Chapter 1

Introduction

This chapter is an introductory one. It introduces the basic concept of a *trustworthy compiler* used throughout the book. Also, it covers the basics of compilers and compilation from a modern viewpoint—different kinds of compilers and the phases of compilation—and summarizes compilers history and compiler development technologies.

In traditional meaning, a *compiler* is a program that translates the *source code* written in some *high-level language* (Pascal, C, C++, Java, C#, etc.) to *object* code—native machine code executable on some hardware platform (e.g., ×86, ×64, or Scalable Processor ARChitecture [SPARC]). In this sense, compilers seem to be an old and well-studied topic. However, right now, the concept of a compiler is much wider. Also, there are many reasons stimulating further growth of compilers and progress in compiling techniques, in particular:

- the development and wide spread of novel hardware architectures, like *multi-core*, *Very Long Instruction Word* (*VLIW*), *Explicit Parallelism Instruction Computers* (*EPIC*), and others, that require much more "intelligence" from the compiler than before, in particular, require from the compiler to perform *static parallelizing* of program execution and *scheduling* parts of hardware working in parallel;
- the progress and wide popularity of two novel software development platforms, Java and .NET, whose computing and compilation model

Trustworthy Compilers, by Vladimir O. Safonov

stimulated research in new approaches to compilation, *just-in-time* (*JIT*) and *ahead-of-time* (*AOT*), that make runtime performance of programs on those platforms more optimal;

- the support of *multi-language programming* on .NET platform. On that new platform, different modules of a large application can be developed in any languages implemented for .NET, for example, C#, Visual Basic.NET, Managed C++.NET, and others. For that reason, now we are witnessing a real *compiler development boom* for .NET platform (exceeding the previous compiler boom of the 1970s when a lot of new programming languages were invented): more than 30 languages are already implemented for .NET, and the number of implemented languages continues to grow. We are also witnessing a related *boom of compiler development tools* (ANother Tool for Language Recognition [ANTLR], SableCC, CoCo/R, and hundreds of others);
- the popularity of *Web programming* that stimulated the evolution of Web programming languages with extensive *dynamic typing* features (Perl, Python, Ruby, JavaScript/ECMAScript). In this relation, the task of their efficient implementation is very important;
- the popularity of *trustworthy computing* (TWC) and, therefore, the challenge to make compilers more trustworthy, verifying, and verified, as described below in this chapter and in Chapter 2;
- the rapid development of *mobile devices*, and the related progress in developing compilers for mobile platforms.

1.1 THE CONCEPT OF A TRUSTWORTHY COMPILER

Keeping in mind the TWC [1] paradigm that inspired this book as well as our previous book [1], one of the major issues in compiler development should be *how to make a compiler trustworthy* and *what is a trustworthy compiler*. Here is our vision of trustworthy compiling and of the concept of a trustworthy compiler, based on our own compiler experience, and on related heuristics and pragmatics.

A *trustworthy compiler* is a compiler that satisfies the following conditions (or, practically speaking, at least some of the conditions) listed below:

1 *A compiler that generates a trustworthy object code*. The users will trust the compiler, at first, if the object code generated by the compiler (any kind of native code, or virtual code) is trustworthy, since code generation is the primary task of any compiler.

2 *A compiler that demonstrates a trustworthy behavior*. The user will trust the compiler, which should do all jobs, from lexical analysis to code generation, with no surprises for the users—no hangs, unexpected stops, or other faults. For example, the user will not trust the compiler after

the first use, if the compiler suddenly issues an unclear message and stops compilation. This requirement looks evident, but the practice shows that such nontrustworthy issues are sometimes demonstrated even by industrial compilers.

3 *A compiler that uses trustworthy data* (sources, tables, intermediate code, libraries, etc.) and *protects them from occasional or intended corruption*. In other words, the *privacy* of all compiler input and intermediate data at any moment should be guaranteed; otherwise, the results of compilation can be falsified at some stage.

4 *A compiler whose design and implementation (source code) is trustworthy*, that is, satisfies the principles of TWC (see Section 2.1 for more details). The most radical and formal and probably the most difficult approach to making the code of the compiler trustworthy is to enable that the compiler is *verified*, following Leroy [6], that is, the compiler code meets its formal specifications. We'll consider Leroy's approach to verified compilers and other related approaches below in Sections 2.2 and 2.3. A simpler, more practical approach to make the compiler code trustworthy is to use *assertions* or *design-by-contract* when designing and developing the compiler code.

5 *A compiler that trustworthily performs error diagnostics and recovery* during all analytical phases of compilation—lexical analysis, parsing, and semantic analysis. As a counter-example, in our compiler practice, when testing some early version of a C compiler in mid-1990s, that compiler, in case of syntax errors like a missing bracket, issued a mysterious message of the kind: "*Syntax error before or at symbol: {"* ; then, after 20–30 lines of code, complained to be unable to compile any more and aborted the compilation. Compiler developers should keep in mind that such nontrustworthy compiler behavior jeopardizes the trust of users to compilers at all. The main principle in this respect, as shown in Chapters 3–5, is as follows: Even if the source code to be compiled contains any kind and any number of lexical, syntax, or semantic errors, the compiler should *issue reasonably clear and brief error diagnostics messages and continue the compilation until the end of the source code*, for the purpose of catching as many errors as possible.

6 *A compiler that works* as a *verifying compiler*, following Hoare's terms [7], that is, the compiler verifies the trustworthiness (including *semantic* correctness) of the source code it compiles, and does not allow nontrustworthy and nonverified source code to pass through. In an ideal case, the compiler should give to the user recommendations how to change the source code to make it trustworthy, rather than just complain of some error. Please see Section 2.2 for more details.

Actually, we don't know any compiler yet (at least in the industry) that would fully satisfy the above definition of compiler trustworthiness. Though,

as we show in the book below, there exist trustworthy compiling techniques implemented in real compilers.

1.2 KINDS OF COMPILERS

The variety of kinds of compilers being used now is amazing. It would be naive to think that there are only compilers to native or virtual machine code. Here are the most widely used kinds of compilers:

- *Traditional* (*classical*) *compilers* are those that compile a source code written in some high-level language (e.g., Pascal, C, or C++) to native machine code. The source language is usually strongly typed (see Chapter 5 for more details). This kind of compilers is covered by classical compiler books such as Dragon Book [5]. From a novel viewpoint, this scheme of compilation can be referred to as *native* compilation. As proved by our experience of research and development work, even in the area of classical native compilers, it is quite possible to make a lot of innovations.
- *Interpreters* [8] are yet another way to implement programming languages. Instead of translating the source code to some machine code, the interpreter *emulates* (*models*) the program execution in terms of the source code, or of some high-level intermediate code to which the program is compiled before its interpretation. An interpreter approach is known to slow down runtime performance of a program, in average, in 100–1000 times, as compared with an equivalent program compiled to native code. However, there are languages that intensively use dynamic types, dynamic data structures, and program extension features: for example, the old symbolic information processing languages LISP and SNOBOL, and a newer language FORTH widely spread in the 1980s and based on the ideas of using postfix notation as a programming language, and an extensible set of *commands*—primitive language constructs. The nature of such dynamic languages makes their interpretation much simpler and adequate than compilation. It's worth mentioning in this relation that Java 1.0 was also implemented in 1995 as a pure interpreter of Java bytecode. JIT compiler was added to Java implementation later, since Java 1.1.
- *Cross-compilers* are compilers that work on one (typically more powerful and comfortable, like ×86 or ×64) hardware platform and generate code for another target hardware platform (typically, an embedded microprocessor with limited resources).
- *Incremental compilers* [9], very popular in the 1970s, are compilers that allow the users to split the source code of a program to *steps*. A step can be a definition, a declaration, a procedure or function header, a statement, or a group of statements. The source program can be entered into an incremental compilation system step-by-step. The selected steps can

be edited and recompiled. The resulting program can be tried and tested for execution, even if not all its steps are already developed. Such approach is comfortable for incremental program development and debugging, but the efficiency of the object program is poor, since such programming system has to use an *interpreter (driver) of steps* at runtime.

- *Converters* are compilers from one high-level language source code to another. They are used for *reengineering*, to port programs from older languages (e.g., COBOL) to newer ones (e.g., Java or C#). Another typical reason of using this approach is a research targeting to extend a programming language with some advanced (e.g., *knowledge management*) features: In this case, the extensions can be implemented by conversion to the basic language constructs and specific API calls. We use this approach in our Knowledge.NET [10] knowledge management toolkit for .NET based on C# extension by knowledge representation constructs.
- *JIT compilers* are compilers that work at runtime and compile each first called method from the intermediate code to the native code of the target platform. JIT compilers are inherent parts of Java and .NET technologies. They are covered in Chapters 6–8.
- *AOT compilers* (or *precompilers*) are also used in Java and .NET to avoid JIT compilation and to improve runtime performance: Those compilers translate platform-independent intermediate code to native code prior to program execution and work similarly to traditional compilers for more conventional languages like C or Pascal.
- *Binary compilers* are compilers directly from binary code of one platform to binary code of another one, without using the source code. Binary compilation is used as a method to port applications from older hardware platforms to newer ones.
- *Graph compilers* are compilers that compile some graph-like representation of a program (rather than its source code) to other graphs, of to native or virtual code; this novel kind of compilers is covered in Chapter 9.

1.3 EVOLUTION OF JAVA COMPILERS

Requirements of modern programming—the need to make an object code platform independent and the need to support rapidly dynamically changing program structure—lead to more complicated scheme of implementing modern languages like Java than traditional native compilation.

The first version of Java implementation (Java 1.0) that was made available to software developers in 1995, included the *compiler from source Java code to bytecode*—intermediate code of the *Java Virtual Machine* (*JVM*) based on postfix notation, and the *JVM* implemented as a pure *interpreter* of Java bytecode. It was kind of a surprise to experienced software developers who had

used highly optimizing compilers for decades before Java appeared. The runtime performance of programs in Java 1.0 based on pure interpretation model was poor.

So, the second version of Java, Java 1.1, shipped in 1996, for the purpose of improvement of runtime performance, included the first Java JIT compiler as part of JVM, alongside with the bytecode interpreter.

The next major version of Java, Java 1.2, which appeared in 1998, made further steps toward making runtime performance of Java applications more optimal and comparable to that of natively compiled applications. Java 1.2 included *HotSpot performance engine*—an enhancement to JVM based on a *profiler* to determine "hot spots" in Java applications—the most often called and resource-consuming methods. Those "hot" methods were JIT compiled, whereas the rest of the methods remained in bytecode representation. The average runtime performance increase due to using HotSpot appeared to be two times, as compared with their performance in the previous version of Java.

It should also be mentioned that, although Sun's Java implementation doesn't contain Java native compilers, many integrated development environments targeted to Java software development (e.g., Borland JBuilder) support native compilation of Java programs as an option. So, native compilation (or AOT compilation) is now used as an alternative way of Java implementation.

1.4 COMPILATION FOR .NET

.NET is a multi-language software development platform. Several dozen languages are already implemented for .NET, and their number continues to increase. The most popular languages used in .NET environment are C#, Visual Basic.NET, and Managed C++.NET, implemented by Microsoft.

The implementation of a language for .NET is based on a similar principle as Java implementation. The compiler translates the source code to *Common Intermediate Language* (*CIL*) code, also known as *Microsoft Intermediate Language* (*MSIL*). The architecture of CIL is similar to Java bytecode and based on postfix notation. Alongside with CIL code, the compiler generates *metadata*—information on the types defined and used in the compilation unit. Then, at runtime, each first called method is JIT compiled to native code. The important distinction of .NET approach from Java is the open nature of .NET—its principles stimulate compiler developers to implement more and more languages, whereas Java technology prescribes using the Java language only; the only alternative of using other languages with Java technology is to use *native methods* that should be implemented in C or C++.

Microsoft's implementation of .NET provides the *ngen* (native generator) utility to precompile the code to avoid JIT compilation.

The .NET's *Common Language Runtime* (*CLR*) supports *managed execution* mode, with full runtime type-checking, security checking, memory management, and garbage collection.

A common principle of developing compilers for .NET is to leave all or most of the optimizations to the JIT compiler.

One of the most important results in .NET area, from the viewpoint of trustworthy compiling, was the development by Microsoft Research of a novel programming system *Spec#* [11], an extension of the C# language by formal specifications in *design-by-contract* [12] style, with a built-in verifier to prove correctness of programs in Spec#. Actually, Spec# can be considered the first worldwide known trustworthy compiler.

1.5 PHASES OF COMPILATION

After the above short introduction to modern compilers and tools, let's summarize the *phases* (*steps*) of compilation.

The formal model of the process of compilation can be represented as the five main successive phases: *lexical analysis*, *parsing* (or *syntax analysis*), *semantic* (or *context-dependent* or *type-dependent*) *analysis*, *code optimization* (as an optional phase), and *code generation*. Each phase takes, as the input, the result of the previous phase and passes its output to the next phase.

Actually, this model is formal, simplified, and nonoptimal, and the architecture of real compilers can differ from it.

The purpose of the *lexical analysis* phase is to translate the *source code* (the program code written in a high-level language and represented as a text file or a logically related group of text files) to *stream of tokens*. A *token* is a primitive unit of a programming language—*identifier* (*name*), *keyword*, *number*, *character literal*, *string literal*, *operator symbol*, or *delimiter* (e.g., left and right parentheses). Lexical analyzer ignores *white space characters* and *comments* in the program source code, and processes *pragmas*—special instructions inserted in the source code to control the compiler behavior and switch on or off its options. Also, during lexical analysis, some (relatively minor) *error diagnostics and recovery* can happen, for example, in case a number has too many digits. Although, from a formal compilation model viewpoint, lexical analysis is a separate compilation phase, in most of the compilers, to save compilation time, lexical analysis is implemented as a *subroutine*, *method*, or *function* to be called by the parser each time it needs the next token from the source code.

The goal of the next phase, *parsing*, is to translate the sequence of tokens, the output of lexical analysis, to a *parse* (or *derivation*) *tree* representing the syntax structure of the program to be compiled. Usually, parsing is based on representing the model of the program syntax by *context-free grammars*. Traditional parsing techniques and algorithms are covered in classical compiler books [5]. Another important goal of parsing is *syntax error diagnostics and recovery*, since program developers often make mistakes like missing bracket, or semicolon, or keyword. The task of the parser in such situations is to keep a trustworthy behavior, provide clear diagnostics, and parse the

erroneous program code up to its end, for the purpose of catching as many more syntax errors as possible.

The next phase, *semantic analysis*, is intended to translate the parse tree to *intermediate representation* (*IR*), more comfortable for the code generator, for example, to *postfix notation*. During this phase, all compile-time checks are performed that couldn't be made at the previous, parsing phase. For the first turn, the semantic analyzer performs *lookup*—for each *applied occurrence* of an identifier, it finds the appropriate *definition* of that identifier, if any. The next major subtask of semantic analysis is *type-checking*—it checks that, in each operation, the types of the operands are appropriate. Finally, the semantic analyzer translates the program code to IR. Surely during this phase, a lot of bugs can be found, so an important task of the semantic analyzer, similar to that of the parser, is to provide *semantic error diagnostics and recovery*, which is an inherent part of a trustworthy compiler.

The next, optional, phase of the compiler is *code optimization*. This phase is usually turned off by default but is very important to enable better runtime performance of the resulting object code. The optimization phase usually works as follows: It takes the IR of the program and generates the optimized IR. Optimization includes solving a number of interested and complicated mathematical problems described in the excellent book by Muchnick [13].

The final phase of the compiler is *code generation*. It takes the IR of the program and generates its *object code*—either *native code* of the target platform (e.g., ×64 or SPARC), or *virtual code* of some *virtual machine* that performs runtime support of the program on the target platform (e.g., Java bytecode or .NET CIL code).

1.6 OVERVIEW OF COMPILER DEVELOPMENT PRINCIPLES AND TECHNOLOGIES

As we've already seen, the architecture of a compiler is complicated and consists a lot of phases. So, compiler developers need to apply specific design principles and technologies to make the development easier and more systematic.

Front-end and back-end. The first compiler design principle used in practically all compilers is the separation of the compiler to two major parts—*front-end* and *back-end* (recall the allegoric picture on the front cover and its explanation in the Preface). Of the five compiler phases explained in Section 1.5 above, the *front-end* of the compiler consists of the three first, source language-dependent phases—*lexical analysis*, *parsing*, and *semantic analysis*. The *back-end* of the compiler is the collection of the two remaining phases, the *optimizer* and the *code generator*. The front-end of the compiler can be otherwise referred to as the *source language-dependent* part, and the back-end as the *target platform-dependent* part. Why is it so convenient to represent the compiler in such a way? Because,

typically, for any hardware platform, a whole *family of compilers* needs to be developed—say, from *M* source languages—C, C++, Pascal, FORTRAN, and so on. Due to the separation of compilers to front-ends and back-ends, we can develop *M* front-ends to use the same common IR, and only *one back-end*, common for the whole family of compilers, that takes the IR and generates a native code for the target platform. If we have *N* target platforms and need to develop the compilers from all *M* source languages for all of them, using this approach, we should develop *M front-ends* and *N back-ends* only, that is, *M* + *N* major compiler components, instead of *M* * *N* components in case we use a straightforward approach and develop each of the front-ends and back-ends from scratch, without any kind of code reusability among them. A good example of such family of compilers developed according to such principles is *Sun Studio* [14], compilers from three languages C, C++, and FORTRAN, working on two target platforms, *Solaris* (on SPARC and ×86/×64 machines) and *Linux* (on ×86/×64 machines). To our knowledge, the best example of a toolkit for developing optimizing compiler back-ends for an extendable set of target platforms is Microsoft Phoenix [3] covered in Chapter 10.

One-pass versus multi-pass compilers. To make the work of developing a compiler more modular, and to parallelize it between several developers, it is quite common to implement the compiler as *multi-pass*. With this scheme, each phase of the compiler (except for lexical analysis) is implemented as a separate *pass*, that is, as a module that analyzes the whole source program (represented by the source code or any kind of intermediate code) and converts it to some output IR suitable for the next pass. As we'll see later on, for most source languages, it is not possible to make the compiler one-pass because of the specifics of the language, typically because of using identifiers or labels defined later in the source code. A legend on an old PL/1 compiler says that the number of its passes was equal to the number of employees in the appropriate department of the compiler developer company.

As an alternative to multi-pass approach, for some simpler languages (e.g., for Pascal), compilers are developed as *one-pass*, for the purpose to make the compiler faster. For one-pass compilers, to resolve the situations like forward references mentioned above, the technique of *applied occurrences lists* can be used (more details in Chapter 5). With the one-pass compiling scheme, the compiler source code combines fragments of different phases in a procedure or method that compiles some source language construct (e.g., *if* statement)—lexical analysis, parsing, semantic analysis, and code generation for the construct are intermixed in one compiler module. Typical compiling technique used in one-pass compilation is *recursive descent* (see Chapter 4). It should be noted, however, that such compiler architecture may be dangerous, since it can lead to design flaws of *temporal cohesion*, speaking in terms by Myers [15]: If in

a compiler module's source code the boundaries are not clear between the statements implementing fragments of one phase and those implementing the other phase, the risk of making a bug of omitting some of the statements is high.

Bootstrapping. Another popular technique of compiler development is referred to as *bootstrapping*, which, as applicable to compilers, means that the source language is used as a tool for developing a compiler from the same language, so *the compiler is used to compile itself*. More exactly, the developer of the compiler uses, as a development tool, the version of the compiler for the subset of the language to be implemented, or uses some other, maybe less efficient, version of the compiler for the same language to compile the source of the "good" compiler written in the same source language.

The first method of bootstrapping is as follows. Let L be the source language to be implemented on the target platform T (let's also denote T as the assembly language of the target platform). To make the task easier, we choose some subset of L referred to as L_0, and develop the compiler from L_0 to T in T. So now we have a tool for further development, more comfortable than the assembly language T. Next, we develop a compiler from some larger subset L_1 of the language L in L_0, and so on. Finally, for some n, we develop the compiler from $L_n = L$ in L_{n-1}, and our job is done. As compared with the straightforward solution—writing a compiler from L to T in low-level language T—due to the use of the bootstrapping technique, we only use low-level language at the first step, when writing the compiler from L_0 to T in T.

The second variant of bootstrapping is used to gradually improve the quality of the compiler. Using the above notation, our first step will be to develop a "good" compiler from L to T written in L. Let's denote that compiler as C_0. Due to the use of high-level language as a tool, it will be easier for us to enable C_0 to generate efficient object code, perform various optimizations, and so on. But C_0 cannot work yet, since we didn't provide a way to translate it into T. To do that, we perform the second step—develop a "poor" compiler C_1 from L to T written in T (with no optimizations and object code quality improvements, just a compiler with straightforward code generation). Next, we perform the first bootstrapping—compile C_0 by C_1. The resulting compiler (let's denote it as C_2) should be as good as C_0 from the viewpoint of the code quality it generates, but this version of the compiler is probably not so fast (since its source was compiled by the poor compiler C_1). To improve the compiler's efficiency, we perform the second bootstrapping—compile C_0 by C_2 and so on. I know Pascal compiler developers who were as patient and persistent as to make 10 bootstraps to improve their compiler written in Pascal. They claimed that the quality of the compiler became better with each bootstrap.

The bootstrapping techniques described above were especially important for older hardware platforms that were lacking high-level languages already implemented on those platforms, ready to be used as tools for compiler development. Nowadays, the situation is quite different: There are enough high-level languages implemented on any platform appropriate for use as compiler writing tools. In practice of modern research and commercial compiler projects, most of the compilers are written in C of C++. Also, to make a compiler or a compiler development tool portable, it has become a common practice to write it in Java.

Compiler compilers. Due to the need of developing more and more compilers, in the late 1960s, the idea of a *compiler compiler* was coined by compiler experts. A compiler compiler is a tool that takes a formal definition of syntax and semantics of a programming language and generates a ready-to-use compiler from that language. Theoretical model of compiler compiler is based on *attributed grammars* [16], a formalism invented by Knuth to combine formal definitions of programming language syntax (by a *context-free grammar*) and semantics (by *semantic actions* attached to each syntax rule to evaluate *semantic attributes* of the grammar symbols participating in the syntax rule). Attributed grammars are described in detail in Chapter 5. In the 1970s, attribute-based compiler compiler projects became very popular. Among them, there were DELTA [17] developed in France, and the Soviet system SUPER [18] developed in the Computing Centre of the Russian Academy of Sciences. The most widely known compiler compiler is *YACC* [19] by Johnson, still used in many industrial compilers, for example, in Sun Studio, and stimulating development of a lot of similar compiler compilers, like *bison* [20] and newer compiler development tools—*JavaCC* [21], ANTLR [22], CoCo/R [23], and SableCC [24] covered in Chapter 4. In short, the approach of tools like YACC is different from the approach of tools like DELTA, since YACC is more practical. The goal of tools like DELTA was to formalize the syntax and the semantics of the source language completely, which is a more complicated task than implement part of the compiler "by hand." Instead, YACC offers a combination of formalized syntax (for automated parser generator) with informal semantic actions, to be written in the compiler implementation language like C, attached to each syntax rule. So YACC's approach provides more flexibility, in particular, in using any other compiler development tools, any algorithms of semantic analysis and code generation. Actually, one of modern approaches to compiler development is to automatically generate lexical analyzer and parser, based on tools like YACC or ANTLR, and develop the semantic analyzer by hand, using its API calls in semantic actions of the grammar. As for code generator and optimizer, there are modern tools like Phoenix that enable automation of developing those phases also.

Technological instrumental package (TIP) technology [1,2,25]. In our own practice in the 1970s and 1980s, for the compiler development for "Elbrus" [2] computers, and later in the 1990s, for the compiler development for Sun, we used our own compiler development technology referred to as *TIP technology*, an enhancement of modular programming and abstract data types. From a modern viewpoint, TIP technology can be characterized as modularization of the compiler, enhanced by using predefined design and implementation schemes for compiler architecture. With TIP technology, a compiler is designed and implemented as a hierarchy of *TIPs*, each of them implementing a set of operations on some data structure used or generated by the compiler—identifier table, table of type definitions, IR to be generated by the front-end, and so on. Each TIP is designed according to the following predefined scheme of abstraction layers and vertical cuts (groups of operations). The *abstraction layers* of a TIP are as follows:

- *representation layer*—a set of operations on the data structure in terms of elements of its concrete representation;
- *definition layer*—a set of operations on the data structure in terms of intermediate concepts;
- *conceptual layer*—a set of operations on the data structure in more adequate and abstract terms, convenient for the user (other compiler developer) who wants to work with the data structure in terms suitable for the task being solved.

The *vertical cuts* (groups of operations) of the TIP are:

- *generation interface*—operations that generate elements of the data structure;
- *access interface*—operations that access elements of the data structure;
- *update interface*—operations that update elements of the data structure;
- *output interface*—operations that output elements of the data structure in symbolic form, or, speaking more generally, convert elements of data structure to some other output format, for example, object code.

The lower layer of the TIP is its *concrete representation* implemented by a group of definitions and declarations.

Each operation of the TIP is implemented by a method, function, procedure (subroutine), or macro definition. Only the conceptual layer of the TIP is directly accessible for the user; other layers are encapsulated in the TIP definition. The TIP is implemented bottom-up, from the representation layer to the definition layer, and then to the conceptual layer. Each layer n uses operations of the layer $n - 1$ only.

For example, let's consider the *Types* TIP representing a table of type denotations (see Chapter 5 for more details). Its representation layer and generation interface can include the operation *TGetMem* (*size*) that returns a pointer to the memory area (table element) of the given size. The definition layer can include the operation *TGenField* (*id*) that generates the element representing a record field with a given identifier. The access interface, at the conceptual layer, can include an operation like *TGetField* (*rec*, *id*) that seeks in a record type a field with the given identifier, and returns a pointer to the field found (if any), or *nil* if not found.

The readers should take into account that the TIP technology was invented in mid-1980s and applied for an implementation language of C level. Our experience with TIP technology has shown that it helped to increase the productivity of the developers and to improve the readability and reliability of the compiler's source code. Due to the application of the TIP technology, we developed a whole family of compilers for "Elbrus" computers—from Pascal, CLU, Modula-2, BASIC, ALGOL (incremental compiler), SNOBOL, FORTH, and REFAL (interpreters) in a few years, with a small group of less than 10 people. Our experience was so successful that we also applied the TIP technology to refurbish the Sun Pascal compiler later in the 1990s [25].

We do think that TIP technology is still quite applicable when developing compilers in C or C-like languages. It is important for compiler development, since it helps to make the design and the source code of the compiler more clear, systematic, more easily maintainable, and therefore more trustworthy.

1.7 HISTORY OF COMPILER DEVELOPMENT IN THE U.S.S.R. AND IN RUSSIA

The history of compiler development is described in classical compiler books [5]. It is well known that the first compilers of the 1950s were written in low-level assembly languages. Later, in the 1960s and 1970s, high-level languages like FORTRAN and C started to be used for compiler development. Among worldwide classicists of theory and methods of compiler development, the first to be mentioned are Alfred Aho, John Hopcroft and Jeffrey Ullman [5], and David Gries [26].

However, the history of compiler development in U.S.S.R. and Russia may appear to be less known to the readers, so in this section, we'll summarize it.

The first widely known Soviet compiler from ALGOL-60 for Soviet computers M-20, M-220, and M-222 was developed in 1962 and named TA-1M (for *Translator from Algol, version 1, Modernized*) [27]. The scientific advisor of this project was my teacher, Dr. Svyatoslav S. Lavrov. This compiler (in terms used in the 1960s—*translator*), as well as the ALGOL-60 language itself,

played an outstanding part in university teaching and scientific computing in the U.S.S.R. When I was a student in mid-1970s, I wrote my educational programs in ALGOL and used TA-1M to compile them (the program source code was input into the computer from a deck of punched cards), as many thousand of Soviet students did. It appeared that, during several years of the TA-1M compiler use, we haven't experienced any compiler bug or issue. This is a good example of compiler implementation quality, even for modern software development companies (recall that TA-1M was an academic project). Due to the wide popularity of ALGOL compilers in the U.S.S.R. and the enthusiastic activity by Dr. Lavrov, ALGOL-60 became, for many years, the most popular programming language for university teaching, and the first programming language for most Soviet students to learn in the 1960s–1970s. Please note that the situation in our country differed from that in the U.S.A., Canada, and Western Europe, where, for many years, FORTRAN, COBOL, and BASIC had been used for university teaching.

Another interesting Soviet compiler systems of the 1960s and 1970s were ALPHA [28] and BETA [29], developed in Novosibirsk, Academgorodok, as the result of the projects supervised by Dr. Andrey Ershov who later became the first Soviet academician in IT. One of the leaders of those projects was also Dr. Igor Pottosin. ALPHA was a compiler from the extension of ALGOL and performed a lot of code optimizations. BETA was an attributed compiler compiler (see Section 1.6); in addition, it used a universal internal representation of source programs, common for all compilers generated by BETA. The latter decision allowed the authors of BETA to use a common back-end within the BETA toolkit.

We already mentioned in Section 1.6 the SUPER-attributed compiler compiler developed in the 1970s in the Computing Center of Russian Academy of Sciences, Moscow. The project was supervised by Dr. Vladimir M. Kurochkin.

A really amazing compiler project in the 1970s was developed at our St. Petersburg University—implementation of ALGOL-68 for Soviet clones of IBM 360 [30]. The project was lead by Dr. Gregory S. Tseytin who gathered a team of about 20 best compiler experts from our university. Due to the enthusiasm of the developers, the compiler and the ALGOL-68 language were used for about 10 years for university teaching at our university, and for large commercial and scientific software projects performed by our university specialists.

As one of the most famous compiler projects in the U.S.S.R., we should also mention our own projects on developing a family of compilers and interpreters for Soviet "Elbrus" computers [2]. We developed in a few years, by a small team of less than 10 young software engineers, the following translators: compilers from Pascal, CLU (that was the first CLU implementation in the U.S.S.R. completed in 1985), Modula-2, and BASIC; interpreters of LISP, SNOBOL-4, FORTH-83, and REFAL [31]; and a translator from system programming language ABC [32] designed by Dr. Lavrov. This project started

with the supervision of Dr. Lavrov and continued under my supervision. It made our team well known in the country. Our compilers for "Elbrus," especially Pascal, were used in major industrial and academic organizations of the U.S.S.R., including the Spaceship Control Center near Moscow. Other compiler groups for "Elbrus" who worked in Moscow, Novosibirsk, and Rostov developed the compiler from EL-76 [33], the high-level language used by "Elbrus" system programmers; compilers from SIMULA-67, C, FORTRAN, COBOL, Ada, and PL/1; and an interpreter of PROLOG. Most of the worldwide known languages, a total of about 30, were implemented for "Elbrus." That phenomenon can only be compared with the current work on implementing a variety of languages for .NET. The "Elbrus" compiler experience and the TIP technology we applied in those compiler projects became a good basis for our subsequent work with Sun on compilers for the SPARC platform. Our original methods we used in our "Elbrus" compilers constitute an important part of this book.

EXERCISES TO CHAPTER 1

1.1 What is a trustworthy compiler, and what do you think is especially important to enable to make a compiler trustworthy?

1.2 In what respects and why Java 1.2 and Java 1.1 compilation schemes differ from those of Java 1.0, and what is the effect on changing Java compilation schemes on runtime performance?

1.3 Please try to classify the compilers you have ever used on the basis of our classification of compilers in Section 1.2.

1.4 What are the main characteristics of the .NET approach to compilers and language implementation, and why do you think .NET is so attractive for compiler developers?

1.5 What are the compiler front-end and the compiler back-end? Please determine and explain how to name, use, and control the front-end and the back-end of the compiler you are using in your programming practice.

1.6 Please list the phases of compilation; give each of them a short characteristic, and summarize the most important (in your opinion) features and mechanisms in each phase to make a compiler trustworthy.

1.7 Why doesn't a one-pass scheme of compilation work for most programming languages to be implemented?

1.8 What is the idea and purpose of bootstrapping technique for compiler development?

1.9 How does a compiler compiler work? What is its input and output? What is the theoretical basis of a compiler compiler?

1.10 Please list and explain the meaning of the abstraction layers and vertical cuts in TIP technology.