; is not allowed in C++ because C++ has a strong type system. If this assignment were allowed, then a later call to csp->size() would be statically valid but may lead to a runtime error because the object pointed at by sp may not support the member function size(). A language with a weak type system, such as Smalltalk, would allow such an assignment and defer error checking to runtime. We will return to the typing issues raised by inheritance in Section 6.2.

According to the concepts developed in Chapter 3, a pointer to a class stack is a polymorphic variable which can also be assigned an object of class counting_stack. Assignments among objects of such types may be checked statically. The assignment of a counting_stack object to a variable of type stack is considered to be legal because a counting_stack object fulfills all the requirements of a stack object, whereas an assignment in the other direction should not be allowed, because a stack object does not have a size() component. In C++, if we do not use pointers, then the situation is different:

```
stack s;
counting_stack cs;
...
s = cs; //okay, object is coerced to a stack (no more size operation available)
cs = s; //not allowed because a later cs.size() would look syntactically okay but not work
at runtime
```

The assignment s = cs; is legal and is an example of ad hoc polymorphism. In fact, what happens is that the values stored in object cs are copied into the corresponding elements of s. This kind of coercion, of course, loses some of the values stored in cs, just as coercion from float to int loses some information.

6.1.4 Dynamic binding of calls to member functions

A derived class may not only add to the functionality of its base class, it may also add new private data and *redefine* or *override* some of the operations provided in the base class. For example, in counting_stack we may decide to provide a new implementation of the push function because we may want to keep track of the number of times push has been called. Now, if sp is a reference to a stack variable and csp is a reference to a counting_stack variable, we expect that csp->push() will call the push of a counting_stack object but what about sp->push()? Since sp is a polymorphic variable, it may be pointing at a counting_stack object or a stack object. This raises an issue of proper binding of operations. Consider the following example:

```
stack* sp = new stack;
counting_stack* csp = new counting_stack;
...
sp.push();  // stack::push
csp.push(); // counting_stack::push
...
sp = csp;  //assignment is okay
...
sp.push();  //which push?
```

Which is the push operation invoked in the last statement, stack::push or counting_stack::push? Because the assignment sp = csp is allowed, at run-time sp may be pointing to a stack object or to a counting_stack object. Should the choice of which routine to call be made statically, in which case stack::push() will be called, or dynamically, in which case counting_stack::push() will be called. So-called purely object-oriented languages, such as Smalltalk and Eiffel, bind the choice dynamically based on the type of the object. In fact, as stated, dynamic binding (often called *dynamic dispatching* in object-oriented terminology) is one of the tenets for object-oriented programming languages. C++, however, not being a purely object-oriented language, provides features for both static and dynamic binding. Section 6.3.1 presents the C++-specific features.

Dynamic binding combined with inheritance is a powerful notion. For example, we may define a class polygon and derive various specialized versions of polygons such as square and rectangle. Suppose that polygon defines a function perimeter to compute the perimeter of a general polygon. Some of the derived classes may define their own special perimeter functions because they are presumably more efficient. We may maintain a list of various types of polygons. Every time we select an element p from the list, the use of dynamic binding ensures that a call p.perimeter for a variable p of type polygon will call the "right" perimeter function based on the type of the object currently assigned to p. Clearly, dynamic binding is more flexible than static binding. In languages that do not support dynamic binding, we may have to use case statements (as in Pascal) or use function pointers (as in C) to achieve the same result but with code that is more verbose and harder to maintain. For example, in Pascal we might implement polygon as a variant record and explicitly call the right perimeter function based on the tag of the variant record. In C, each object could contain a pointer to its perimeter function and the call would have to be made indirectly through this pointer. Both of these solutions are not only more verbose but also less secure and maintainable than the solution using inheritance and dynamic binding.

As we have seen in Chapter 3, dynamic binding of the function call may create possibilities of type violations. Indeed, in dynamically typed languages such as Smalltalk, this type of error may easily occur. A call to member function f of an object v may fail at runtime because the object bound to v belongs to a class that does not provide function f, or if it does, because the types of the actual parameters (or the result type) are incompatible with those of the formal parameters. Several languages, such as Eiffel and C++, as we shall see later, combine polymorphism and dynamic binding with static type checking.

6.2 Inheritance and the type system

In the previous section, we have described the basic components of object-oriented programming languages. The interaction between inheritance and type consistency rules of the language raises a number of interesting issues. In this section, we consider some of these issues.

6.2.1 Subclasses versus subtypes

In Chapter 3, we saw the concept of subtype with which we defined a new type as a subrange of an existing type. For example, we defined week_day as a subrange of day. Subtyping introduces a relationship among objects of the subtype and objects of the the parent type such that objects of a subtype may also be viewed as objects of the parent type. For example, a week_day is also a day. This relationship is referred to as the *is-a* relationship: week_day *is-a* day. The subtype relationship is generalizable to user-defind types such as those defined by classes. For example, a counting_stack *is-a* stack but not vice versa.

But not all subclasses create subtypes. If a derived class only adds member variables and functions or redefines existing functions in a *compatible* way, then the derived class defines a subtype. If it hides some of the parent's member variables and functions or modifies them in an incompatible way, then it does not create a subtype. Therefore, whether a derived class defines a subtype depends on the definition of the derived class and is not guaranteed by the language.

What does it mean for a function f in a derived class to override a function f in a base class in a *compatible* way? Informally, it means that an occurrence of base::f(x) may be replaced by derived::f(x) without risking any type errors. For example, if the signature of the function derived::f is identical to the signature of base::f, no type errors will be introduced as a result of the replacement. We will examine this issue more deeply in the next subsection.

6.2.2 Strong typing and polymorphism

In Chapter 2 we defined a strong type system as one which guarantees type safety. Strong type systems have the advantage of enabling type errors to be caught at compile-time. Statically typed languages provide a strong type system. In this section we discuss how object oriented languages can rely on dynamic dispatch as a fundamental principle and yet adopt a strong type system.

Let us assume that we have a base class base and a derived class derived and two objects derived from them:

```
class base {...};
class derived: public base {...};
...
base* b;
derived* d;
```

We have seen that we may assign d to b but not b to d. The question is under what circumstances can we guarantee that an assignment

```
b = d;
```

will not lead to a type violation at runtime? We may ask this question in terms of *substitutability*: can an object of type derived be substituted for an object of class base in every context? Or in general, can an object of a derived type be substituted for an object of its parent type in every context, and is such a kind of polymorphism compatible with strong typing? If substitutability is ensured, the derived type can be viewed as a subtype of the parent type. To answer this question we need to examine the contexts under which objects are used. By imposing some restrictions on the use of inheritance, the language can ensure substitutability. Below we examine several sufficient (but not necessarily necessary) restrictions.

6.2.2.1 Type extension

If the derived class is only allowed to extend the type of the parent class, then substitutability is guaranteed. That is, if derived does not modify any member functions of base and does not hide any of them, then it is guaranteed that any call b.f(...) will be valid whether b is holding a base object or a derived object. The compiler may do type-checking of any uses of base variables solely based on the knowledge that they are of type base. Therefore static type checking can guarantee the lack of runtime violations. Type extension is one of the mechanisms adopted in Ada 95.

6.2.2.2 Overriding of member functions

Restricting derived classes to only extend the parent type is a severe limitation on an object-oriented language. In fact, it rules out dynamic dispatch completely. Many languages allow a derived class to redefine an inherited member function. For example, as in our previous example, we may derive a square class from a polygon class. The square class may redefine the general perimeter function from the base class by a more efficient version of a perimeter function. In C++, the base class must specify the function perimeter as a virtual function, giving derived classes the opportunity to override its definition. That is,

```
class polygon {
    public:
        polygon (...) {...} //constructor
        virtual float perimeter () {...};
        ...
};
class square: public polygon {
    public:
        ...
        float perimeter() {...};  //overrides the definition of perimeter in polygon
};
```

The typing question is: under what conditions can we guarantee that a use of a square object may substitute the use of a polygon object in all contexts? That is, under what conditions can we guarantee that a call p->perimeter() will always be valid whether *p is a polygon or a square object? C++ requires that the signature of the overriding function must be *exactly* the same as the signature of the overridden function. This rule ensures that the compiler can do the type checking based only on the static type of p and the signature of the function polygon::permiter. If at runtime p happens to hold a square object or any other derived type object, a function other than polygon::perimeter will be called but that function will have exactly the same parameter requirements and no runtime type violations will occur.

Can we relax the C++ requirement and still ensure type safety? Again, we can analyze the semantic requirements of the relationship between an overridden function and the function it overrides in terms of substitutability. In general, we must be able to do the type checking based on the parent class, knowing that a derived member function may be substituted for the overridden function. Clearly, we must require exactly the same number of parameters in the overriding function and the overridden function. The question is how should the signatures of the two functions be related?

Consider the following program fragment (we will use the syntax of C++ but we will take some liberty with its semantics):

```
//not C++
class base {
public:
    void virtual fnc (s1) (...) //s1 is the type of formal parameter
...};
class derived: public base {
public:
    void fnc (s2) (...)  //C++ requires that s1 is identical to s2
...};
...
base* b;
derived* d;
s1 v1;
s2 v2;
...
if (...) b = d;
...
b->fnc(v1); // okay if b is base but what if it is derived?
```

To ensure substitutability, the call b->fnc(v1) must work at runtime whether b holds a base object or a derived object. That is, the parameter v1 must be acceptable to both base::fnc() and to derived::fnc(). This, in turn, means that derived::fnc() must be able to accept v1, which is of type s1. In other words, type s1 must be a derived type that can be substituted for type s2. That is, either class s1 simply extends class s2 (Section 6.2.2.1), or class s1 redefines member functions of s2, but redefinitions satisfy the constraints we are discussing in this section. Informally, this rule means that the overriding function must not impose any more requirements on the input parameters than the overridden function does.

Now let us consider a member function with a result parameter.

```
//not C++
class base {
public:
    t1 virtual fnc (s1) (...);    //s1 is the type of formal paramter;
                                  // t1 is the type of result parameter
...
};
class derived: public base {
public:
    t2 fnc (s2) (...);  //C++ requires that s1 is identical to s2 and t1 is identical to t2
...
};
...
base* b;
derived* d;
s1 v1;
s2 v2;
t0 v0;
...
if (...) b = d;
...
v0 = b->fnc(v1); // okay if b is base but what if it is derived?
```

Again, substitutability means that if b holds a derived object, the call fnc() will work at runtime and a proper result will be returned to be assigned to v0 without any type violations. That means that the result type of the overriding function (t2) must be substitutable to the result type of the overridden function (t1), which must be substitutable to the type of v0, if the last assignement of the fragment is considered to be legal by the compiler. In other words, t2 must be a subtype of t1 which must be a subtype of t0. Combining the rules on input parameters and result parameter together, we can state intuitively that an overriding function must be able to accept the parameters of the overridden function and return at least what the overridden function returns: it may accept less but it may return more.

Stated more precisely:

- The input parameters of the overriding function must be supertypes of the corresponding parameters of the overridden function;
- The result parameters of the overriding function must be subtypes of, the result parameters of the overridden function.

These two type checking rules are known respectively as the *contravariance* rule on input parameters and the *covariance* rule on result parameters. We have stated them here as syntactic rules that ensure type safety. However, the contravariance rule on input arguments seems rather counter-intuitive. It says that even though we may define a more specialized function in a derived class, the input parameters of the specialized function may not impose any more specific requirements.

Few programming languages enforce these rules completely. Emerald is one language that does. C++, Java, Object Pascal, and Modula-3 follow neither: they requires the exact identity of the two functions. Eiffel and Ada require covariance of both result and input arguments. The assertions of Eiffel, as seen in Chapter 3, may be used to check the contravariance requirements at runtime.

```
class point{
    public:
        x: float;
        y: float;
        bool equal (point p) //bool is defined as a boolean type
            {return (x == p.x && y == p.y);}
};
class colorPoint: public point{
    public:
        color: float;
        bool equal (colorPoint p) //bool is defined as a boolean type
            {return (x == p.x && y == p.y && color == p.colorPoint);}
};
```

FIGURE 78. Classes point and colorPoint

The example in Figure 78 shows the counterintuitive nature of the contravariance requirement on input parameters and the basic difficulty of equating subtyping as an abstract concept and inheritance as a language construct that implements it. We have defined a class point characterized by x and y coordinates and a member function equal that can compare itself for equality with another point. We then derive a colorPoint which inherits x and y from class point, adds another instance variable color, and redefines the equal member function. The member function must be redefined because the colorPoint equality test must compare two colorPoints. For this reason, the input parameter to colorPoint::equal must be of type colorPoint. But if we allow such a redefinition, colorPoint may not be considered a subtype of point. This redefinition, although intuitively necessary, goes against our rule of contravariance of input arguments which requires the parameter to colorPoint::equal() to be a supertype of the input parameter of point::equal(). We can see the problem by considering a call p1.equal(p2). This call works if p1 and p2 are both points. But the call will fail if we substitute a colorPoint for p1 because the equality test will attempt to access the nonexistent color variable of p2.

In conclusion, if inheritance is constrained by requiring that either (a) the derived class only provides extensions, or (b) redefinitions are also allowed, but contravariance and covariance are required for input and output parameters, respectively, then substitutability is ensured and derived classes can be considered to define subtypes of their parent class. The resulting type system would be polymorphic (inclusion polymorphism) and yet it would be strong. The price to pay for this conceptual integrity of the language, however, is that the restrictions imposed on inheritance are severe, and even counterintuitive.

6.2.3 Inheritance hierarchies

Hierarchical organization is an effective method of controlling complexity. The inheritance relationship imposes a hierarchy and provides a mechanism for the development of a hierarchically organized families of classes. In this section we discuss several issues raised by inheritance hierarchies.

6.2.3.1 Single and multiple inheritance

In Simula 67, Ada, and Smalltalk, a new class definition is restricted to have only one base class: a class has at most one parent class. These languages have a *single-inheritance* model.

C++ and Eiffel have extended the notion of inheritance to allow a child to be derived from more than one base class. This is called *multiple inheritance*. For example, if we have a class displayable and a class polygon, we might inherit from both to define a displayable rectangle:

```
class rectangle: public displayable, public polygon {
...
}
```

The introduction of multiple inheritance into a language raises several issues. For example, there may be name clashes between parents. For example, a displayable class and a bank_account class may both provide a member function draw() and inheriting from both to build a displayable bank_account may cause problems. In this case, which draw() function is exported by the derived class? The derived class needs a way to both access and to export the desired members. Another issue is what if several of the parents are themselves derived from the same base class. The members of the parent will be repeated multiple times in the child class. This may or may not be desirable. Some languages provide features to resolve such name conflicts. For example, Eiffel has a construct to **undefine** an inherited feature; it also has a construct to **rename** an inherited feature.

The successful use of multiple inheritance requires not only well-designed inheritance hierarchies but also orthogonally designed classes that may be combined without clashing. In practice, the use of multiple inheritance requires much care. Whether its benefits outweigh its complexity is an open question. Java, which adopts many features of C++, uses only single inheritance but introduce separate interfaces and supports the idea of inheriting from multiple interfaces.

6.2.3.2 Implementation and interface inheritance

One of the promises of object-oriented programming is that new software components may be constructed from existing software components. This would be a significant contribution to programming in the large issues. To what extent does inheritance support a methodology for such incremental building of components?

In the last chapter, we discussed the importance of encapsulation in achieving independent modules whose internals may be modified without affecting their interfaces and thus their *clients*, who *use* the resources specified in the interface. Inheritance complicates the issue of encapsulation because the derived classes of a class are a different type of client for the class. On the one hand, they may want to extend the facilities of a parent class and may be able to do so solely by using the public interfaces of the parent class; on the other hand, the facilities they provide to their clients may often be implemented more efficiently if they access the internal representations of their parent classes. C++ introduces protected members (see Section 6.3.1.4) and friend classes exactly for these special clients of a base class. Eiffel, on the other hand, allows derived classes to access all features defined in the parent class.

There is a trade-off. If the derived class uses the internal details of the parent class—it inherits the implementation of the parent—it will be affected any time the implementation of the parent class is modified. This means that, at the very least, it will have to be recompiled but most likely it will also have to be modified, leading to familiar maintenance problems. This is not a major problem if the base class and the derived class are part of the same package produced and maintained by the same organization. It is a serious problem, however, if the base class is supplied by a library and a different organization creates the derived class.

From a software engineering view, interface inheritance is the right methodology but to rely only on interface inheritance requires both a well-designed base class and efficient language implementations. A well-designed inheritance hierarchy is a requirement for the successful use of object-oriented programming in any case. Any hierarchy implies that the nodes closer to the root of the hierarchy affect a larger number of the leaf nodes of the hierarchy. If a node close to the root needs to be modified, all of its children are affected. As a result, even though inheritance supports the incremental creation of software components, it also creates a tightly-dependent set of components. Modifications of base classes may have far reaching impact.

6.3 Object-oriented programming support in programming languages

The way different languages support object-oriented programming is related to the philosophy of the language and more specifically to the language's object and encapsulation models. In C++, the class construct defines a user-defined type. Object-oriented features have been added to the language to allow programmers who want to use object-oriented programming to do so. In Eiffel, the class construct defines an abstract data type. The language has been designed to support the object-oriented programming style exclusively. In Ada 95, the package is simply an encapsulation mechanism for packaging a set of related entities. It is neither necessarily a type, nor an abstract data type. It may be used to support those notions, however. Ada 95 has some object-orientation features but the language remains a module-oriented language. In this section, we examine these languages a little more closely from an object-oriented view.

6.3.1 C++

C++ supports object-oriented programming by providing classes for abstract data types, derived classes for inheritance, and virtual functions for dynamic binding. This support is provided with static type checking.

As we have seen, a C++ class defines a user-defined type. Indeed the programmer can create first-class types because the language allows the definition of initialization, assignment, and equality test for the type being defined. As a result, objects of user-defined types may behave quite like objects of language-defined types: they may be created on the stack or in free store, passed as parameters, and returned by functions as results.

The language supports both inheritance and multiple inheritance for defining new classes.

6.3.1.1 Classes

We have already seen the use of C++ classes as a definition mechanism for abstract data types. C++ provides the programmer with control over the creation, initialization, and cleanup of objects of such abstract data types. In particular, one or more constructors may be defined for a class. Such constructors are invoked automatically to initialize objects of the class at creation time. A constructor has the same name as the class. By analogy to a constructor, a *destructor* is invoked automatically when the object is destroyed—either explicily through a call to delete or implicitly when the object goes out of scope. The ability to control what happens when an object is destroyed is critical for complex data types that may have allocated substructures in the heap. For example, simply deleting a pointer to the head of a list may leave the entire list inaccessible in the free store. A destructor gives the progammer the possibility to clean up after the object properly based on the requirements of the object. The name of a destructor is the same as the class name preceded by ~ (i.e. the complement of the constructor).

We have seen in Chapter 3 that garbage collection is an important issue in programming languages that support dynamic object allocation. Constructors and destructors may be used by the programmer to design object-specific storage allocation and garbage collection policies. For example, the destructor included in a class might link the released object in a free list. The constructor included in the same class, would first try to extract an object from the free list and, if the free list is empty, would allocate a new object from scratch. This policy would be similar to a garbage collection service that collects and recycles different kinds of garbage (paper, plastic, etc.) separately.

For a derived class, its constructor is invoked after that of its base class. This order guarantees that the derived class constructor may rely on the availability of its inherited variables. The destruction of a derived class proceeds in the opposite direction: first the constructor of the derived class is invoked followed by that of its parent.

Besides construction and destruction, programmer control is important over two other operations for user-defined types: assignment and equality comparison. These two operations are related semantically. In general, we expect that after assigning an object a to object b, the two objects are equal. C++ by default uses member-wise copy and member-wise comparison for assignment and comparison of class objects. This is often inadequate if the structure of the object involves heap-allocated components. In these cases, the programmer may define class-specific assignment and equality operations. There are no special C++ features for this: as any other operators, the programmer may overload = and ==.

Most languages treat these operations in a special way. For examle, we have seen that Ada lets the programmer designate a type as **private** to indicat that the language-defined assignement and equality apply to the type; **limited private** means that they do not.

6.3.1.2 Virtual functions and dynamic binding

By default, a function call is bound to a function definition statically. If the programmer wants a function to be selected dynamically, the function must be declared as virtual in the base class and then redefined in any derived classes. For example, suppose we define a class student that supports some operations including a print() operation. We expect to derive different types of students from this class, such as college_student and graduate_student. First we define the student class and define a default print() function:

```
class student{
public:
...
virtual void print(){...};
};
```

The virtual qualifier on print() says that classes derived from student may redefine the function print(). If they do, then the appropriate print() function will be selected dynamically. For example:

```
class college_student: public student{
    void print() {
    . . . // specific print for college_student
```

```
        }
    };
```

defines a derived class that inherits all its operations from student but supplies a new definition for print(). Now, the following code sequence shows the effect of dynamic binding:

```
    student* s;
    college_student* cs;
    ...
    s->print(); //calls student::print()
    s = cs; // okay
    s->print(); //calls college_student::print()
```

Remember that in C++, the binding is normally static. The virtual function and pointers are used to effect dynamic binding.

To ensure the type safety of virtual functions, first the virtual function must be defined the first time it is declared, that is, in the base class. This means that the function will be available in any derived classes even if the derived class does not define it. Second, any redefinition of a virtual function may not change the signature of the function. That means that no matter which function is invoked, it is guaranteed to have the same signature. We have discussed this second rule in Section 6.2.2.2.

6.3.1.3 Use of virtual functions for specification

Virtual functions may be used to define abstract classes. For example, we may specify that a shape must have three functions named draw, move, and hide, without providing an implementation for these functions. A virtual function that does not have an implementation is called a *pure virtual function*. To write a pure virtual function, its body is written as = 0;. If one of the functions of a class are pure virtual, the class is called *abstract*. We may not create objects of an abstract class. In the example, objects of type shape cannot be created because such a class does not have an implementation. The pure virtual designation for a function says that a derived class based on shape must define such functions concretely. Figure 79 shows the outline of the class shape and a class rectangle derived from it.

```
class shape{
public:
    void draw() = 0; // this and the others are pure virtual function
    void move() = 0;
    void hide() = 0;
    point center;
};

class rectangle: public shape{
    float length, width; //specific data for rectangle
public:
    void draw() {...}; //implementation for the derived pure virtual function
    void move() {...};
    void hide() {...};
};
```

FIGURE 79.A C++ abstract class using pure virtual functions

We may view abstract classes as a specification for a set of derived classes. The abstract class specifies the interface and the derived classes must provide the implementation.

6.3.1.4 Protected members

Let us go back to the example of counting_stack in Section 6.1.2. We want counting_stack to provide an additional function called size(). How are we going to implement size()? The simplest way to do it is to return the value of top. That is, define size() as:

```
counting_stack::size(){return top;};  //not quite right!
```

But there is a problem here. The variable top was declared to be private in stack. This means that it is only known within stack, and not even within classes derived from stack. We do not want to make top public because that would make it available to all clients also. For this reason, C++ has a third class of visibility for class entities: *protected* entities are visible within the class and any derived subclasses. So, if we declare top to be a protected variable of stack, rather than a private variable, then the above implementation of size() will work properly. We show in Figure 80 the code for both stack and counting_stack.

```
class stack{
    public:
        stack();  {top = 0;} //constructor
        void push(int) {s[top++] = i;};
        int pop() {return s[--top];};
    protected:
        int s[100];
        int top;
};

class counting_stack : public stack {
    public:
        int size(){return top;};  //return number of elements on the stack
};
```

FIGURE 80. Example of class inheritance (derivation) in C++

Thus the general form of a C++ class is shown here:

```
class C {
private:
    // accessible to members and friends only
protected:
    // accessible to members and friends and
    // to members and friends of derived classes only
public:
    // accessible to the general public
};
```

In summary, then, the C++ language provides three levels of protection. Entities defined in a class may be private (default case), in which case they are only accessible inside the class itself; they may be defined as protected, in which case they are accessible inside the class and inside any classes derived from it; or they may be defined as public, in which case they are accessible generally. The public entities define the services provided by the class and constitute its interface. The private entities deal with the internal details of the class such as the representation of data structures. The protected entities are not of interest to users of the class but they are of interest to any classes that are derived from this class and need to provide services based on the services already provided by this class.

Using the terminology of Section 6.2.3.2, we can say that protected members support the use of implementation inheritance whereas public members support the use of interface inheritance.

6.3.1.5 Overloading, polymorphism, and genericity

In Chapter 3 we discussed several kinds of polymorphism. Now we have seen that all the forms exist in C++. First, we have seen the use of ad-hoc polymorphism in the support of overloading of operators and functions, in which case the binding of the operator or function is done at compile-time based on the types of the arguments. If a derived class redefines a virtual function f of a base class b, the base class defines a polymorphic type and the function call b.f() is a call to a polymorphic function resolved dynamically on the basis of the type of the object referred to by b. Since the object must belong to a subclass of the class to which b is declared to point, this is a case of inclusion polymorphism. If the function is not declared to be virtual in the base class, then the two functions in the base and derived classes are treated as simply overloading the same function name: the proper function to call is selected at compile-time. Finally, we have seen that C++ also supports generic functions. If a function f(a,b) is generic, the types of a and b are used at compile time to instantiate a function that matches the types of the parameters a and b. There is no dynamic dispatch in this case.

6.3.2 Ada 95

The original version of the Ada language, introduced in 1983, was an object-based language. The package construct may be used to create objects that encapsulate both data, possibly private, and related operations. This enabled object-based programming. Since the introduction of Ada, however, the concepts of object-oriented programming have become better understood. As a result, a number of features were added to Ada 95 to support object-oriented programming techniques. In particular, **tagged types** support derivation of a new type from an existing type, and a technique called “classwide programming” combined with tagged types supports dynamic dispatch. Another feature of Ada 95 is the ability to define abstract types which in turn enable the association of multiple implementations with the same interface. Such flexibility is usually associated with object-oriented programming. We discuss these features below.

6.3.2.1 Tagged types

In Chapter 3, we have seen how new types may be derived based on existing types. We have also seen how subtypes of discrete types may be defined that inherit the operations of the parent type but restrict the range of values of the type. In Chapter 5, we have seen that we can use a package to group together a type and its associated operations, thus creating a user-defined type. With Ada 95, a type declaration may be designated as *tagged*, in which case it is possible to derive new types from it by extending it. This facility allows us to define a hierarchy of related types. For example, Figure 81 defines a tagged type named Planar_Object as having certain set of basic properties such as X and Y coordinates of its center and three functions: one to compute its Distance from the origin, another to Move it to a new position, and another to Draw it as a predefined icon on the screen. We might then derive other objects such as *points* and *circles* which each extend the basic object in their own ways.

```
package Planar_Objects is
    type Planar_Object is tagged
    record
        X: Float := 0.0; --default initial value of the center's x coordinate
        Y: Float := 0.0; --default initial value of the center's y coordinate
    end record;
    function Distance (O: Planar_Object) return Float;
    procedure Move (O: inout Planar_Object; X1, X2: Float);
    procedure Draw (O: Planar_Object);
end Planar_Objects;
```

FIGURE 81. An Ada 95 package that defines a tagged type Planar_Object

We will assume that the body (implementation) of the package Planar_Objects is given elsewhere. With this definition we may declare objects of type Planar_Object and apply Distance, Move and Draw operations to them. Next we can define new types Point, Rectangle, and Circle that inherit the properties of the type Planar_Object. For a Point, the X and Y coordinates are enough, thus the data representation of Planar_Object does not need to be extended. But for a Circle, we will add a Radius field and for a Rectangle, we will add the sizes of the two edges. Finally, it is necessary to redefine Draw for all of them. These shapes are defined in the package in Figure 82.

```
with Planar_Objects; use Planar_Objects;
package Special_Planar_Shapes is
    type Point is new Planar_Object with null record; --indicates no additions
    procedure Draw (P: Point);
    type Circle is new Planar_Object with
        record
            Radius: float;
        end record;
    procedure Draw (C: Circle);
    type Rectangle is new Planar_Object with
    record
        A, B: Float;
    end record;
    procedure Draw (T: Rectangle);
end Special_Planar_Shapes;
```

FIGURE 82. Extending tagged types in Ada 95

First, note that we may only *extend* a type, not remove any properties. Therefore, the new types are guaranteed to have all the fields of the parent type. As a result, the derived types can easily be coerced to the parent type by simply ignoring the additional fields. Thus, the following statements are valid:

```
O1: Planar_Object; -- basic object at origin
O2: Planar_Object (1.0, 1.0); -- on the diagonal
C: Circle := (3.0, 4.5, 6.7);
...
O1 := Planar_Object(C); -- coercion by ignoring the third field of C
```

What if we want to do the assignment in the opposite direction? As opposed to C++, this can be done but since the object on the right hand side does not have all the necessary fields, they must be provided by the programmer explicitly. For example:

```
C := (O2 with 4.7);
```

In our example, the three newly defined types inherit the operations Distance and Move from Planar_Objects. When they need to redefine (that is, override) an operation such as Draw, the rule on redefinition is similar to Eiffel, that is, the parameters of the overriding operations must be subtypes of (more specific than) the parameters of the overridden operations.

Thus, the tagged types of Ada 95 support the development of a tree of types through the use of inheritance, overriding, and extension. Type coercion is supported from a derived to any ancestor type. One of the major goals of Ada in adopting the type extension model of inheritance has been to ensure that extension of a type does not force the recompilation of either the type being extended, or the clients of that type.

6.3.2.2 Dynamic dispatch through classwide programming

The tagged types of Ada 95 are also used to support dynamic binding of function calls. The tag is an implicit field of an object and is used at runtime to identify the object's type. For example, suppose that we want to write a procedure Process_Shapes that will process a collection of objects that may be Points, Rectangles, or Circles. We need to declare this procedure as accepting a polymorphic type that includes all these types. The 'Class attribute of Ada 95 constructs exactly such a class. That is, the expression T'Class applied to a tagged type T is the union of the type T and all the types derived from T. It is called a *class-wide* type. In our example, we can write our procedure as:

```
procedure Process_Shapes (O: Planar_Object'Class) is
...
begin
    ...
    ... Draw (O) ...;  --dispatch to appropriate Area procedure
    ...
end Process_Shapes;
```

Since it is often useful to access objects through pointers, it is also possible to declare polymorphic pointers such as:

```
        type Planar_Object_Ptr is access all Planar_Object'Class;
```

6.3.2.3 Abstract types and routines

As in C++ and Eiffel, Ada 95 supports the notion of top-down design by allowing tagged types and routines (called subprograms in Ada) to be declared abstractly as a specification to be implemented by derived types. For example, we might have declared our Planar_Object type before as an abstract type as in Figure 83.

```
package Planar_Objects is
    type Planar_Object is abstract tagged null record;
    function Distance (O: Planar_Object'Class) return Float is abstract;
    procedure Move (O: inout Planar_Object'Class; X, Y: Float)  is abstract;
    procedure Draw (O: Planar_Object'Class) is abstract;
end Objects;
```

FIGURE 83. An abstract type definition in Ada 95

This package will not have a body. It is only a specification. By applying derivation to the type Planar_Object, we can build more concrete types. As before, we may derive a tree of related types. Once concrete entities (records and subprograms) have been defined for all the abstract entities, we have defined objects that may be instantiated.

6.3.3 Eiffel

Eiffel was designed as a strongly-typed object-oriented programming language. It provides classes for defining abstract data types, inheritance and dynamic binding. Classes may be generic based on a type parameter.

6.3.3.1 Classes and object creation

An Eiffel class is an abstract data type. As we have seen in Chapter 3, it provides a set of **feature**s. As opposed to C++, all Eiffel objects are accessed through a reference. They may not be allocated on the stack. Again in contrast to C++, object creation is an explicit, separate step from object declaration. In one statement, a reference is declared and in a following statement, the object is created. As in:

```
b: BOOK; --declaration of a reference to BOOK objects
!!b;    --allocating and initializing an object that b points to
```

The language provides a predefined way of creating and initializing objects, based on their internally defined data structure. It is also possible to provide user-defined “creator” routines for a class which correspond to constructors of C++.

6.3.3.2 Inheritance and redefinition

A new class may be defined by inheriting from another class. An inheriting class may **redefine** one or more inherited features. We saw that C++ requires the redefined function to have exactly the same signature as the function it is redefining. Eiffel has a different rule: the signature of the redefining feature must conform to the signature of the redefined feature. This means that for redefined routines, the parameters of the redefining routine must be assignment compatible with the parameters of the redefined routine.

Consider the following Eiffel code sequence:

```
class A
feature
    fnc (t1: T1): T0 is
    do
    ...
    end -- fnc
end --class A

class B
inherit
    A redefine fnc
end
feature
fnc (s1: S1): S0 is
    do
    ...
    end -- fnc
end --class B
...
a: A;
b: B;
...
a := b;
...
... a.fnc (x)...
```

The signature of fnc in B must conform to the signature of fnc of A. This means that S0 and S1 must be assignment compatible with T0 and T1 respectively. Referring to the discussion of Section 6.2.2.2, the Eiffel rule follows the covariance rule on both input and output arguments. The Eiffel **require** and **ensure** clauses which are used to specify pre- and post-conditions for routines and classes may be used as a design tool to impose a stronger discipline on redefinitions. In particular, restating the rules of Section 6.2.2.2, the fnc of B must **require** no more than the fnc of A and must **ensure** at least as much as the fnc of A.

Eiffel has a different view of inheritance from what we have described in Section 6.1.2. In particular, Eiffel views a subclass (derived class using C++ terminology) as either extending its parent class or specializing it. For example, a subclass may **undefine** a feature of the parent class. In such a case, the child class may no longer be viewed as satisfying the is-a relationship with its parent.

The **deferred** clause of Eiffel may be used as the virtual specifier of C++ to implement abstract classes.

Eiffel supports multiple inheritance. To resolve the name conflicts that may occur due to inheriting from more than one base class, the **undefine** construct may be used to hide some features and **rename** may be used to rename others.

6.3.4 Smalltalk

Smalltalk was the first purely object-oriented language, developed for a special purpose machine, and devoted to the devopelopment of applications in a highly interactive single-user personal workstation environment. It is a highly dynamic language, with each object carrying its type at runtime. The type of of a variable is determined by the type of the object it refers to at runtime. Even the syntax of Smalltalk reflects its object orientation.

All objects are derived from the predefined class object. A subclass inherits the instance variables and methods of its parent class (called its superclass) and may add instance varaiables and methods or redefine existing methods. A call to an object is bound dynamically by first searching for the method in the subclass for the method. If not found, the search continues in the superclass and so on up the chain until either the method is found or the superclass object is reached and no method is found. An error is reported in this case.

A class defines both *instance* variables and methods, and *class* variables and methods. There is a single instance of the class variables and methods available to all objects of the class. In contrast, a copy of each instance variable and method is created for each instance of an object of the class.

6.4 Object-oriented analysis and design

In this chapter we have described programming language support for object-oriented programming. The object-oriented approach to software development has grown to encompass not just programming but most other phases of software development. Object-oriented analysis tries to analyze the application domain in terms of objects, their associated operations, and relationships among objects. Object-oriented design tries to design a system that consists of objects. Such designs are implemented more easily in object-oriented languages. Indeed, the use of object-oriented languages is only effective if the design is object-oriented. It is at the design stage that component objects and their relationships are identified. Constructs such as abstract classes that we have seen in Ada, Eiffel, and C++ may be used to document object-oriented designs that can then be implemented in programming languages.

The substitutability and proper inheritance properties that we have discussed for programming languages are treated in terms of *is-a* relationship at the analysis and design stages. In our example, a counting_stack *is-a* stack and therefore may be substituted anywhere a stack is needed (for example passed to a procedure that expects a stack). But stack is *not* a counting_stack and therefore a stack may not be substituted for a counting_stack. A good design rule is to use inheritance to derive a new class when derived_class *is-a* base_class. The C++ rule on assignments among derived and base classes may also be defined using the *is-a* relationship. The assignment a = b is allowed if b *is-a* a. While this rule is intuitive and simple to state, it is not always easy to determine whether two objects are related with the *is-a* relation. For example, we have seen that colorPoint is not necessarily a point (Section 6.2.2.2). Usually, the relationship that holds is "*is-a-kind-of*." It often takes great care to create *is-a* relationships.

6.5 Summary

Object-oriented programming is an effective style of programming for many situations. In recent years, however, it has been advertised rather as a panacea to all software development problems. It is important to realize that the design of large software systems is an inherently difficult activity. Programming language features may help in implementing good designs but they do not remove the deficiencies of a bad design. More importantly, designs are not necessarily bad or good. For example, consider developing tree abstractions for use in a system. Suppose we need both general trees and binary trees. Should the two classes be related by an inheritance relationship? If so, which one should be the base class? Depending on how the rest of the design fits together, one or the other class as a base class would be the better choice.

In practice, the initial development of the design is not the major problem. The designer is often able to build an inheritance hierarchy that fits the problem at hand. The problems occur later when the software is extended to meet new requirements. The difficulties arise when new classes needed to be defined introducing new is-a relationships that are not compatible with previous such relationships. If the inheritance tree needs to be modified significantly, then the impact on the rest of the software can be significant.

6.6 Bibliographic notes

Simula 67 was the first object-oriented language. It introduced the notions of class and inheritance. All other object-oriented languages have their ancestry in Simula 67 and later to Smalltalk. Smalltalk was the first popular object-oriented language. It is a dynamically typed language and was initially developed on special hardware but current implementations of the language are available on many computers including personal computers. CLOS (Common LISP Object System) was an early attempt to introduce objects into an existing language. C++ added object-orientation support to an existing imperative language (Stroustrup). The book by Stroustrup (Design and Evolution) gives a fascinating account of how C++ grew from C with classes to a full language on its own. It also explains the differences between an object-oriented language and a language that supports object-oriented programming. Meyer is a comprehensive treatment of object-oriented programming using the Eiffel language. The language itself is described fully in Meyer.

References to languages: Dylan, Beta, Self, Emerald, Oberon, Modula, Java.

Snyder (Encapsulation and inheritance) pointed out the differences between implementation and interface inheritance.

Wegner paper is the source for the classification of languages into object-based and object-oriented.

Bruce et al. is the source of the example in Section 6.2.2.2. The type structure of strongly typed object oriented languages has sparked a large amount of type-theoretic research in recent years. A number of papers have been written to clarify the rules of covariance and contravariance (Liskov, Castagna, Bruce). Cardelli and Wegner was the first of these papers. Barnes 95 is the source for our treatment of Ada 95. Wirth (Wirth 93) descirbes the idea of type extension.

The use of SIMULA 67’s prefix mechanism (its inheritance) in top-down design is illustrated by several examples in (Birtwistle et al. 1976). A number of approaches to object-oriented analysis and design are described in (Fowler).

6.7 EXERCISES

1. Implement a C++ class employee that supports a virtual method print() which prints the name and age of an employee object. Next derive a class manager which supplies its own print() method which, in addition to the employee information, prints the group number for which the manager is responsible (this is an additional field of manager). Also derive another class from employee called part_time. The part_time class also supplies its own print() which prints how many hours a week the employee works.
   - Can you use the print of employee in manager?
   - Explain how you would implement the same program in Pascal.
   - Compare the object-oriented and the procedual solution in terms of maintainability. What changes are necessary in the two solutions if we need to add a new type of employee?
   - In the C++ solution, how would you implement a part_time_manager? Does your solution allow you to implement this new class using multiple inheritance?

2. Consider the following C++ program fragment:

```
class point {
public:
  float x;
  float y;
  point ( float xval, float yval): x(xval), y(yval) {};
  virtual int equal ( point& p)
  {cout << "calling equal of point.\n";
  return x == p.x && y == p.y;
  };
};

class colorPoint: public point{
public:
  int color;
  colorPoint (float xval, float yval, int cval): point(xval, yval), color(cval){};
  int equal (colorPoint& cp)
  {cout << "calling equal of colorPoint.\n";
  return x == cp.x && y == cp.y /*&& color == cp.color*/;
  };
  int equal (point& cp)
 {cout << "calling equal of colorPoint with point.\n";
  return x == cp.x && y == cp.y;};
};
```

```
...
point p1 (2.0, 6.0);
colorPoint cp1 (2.0, 6.0, 3);
...
p1 = cp1;
```

- Which function will be called with the calls p1.equal (p1), p1.equal(cp1) and according to what rule (polymorphism or overloading)?
- If we create a point* pp and assign cp1 to it, which functions will be called with the calls p1.equal (p1), p1.equal(cp1) and according to what rule (polymorphism or overloading)?
- Explain the differences between the answers to the first and second question.

3. In Section 6.1.4 we suggested that a language that does not support dynamic binding may use case statements or function pointers to achieve the same result. Explain how this can be done and discuss the drawbacks of such solutions.
4. In Exercise 2, what is the type of p1 after the assignment p1 = cp1? What is the type of pp after the assignment pp = cp1? Explain the differences between the object held by p1 and the object pointed to by pp. What about the computational model of the C++ makes this difference necessary? Propose a change to the language that would remove this difference. What is the cost of your proposal?
5. Let us define a relation s<.t to mean that s is a subtype of t. We want to define a subtype relation for function signatures. First we represent a signature as t1 -> t2 indicating a function that takes an argument of type t1 and produces a result of type t2. How would you state the covariance and contravariant requirements using the subtype relation. That is, complete the equivalence relation below: s1 -> s2 <. t1 -> t2 iff s1 ? t1 && s2 ? t2. How would you describe the Eiffel rule?
6. In C++, it is possible for a derived class to hide the public members of its base class from its clients. Give an argument to show that this is not a good design practice. (hint: substitutability)
7. (Multimethods) Assume the following class definitions:

```
class shape{
    public:
        virtual void move();
        ...
};
class circle: public shape{
    public
        move () {...}
        ...
};
class square: public shape{
    public
        move () {...}
        ...
};
```

The dynamic binding associated with virtual functions makes it possible to call s.move()

for a shape s, which may be a circle or a square at runtime, and the appropriate move function will be called depending on the runtime type of s. As we have seen, this is one of the essential features of object-oriented programming languages. Now, suppose that we also want to define a sort of conversion function that will reshape one shapes into another, for example a square into a circle or a circle into a square. In particular, we want to define a virtual function in shape:

```
        virtual shape reshape(shape&);
```

And in square we want to define:

```
        rectangle shape reshape(circle&);
```

And in circle we want to define:

```
        circle reshape(rectangle&);
```

What is the difference between reshape and move? Can a call s.reshape(s) be statically checked for consistency? Can the identity of the function that needs to be called be determined at compile-time? If not, what is necessary to be able to call the right function at runtime? Does the usual virtual function table mechanism be used?

8. Sometimes inheritance is used improperly. For example, consider defining an automobile class. We have an existing class window. Since automobiles have windows, we have two options: we can derive automobile from window, assuring that the automobile will have a window or we can define the class automobile and use window as a member of the class. Why is the second solution better? Explain the proper use of inheritance in terms of the is-a relation.
9. When a class b is derived from a class a, class b may add new properties, or it may redefine properties defined in a. How do addition of properties affect the subtyping relation between parent and child? How do redefinitions affect the relationship?
10. In Chapter 3, we defined two classes POINT and NON_AXIAL_POINT. Is NON_AXIAL_POINT a subtype of POINT?
11. Some languages support the concept of multiple inheritance, that is a new class may be derived based on more than one parent classes. For example, we may define class displayable_rectangle inheriting from both polygon and displayable classes. This can be done in C++:

```
class displayable_rectangle: public polygon, public displayable {...}
```

Multiple inheritance is supported in both C++ and Eiffel. It is conceptually appealing but it does exacerbate the maintenance problems associated with tightly-related inheritance hierarchies.

12. From an implementation point of view, multiple inheritance introduces two issues:

- If an operation is defined in more than one of the base classes, which one is inherited by the derived class?
- What does an object of a derived class look to an operation of the base class. For example, a displayable_rectangle passed to a polygon operation should appear as a polygon object and passed to an operation of displayable should look like a displayable object.

Find out and explain how C++ and Eiffel handle the first issue. Devise an implementation to solve the second issue.

13. Section 6.3.2.2 introduced the class-wide types of Ada 95. Consider the following code:

```
Poly: Object'Class;
Mono: Circle;
...
```

```
Poly := Mono;
...
Mono := Poly;
```

Which of the two assignment statements are statically type-safe? Which one may raise a runtime exception? Based on what you know from this chapter, is Ada able to detect such an exception?

Assume the classes point and colorPoint of Section 6.2.2.2. Given the procedure below:

```
void problem(point p)
{
    colorPoint n = new colorPoint(...);
if p.equal(n) {...}
}
```

can we call the procedure with a colorPoint parameter? with a point parameter? Will the class definitions pass type checking in Eiffel? Will they pass type checking in C++? Will a call to procedure problem cause a runtime error in C++? Will in cause a runtime error in Eiffel? Can we modify the class definitions to avoid a runtime error?

Functional programming languages

CHAPTER 7

So far in this book we have been concerned primarily with languages which may be described as statement-oriented or imperative. These languages are affected strongly by the architecture of conventional computers. Functional programming languages take as their basis not the underlying computing engine, but rather the theory of mathematical functions. Rather than efficient execution, these languages are motivated by the questions: what is the proper unit of program decomposition and how can a language best support program composition from independent components.

We have seen that procedural languages use procedures as the unit of program decomposition. Procedures generally use side effects on global data structures to communicate with other procedures. Abstract data types attempt to modularize a program by packaging data structures and operations together in order to limit the scope of side-effects. Functional programs reduce the impact of side-effects further, or even eliminate them entirely, by relying on mathematical functions, which operate on *values* and produce *values* and have no side-effects.

We start in the next section by describing the main elements of imperative programming. These elements help illustrate the main differences with functional programming. To contrast these differences further, we will then compare mathematical functions with programming langauge functions. In Section 7.3.2 we present Lambda calculus as a model for function definition, evaluation and composition. We then look at ML and LISP as examples of functional programming languages. Early functional languages, starting from LISP, were dynamically typed and scoped. Scheme is a dialect of |LISP that introduces static scoping into the language. Later functional languages, such as ML, not only include static scoping but also static typing. Many functional languages, include both Scheme and ML, have also added a module construct to address programming in the large.

7.1 Characteristics of imperative languages

Imperative languages are characterized by three concepts: variables, assignment, and sequencing. The state of an imperative program is maintained in program variables. These variables are associated with memory locations and hold values and have addresses. We may access the value of a variable either through its name (directly) or through its address (indirectly). The value of a variable is modified using an assignment statement. The assignment statement introduces an order-dependency into the program: the value of a variable is different before and after an assignment statement. Therefore, the meaning (effect) of a program depends on the order in which the statements are written and executed. While this is natural if we think of a program being executed by a computer with a program counter, it is quite unnatural if we think of mathematical functions. In mathematics, variables are bound to values and once bound, they do not change value. Therefore, the value of a function does not depend on the order of execution. Indeed, a mathematical function defines a mapping from a value domain to a value range. It can be viewed as a set of ordered pairs which relate each element in the domain uniquely with a corresponding element in the range. Imperative programming language functions, on the other hand, are described as algorithms which specify how to compute the range value from a domain value with a prescribed series of steps.

One final characteristic of imperative languages is that repetition—loops—are used extensively to compute desired values. Loops are used to scan through a sequence of memory locations such as arrays, or to accumulate a value in a given variable. In contrast, in mathematical functions, values are computed using function application. Recursion is used in place of iteration. Function composition is used to build more powerful functions.

Because of their characteristics, imperative languages have been given labels such as state-based and assignment-oriented. In contrast, functional languages have been called value-based and applicative.

7.2 Mathematical and programming functions

A function is a rule for mapping (or associating) members of one set (the domain set) to those of another (the range set). For example, the function “square” might map elements of the set of integer numbers to the set of integer numbers. A function definition specifies the domain, the range, and the mapping rule for the function. For example, the function definition

square(x) ≡ x*x, x is an integer number

defines the function named “square” as the mapping from integer numbers to integer numbers. We use the symbol “≡” for “is equivalent to.” In this definition, *x* is a *parameter*. It stands for *any* member of the domain set.

Once a function has been defined, it can be *applied* to a particular element of the domain set: the application yields (or *results* in, or *returns*) the associated element in the range set. At application time, a particular element of the domain set is specified. This element, called the *argument*, replaces the parameter in the definition. The replacement is purely textual. If the definition contains any applications, they are applied in the same way until we are left with an expression that can be evaluated to yield the result of the original application. The application

square (2)

results in the value 4 according to the definition of the function square.

The parameter *x* is a mathematical variable, which is not the same as a programming variable. In the function definition, x stands for any member of the domain set. In the application, it is given a specific value—*one* value. Its value never changes thereafter. This is in contrast to a programming variable which takes on different values during the course of program execution.

New functions may be created by combining other functions. The most common form of combining functions in mathematics is function composition. If a function F is defined as the composition of two functions G and H, written as

F ≡ G o H,

applying F is defined to be equivalent to applying H and then applying G to the result.

In conventional programming languages, a function is defined proceduraly: the rule for mapping a value of the domain set to the range set is stated in terms of a number of steps that need to be “executed” in certain order specified by the control structure. Mathematical functions, on the other hand, are defined applicatively—the mapping rule is defined in terms of combinations or applications of other functions.

Many mathematical functions are defined recursively, that is, the definition of the function contains an application of the function itself. For example, the standard mathematical definition of factorial is:

```
n! ≡ if n = 0 then 1 else n * (n - 1)!
```

As another example, we may formulate a (recursive) function to determine if a number is a prime:

```
prime (n) ≡ if n = 2 then true else p (n, n div 2)
```

where function p is defined as:

```
p (n, i) ≡  if (n mod i) = 0 then false
            else if i = 2 then true
                else p (n, i - 1)
```

Notice how the recursive call to p(n, i-1) takes the place of the next iteration of a loop in an imperative program. Recursion is a powerful problem-solving technique. It is a used heavily when programming with functions.

7.3 Principles of functional programming

A functional programming language has three primary components:

1. A set of data objects. Traditionally, functional programming languages have provided a single high level data structuring mechanisms such as a list or an array.
2. A set of built-in functions. Typically, there are a number of functions for manipulating the basic data objects. For example, LISP and ML provide a number of functions for building and accessing lists.
3. A set of functional forms (also called high-order functions) for building new functions. A common example is function composition. Another common example is function reduction. *Reduce* applies a binary function across successive elements of a sequence. For example, reducing + over an array yields the sum of the elements of the array and reducing * over the elements of an array yields the product of the elements of the array. In APL, /

is the reduction functional form (called operator in APL) and it takes one operation as argument. The plus reduction can be accomplished by /+ and the multiplication reduction by /*. The use of functional forms is what distinguishes a functional program. Functional forms support the combination of functions without the use of control structures such asses iteration conditional statements.

The execution of functional programs is based on two fundamental mechanisms: binding and application. *Binding* is used to associate values with names. Both data and functions may be used as values. Function application is used to compute new values.

In this section we will first review these basic elements of functional programs using the syntax of ML. We will then introduce Lambda calculus, a simple calculus that can be used to model the behavior of functions by defining the semantics of binding and application precisely.

7.3.1 Values, bindings, and functions

As we said, functional programs deal primarily with values, rather than variables. Indeed, variables denote values. For example 3, and “a” are two constant values. A and B are two variables that may be *bound* to some values. In ML we may bind values to variables using the binding operator =. For example

```
val A = 3;
val B = "a";
```

The ML system maintains an environment that contains all the bindings that the program creates. A new binding for a variable may hide a previous binding but does not replace it. Function calls also create new bindings because the value of the actual parameter is bound to the name of the formal parameter.

Values need not be just simple data values as in traditional languages. We may also define values that are functions and bind such values to names:

```
val sq = fn(x:int) => x*x;
sq 3;
```

will first bind the variable sq to a function value and then apply it to 3 and print 9. We may define functions also in the more traditional way:

```
fun square (n:int) = n * n;
```

We may also keep a function anonymous and just apply it without binding it to a name:

```
(fn(x:int) = x*x) 2;
```

We may of course use functions in expressions:

```
2 * sq (A);
```

will print the value of the expression $2A^2$.

The role of iteration in imperative languages is played by recursion in functional languages. For example, Figure 84 shows the function factorial written in C++ using iteration and ML using recursion.

```
int fact(int n)
{   int i=1;
    assert (n>0);
     {for (int j=n;  j>1; ++j)
         i= i*n;
    return i;
}
```

```
fun fact(n) =
        if n = 0 then 1
        else n*fact(n-1);
```

FIGURE 84. Definition of factorial in C++ and ML

We saw in Chapter 4 that functions in ML may also be written using patterns and case analysis. The factorial program in the figure may be written as composed of two cases, when the argument is 0 and when it is not:

```
fun fact(n) =
         fact(0) = 1
    |    n*fact(n-1);
```

In addition to function definition, functional languages provide functional forms to build new functions from existing functions. We have already mentioned mathematical function composition operator o as such a higher order function. It allows us to compose two functions F and G and produce a new function FoG. Functional programming languages provide both built-in higher order functions and allow the programmer to define new ones. Most languages provide function composition and reduction as built-in functional forms.

7.3.2 Lambda calculus: a model of computation by functions

In the previous section, we have seen the essential elements of programming with functions: binding, function definition, and function application. As opposed to an imperative language in which the semantics of a program may be understood by following the sequence of steps specified by the program, the semantics of a functional program may be understood in terms of the computation implied by function applications. Lambda calculus is a surprisingly simple calculus that models the computational aspects of functions. Studying lambda calculus helps us understand the elements of functional programming and the underlying semantics of functional programming languages independently of the syntactic details of a particular programming language.

Lambda *expressions* represent values in the lambda calculus. There are only three kinds of expressions:

1. An expression may be a single identifier such as x.
2. An expression may be a function definition. Such and expression has the form λ x.e which stands for the expression e with x designated as a *bound* variable. The expression e represents the body of the function and x the paramter. The expression e may contain any of the three forms of lambda expressions. Our familiar square function may be written as λ x.x*x.
3. An expression may be a function application. A function application has the form e1 e2 which stands for expression e1 applied to expression e2. For example, our square function may be applied to the value 2 in this way: (λ x.x*x) 2. Informally, the result of an application can be derived by replacing the parameter and evaluating the resulting expression.

```
(λ x.x*x) 2=
2*2 =
4
```

In a function definition, the parameters following the "λ" and before the "." are called *bound* variables. When the lambda expression is applied, the occurrences of these variables in the expression following the “.” are replaced by the arguments. Variables in the definition that are not bound are called *free* variables. Bound variables are like local variables, and free variables are like nonlocal variables that will be bound at an outer level.

Lambda calculus capture the behavior of functions with a set of rules for rewriting lambda expressions. The rewriting of an expression models a step in the computation of a function. To apply a function to an argument, we rewrite the function definition, replacing occurrences of the bound variable by the argument to which the function is being applied.

Thus, to define the semantics of function application, we first define the concept of *substitution*. Substitution is used to replace all occurrences of an identifier with an expression. This is useful to bind parameters to arguments and to avoid name conflicts that arise if the same name appears in both the expression being applied and the argument expression to which it is being applied. We will use the notation [e/x]y to stand for “substitute e for x in y.” We will refer to variables as xi. Two variables xi and xj are the same if i=j. They are not the same if i=/=j. We can define substitution precisely with the following three rules, based on the form of the expression y:

1. If the expression is a single variable:
   [e/xi]xj= e, if i = j
   = xj, if i=/= j

2. If the expression is a function application, we first do the substitution both in the function definition and in the argument, and then we apply the resulting function to the resulting argument expression:

   [e1/x](e2 e3)= ([e1/x]e2)([e1/x]e3)

   In doing the substitutions before the function application, we have to be careful not to create any bindings that did not exist before or invalidate any previous bindings. This means that we may not rename a variable and make it bound if it were free before or make it free if it were bound before. The next rule takes care of these situations.

3. If the expression is a function definition, we must do the substitution carefully:

   [e1/xi](λ xj.e2)= λ xj.e2, if i=j

   = λ xj.[e1/xi]e2, if i=/=j and xj is not free in e1 (otherwise, it would become newly bound)

   = λ xk.[e1/xi]([xk/xj]e2), otherwise, where k=/=i, k=/=j, and xk is not free in either e1 or e2

The last rule serves to rename all occurrences of a variable by another name to avoid name clashes.

Using the substitution rules above, we can define the semantics of functional computations in terms of rewrite rules. That is, we define the result of a function application in terms of rewriting the definition of the function, replacing the bound variables of the function with corresponding arguments. The following three rewrite rules define the concept of function evaluation:

1. Renaming: λ xi.e <=> λ xj.[xj/xi]e, where xj is not free in e. The renaming rules says that we can replace all occurrences of a bound variable with another name without affecting the meaning of the expression. In other words, a function is abstracted over the bound variables.

2. Application: (λ x.e1)e2 <=> [e2/x]e1. This rule says function application means replacing the bound variable with the argument of the application.

3. λ x.(e x) <=> e, if x is not free in e.

The last rule says that free variables are the only way for an environment to change the effect of a function. That is, a function is a self-contained entity with the parameters being its only interface.

These rules may be used in the forward direction to “reduce” a lambda expression. In fact, any lambda expression may be reduced using these three rules until no further reduction is possible. An expression that may no longer be reduced is said to be in *normal form*.

For example, the following shows the application of the three rules to reach a normal form for the original expression.

```
(λ x.(λ y.x+y) z) (λ y.y*y) =
(λ x.x+z)) (λ y.y*y) =
(λ y.y*y)+z
```

The simple semantics that we have described here capture the semantics of binding, function definition and function application, which are the primitive elements of functional programming languages. The clear semantics of functional languages is due to the fact that the semantics of function definition and application can be defined with these three simple rules.

One of the interesting aspects of lambda calculus is that we can define the semantics of functions using only one-argument functions. To deal with functions of more than one argument, a list of arguments is passed to the function f which applies to the first argument and produces as result a function that is then applied to the second argument, and so on. This technique is called *currying* and a function that works this way is called a *curried function*.

For example, consider a function to sum its two arguments. We could write it as λ x,y.x+y. This is a function that requires two arguments. But we could also write it as λ x.λ y.x+y. This new function is written as the composition of two functions, each requiring one parameter. Let us apply it to arguments 2 3:

```
(λ x.λ y.x+y) 2 3 =
((λ x.λ y.x+y) 2) 3 =
(λ y.2+y) 3 =
2+3 =
5
```

This is a common technique in functional programming to deal with a variable number of arguments. Each argument is handled in sequence through one function application. Each function application replaces one of the bound variables, resulting in a “partially evaluated” function that may be applied again to the next argument. Symbolically, (f x y z) is considered to be (((f(x)) y) z). Indeed, in ML, the function application f(x,y,z) may also be written in the curried form f x y z.

7.4 Representative functional languages

In this section, we examine pure LISP, APL, and ML. LISP was the first functional programming language. The LISP family of languages is large and popular. LISP is a highly dynamic language, adopting dynamic scoping and dynamic typing, and promoting the use of dynamic data structures. Indeed garbage collection was invented to deal with LISP’s heavy demands on dynamic memory allocation. One of the most popular descendants of LISP is Scheme, which adopts static scope rules.

APL in an expression-oriented language. Because of the value-orientation of expressions, it has many functional features. As opposed to LISP’s lists, the APL data structuring mechanism is the multidimensional array.

ML is one of the recent members of the family of functional programming languages that attempt to introduce a strong type system into functional programming. We will examine ML in more detail because of its interesting type structure. In the next section, we look at C++ to see how the facilities of a conventional programming language may be used to implement functional programming techniques.

Most functional programming languages are *interactive*: they are supported by an interactive programming system. The system supports the immediate execution of user commands. This is in line with the value-orientation of these languages. That is, the user types in a command and the system immediately responds with the resulting value of the command.

7.4.1 ML

ML starts with a functional programming foundation but adds a number of features found to be useful in the more conventional languages. In particular, it adopts polymorphism to support the writing of generic components; it adopts strong typing to promote more reliable programs; it uses type inference to free the programmer from having to make type declarations; it adds a module facility to support programming in the large. The most notable contribution of ML has been in the area of type systems. The combination of polymorphism and strong typing is achieved by a “type inference” mechanism used by the ML interpreter to infer the static type of each value from its context.

7.4.1.1 Bindings, values, and types

We have seen that establishing a binding between a name and a value is an essential concept in functional programming. We have seen examples of how ML establishes bindings in Section 7.3.1. Every value in ML has an associated type. For example, the value 3 has type int and the value fn(x:int) =>x*x has type int->int which is the signature of the functional value being defined.

We may also establish new scoping levels and establish local bindings within these scoping levels. These bindings are established using let expressions:

```
let x = 5
in 2*x*x;
```

evaluates to 50. The name x is bound only in the expression in the let expression. There is another similar construct for defining bindings local to a series of other declarations:

```
local
     x = 5
in
    val sq  = x*x
    val cube = x*x*x
end;
```

Such constructs may be nested, allowing nested scoping levels. ML is statically scoped. Therefore, each occurrence of a name may be bound statically to its declaration.

7.4.1.2 Functions in ML

In ML, we can define a function without giving it a name just as a lambda expression. For example, as we have seen:

```
fn(x, y):int => x*y
```

is a value that is a function that multiplies its two arguments. It is the same as the lambda expression λx,y.x*y. We may pass this value to another function as argument, or assign it to a name:

```
val intmultiply = fn(x, y):int => x*y;
```

The type of this function is fn:int*int->int.

We have seen that functions are often defined by considering the cases of the input arguments. For example, we can find the length of a list by considering the case when the list is empty and when it is not:

```
fun length(nil) = 0
    |    length([_::x]) = 1+length(x);
```

The two cases are separated by a vertical bar. In the second case, the underscore indicates that we do not care about the value of the head of the list. The only important thing is that there exists a head, whose value we will discard.

We may also use functions as values of arguments. For example, we may define a higher-order function compose for function composition:

```
fun compose (f, g)(x) = f(g(x));
```

The type of compose is ('a->'b * 'c->'a)->('c->'a). ML provides some built-in functional forms as well. The classic one is map which takes two arguments, a function and a list. It applies the function to each element of the list and forms the results of the applications into a list. For example, the result of:

```
val x= map (length,[[], [1,2,3],[3]);
```

is [0,3,1].

For example, the reduce function takes a function F of two arguments and a nonempty list [a1,a2,...,an] as arguments and produces as result the value F(a1, F(...F(an-1,F(an))). The basis case F(x) applied to a singleton list is defined to be the singleton element. So, the result of

```
val x = reduce(+, [1,2,3,4]);
```

is 10. Some systems provide the function reduce as a built-in function. If it is not available, we can easily define it for nonempty lists:

```
fun reduce(F, [x]) = x
| reduce(F,[x::xs]) = F(x,reduce(F,xs));
```

Another class of such high order functions is filters that apply a predicate function to elements of a list and return only those elements that satisfy the predicate. We can easily write such functions in ML.

A useful functional programming technique is to partially evaluate a function by binding some of its arguments. The result is still a function that may be applied to the remaining arguments. A function some of whose arguments have been bound is called a *closure*. As an example, consider a function TranslateWord that takes two arguments: a dictionary to use for translation and the word to translate. The function looks up the word in the dictionary and returns the translation found in the dictionary. We might define closures of this function by binding the dictionary argument to different language dictionaries and producing special translator functions such as ItalianEnglish, ItalianGerman, and EnglishGerman. These new functions are single-argument functions because the dictionary argument has already been bound.

Curried functions may also be used in ML. For example, we can define the function to multiply two integers in curried form:

```
fun times (x:int) (y:int) = x*y;
```

The signature of this function is fn: int-> (int->int). We can build a new function, say multby5, by binding one of the arguments of times:

```
fun multby5(x) = times(5)(x);
```

7.4.1.3 List structure and operations

The *list* is the major data structuring mechanism of ML; it is used to build a finite sequence of values of the same type. Square brackets are used to build lists: [2, 3, 4], ["a", "b", "c"], [true, false]. The empty list is shown as [] or nil. A list has a recursive structure: it is either nil or it consists of an element followed by another list. The first element of a nonempty list is known as its *head* and the rest is known as its *tail*.

There are many built-in list operators. The two operators hd and tl return the head and tail of a list, respectively. So: hd([1,2,3]) is 1 and tl([1,2,3]) is [2,3]. Of course, hd and tl are polymorphic. The construction operator :: takes a value and a list of the same type of values and returns a new list: 1::[2,3] returns [1,2,3]. We can combine two lists by concatenation: [1,2]@[3] is [1,2,3].

Let us look at some functions that work with lists. First, recall from Chapter 4 the function to reverse a list:

```
fun reverse(L) = reverse([]) = []
    |   reverse(x::xs) = reverse(xs) @ [x]
```

Let us write a function to sort a list of integers using insertion sort:

```
fun sort(L) = sort([]) = []
    |   sort(x::xs) = insert (x,xs)
fun insert(x, L) = insert (x,[]) = [x]
    |   insert (x:int, y::ys) =
            if x < y then x::y::ys
                    else y::insert(x,ys);
```

The recursive structure of lists makes them suitable for manipulation by recursive functions. For this reason, functional languages usually use lists or other recursive structures as a basic data structuring mechanism in the language.

7.4.1.4 Type system

Unlike LISP and APL, ML adopts a strong type system. Indeed, it has an innovative and interesting type system. It starts with a conventional set of built-in primitive types: bool, int, real, and string. Strings are finite sequences of characters. There is a special type called unit which has a single value denoted as (). It can be used to indicate the type of a function that takes no arguments.

There are several built-in type constructors: lists, records, tuples, and functions. A list is used to build a finite sequence of values of a single type. The type of a list of T values is written as T list. For example, [1,2,3] is an int list and ["a","b","cdef"] is a string list. An empty list is written as nil or []. The type of an empty list is ’t list. ’t is a *type variable* which stands for any type. The empty list is a polymorphic object because it is not specifically an empty int list or an empty bool list. The expression ’t list is an example of a polymorphic type expression (called *polytype* in ML).

Tuples are used to build Cartesian products of values of different types. For example, (5, 6) is of type int*int and (true, "fact", 67) is of type bool*string*int. We can of course use lists as elements of tuples: (true, []) is of type bool* (t’ list).

Records are similar to Pascal records: they are constructed from named fields. For example, {name="Darius", id=56789} is of type {name: string, id: int}. Tuples are special cases of records in which fields are labeled by integers starting with 1. The equality operation is defined for records based on comparing corresponding fields, that is, the fields with the same names.

As we have seen before, a function has a—possibly polymorphic—signature, which is the type of the function. For example, the built-in predicate null which determines whether its argument is the empty list is of type ’t list -> bool. Null is a polymorphic function.

In addition to the built-in type constructors, the programmer may define new type constructors, that is, define new types. There are three ways to do this: type abbreviation, datatype definition, and abstract data type definition. The simplest way to define a new type is to bind a type name to a type expression. This is simply an abbreviation mechanism to be able to use a name rather than the type expression. Some examples are:

```
type intpair = int * int;
type ’a pair = ’a * ’a;
type boolpair = bool pair;
```

In the second line, we have defined a new polymorphic type called pair which is based on a type ’a. The type pair forms a Cartesian product of two values of type ’a. We have used this type in the third line to define a new monomorphic type.

The second way to define a new type is to specify how to construct values of the new type. For example, similar to Pascal enumeration types, we can define the new type color as:

```
datatype color = red | white | blue;
```

This definition defines a new type color and three value constructors for it. These value constructors are simple because they do not take any parameters. In general, we may have more complex value constructors. In any case, the name color has now been defined as a new type. We may use the new constructors to build new values. For example, [red, blue] is of type color list.

Value constructors may take parameters. We might define a new type money as:

```
datatype money = nomoney | coin of int | note of int;
```

based on three value constructors: nomoney, coin, and note. The first constructor takes no arguments while the latter two are monadic constructors. Some values of type money are: nomoney, coin(5), note(7).

We can also define recursive type constructors. For example, we might define a binary tree as:

```
datatype ’t Btree = null | Node of ’t * ’t Btree * ’t Btree;
```

We have defined a Btree of a particular type ’t as being either null or consisting of a node which has three components. One component is simply a value of type ’t. The other two components are each a ’t Btree themselves.

To show the power of value constructors in defining new types, next we define a stack in terms of the operation push that can be used to construct it.

```
datatype ’a stack = empty | push of ’t * ’t stack;
```

This definition says that the following are example values of a stack data type:

```
empty
push(2, empty)
push(2,(push(3,push(4,empty))))
```

Notice how we have used push as a constructor rather than an operation defined on stacks. Given this definition of ’t stack, we can define functions such as pop, and top on the new data type stack. For example, we might define pop and top as shown here:

```
fun pop (empty) = raise error
|     pop(push(x,xs)) = x;
fun top (empty) = raise error
|    top (push(x, xs)) = x;
```

This stack does not hide its representation from its clients. If we want to do that, we must use data abstraction, which is the third way to define a new type in ML An abstype defines a new type and hides its concrete representation. For example, Figure 85 shows a stack abstract data type. This type defines the operations create, push, pop, top, and length for the abstract type stack. From the outside, the representation of stack, which is a list, is not accessible. The stack is only accessible through its exported operations, which are the functions specified after the with expression.

```
abstype 'a stack = stack of 'a list
with    val create = [];
        fun push(x, stack xs) = stack (x::xs);
        fun pop  (stack nil) = raise poperror
        |   pop (stack [e]) = []
        |   pop (stack [x::xs]) = stack [xs];
        fun top(stack nil) = raise toperror
        |   top(stack [x::xs]) = x;
        fun lenght(stack []) = 0
        |   length (stack [x::xs]) = length (stack [xs]) + 1;
end;
```

FIGURE 85. An abstract data type stack in ML

We can now use the operations of this type. For example push(7, create) will produce an int stack and push("how",(push ("now", create)) will produce a string stack of two elements.

We can see that ML has a rich and consistent type system. Every value has a single type but the type may be polymorphic (called *polytype* in ML). The use of type variables supports the creation of polymorphic types. These types may then be used to write polymorphic functions.

7.4.1.5 Type inference

Earlier functional languages used dynamic typing. As we saw in Chapter 3, this means that variables are not required to be declared: a variable simply takes on the type of the value that is assigned to it. Both LISP and APL are based on such dynamic type systems. ML tries to achieve the advantages of a strongly-typed language without burdening the user with the requirement for type declarations. A name always has to be declared before it is used but its type is not required in the declaration. The ML system (compiler or interpreter) infers the types of variables based on their use. Because the language is strongly-typed, each value produced by the program has to be assigned a particular type. If the system cannot infer the proper type for a value, an error message is produced at compile-time. The programmer must specify the type in such situations. For example, an expression x>y does not contain enough information to infer whether x an y are integers or reals. If a function of x and y contains such an expression and no other indication of the types of x and y, then the program must declare the type of either x or y. Only one type declaration is necessary because the other one is constrained to have the same type. This was the reason that in the definition of the function square in Section 7.3.1, we had to declare the type of the argument as being int, while in the definition of the factorial function we did not have to declare the type of the argument. The ML system uses the expression "if x = 0" in the definition of factorial to infer the type of x to be int.

ML combines type inference with extensive support for polymorphism. Consider a simple polymorphic identity function which returns its argument as its result. This function does not care what the type of its argument is. If we were to write such a function in C or Pascal, we would have to write a function for each type that we expect to use. In ML, we only need one function:

```
fun id(x) = x;
```

We could write a similar function in C++ using templates, where the type of the parameter is a template parameter (Exercise 10). This function may be applied to a value of any type and it will return the same value. Of course, we may apply id to a function also, for example to itself: id (id) returns id.

The signature of this function is id: ’a -> ’a, that is, it maps from a domain of some type to a range of the *same* type. From its signature, we can see clearly that the function is polymorphic. Such a function can be type-checked statically even though we do not know what type will be passed at the time the function is applied. Each application of a function uses the type of the argument to produce a value of the appropriate type as result. For example id(3) returns an integer and id(id) returns a value of type ’a->’a.

Some built-in functions such as list handling functions are naturally polymorphic. For example, hd has signature hd: t’list->’t. Rather than requiring the programmer to declare the types of variables, the ML system uses such information to do its static type checking.

As we have seen, some operators used in function definitions limit the ability of the system to do type inferencing. For example, the function max defined as:

```
fun max (x:int, y) = if x > y then x else y;
```

requires the declaration of the x because the system cannot tell whether, for example, the signature of max should be (int*int)->bool or (real*real)->bool. The signature ('t*'t)->bool is not correct either because not any type is acceptable. Only types that support the > operator are acceptable. We can use the signature facility of ML to specify such type requirements and if we do so, the system can make use of them.

One particularly important class of types is the equality type, denoted by "a. These are types that support the equality and inequality operators = and <>. If we define a function eq:

```
fun eq(x, y) = x=y;
```

the type of the function is: eq: "a*"a-> bool. That is, eq is a polymorphic function that requires two equality types and returns a boolean. Of course, a type expression including a type variable "t has stricter requirements than one having only 't variables.

7.4.1.6 Modules

We have already seen an example of an ML module in Chapter 5. ML modules are separately compilable units. There are three major building blocks:

1. A *structure* is the encapsulation unit. A structure contains a collection of definitions of types, datatypes, functions, and exceptions.
2. A *signature* is a type for a structure. A signature is a collection of type information about some of the elements of a structure: those we wish to export. We may associate more than one signature with a structure.
3. A functor is an operation that combines one or more structures to form a new structure.

As an example of a module, Figure 86 shows the stack definition of Figure 85 encapsulated in a structure. The figure defines a polymorphic stack implemented as a list.

```
structure Stack = struct
     exception Empty;

     val create = [];
     fun push(x, stack xs) = stack (x::xs);
     fun pop  (stack nil) = raise Empty;
     |     pop (stack [e]) = []
     |     pop (stack [x::xs]) = stack [xs];
     fun top(stack nil) = raise Empty;
     |     top(stack [x::xs]) = x;
     fun lenght(stack []) = 0
     |     length (stack [x::xs]) = length (stack [xs]) + 1;
end;
```

FIGURE 86. A stack module in ML

We can use signatures to specialize a structure. For example, we can make a stack of strings out of the stack of Figure 86. Not only that, we can also restrict the operations that are exported using the signature mechanism. Figure 87 is a signature for Figure 86 which specifies a stack of strings which supports create, push, pop, top but not length. A signature may be viewed as a specification for a module. Several specifications may be associated with the same module and the same signature may be used with different modules. We can use the signature we have just defined to create a new structure:

```
signature stringStack = sig
     exception Empty;
     val create = string list;
     val push: string * string list -> string list;
     val pop:  srting list -> sring list;
     val top:  string list -> string;
end;
```

FIGURE 87. A signature for string stack module that hides length

```
structure SS:stringStack = Stack;
```

The new structure SS is built from Stack and has the signature stringStack. Structure elements are accessed either using dot notation (e.g. SS.push("now",SS.create)) or by "opening" the structure and gaining access to all the elements:

```
open SS;
```

```
push("now", create);
```

7.4.2 LISP

The original LISP introduced by John McCarthy in 1960, known as pure LISP, is a completely functional language. It introduced many new programming language concepts, including the uniform treatment of programs as data, conditional expressions, garbage collection, and interactive program execution. LISP used both dynamic typing and dynamic scoping. Later versions of LISP, including Scheme, have decided in favor of static scoping. Common Lisp is an attempt to merge the many different dialects of LISP into a single language. In this section, we take a brief look at the LISP family of languages.

7.4.2.1 Data objects

LISP was invented for artificial intelligence applications. It is referred to as a language for symbolic processing. It deals with symbols. Values are represented by symbolic expressions (called *S-expressions*). An expression is either an *atom* or a *list*. An atom is a string of characters (letters, digits, and others). The following are atoms:

```
A
AUSTRIA
68000
```

A list is a sequence of atoms or lists, separated by space and bracketed by parentheses. The following are lists:

```
(FOOD VEGETABLES DRINKS)
((MEAT CHICKEN) (BROCCOLI POTATOES TOMATOES) WATER)
(UNC TRW SYNAPSE RIDGE HP TUV)
```

The empty list “()”, also called NIL. The truth value false is represented as () and true as T. The list is the only mechanism for structuring and encoding information in pure LISP. Other dialects have introduced most standard data structuring mechanisms such as arrays and records.

A symbol (as atom) is either a number or a name. A number represents a value directly. A name represents a value bound to the name.

There are different ways to bind a value to a name: SET binds a value globally and LET binds it locally. (SET X (A B C)) binds A to the list value (A B C).

SET shows an example of a function application. A function application is written as a list: the first element of the list is the function name and the rest of the elements are parameters to the function. Thus, functions and data have the same representation. Representing the function application in this way implies the use of prefix notation, as opposed to infix of other languages: LISP uses (PLUS A B) instead of A+B.

7.4.2.2 Functions

There are very few primitive functions provided in pure LISP. Existing LISP systems provide many functions in libraries. It is not unusual Such libraries may contain as many as 1000 functions.

QUOTE is the identity function. It returns its (single) argument as its value. This function is needed because a name represents a value stored in a location. To refer to the value, we use the name itself; to refer to the name, we use the identity function. Many versions of LISP use 'A instead of the verbose QUOTE A. We will follow this scheme.

The QUOTE function allows its argument to be treated as a constant. Thus, 'A in LISP is analogous to "A" in conventional languages.

Examples

```
    (QUOTE A) = 'A = A
    (QUOTE (A B C)) = '(A B C) = (A B C)
```

There are several useful functions for list manipulations: CAR and CDR are selection operations, and CONS is a structuring operation. CAR returns the first element of a list (like hd in ML); CDR returns a list containing all elements of a list except the first (like tl in ML); CONS adds an element as the first element of a list (like :: in ML). For example

```
    (CAR '(A B C)) = A
```

The argument needs to be “quoted,” because the rule in LISP is that a function is applied to the *value* of its arguments. In our case the evaluation of the argument yields the list (A B C), which is operated on by CAR. If QUOTE were missing, an attempt would be made to evaluate (A B C), which would result in using A as a function operating on arguments B and C. If A is not a previously defined function, this would result in an error.

Other examples:

```
(CDR '(A B C)) = (B C)
(CDR '(A)) = ( ) = NIL
(CONS 'A '(B C)) = (A B C)
(CONS '(A B C) '(A B C)) = ((A B C) A B C)
```

A few predicates are also available. A true value is denoted by the atom T and a false value by NIL.

ATOM tests its argument to see if it is an atom. NULL, as in ML, returns true if its argument is NIL. EQ compares its two arguments, which must be atoms, for equality.

Examples:

```
(ATOM ('A)) = T
(ATOM ('(A))) = NIL
(EQ ('A) ('A)) = T
(EQ ('A) ('B)) = NIL
```

The function COND serves the purpose of if-then-else expressions. It takes as arguments a number of (predicate, expression) pairs. The expression in the first pair (in left to right order) whose predicate is true is the value of COND.

Example:

```
(COND ((ATOM '(A))) 'B) (T 'A) = A
```

The first condition is false because (A) is not an atom. The second condition is identically true. The COND function, known as the McCarthy conditional, is the major building block for user-defined functions.

Function definition is based on lambda expressions. The function

$\lambda\, x,y.x+y$

is written in LISP as

```
(LAMBDA (X Y) (PLUS X Y))
```

Function application also follows lambda expressions.

```
((LAMBDA (X Y) (PLUS X Y)) 2 3)
```

binds X and Y to 2 and 3, respectively, and applies PLUS yielding 5.

The binding of a name to a function is done by the function DEFINE, which makes the function name known globally. Another function, LABEL, is used if we want to define the function to be known only locally.

```
(DEFINE (ADD (LAMBDA (X Y) (PLUS X Y))))
```

Now, the atom ADD can be used in place of the function above, that is, the atom ADD has a value that is a function.

The ability to name a function is especially useful in defining recursive functions. For example, we can define a function REVERSE to reverse the elements of a list:

```
(DEFINE (REVERSE (LAMBDA (L)
          (REV NIL L))))
(DEFINE (REV (LAMBDA (OUT IN)
          (COND ((NULL IN) OUT)
          (T (REV (CONS (CAR IN) OUT) (CDR IN))))))
```

The REVERSE function calls a subsidiary function REV that works by picking the first element of a list and calling REV on the rest of the list.

The use of DEFINE is one of two ways in pure LISP that an atom can be bound to a value. The other is through function application, at which time the parameters are bound to the arguments. The conventional assignment is not present.

The variables in pure LISP are more like the variables in mathematics than those in other languages. In particular, variables may not be modified: they can be bound to a value and they retain that value throughout a given scope (i.e., function application); and at any moment, there is only at most one access path to each variable.

7.4.2.3 Functional forms

Function composition was the only technique for combining functions provided by original LISP. For example, the “to_the_fourth” function of Section 7.2 can be defined in LISP as

```
(LAMBDA(X) (SQUARE (SQUARE X)))
```

(We assume SQUARE has been defined.) All current LISP systems, however, offer a functional form, called MAPCAR, which supports the application of a function to every element of a list. For example

```
(MAPCAR TOTHEFOURTH L)
```

raises every element of the list L to the fourth power.

Rather than provide many functional forms, the choice in LISP has been to supply a large number of primitive functions in the library.

7.4.2.4 LISP semantics

One of the most remarkable points about LISP is the simplicity and elegance of its semantics. In less than one page, McCarthy was able to describe the entire semantics of LISP by giving an interpreter for LISP written in LISP itself. The interpreter is called eval.

7.4.3 APL

APL was designed by Kenneth Iverson at Harvard University during the late 1950s and early 1960s. Even though APL relies heavily on the assignment operation, its expressions are highly applicative. We will only look at these features here to see the use of functional features in a statement-oriented language.

7.4.3.1 Objects

The objects supported by APL are scalars, which can be numeric or character, and arrays of any dimension. An array is written as sequence of space-separated elements of the array. Numeric 0 and 1 may be interpreted as boolean values. APL provides a rich set of functions and a few higher-order functions for defining new functions.

The assignment operation (←) is used to bind values to variables. On assignment, the variable takes on the type of the value being assigned to it. For example, in the following, the variable X takes on an integer, a character, and an array, in successive statements:

```
X ← 123;
X ← 'b';
X ← 5 6 7 8 9;
```

The assignment is an operation that produces a value. Therefore, as in C, it may be used in expressions:

```
X ← (Y ← 5 6 7 8 9) × (Z ← 9 9 7 6 5);
W← Y - Z;
```

will set the value of Y to 5 6 7 8 9, Z to 9 9 7 6 5, and W to -4 -3 0 2 4.

7.4.3.2 Functions

In contrast to pure LISP, APL provides a large number of primitive functions (called *operations* in APL terminology). An operation is either monadic (taking one parameter) or dyadic (taking two parameters).

All operations that are applicable to scalars also distribute over arrays. Thus, A x B results in multiplying A and B. If A and B are both scalars, then the result is a scalar. If they are both arrays and of the same size, it is element-by-element multiplication. If one is a scalar and the other an array, the result is the multiplication of every element of the array by the scalar. Anything else is undefined.

The usual arithmetic operations, +, -, ×, ÷, | (residue), and the usual boolean and relational operation, ∧, ∨, ~, <, ≤, =, >, ≥, ≠, are provided. APL uses a number of arithmetic symbols and requires a special keyboard.

There are a number of useful operations for manipulating arrays. The operation "ι" is a "generator" (or constructor, using ML terminology) and can be used to produce a vector of integers. For example, ι5 produces

```
1 2 3 4 5
```

The operation ";" concatenates two arrays. So i4; i5 results in

```
1 2 3 4 1 2 3 4 5
```

The operation "ρ" uses its left operands as dimensions to form an array from the data given as its right operands. For example:

$$2\ 2\ \rho\ 1\ 2\ 3\ 4 \equiv \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

and

The *compress* operation "/" takes two arguments of the same dimensions and

$$2\;3\;\;\rho\;1\;2\;3\;4\;5\;6 \equiv \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

selects elements of its right-hand argument, depending on whether the corresponding left-hand argument is a (boolean) 1 or 0. For example

$1\;0\;0\;1\;/\;\iota 4 \equiv 1\;4$

The left argument may consist of boolean expressions. For example

A<B B<C C<D / X

will pick certain values from X, depending on the comparisons on the left. X must be a three-element vector in this case.

There are many other primitive operations in APL. They can be regarded as mathematical functions because they operate on operands and produce values. User-defined functions are similar to the primitive functions in that they also are either monadic or dyadic (niladic functions correspond to subroutines). They are used in infix notation and thus can be used in expressions, in the same way as built-in functions can.

7.4.3.3 Functional Forms

As we have seen, functional forms give the programmer the ability to construct new functions. APL provides three functional forms (*operator* in the APL terminology) that may be used uniformly to combine the many built-in functions of APL. The functional forms are particularly useful for mathematical manipulations. They functional forms are:

> **a.** The *reduction* operator "/" (same symbol as compress). For example, the sum of the elements of the vector A is given by +/A. Contrasting this with adding the elements of a vector in an imperative language shows that the iteration and step-by-step computation are handled by the functional form. If the right operand of "/" is a matrix, the reduction operation applies to successive rows, that is, if A is the matrix
> 1 2
> 3 4
> then +/A is
> 3
> 7

which is represented as 3 7. In general, a reduction applied to an n-dimensional array results in an (n -1) dimensional array.

b. The *inner product* operator "." takes two primitive binary operations as arguments and produces a binary operation as result. The operands of the resulting operation must be arrays that "conform" in size. For example, if they are matrices, the number of rows of the left operand must be the same as the number of columns of the right operand; the result will be a matrix with as many rows as the left operand and as many columns as the right operand. If f and g are two primitive binary functions, the effect of A f.g B is to apply g, element by element, to the corresponding rows of A and columns of B (i.e., first row of A with first column of B, and so on). This is followed by an f reduction (/f) on the resulting vector.

As an example of the power of inner product in building operations, matrix multiplication can be accomplished by: +.x. This time, the functional form accomplishes the equivalent of two nested loops necessary in a procedural way to do the same job.

c. The third functional form of APL is the outer product "^{o}", which takes one primitive operation as operand and results in a binary. The operation o.f applied to arrays A and B (i.e., A ^{o}.f B) has the effect of applying f between each element of A and every element of B. For example, if A has the value (1 2 3) and B has the value (5 6 7 8), the result of A ^{o}.x B is the matrix

```
5   6   7   8
10  12  14 16
15  18  21 24
```

The effect can be seen as forming a matrix with the rows labeled with elements of A and columns labeled with elements of B. The entries of the matrix are the result of applying the operation to the row and column labels. So the above matrix was derived from

```
x   5   6   7   8
1   5   6   7   8
2   10  12 14  16
3   15  18 21  24
```

The outer product finds many applications in data processing when producing tables of interest rates, taxes, and so on. It has other uses as well. As an example, to find which elements of A occur in B, A^{o}.= B provides a map of boolean values, with a 1 in the position where an element of A equals an element of B.

The many operations of APL and its functional forms support an expression-oriented style of programming that reduces the reliance of loops and computation of intermediate values. Instead of loops, the functional forms are used to build powerful operations that hide the internal details of the functions, which may in fact be accomplished using loops.

7.4.3.4 An APL Program

As an example of the power of functional programming, in this section we will look at how we may derive an APL program to compute prime numbers in the range 1 to N. Despite the limited exposure to APL provided in this section, we have seen enough to construct the desired program.

The style of programming emphasizes exploiting arrays and expressions rather than scalars, assignments, and iteration. Because of the array orientation of APL, we plan to produce a vector of prime numbers. We can start with a vector of numbers in the range 1 to N and compress it, using the compress operator, to remove the nonprimes. In other words, our task is to find the vector of boolean expressions in the following APL program:

```
    vector of boolean expressions /  ιN
```

We can start with the definition of a prime number: a number that is divisible only by 1 and itself. So, for each number in the range of interest, 1 to N, we can (a) divide it by all the numbers in the range and (b) select those which are divisible only by two numbers.

Step (a) can be done with the residue operation and an outer product:

```
    (ιN)°.|(ιN)
```

The result of this operation will be a vector of remainders. We are interested in whether the remainder is equal to 0:

```
    0 = (ιN)°.|(ιN)
```

Now we have a boolean two-dimensional matrix indicating whether the numbers were divisible (1) or not (0).

In step (b), we want to see how many times the number was divisible, that is, the number of 1's in each row:

```
    +/[2] 0 = (ιN) °.| (ιN)
```

But we are only interested in those rows that have exactly two 1's.

```
    2 = (+/[2] 0 = (ιN) °.| (ιN))
```

The result is a boolean vector indicating whether the index is a prime (1) or not (0). This is the desired vector of boolean expressions. To get the actual prime numbers, we apply compression:

```
    (2 = (+/[2] 0 = (ιN) °.| (ιN))) / ιN
```

The essence of this solution is that it builds successively more complex expressions from simpler ones. We may compose parts easily because the parts do not interfere with one another. Lack of interference is due to the lack of side-effects in the expressions that we have built This is indeed the promise of functional programming: functions are appropriate building blocks for program composition.

7.5 Functional programming in C++

In this chapter, we have studied the style of functional programming as supported by languages designed to support this style of programming. It is interesting to ask to what degree traditional programming languages can support functional programming techniques. It turns out that the combination of classes, operator overloading, and templates in C++ provides a surprisingly powerful and flexible support for programming with functions. In this section, we explore these issues.

7.5.1 Functions as objects

A C++ class encapsulates an object with a set of operations. We may even overload existing operators to support the newly defined object. One of the operator we can overload is the application operator, i.e. parentheses. This can be done by a function definition of the form: operator()(parameters...){...}. We can use this facility to define an object that may be applied, that is, an object that behaves like a function. The class Translate whose outline is shown in Figure 88 is such an object. We call such objects function or functional object. They are defined as objects but they behave as functions.

```
...definitions of types word and dictionary
class Translate {
private: ...;
public:
    word operator()(dictionary& dict, word w)
    {
        //   look up word w in dictionary dict
        //   and return result
    }
}
```

FIGURE 88. Outline of a function object in C++

We may declare and use the object Translate in this way:

```
Translate Translator(); //construct a Translate object
cout << Translate(EnglishGermanDict, “university”);
```

which would presumably print “universitaet”, if the dictionary is correct.

The ability to define such objects means that we have already achieved the major element of functional programming: we can construct values of type function in such a way that we can assign them to variables, pass them as arguments, and return them as result.

7.5.2 Functional forms

Another major element of functional programming is the ability to define functions by composing other functions. The use of such high-order functions is severely limited in conventional languages and they are indeed one of the distinguishing characteristics of functional languages.

It turns out, however, that with the use of templates in C++, we can simulate high-order functions to a high degree. First, we can use function objects as closures and bind some of their parameters. For example, we can modify the class definition of Figure 88 to add a constructor that accepts the dictionary to be used in lookup. By constructing a translator object this way, we bind the dictionary and produce a function object that works only with that dictionary. The new class definition and its use are shown in Figure 89. A closure is a function with some of its free variables bound. In this example, the function application operator () uses d as a bound variable. We use a constructor to bind this free variable. We can bind it to different values in different instantiations of the object. The difference between partial instantiation in this way in C++ and the general use of closures in functional languages is that here we can only bind a particular set of variables that are the parameters in the constructor of the object.

```
...definitions of types word and dictionary
class Translate {
private:
    dictionary D; //local dictionary
public:
    Translate(dictionary& d)
        {D = d;}
    word operator()(word w)
    {
        //   look up word w in dictionary D
        //   and return result
    }
}
...
//construct a German to English translator
Translate GermanToEnglish (GermanEnglishDictionary);
//construct a German to English translator
Translate EnglishToItalian (EnglishItalianDictionary);
...
cout << EnglishToItalian (GermanToEnglish(“universitaet”));
...
```

FIGURE 89. Outline of a partially instantiated function object in C++

The 1995 ANSI proposal for the C++ standard library contains a number of function objects and associated templates to support a functional style of programming. For example, it includes a predicate function object greater which takes two arguments and returns a boolean value indicating whether the first argument is greater than the second. We can use such function objects, for example, as a parameter to a sort routine to control the sorting order. We construct the function object in this way: greater<int>() or less<int>().

The library also includes a higher-order function find_if, which searches a sequence for the first element that satisfies a given predicate. This find_if takes three arguments, the first two indicate the beginning and end of the sequence and the third is the predicate to be used. Find_if uses the iterators that we discussed in Chapter 5. Therefore, it is generic and can search arrays, lists, and any other linear sequence that provides a pointer-like iterator object. Here, we will use arrays for simplicity. To search the first 10 elements of array a for an element that is greater than 0, we may use something like the following statement:

```
int* p= find_if (a, a+10, "...positive..." ); //not right******
```

What function can we use to check for positiveness? We need to check that something is greater than 0. Given template function objects such as greater, we can build new functions by binding some of their parameters. The library provides *binder* templates for this purpose. There is a binder for binding the first argument of a template function object and a binder for binding the second argument. For example, we might build a predicate function positive from the function object greater by binding its second argument to 0 in the following way:

```
bind2nd<int>(greater<int>, 0)
```

The library also provides the usual high-order functions such as reduce, accumulate, and so on for sequences. The combination of the high level of genericity for sequences and the template function objects to a great degree enable the adoption of a functional style of programming in C++.

7.5.3 Type inference

The template facility of C++ provides a surprising amount of type inference. For example, consider the polymorphic max function given in Figure 90. First, the type of the arguments is simply stated to be of some class T. The C++ compiler accepts such a definition as a polymorphic function parameterized by type T. We have seen that the ML type inferencing scheme rejects such a function because it cannot infer the type of the operator > used in the function definition. It forces the programmer to state whether T is int or float. C++, on the other hand, postpones the type inferencing to template instantiation time. Only when max is applied, for example in an expression ...max(a, b), does C++ do the required type checking. This scheme allows C++ to accept such highly generic functions and still do static type checking. At function definition time, C++ notes the fact that the function is parametric based on type T which requires an operation > and assignment (to be able to be passed and returned as arguments). At instantiation time, it checks that the actual

```
template <class T>
T max (T x, T y)
    {if (x>y) return x;
        else return y;
}
```

FIGURE 90.A C++ generic max function parameters satisfy the type requirements.

We have already contrasted the C++ polymorphic functions with those of ML in terms of type inference. It is also instructive to compare them with those of Ada. In the definition of a polymorphic function based on a type parameter T, neither C++, nor ML require the programmer to state the requirements on type T explicitly: they infer them from the text of the function definition. For example, both discover that type must support the > operation. ML rejects the function definition because of this requirement and C++ accepts it. In Ada, in contrast, the specification of the function must state explicitly that they type T must support the operation >. This is intended to allow the function specification to be compiled without the body of the function. Both ML and Ada accord special treatment to the assignment and equality operators: Ada refers to types that support these two operations as **private** and ML infers a type that uses the equality operator as not just any type but an *equality* type. Each language tries with its decisions to balance the inter-related requirements of strong typing, ability to describe highly generic functions, writability and readability.

7.6 Summary

In this chapter, we have examined the concepts and style of functional programming and some of the programming languages that support them. The key idea in functional programming is to treat functions as values. Functional programming has some of the elegance and other advantages of mathematical functions and therefore, it is easier to prove properties about functional programs than about iterative programs. On the other hand, because of its reliance on mathematics rather than computer architecture as a basis, it is more difficult to achieve efficient execution in functional programs.

Modern functional languages have adopted a number of features such as strong typing and modularity that have been found useful in conventional languages. In turn, conventional languages such as C++ and Ada have adopted some functional programming ideas that make it easier to treat functions as objects.

7.7 Bibliographic notes

Even though work on functional programming dates back to the 1930s, interest in functional programming was sparked by the Turing award lecture of John Backus[Backus 76]. In this paper, the inventor of FORTRAN argued that imperative programming simply could not support programming large systems and the mathematically-based functional programming had a much better chance. He introduced a family of functional programming languages called FP as a candidate language. References to Miranda, Hope, Scheme, ...Haskell is an attempt to standardize the functional programming syntax. The paper by Hudack is an excellent treatise on functional programming languages. The document by Bob Harper, available on the net, is an excellent introduction to ML and is the source of several examples in this chapter.

7.8 Exercises

1. Reduce the lambda expression [y/x]((λ y.x)(λ x.x)x).
2. Reduce the lambda expression (λ x.(x x))(λ x.(x x)). What is peculiar about this expression?
3. What is the type of this ML function:
```
fun f(x, y) = if hd(x) = hd(y)
                 then f(tl(x), tl(y))
                 else false;
```
4. Write an ML function to merge two lists.
5. Write an ML function to do a sortmerge on a list.
6. Explain why the ML function bigger gives a type error:
```
fun bigger(x, y) = if x> y then x else y;
```
7. Write a function in ML to compute the distance between two points, represented by (x, y) coordinates. Next, based on this function, define a new function to compute the distance of its single argument from the origin.
8. Define the two functions of Exercise 7 in C++.
9. In C++, we can use a function template to write a function bigger similar to the one in Exercise 6. Why does this program not cause a type error at compile-time?
```
template<class N>
int bigger(N x, N y)
   {if (x>y) return x; return y;}
```
Use Exercises 6 and 7 to compare the type inference support in C++ and ML in terms of flexibility, generality, and power.
10. Write the identify function of Section 7.4.1.5 in C++ (using templates).
11. Define a signature for the Stack of Figure 86 which exports only create and push. Is it useful to have a signature that does not export create?
12. In this chapter, we have seen the use of partially instantiated functions in both C++ and functional programming languages. In Chapter 4, we saw the use of default values for function parameters. In what sense is the use of such default values similar to constructing a closure and in what ways is it different?

1

Logic and rule-based languages

CHAPTER 8

This chapter presents a nonconventional class of languages: logic and rule-based languages. Such languages are different from procedural and functional languages not only in their conceptual foundations, but also in the programming style (or paradigm) they support. Programmers are more involved in describing the problem in a declarative fashion, then in defining details of algorithms to provide a solution. Thus, programs are more similar to specifications than to implementations in any conventional programming language. It is not surprising, as a consequence, that such languages are more demanding of computational resources than conventional languages.

8.1 The"what" versus "how" dilemma: specification versus implementation

A software development process can be viewed abstractly as a sequence of phases through which system descriptions progressively become more and more detailed. Starting from a software requirements specification, which emphasizes *what* the the system is supposed to do, the description is progressively refined into a procedural and executable description, which describes *how* the problem actually is solved mechanically. Intermediate steps are often standardized within software development organizations, and suitable notations are used to describe their outcomes (software artifacts). Typically, a design phase is specified to occur after requirements specification and before implementation, and suitable software design notations are provided to document the resulting software architecture. Thus the "what" stated in the requirements is transformed into the "how" stated in the design document, i.e., the design specification can be viewed as an abstract implementation of the requirements specification. In turn, this can be viewed as the specification for the subsequent implementation step, which takes the design specification and turns it into a running program.

In their evolution, programming languages have become increasingly higher level. For example, a language like Ada, Eiffel, and C++ can be used in the design stage as a design specification language to describe the modular structure of the software and module interfaces in a precise and unambiguous way, even though the internals of the module (i.e., private data structures and algorithms) are yet to be defined. Such languages, in fact, allow the module specification (its interface) to be given and even compiled separately from the module implementation. The specification describes "what" the module does by describing the resources that it makes visible externally to other modules; the implementation describes "how" the internally declared data strucures and algorithms accomplish the specified tasks.

All of the stated steps of the process that lead from the initial requirements specification down to an implementation can be guided by suitable systematic methods. They cannot be done automatically, however: they require engineering skills and creativity by the programmer, whose responsibility is to map–translate–requirements into executable (usually, procedural) descriptions. This mapping process is time-consuming, expensive, and error-prone activities.

An obvious attempt to solve the above problem is to investigate the possibility of making specifications directly executable, thus avoiding the translation step from the specification into the implementation. Logic programming tries to do exactly that. In its simplest (and ideal) terms, we can describe logic programming in the following way: A programmer simply declares the properties that describe the problem to be solved. The problem description is used by the system to solve the problem (*infer a solution*). To denote its distinctive capabilities, the run-time machine that can execute a logic language is often called an *inference engine*.

In logic programming, problem descriptions are given in a logical formalism, based on first-order predicate calculus. The theories that can be used to describe and analyze logic languages formally are thus naturally rooted into mathematical logic. Our presentation, however, will avoid delving into deep mathematical concepts, and will mostly remain at the same level in which more conventional languages were studied.

The above informal introduction and motivations point out why logic programming is often said to support a *declarative* programming paradigm. As we will show, however, existing logic languages, such as PROLOG, match this description only partially. To make the efficiency of the program execution acceptable, a number of compromises are made which dilute the purity of the declarative approach. Efficiency issues affect the way programs are written; that is, the programmer is concerned with more than just the specification of what the program is supposed to do. In addition, nondeclarative language features are also provided, which may be viewed as directions provided by the programmer to the inference engine. These features in general reduce the clarity of program descriptions.

8.1.1 A first example

In order to distinguish between specification and implementation, and to introduce logic programming, let us specify the effect of searching for an element x in a list L of elements. We introduce a predicate is_in (x, L) which is true whenever x is in the list L. The predicate is described using a self-explaining hypothetical logic language, where operator "•" denotes the concatenation of two lists and operator [ ] transforms an element into a list containing it and "iff" is the conventional abbreviation for "if and only if.".

```
for all elements x and lists L: is_in (x, L)  iff
    L = [x]
or
    L = L1 • L2 and
        (is_in (x, L1) or is_in (x, L2))
```

The above specification describes a binary search in a declarative fashion. The element is in the list if the list consists exactly of that element. Otherwise, we can consider the list as decomposed into a left sublist and a right sublist, whose concatenation yields the original list. The element is in the list, if it is in either sublist.

Let us now proceed to an implementation of the above specification. Besides other details, an implementation of the above specification must decide

- how to split a list into a right and a left sublist. An obvious choice is to split it into two sublists of either the same length, or such that they differ by at most one;
- how to store the elements in the list. An obvious choice is to keep the list sorted, so that one can decide whether to search the left or the right sublist and avoid searching both;
- how to speed up the search. Instead of waiting until a singleton list is obtained via repeated splitting, the algorithm can check the element that separates the two sublists. If the separator equals the desired element, the search can stop. Otherwise, it proceeds to check either in the right or in the left sublist generated by the splitting, depending on the value of the separator.

A possible C++ implementation of the specification is shown in Figure 91.By looking carefully at both the logic specification and the C++ implementation, one can appreciate the differences between the two in terms of ease of writing, understandability, and self-confidence in the correctness of the description with respect to the initial problem.

Instead of transforming the specification into an implementation, one might wonder whether the specification can be directly executed, or used as a starting point for a straightforward derivation process yielding an implementation. To do so, we can read the above declarative specification procedurally as follows:

```
Given an element x and a list L, in order to prove that x is in L,  proceed as follows
(1) prove that L is [x];
(2) otherwise split L into L1 • L2 and prove one of the following:
    (2.1) x is in L1, or
    (2.2) x is in L2
```

A blind mechanical executor which follows the procedure can be quite inefficient, especially if compared to the C++ program. This is not surprising. Direct execution is less efficient than execution of an implementation in a traditional procedural language, but this is the obvious price we pay for the savings in programming effort.

```
int binary_search (const int val, size, const int array[ ]) {
    // return the index of the desired value val, if it is there
    // otherwise return -1
    if size ð 0 {
        return (-1);
    }
    int high = size; // the portion of array to search is
    int low = 0;        // low. .high-1
    for ( ; ; ) {
        int mid = (high + low) / 2;
        if (mid = low) {
            // search is finished
            return (test != array [low]) ? -1 : mid;
        }
        if (test < array [mid]) {
            high = mid;
        }
        else if (test > array [mid]) {
            low = mid;
        else {
            return mid;
        }
    }
}
```

FIGURE 91. A C++ implementation of binary search

8.1.2 Another example

Suppose we wish to provide a logical specification of sorting a list of integers in ascending order. Our goal is thus to describe a predicate sort (X, Y) which is true if the nonempty list Y is the sorted image of list X. The description of such a predicate can be provided top-down by introducing two lower-level predicates permutation (X, Y), which is true if list Y is a permutation of list X, and is_sorted (Y), which is true if list Y is sorted. We can in fact write

```
for all integer lists X, Y: sort (X, Y) iff
    permutation (X, Y) and sorted (Y)
```

In order to describe predicate sorted, let us assume that our logic notation provides the notion of an indexable sequence of integers (we use subscripts in the range 1. .length (X) for this purpose):

sorted (Y) iff forall j such that 1 ð j < length (Y), Y_j ð Y_{j+1}

In order to describe predicate permutation (X, Y), we assume that the following built-in predicates are available for lists (of integers):

- is_empty (X), which is true if list X is empty;
- has_head (X, Y), which is true if the integer Y is the first element in the (nonempty) list X;
- has_tail (X, Y), which is true if Y is the list obtained by deleting the first element of the (nonempty) list X;
- delete (X, Y, Z) , which is true if list Z is the result of deleting an occurrence of element X from list Y

Predicate permutation (X, Y) can thus be specified as follows:

```
permutation (X, Y) iff
    is_empty (X) and is_empty (Y)
or else
    has_head (Y, Y1) and has_tail (Y, Y2) and delete (Y1, X, X2) and permutation (X2, Y2 )
```

(The logical connective or else has the following intuitive meaning: A or else B means A or ((not A) and B).)

The declarative specification can be read procedurally as follows, assuming that two lists X and Y are given:

```
Given two integer lists X and Y, in order to prove that the sort operation applied to X yields
Y,  prove that Y is a permutation of X and prove that Y is sorted.

    In order to prove that Y is a permutation of  X, proceed as follows
        (1) prove that both are empty;
        (2) otherwise, eliminate the first element of Y from both X and Y, thus
            producing X2 and Y2, and prove that Y2 is a permutation of X2.

    In order to prove that Y is sorted, prove that each element is less than the one
    that follows it.
```

The declarative specification can also be read as ad constructive recursive procedure. Assume that X is a given list and its sorted image Y is to be provided as a result:

```
Given an integer list X, construct its permutations and prove that one such permutation Y
exists that is sorted.

    In order to construct Y, a permutation of  X, proceed as follows
        (1) Y is the empty list if X is an empty list;
        (2) otherwise, Y is constructed as a list whose head is an element X1
            of X and whose tail Y2 is constructed as follows.
            (2.1) delete X1 from X, thus obtaining the list X2;
            (2.2) Y2 is constructed as a permutation of X2
```

This example confirms that a direct implementation of the specification, according to its procedural interpretation, can be quite inefficient. In fact, one might need to generate all permutations of a given list, before generating the one which is sorted. All different permutations can be generated because in step (2) above there are many ways of deleting an element X1 from X. Any such way provides a different permutation, and all such different permutations must be generated, until a sorted one is finally found.

8.2 Principles of logic programming

To understand exactly how logic programs can be formulated and how they can be executed, we need to define a possible reference syntax, and then base on it a precise specification of semantics. This would allow some of the concepts we used informally in Section 8.1 (such as "procedural interpretation") to be stated rigorously. This is the intended purpose of this section. Specifically, Section 8.2.1 provides the necessay background definitions and properties that are needed to understand how an interpreter of logic programs works. The interpreter provides a rigorous definition the program’s "procedural interpretation". This is analogous to SIMPLESEM for imperative programs.

8.2.1 Preliminaries: facts, rules, queries, and deductions

Although there are many syntactic ways of using logic for problem descriptions, the field of logic programming has converged on PROLOG, which is based on a simple subset of the language of first-order logic. Hereafter we will gradually introduce the notation used by PROLOG.

The basic syntactic constituent of a PROLOG program is a *term*. A term is a constant, a variable, or a compound term. A compound term is written as a *functor symbol* followed by one or more arguments, which are themselves terms. A *ground term* is a term that does not contain variables. Constants are written as lower-case letter strings, representing atomic objects, or strings of digits (representing numbers). Variables are written as strings starting with an upper-case letter. Functor symbols are written as lower-case letter strings.

```
alpha               --this is a constant
125                 --this is a constant
X                   --this is a variable
abs (-10, 10)       --this is a ground compound term; abs is a functor
abs (suc (X), 5)    --this is a (nonground) compound term
```

The constant [ ] stands for the empty list. Functor "." constructs a list out of an element and a list; the element becomes the head of the constructed list. For example, .(alpha, [ ]) is a list containing only one atomic object, alpha. An equivalent syntactic variation, [alpha, [ ]], is also provided. Another example would be

```
.(15, .( toot, .(duck, donald)))
```

which can also be represented as

```
[15, [toot, [duck, donald]]]
```

The notation is further simplified, by allowing the above list to be written as

```
[15, toot, duck, donald]
```

and also as

```
[15 | [toot, duck, donald]]
```

In general, the notation

```
[X | Y]
```

stands for the list whose head element is X and whose tail list is Y.

A predicate is represented by a compound term. For example

```
less_than (5, 99)
```

states the "less than" relationship between objects 5 and 99.

PROLOG programs are written as a sequence of *clauses*. A clause is expressed as either a single predicate, called *fact,* or as a *rule* (called *Horn clause*) of the form

```
conclusion :- condition
```

where :- stands for "if", conclusion is a single predicate, and condition is a conjunction of predicates, that is, a sequence of predicates separated by a comma, which stands for the logical and. Facts can be viewed as rules without a condition part (i.e., the condition is always true). Thus the term "rule" will be used to indicate both facts and rules, unless a distinction will be explicitly made. A rule's conclusion is also called the rule's *head*. Clauses are implicitly quantified universally. A PROLOG rule

```
conclusion :- condition
```

containing variable X1, X2, . . ., Xn would be represented in the standard notation of mathematical logic as

$$\forall X1, X2, \ldots, Xn \ (\text{condition} \supset \text{conclusion})$$

where $\supset$ is the logical implication operator. In a procedural program, it would be represented as

if condition **then** conclusion;

For example, the following program

```
length ([ ], 0).      --this is a fact
length ([X | Y], N) :- length (Y, M), N = M + 1.       --this is a rule
```

says that

- the length of the null string is zero,
- for all X, Y, N, M, if M is the length of list Y and N is M + 1, then the length of a nonnull string with head X and tail Y is one more than the length of Y.

As another example, the sort problem of Section 8.1.2 can be represented in PROLOG as follows:

```
sort (X, Y) :- permutation (X, Y), sorted (Y).
sorted ([ ]).               --the empty list is sorted
sorted ([X | [ ]]. -        - the sigleton list is sorted
sorted ([X | [Y | Z]]) :- X ð Y and sorted (Z).
permutation ([ ], [ ]).
permutation (X, [Y1 | Y2]) :- delete (Y1, X, X2), permutation (X2, Y2 ).
delete (A, [A | B], B).
delete (A, [B | C], [B | D]) :- A ¦ B, delete (A, C, D).
```

The examples we gave so far show implicitly that PROLOG is an untyped language. No type declarations are provided for variables. The value that is dynamically bound to a variable determines the nature of the object, and thus the legality of the operations applied to it. For example, in the case of sort, operators "less than" and "not equal" must be applicable to the elements of the list. For example, it might be a list of numbers, or a list of characters.

Facts and rules are used to express the available knowledge on a particular domain: they provide a declarative specification. They are used to solve problems, specified as *queries*. A query can also be viewed as a *goal* that must be proved.

From a logical viewpoint, the answer to a query is YES if the query can be derived by applying *deductions* from the set of facts and rules. For example:

```
?-sort ([3, 2, 7, 1], [1, 2, 3, 7]).
```

is a query, to which the answer would be YES.

In order to understand how deductions are made from a logic program, we need to provide some mathematical preliminaries. A *substitution* is a function, defined as a (possibly empty) finite set of pairs of the form <Xi, ti>, where Xi is a variable and ti is a term, Xi ¦ Xj for all i, j with i ¦ j and Xi does not occur in tj for all i, j. A substitution µ may be extended to apply to terms; i.e., it is applied to any of the variables appearing in a term. The result of applying a substitution µ to term t1, µ (t1), yields a term t2, which is said to be an *instance* of t1. A substitution may also be applied to a rule; i.e., it is applied to all its component terms to produce an instance rule.

For example, the substitution

```
{<A, 3>, <B, beta (X, xyz>}
```

applied to term

```
func (A, B, C)
```

yields

```
func (3, beta (X, xyz), C)
```

The fundamental rule used in logic to make deductions is called *modus ponens*. Such rule can be stated as follows, using the syntax of logic programming:

```
from the rule R:
        P :- Q1, Q2, . . ., Qn
and the facts
        F1, F2, . . ., Fn
we can deduce D as a logical consequence
if D :- F1, F2, . . ., Fn is an instance of R
```

If we submit a ground query to a logic program, the answer to the query is YES if the repeated application of modus ponens proves that the query is a logical consequence of the program. Otherwise, if such deduction cannot be generated, the answer is false. For example, the answer to the query sorted ([1, 5, 33]) is YES because the following deduction steps can be performed using modus ponens:

```
i1. sorted ([33| [ ])
```

i2. from the previous step, our knowledge that 5 ð 33, and from the rule sorted ([X | [Y | Z]]) :- X ð Y and sorted (Z) we can deduce sorted ([5 | [33 | [ ]]])

i3. from the previous step, our knowledge that 1 ð 5, and from sorted ([X | [Y | Z]]) :- X ð Y and sorted (Z) we can deduce sorted ([1 | [5 | [33 | [ ]]]]), i.e., sorted ([1, 5, 33]).

PROLOG allows existential queries to be submitted. An *existential query* is a query which contains a variable. For example,

```
?-sort ([5, 1, 33], X)
```

means "is there an X such that the sort of [5, 1, 33] gives X"? To accommodate existential queries in the deduction process, another rule, called *existentiail rule*, is provided. The rule states that an existential query Q is a consequence of an instance of it, µ (Q), for any µ. In the above example, the answer would be YES since

j1. sorted ([1, 5, 3]) can be proved by repeated application of modus ponens, as shown above

j2. permutation ([5, 1, 33], [1, 5, 33]) can be proved in a similar way

j3. from j1 and j2 we can deduce sort ([5, 1, 33], [1, 5, 33])

j4. from the existential rule, we can conclude that the answer to the query is YES.

Modus ponens and the existential rule are the conceptual tools inherited from mathematical logic that can be used to support deductive reasoning. But in order to make logic specifications executable, we need to devise a practical approach that is amenable to mechanical execution: we need to interpret logic programs procedurally.

Intuitively, the *procedural interpretation* of a logic program consists of viewing a query as a procedure call. A set of clauses for the same predicate, in turn, can be viewed as a procedure definition, where each clause represents a branch of a case selection. The basic computational step in logic programming consists of selecting a call, identifying a procedure corresponding to the call, selecting the case that matches the call, and generating new queries, if the matched case is a rule. This is in accordance to the concepts of case analysis and pattern matching that were introduced in Chapter 4. For example, the above query ?-sort ([5, 1, 33], X), which is matched by the sort rule, generates the following queries:

```
?-permutation ([3, 2, 7, 1], [1, 2, 3, 7]).
```

and

```
?-sorted ([1, 2, 3, 7]).
```

The procedure corresponding to the call described by the first of the above two queries has two cases:

```
permutation ([ ], [ ]).
permutation (X, [Y1 | Y2]) :- delete (Y1, X, X2), permutation (X2, Y2 ).
```

To select the appropriate case, a special kind of pattern matching is performed between the query and the head of the rule describing each case. Intuitively, in our example, the query does not match the first case, which is the rule for empty lists. The match against the other rule’s head binds X to [3, 2, 7, 1], Y1 to 1, Y2 to [2, 3, 7], and generates two further queries

```
delete (1, [3, 2, 7, 1], X2)
```

and

```
permutation (X2, [2, 3, 7])
```

Interpretation proceeds in much the same manner for each generated query, until all queries are processed by the interpreter. The intuitive, yet informal, treatment of the interpretation procedure described so far will be formally described in the next section.

8.2.2 An abstract interpretation algorithm

In this section we discuss in detail how logic programs can be procedurally interpreted by an abstract processor. As we mentioned earlier, the abstract processor must be able to take a query as a goal to be proven, and match it against facts and rule heads. The matching process, which generalizes the concept of procedure call, is a rather elaborate operation, called *unification*, which combines pattern matching and binding of variables.

Unification applies to a pair of terms t1 (representing the goal to prove) and t2 (representing the fact or rule’s head with which a match is tried). To define it, we need a few other background definitions. Term t1 is said to be *more general* than t2 if there is a substitution µ such that t2 = µ (t1), but there is no substitution ¼ such that t1 = ¼ (t2). Otherwise, they are said to be *variants*, i.e., they are equal up to a renaming of variables.

Two terms are said to unify if a substitution can be found that makes the two terms equal. Such a substitution is called a *unifier*. For example, the substitution

```
s1 = {<X, a>, <Y, b>, <Z, b>}
```

is a unifier for the terms f (X, Y) and f (a, Z). A *most general unifier* is a unifier μ such that μ (t1) = μ (t2) is the most general instance of both t1 and t2. It is easy to prove that all most general unifiers are variants. We will therefore speak of "the" most general unifier (MGU), assuming that it is unique up to a renaming of its variables. In the previous example, s1 is not the MGU. In fact the substitution

s2 = {<X, a>, <Y, W>, <Z, W>}

is more general than s1, and it is easy to realize that no unifier can be found that is more general than s2.

MGUs are computed by the unification algorithm shown in Figure 92. The algorithm keeps the set of pairs of terms to unify in a working set which is initialized to contain the pair <t1, t2>.The algorithm is written in a self-explaining notation. If the two terms given to the algorithm do not unify, the exception unification_fail is raised.

The algorithm operates on two terms t1 and t2 and returns their MGU.
It raises an exception if the unification fails.

```
MGU = { }; --MGU is the empty set
WG = {<t1, t2>}; --working set initialized
repeat
    extract a pair <x1, x2> from WG;
    case
        • x1 and x2 are two identical constants or variables:
            do nothing
        • x1 is a variable that does not occur in x2:
            substitute x2 for x1 in any pair of WG and in MGU;
        • x2 is a variable that does not occur in x1:
            substitute x1 for x2 in any pair of WG and in MGU;
        • x1 is f (y1, y2, . . ., yn), x2 is f (z1, z2, . . ., zn), and f is a functor and n Š 1:
            insert <y1, z1>, <y2, z2>, . . ., <yn, zn> into WG;
        otherwise
            raise unification_fail;
    end case;
until WG = { } --working set is empty
```

FIGURE 92. Unification algorithm

To ensure termination the unification algorithm does not attempt to unify such pairs as <f (. . .X. . .), X>, in order to enforce termination. This is achieved by the so-called *occur check* (see the second and third case in the repeat loop of the algorithm in Figure 92).

We are finally in a position to provide a precise meaning for "procedural interpretation" by showing how logic programs can be interpreted (Figure 93). The algorithm assumes that whenever a unification is applied to a goal and a rule’s head all variables appearing in the rule’s head are automatically renamed with brand new variable names. Remember that variables with the same name appearing in different clauses of the logic language are different; this is obvious since clauses are implicitly universally quantified. The renaming ensures that such variables are indeed treated as different.

The algorithm shown in Figure 93 is *nondeterministic*, i.e., it describes several possible computations for a given input goal. The goal is solved successfully if there is a computation that stops with the answer YES. In such a case, if the goal contains variables, when the interpreter stops, all variables are bound to a ground term. A computation may raise the exception fail, if the attempt to solve a goal fails during the process. It is also possible that a computation does not terminate, i.e., the set of goals to be proven never becomes empty.

In order to illustrate how the nondeterministic interpretation algorithms operates, consider the following example of a logic program, which describes a binary relation (rel) and its closure (clos):

```
(1)     rel (a, b).
(2)     rel (a, c).
(3)     rel (b, f).
(4)     rel (f, g).
(5)     clos (X, Y) :- rel (X, Y).
(6)     clos (X, Y) :- rel (X, Z), clos (Z, Y).
```

Predicate rel lists all object pairs that constitute the relation. Pairs belonging to the closure are specified by the recursive predicate clos.

```
Given a goal G submitted as a query to a logic program P, the algorithm answers
YES, and provides bindings for the variables appearing in G, or  it answers NO
SG = {G}; --initialize the set of goals to solve to G, the submitted query
repeat
    begin
        extract an element E from SG
            and select a (renamed) clause X :- X1, X2, ..., Xn from P (n = 0 for a fact)
            such that <E, X> unifies with MGU μ;
        insert X1, X2, . . ., Xn into SG;
        apply μ to all elements of SG and to G;
    exception
        when unification_fail => exit;
    end;
until SG = empty;
if SG = empty then
    the answer is YES and G describes a solution
else
    raise fail
```

FIGURE 93.A nondeterministic interpretation algorithm

Suppose that the query

```
?-clos (a, f)
```

is submitted to the nondeterministic interpreter of Figure 93. Some of the many possible different computation paths for the query are shown in Figure 94. Computation paths are described by showing how goals are successively matched against facts or rule heads, and new subgoals are generated after the match. The goal chosen at each step for the match is shown in bold in the computation path. Since clauses are numbered in the logic program, clause numbers are shown in the computation paths to indicate the clause selected by the unification algorithm.

By examining Figure 94, it is easy to understand that in case (b) the computation solves the goal; cases (a) and (c) describe computations that fail because of the wrong choices made to solve nondeterminism; case (d) describes a nonterminating computation where clause 6 is chosen repeatedly in the unification procedure.

Let us try to understand the effect of the different choices that can be made during the execution of the interpretation algorithm because of nondeterminism. First, when a goal is being solved, it may be necessary to choose one out of several clauses to attempt unification. It may happen that the choice we make eventually results in a failure, while another choice would have led to success. For example, in the case of computation (a) the choice of clause 5 instead of 6 for the first match leads to a failure. Second, when several goals can be selected from SG, there is a choice of which is to be solved first. For example, when the goals to be solved are rel(a, Z1), clos (Z1, f), computations (c) and (d) make their selection in a different order. In general, when we have to choose which of two goals G1 and G2 is to be solved first, it can be shown that if there is a successful computation choosing G1 first, there is also a successful computation choosing G2 first. The choice can only affect the efficiency of searching a solution. From now on, we will assume that whenever a goal unifies with a rule's head, the (sub)goals corresponding to the righthand side of the rule will be solved according to a deterministic policy, from left to right.

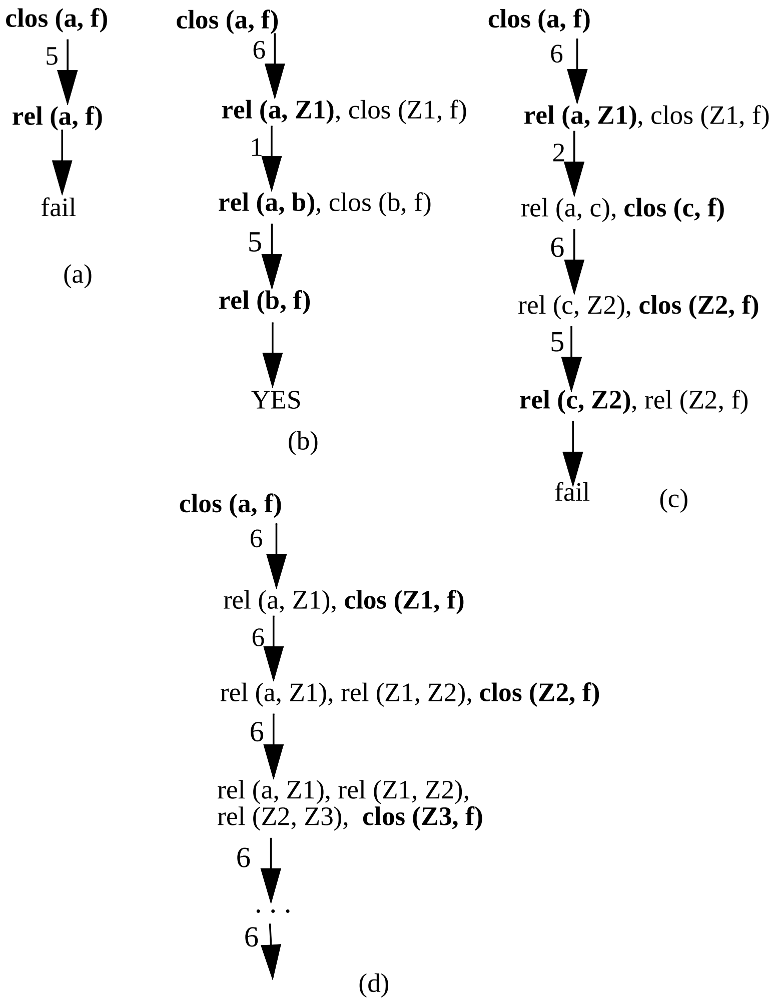

clos (a, f)
5
rel (a, f)
fail
(a)
clos (a, f)
6
rel (a, Z1), clos (Z1, f)
1
rel (a, b), clos (b, f)
5
rel (b, f)
YES
(b)
clos (a, f)
6
rel (a, Z1), clos (Z1, f)
2
rel (a, c), clos (c, f)
6
rel (c, Z2), clos (Z2, f)
5
rel (c, Z2), rel (Z2, f)
fail
(c)
clos (a, f)
6
rel (a, Z1), clos (Z1, f)
6
rel (a, Z1), rel (Z1, Z2), clos (Z2, f)
6
rel (a, Z1), rel (Z1, Z2),
rel (Z2, Z3), clos (Z3, f)
6
. . .
6
(d)

FIGURE 94. Different computations of the nondeterministic interpreter

Another way of viewing the behaviors of the nondeterministic interpreter of Figure 94 is to view them as a tree of computations (called the *search tree*). The arcs exiting a node represent all possible clauses with which unification is performed. Figure 95 shows the search tree for the query clos (a,f).

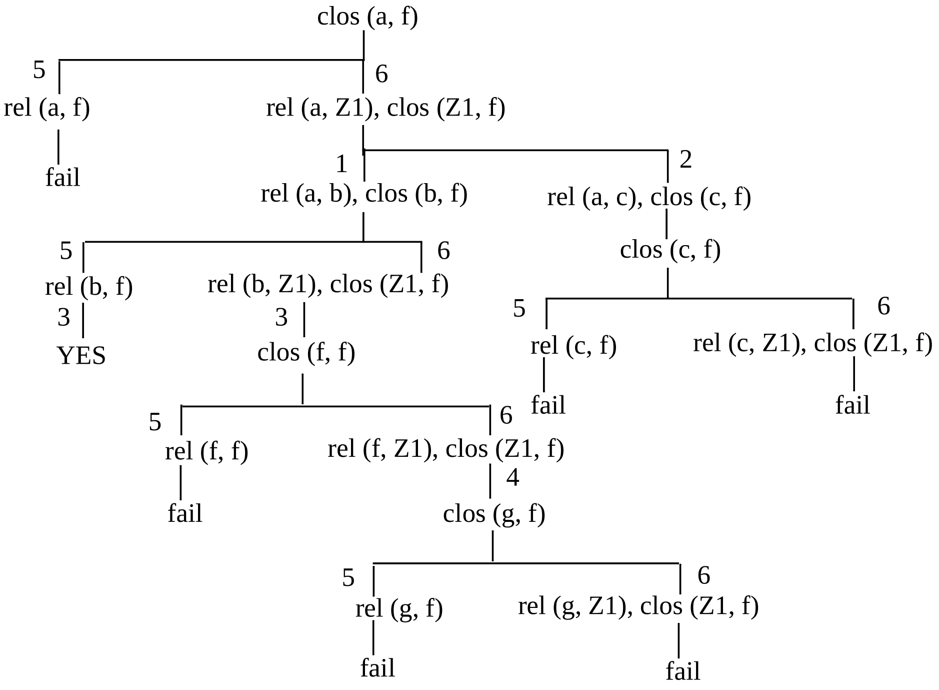


FIGURE 95. Search tree for the query clos (a, f)

To implement the nondeterministic interpreter on a conventional processor, it is necessary to define a traversal of the search tree according to some policy. One possibility is to search all branches in parallel (*breadth-first* policy). Another possibility is *depth-first* search, for example always choosing the first clause in the list for unification. In such a case, when the computation fails along one path, it is necessary to backtrack to a previously unexplored choice, to find an alternative path. A breadth-first searching algorithm is said to provide a *complete proof procedure* for logic programs: it guarantees that if there is a finite proof of the goal, it will be found. In addition, the breadth-first algorithm is said to provide a *sound proof procedure*, since any answer derived by the interpreter is a correct answer to the query (i.e., it is a logical consequence that can be derived from the program via modus ponens and the existential rule). Completeness and soundness are indeed the most desired properties of a proof procedure. Depth-first search is a sound procedure; it is not complete, however, since the searching engine might enter a nonterminating computation, which would prevent backtracking to attempt another path which might lead to the solution.

Conventional logic programming languages, such as PROLOG, follow a depth-first search policy, as we will see in the next section. Several experimental languages have tried to improve the search method, by supporting breadth-first search via parallel execution of the different branches of the search tree.

As a final point, let us discuss when the answer to a query submitted to a logic program can be NO. This can only occur if all computations for that query terminate with a failure; i.e., the search tree is finite, and no leaf of the tree is labelled YES. Similarly, a goal submitted as a query ?-not Q yields YES if Q cannot be proven from the given facts and rules. This happens if the search tree is finite, and all computations corresponding to the different branches fail. In other terms, logic programs are based on the concept of *negation as failure*. A common way to describe negation by failure is to say that logic language interpreters work under the "closed world assumption". That is, all the knowledge possessed by the interpreter is explicitly listed in terms of facts and rules of the program. For example, the answer to the query

```
?- rel (g, h)
```

would be NO, which means that "according to the current knowledge expressed by the program it cannot be proved that rel (g, h) holds".

8.3 PROLOG

PROLOG is the most popular example of a logic language. Its basic syntactic features were introduced informally in Section 8.2. As we anticipated, PROLOG solves problems by performing a depth-first traversal of the search tree. Whenever a goal is to be solved, the list of clauses that constitutes a program is searched from the top to the bottom. If unification succeeds, the subgoals corresponding to the terms in the righthand side of the selected rule (if any) are solved next, in a predined left-to-right order. This particular way of operating makes the behavior of the interpreter quite sensitive to the way programs are written. In particular, the ordering of clauses and subgoals can influence the way the interpreter works, although from a conceptual viewpoint clauses are connected by the logical operator "or" and subgoals are connected by "and". These connectives do not exhibit the expected commutativity property.

As an example, Figure 96 shows all possible permutations of terms for the closure relation that was discussed in Section 8.2.2. It is easy to verify that any query involving predicate clos would generate a nonterminating computation in case (4). Similarly, a query such as ?-clos (g, c) causes a nonterminating interpretation process in cases (2) and (4), whereas in (1) and (3) the interpreter would produce NO.

```
(1)  clos (X, Y) :- rel (X, Y).                (2)  clos (X, Y) :- rel (X, Y).
     clos (X, Y) :- rel (X, Z), clos (Z, Y).        clos (X, Y) :- clos (Z, Y), rel (X, Z).


(3)  clos (X, Y) :- rel (X, Z), clos (Z, Y).   (4)  clos (X, Y) :- clos (Z, Y), rel (X, Z).
     clos (X, Y) :- rel (X, Y).                     clos (X, Y) :- rel (X, Y).
```

FIGURE 96. Variations of a PROLOG program

PROLOG provides several extra-logical features, which cause its departure from a pure logical language. The first fundamental departure is represented by the *cut* primitive, written as "!", which can appear as a predicate in the condition part of rules. The effect of the cut is to prune the search space by forbidding certain backtracking actions from occurring. Its motivation, of course, is to improve efficiency of the search by reducing the search space. It is the programmer’s responsibility to ensure that such a reduction does not affect the result of the search. The cut can be viewed as a goal that never fails and cannot be resatisfied. That is, if during backtracking one tries to resatisfy it, the goal that was unified with the lefthand side of the rule fails.

In order to illustrate how the cut works, consider the following rule:

```
A :- B, C, !, D, E
```

Suppose that, after a match between the rule’s head and a goal A', subgoals B, C, and D (with suitably applied substitutions) have been solved successfully. If subgoal E fails, the PROLOG interpreter backtracks and tries to solve D by matching it to the next available rule’s head, if any, found in scanning the program clauses from the top down. If no successful match can be found, the PROLOG interpreter would normally backtrack further, trying to find a new solution for C, and then B. Eventually, if all these fail, the match of A' with the rule would fail and another rule or fact would be tried. The presence of the cut, however, forbids the backtracking procedure from retrying C, then B, and then a further alternative for the match with A': the current goal A' would fail right away. In other terms, the cut, viewed as a predicate, always succeeds, and commits the PROLOG interpreter to all the choices made since the goal A' was unified with the head of the rule in which the cut occurs.

Let us consider as examples the simple programs shown in Figure 97 (a). The program contains the relational predicate ð. Relational predicates (i.e., <, ð, =, ¦, >, Š) are such that when a goal, like A ð B, is to be solved, both operands A and B must be bound to arithmetic expressions that can be evaluated (i.e., if they contain variables, they must be bound to a value). The goal A ð B succeeds if the result of evaluating A is less than or equal to the result of evaluating B. If not, or if A and B cannot be evaluated as arithmetic expressions, the goal fails. The presence of the cut implies that if the first alternative is chosen (i.e., X ð Y), the backtracking that may occur due to the failure in the proof of some later goal will not try to find another solution for max, because there is no possibility for the second alternative to be chosen.

Relational predicates represent another departure of PROLOG from logical purity. In fact, for example, the evaluation of the following goal

```
?- 0 < X
```

which is read as

is there a positive value for X?

does not succeed by binding X to an integer value greater than zero, as the logical reading of the clause might suggest. It simply fails, since X in the query is unbound, and the arithmetic expression cannot be evaluated. To guard against such a situation, the programmer must ensure that variables are bound if they are expected to participate in expression evaluations.

```
max (X, Y, Y) :- X ðY, !.
max (X, Y, X) :- X > Y, !.
```

(a)

```
if_then_else (A, B, C) :- A, !, B.
if_then_else (A, B, C) :- C.
```

(b)

FIGURE 97. Sample PROLOG fragments using cut

The fragment of Figure 97 (b) defines an if_then_else predicate. If clause A describes a goal whose proof succeeds, then goal B is to be proved. If the execution fails to prove that A holds, then C is to be proved. A possible use is shown by the following query, where rel and clos have been introduced in Section 8.1.2 and assuming that some clause exists in the program for goal g:

```
?- if_then_else (rel (a, X), retract (rel (a, X)), g (X)).
```

The example shows another extralogical feature of PROLOG: retract. This feature removes from the program the first clause that unifies with its argument. Thus the effect of the query is to remove from the relation rel a pair whose first element is a, if there is one. If the choice of executing retract is made, it cannot be undone through backtracking. Instead, if a pair whose first element is a does not exist, goal g is solved.

The reciprocal effect of retract is provided by the extra-logical primitives assert and asserta, which allow their argument to be added as a clause at the end of the program or at the beginning, respectively. Thus retract and assert allow logic programs to be modified as the program is executed. They can be used, for example, to add new ground facts to the program, to represent new knowledge that is acquired as the program is running.

Another departure from logic is represented by the assignment operator is, illustrated by the PROLOG program of Figure 98, which defines the factorial of a natural number. When the operator is encountered during the evaluation, it must be possible to evaluate the expression on its lefthand side (i.e., if the expression contains variables, they must be bound to a value); otherwise the evaluation fails. If the righthandside variable is also bound to a value, then the goal succeeds if the variable’s value is equal to the value of the expression. Otherwise, the evaluation succeeds and the lefthand side variable is bound to the value of the expression. In the example, when the subgoal

```
F is N * F1
```

is encountered in the evaluation, N and F1 must be bound to a value. For example, suppose that N and F1 are bound to 4 and 6, respectively. If F is also bound, the value bound to it must be equal to the value evaluated by the expression (24, in the example). If F is not bound, it becomes bound to the value of the expression. You should examine the behavior of the PROLOG interpreter for the following query:

```
?- fact (3, 6).
```

In the evaluation process, both previous cases arise.

As the above discussion illustrates, PROLOG variables behave differently from variables of a conventional procedural programming language. As in functional languages, logic language variables can be bound to values, but once the binding is established, it cannot be changed.

```
fact (0, 1).
fac (N, F) :- N > 0, N1 is N - 1, fact ( N1, F1), F is N * F1.
```

FIGURE 98. Factorial in PROLOG

8.4 Functional programming versus logic programming

The most striking difference between functional and logic programming is that programs in a pure functional programming language define functions, whereas in pure logic programming they define relations. In a sense, logic programming generalizes the approach taken by relational databases and their languages, like SQL. For example, consider the simple PROLOG program shown in Figure 99, consisting of a sequence of facts. Indeed, a program of this kind can be viewed as defining a relational table; in the example, a mini-database of classical music composers, which lists the composer’s name, year of birth, and year of death. (See the sidebar on relational database languages and their relation to logic languages.)

In a function there is a clear distinction between the domain and the range. Executing a program consists of providing a value in the domain, whose corresponding value in the range is then evaluated. In a relation, there is no predefined notion of which is the input domain. In fact, all of these possible queries can be submitted for the program of Figure 99:

```
?- composer (mozart, 1756, 2001).
?- composer (mozart, X, Y).
?- composer X, Y, 1901).
?- composer (X, Y, Z).
```

In the first case, a complete tuple is provided, and a check is performed that the tuple exists in the database. In the second case, the name of the composer is provided as the input information, and the birth and death years are evaluated by the program. In the second case, we only provide the year of death, and ask the program to evaluate the name and year of birth of a composer whose year of death is given as input value. In the fourth case, we ask the system to provide the name, year of birth, and year of death of a composer listed in the database.

```
composer (monteverdi, 1567, 1643).
composer (bach, 1685, 1750).
composer (vivaldi, 1678, 1741).
composer (mozart, 1756, 1791).
composer (haydn, 1732, 1809).
composer (beethoven, 1770, 1827).
composer (schubert, 1797, 1828).
composer (schumann, 1810, 1856).
composer (brahms, 1833, 1897).
composer (verdi, 1813, 1901).
composer (debussy, 1862, 1918).
```

FIGURE 99.A PROLOG database

As most functional languages are not purely functional, PROLOG is not a pure logic language. Consequently, it is not fully relational in the above sense. In particular, the choice of the input domains of a query is not always free. This may happen if the program contains relational predicates, assignment predicates, or other extralogical features. For example the factorial program of Figure 98 cannot be invoked as follows

```
?- fact (X, 6).
```

to find the integer whose factorial is 6. The query would in fact fail, because the extralogical predicate is fails. Similarly, the following query

```
?- max (X,99, 99).
```

for the program fragment of Figure 97 does not yield a value less than or equal to 99, as the logical reading might suggest. It fails, since one of the arguments in the invocation of ð is not bound to a value.

sidebar on Relational database languages

A relational database can be viewed as a table of records called *tuples*. This form is quite similar to a logic program written as a sequence of ground terms. For example, the logic program of Figure 99 can be represented in a relational database as a table (relation) COMPOSER with fields NAME, BIRTH_YEAR, and DEATH_YEAR. The best known language for relational databases is SQL. In SQL, retrieval of data from the relational database is accomplished by the SELECT statement. Here are two sample SQL queries:

```
SELECT BIRTH_YEAR
FROM COMPOSER
WHERE NAME = "BEETHOVEN"
```

This query selects the field BIRTH_YEAR from a tuple in the relation COMPOSER such that the value of field NAME is BEETHOVEN.

The following query selects all tuples representing composers who were born in the 19th century:

```
SELECT *
FROM COMPOSER
WHERE BIRTH_YEAR Š 1800 and BIRTH_YEAR ð 1899
```

It is easy to see that the query language selects information stored in the database by specifying the logical (relational) properties that characterize such information. Nothing is said about how to access the information through suitable scanning of the database. It is interesting to note that earlier generations of database systems were imperative, requiring the user to state how to find the desired tuples through pointers and other such mechanisms. The current declarative approach is more oriented to the end-user who is not necessarily a computer programmer.

Logic and relational databases fit together quite nicely. In fact, extensions have been proposed to relational databases that add PROLOG-like rules to relational tables.

sidebar end***

8.5 Rule-based languages

Rule-based languages are common tools for developing expert systems. Intuitively, an *expert system* is a program that behaves like an expert in some restricted application domain. Such a program is usually structured as a *knowledge base (KB)*, which comprises the knowledge that is specific to the application domain, and an *inference engine*. Given the description of the *current situation (CS)*, often called the database, expressed as a set of facts, the inference engine tries to match CS against the knowledge base to find the rules that can be *fired* to derive new information to be stored in the database, or to perform some action.

An important class of expert system languages (called *rule-based languages*, or *production systems*) uses the so-called *production rules*. Production rules are syntactically similar to PROLOG rules. Typical forms are:

```
if condition then action
```

For example, the MYCIN system for medical consultation allows rules of this kind to be written:

```
if
    description of symptom 1, and
    description of symptom 2, and
    . . .
    description of symptom n
then
    there is suggestive evidence (0.7) that the identity of the bacterium is . . .
```

The example shows that one can state the "degree of certainty" of the conclusion of a rule. In general, the action part of a production rule can express any action that can be described in the language, such as updating CS or sending messages.

Supposing that knowledge is represented using production rules, it is necessary to provide a reasoning procedure (inference engine) that can draw conclusions from the knowledge base and from a set of facts that represent the current situation. For production rules there are two basic ways of reasoning:

- *forward chaining*, and
- *backward chaining*.

Different rule-based languages provide either one of these methods or both.

In order to understand forward and backward chaining, let us introduce a simple example described via production rules. The knowledge base provides a model of a supervisory system that can be in two different danger states, characterized by levels 0 and 1, indicated by the state of several switches and lights:

```
if switch_1_on and switch_2_on
    then notify danger_level_0.
if switch_1_on and switch_3_on
    then assert problem_1.
if light_red or alarm_on
    then assert problem_2.
if problem_1 and problem_2
    then notify danger_level_1.
```

An equivalent representation for the set of production rules is described by the and-or tree representation of Figure 100, which uses the convention introduced in Section 5.6.

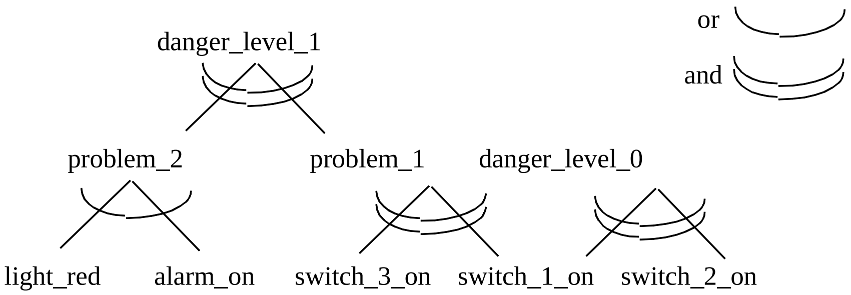


FIGURE 100. An and-or tree representation of production rules

Consider the following initial CS: the alarm is on and switches 1 and 3 are on. The inference engine should help the supervisory system determine the danger level. Forward chaining matches CS against KB, starting from leaf nodes of the and-or tree, and draws conclusions. New facts that are asserted by the rules are added to CS as the rules are fired. In our case, both problems 1 and 2 are asserted, and danger_level_1 is subsequently notified, since both problems 1 and 2 have been discovered.

Suppose now that the purpose of the reasoning procedure was to understand if we are in level 1 of danger. Forward chaining worked fine in the example, since the deduction succeeded. But in general, for a large KB, the same facts might be used to make lots of deductions that have nothing to do with the goal we wish to check. Thus, if we are interested in a specific possible conclusion, forward chaining can waste processing time. In such a case, backward chaining can be more convenient. Backward chaining consists of starting from the hypothesized conclusion we would like to prove (i.e., a root node of the and-or tree) and only executing the rules that are relevant to establishing it. In the example, the inference engine would try to identify if problems 1 and 2 are true, since these would cause danger_level_1. On the other hand, there is no need to check if danger_level_0 is true, since it does not affect danger_level_1.To signal problem 1, switches 1 and 3 must be on. To signal problem 2, either the alarm is on or the light is red. Since these are ensured by the facts in CS, we can infer both problems 1 and 2, and therefore the truth of danger_level_1.

Different expert system languages based on production rules are commercially available, such as OPS5 and KEE. It is also possible to implement production rules and different reasoning methods in other languages; e.g., in a procedural language like C++ or in a functional language like LISP. An implementation in PROLOG can be rather straightforward.

The main difference between logic and rule-based languages is that logic languages are firmly based on the formal foundations of mathematical logic, while rule-based languages are not. Although they have a similar external appearance, being based on rules of the form "if *condition* then *action*", in most cases rule-based languages allow any kind of state-changing actions to be specified to occur in the *action* part.

8.6 Bibliographic notes

The reader interested in the theory of logic, upon which logic programming is founded, can refer to (Mendelson 1964). (Manna and Waldinger 1985) present logics as a foundation for computer science. (Kowalski 1979) pioneered the use of logic in computer programming. (Lloyd 1984) provides the foundations for logic programming languages. PROLOG and the art and style of writing logic programs are discussed at length by (Sterling and Shapiro 1986).

(Bratko 1990) discusses the use of PROLOG in artificial intelligence applications. For example, it shows how PROLOG can be used to write a rule-based expert system along with different searching schemes (forward and backward chaining). Commercially available rule-based systems include KEE (***) and OPS5 (***).

Relational databses and the SQL query language are presented in most textbooks on databases, such as (Ullman ***). (Ceri et al. **) is an example of extension to relational databses to incorporate features from logic programming.

8.7 Exercises

1. Comment the following statement: "In logic programming, a program used to generate a result can be used to check that an input value is indeed a result." Discuss how existing programming languages approximate this general statement.
2. Find the most general unifier for
   f (X, g (a, Z, W), a, h (X, b, W))

   and

f (h (a, Z), g (a, h (Z, b), X), Z, h (d, b, a))

3. Given the PROLOG sort program of Section 8.1.2, show the search tree for the query ?-sort ([3, 5, 1], [1, 3, 5]).
4. Show the different computations of the PROLOG interpreter for the fragments of Figure 9.6, given the folowing facts:

rel (a, b).
rel (a, c).
rel (b, f).
rel (f, g).

5. Predicate even (n) is true for all even numbers. Write a PROLOG program implementing predicate even.
6. Write a PROLOG program which checks if a list contains another as a sublist. The program returns YES for queries of the following kind:

?-sublist ([1, 5, 2, 7, 3, 10], [5, 2, 7]).

What is the intended meaning of the following queries?

?-sublist ([1, 5, 2, 7, 3, 10], X).
?-sublist (X, [5, 2, 7]).
?-sublist (X, Y).

Does the behavior of the PROLOG interpreter correspond to the expected meaning if these queries are submitted for evaluation?

7. Consider the program of Figure 98. Discuss what happens if the following query is submitted:

fact (-5, X)

- If the program does not behave as expected, provide a new version that eliminates the problem.

8. Consider the following PROLOG program.

belongs_to (A, [A | B]) :- !.
belongs_to (A, [C | B]) :- belongs_to (A, B).

- what does this program do?
- can the cut be eliminated from the first clause without affecting the set of solutions computed by the program?

9. Consider the fragment of the previous exercise. Suppose you eliminate the cut from the first clause and, in addition, you interchange the two clauses. Describe the behavior of the PROLOG interpreter when the following goal is submitted:

?-belongs_to (a, [a, b, c])

10. The special goal fail is yet another extralogical feature provided by PROLOG. Study it and discuss its use.
11. It has been argued that the cut can be viewed as the logical counterpart of the goto statement of imperative programming languages. Provide a concise argument to support the statement.
12. Write a PROLOG program which defines the predicate fib (I, X), where I is a positive integer and X is the I-th Fibonacci number. Remember that the first two Fibonacci numbers are 0 and 1, and any other Fibonacci number is the sum of the two Fibonacci numbers that precede it.
13. Take the list of courses offered by the Computer Science Department and the recommended prerequisites. Write a PROLOG application that can answer questions on

the curricula; in particular, it should able to check whether a certain course sequence conforms to the recommendations.

14. Write a PROLOG program which recognizes whether an input string is a correct expresion with respect to the EBNF grammar illustrated in Chapter 2. You may assume, for simplicity, that expressions only contain identifiers (i.e., numbers cannot appear), and identifiers can only be single-letter names.
15. Suppose you are given a description of a map in terms of the relation from_to. from_to (a, b) means that one can directly reach point b from point a. Assume that from any point X one cannot return to the same point by applying the closure of relation from_to (i.e., the map contains no cycles). A special point, called exit, represents the exit from the map. Write a PROLOG program to check if, given a starting point, one can reach exit.
16. Referring to the previous exercise, write a PROLOG predicate to check if the assumption that the map contains no cycles hold; i.e., from any point X one cannot return to the same point by applying the closure of relation from_to.
17. Consider the fragment of Figure 97. Suppose that the second rule is changed in the following way:

```
max (X, Y, Y).
```

- Is the new fragment equivalent to the previous? Consider, in particular, the case of the following queries:

```
?- max (2, 5, A).
?- max (5, 2, B).
?- max (2, 5, 3).
?- max (2, 5, 5).
?- max (2, 5, 2).
```

18. Consider the PROLOG sort program discussed in Section 8.1.2. Discuss if (and how) the goal specified by the following query can be solved:

```
sort (X, [1, 3, 5, 99]).
```

19. Consider a problem of the kind shown in Figure 100. Write a PROLOG implementation that does both forward and backward chaining, as illustrated in Section 8.5.