

Chapter 2

Compilers and Interpreters

Before looking at the details of programming language implementation, we need to examine some of the characteristics of programming languages to find out how they are structured and defined. A compiler, or other approach to implementation, is a large and complex software system and it is vital to have some clear and preferably formal structure to support its construction.

This chapter examines some of the approaches that can be used for high-level programming language implementation on today's computer hardware and provides some of the background to enable high-level to low-level language translation software to be designed in a structured and standard way.

2.1 Approaches to Programming Language Implementation

The traditional approach for the implementation of a programming language is to write a program that translates programs written in that language into equivalent programs in the machine code of the target processor. To make the description of this process a little easier, we shall assume that the source program is written in a language called *mylanguage* and we are producing a program to run on *mymachine*. This view is shown in Fig. 2.1.

This trivial diagram is important because it forces us to consider some important issues. First, what precisely is the nature of the source program? It is obviously a program written in *mylanguage*—the language we are trying to implement. But before we can contemplate an implementation of the translator software, we have to have a precise definition of the rules and structure of programs written in *mylanguage*. In Sect. 2.2, we consider the nature of such a definition.

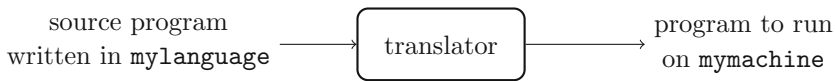


Fig. 2.1 A simple view of programming language implementation

Second, what sort of programming language is `mylanguage`? At this stage it may not really matter, but the nature of the language will of course affect the design of the translator software. For example, if `mylanguage` is a high-level programming language and `mymachine` is a hardware-implemented processor, then the translator program is usually called a *compiler*. This book concentrates on this particular configuration. If, however, `mylanguage` is an assembly language (for `mymachine`) then the translator is usually called an *assembler* and would be significantly easier to implement than a compiler.

Can programs in `mylanguage` be passed directly into the translator or does it make more sense to preprocess the programs first? For example, programs written in C can include language features best dealt with by a preprocessor. This stage could of course be regarded as being an integral part of the translation process.

Third, what sort of language is used to express the program to run on `mymachine`? Again if `mymachine` is a hardware-implemented processor, then we are probably looking at the generation of machine code programs encoded in some object file format dependent on the design of `mymachine` and probably on the operating system running on `mymachine`. Generating assembly language may be the right thing to do, requiring the existence of a separate assembler program to produce code directly runnable on `mymachine`. And there are other possibilities too. A popular approach to language implementation assumes that `mymachine` is a *virtual machine*. This is a machine for which there is no corresponding hardware and exists only as a consequence of a piece of software which emulates the virtual machine instructions. This approach is examined in Sect. 2.1.1. Furthermore, the translator could have a somewhat different role. It could be used to translate from `mylanguage` to a high-level language rather than to a low-level language for `mymachine`. This would result in a software tool which could be used to translate programs from one high-level language to another, for example from C++ to C. In this case, aspects of the internal design of the translator may be rather different to that of a conventional compiler and there is an assumption that `mymachine` somehow runs programs written in the target high-level language. There are many common principles and algorithms that can be used in all these language translation tasks. Whatever the form of `mymachine` and the generated code, a precise specification is required, just as for `mylanguage`.

At the beginning of this section is a statement that the translator generates “equivalent programs” for `mymachine`. This is an important issue. The translator should preserve the semantics of the `mylanguage` program in the running of the generated code on `mymachine`. The semantics of `mylanguage` may be specified formally or informally and the user of `mylanguage` should have a clear idea of what each valid program should “mean”. For example, translating the statement `a = a + 2`

into code that increments the value of `a` by 3 is not right, certainly for a sensible programming language! The translator should translate correctly.

Assuming that `mylanguage` and `mymachine` are both non-trivial, the translator is going to be a complex piece of software. It is the role of this book to help explain how this software can be structured to make it feasible to produce a reliable translator in a reasonable time. We concentrate in this book on the structure of compilers and later in this chapter a traditional internal structure of a compiler is described. But it is helpful now to say that a compiler can be built of two distinct phases. The first is the *analysis phase*, reading the source program in `mylanguage`, creating internal data structures reflecting its syntactic and semantic structure according to the definition of `mylanguage`. The second is the *synthesis phase*, generating code for `mymachine` from the data structures created by the analysis phase. Thinking about a compiler in terms of these two distinct phases can greatly simplify both design and implementation.

2.1.1 *Compile or Interpret?*

Figure 2.1 illustrates the conventional view of a compiler used to generate code for a target machine from a source program written in a high-level language. This book concentrates on the design of this type of translator, how it can be structured and implemented. The term *compiler* is usually used for a translator from a high-level programming language to a low-level language such as the machine code of a target machine. However, as we have seen, the term is sometimes used to cover translation from and to a wider range of programming language types, such as high-level language to another high-level language.

When a program is passed through a compiler generating code for some target machine (say, the `mymachine` processor), the code can be run on the `mymachine` architecture and this has the effect of “running” the original program. The processor hardware *interprets* the machine instructions generated by the compiler and the cpu state is altered according to the nature of the sequence of instructions executed. However, other implementation routes are possible and in particular there is no fundamental necessity for the instructions generated by the compiler to be interpreted directly by the hardware.

There are many implementations of high-level programming languages where the compiler generates code for a *virtual machine* and then a separate program, the *interpreter*, reads the virtual machine code and emulates the execution of the virtual machine, instruction by instruction. At first sight this may seem a strange thing to do—why not generate target machine code directly? But this approach has several significant advantages, including the following.

- The design of the code generated by the compiler is not constrained by the architecture of the target machine. This can simplify the design of the compiler because it does not have to deal with the quirks and restrictions of the real hardware. For

example, it may be convenient to use a stack-based architecture for the virtual machine, not directly supported by the target hardware.

- Portability is enhanced. If the interpreter is written in a portable language, the interpreter and the virtual machine code can be shipped and easily run on machines with different architectures or operating systems. This ties in well with today's prevalence of heterogeneous networked environments.
- The virtual machine code can be designed to be particularly compact. There are application areas where this may be very important.
- Runtime debugging and monitoring features can be incorporated in the virtual machine interpreter allowing improved safety and security in program execution. The virtual machine code can run in a sandbox, preventing it from performing illegal operations. For example, the virtual machine can operate on typed data, and runtime type checking can provide helpful debugging information.

The obvious disadvantage of this approach concerns the question of efficiency. Interpreted code is likely to be slower than native execution. But for most applications this turns out not to be of real significance. The advantages usually easily outweigh this disadvantage and so many modern programming languages are implemented in this way.

The nature of the virtual machine poses interesting questions. The design of the virtual machine should not be too close to that of the hardware because the advantages of compiler simplification essentially disappear. But if the virtual machine's design is made very close or identical to the language that is being implemented, the compiler is made very simple, but the interpreter has to deal with the detail of decoding and analysing this potentially complex code. The functions of the compiler are being shifted into the interpreter. However, several languages have been implemented successfully in this way, where the interpreter does all the work, removing the necessity for the separate compiler. Some implementations of the BASIC language have been implemented in this way. Because of the need for repeated analysis of the source language statements in the interpreter, this is rarely a practical approach for a production system.

These implementation issues are examined again in Chap. 9.

2.2 Defining a Programming Language

The definition of a programming language is fundamentally important to users of the programming language as well as to the compiler writer. The definition has to provide information about how to write programs in the language. It has to specify the set, presumably infinite, of valid programs and also what each valid program "means". The specification of *syntax* is central here. Syntax defines the sequences of characters that could form a valid program. And the meaning of these programs is specified by the *semantics* of the language.

The language definition should be clear, precise, complete, unambiguous and preferably understandable by all language users. Specifying a programming language in an informal way using a language such as English makes the definition accessible, but precision can suffer. Aspects of many programming languages have been defined using natural language and ambiguities have been common, particularly in early revisions of the definitions. The definition of the language's syntax is usually done using a more formal approach and a range of *metalanguages* (languages used to define other languages) have been developed. A few of these metalanguages are described in Sect. 2.2.1. For most of the current and popular programming languages, the use of a simple metalanguage to define syntax results in a compact and largely complete syntactic specification. We shall see in Chap. 4, how this specification can be used as the starting point for the design of a syntax analyser for the language.

2.2.1 BNF and Variants

Backus–Naur Form or *Backus Normal Form* is a metalanguage which was popularised by its use in the definition of the syntax of ALGOL 60 [1]. It is a very simple yet powerful metalanguage and it has been used extensively to support the formal definitions of a huge range of languages. It has become one of the fundamental tools of computer science.

A BNF specification consists of a set of rules, each rule defining a symbol (strictly a *non-terminal symbol*) of the language. The use of BNF is best illustrated by some examples. This first example (see Fig. 2.2) makes use of some of the simpler rules and terminology of English grammar to define the syntax of some trivial sentences.

```

<sentence> ::= <subject> <verb> <object>
<subject> ::= <article> <noun>
<object> ::= <article> <noun> | <article> <adjective> <noun>
<verb> ::= watches | hears | likes
<article> ::= a | the
<noun> ::= man | woman | bicycle | book
<adjective> ::= red | big | beautiful

```

Fig. 2.2 A trivial language

Here, we define a structure called a `<sentence>` as a `<subject>` followed by a `<verb>` followed by an `<object>`. An `<object>` includes an optional `<adjective>`. The tokens on the right-hand sides of `<verb>`, `<article>`, `<noun>` and `<adjective>` are *terminal* tokens implying that they cannot be expanded further. Tokens enclosed in angle brackets in BNF are called *non-terminal* tokens. Terminal tokens are just treated as sequences of characters. The symbol `::=` separates the token being defined

from its definition. The symbol $|$ separates alternatives. It is the *alternation* operator. The symbols $::=$, $|$, $<$ and $>$ are the *metasymbols* of BNF.

This set of rules can be used to generate random “sentences”. Where there are alternatives, random choices can be made. For example, starting with the non-terminal $<\text{sentence}>$, expanding just a single non-terminal at each step:

```

<sentence>
<subject> <verb> <object>
<article> <noun> <verb> <object>
the <noun> <verb> <object>
the woman <verb> <object>
the woman watches <object>
the woman watches <article> <adjective> <noun>
the woman watches a <adjective> <noun>
the woman watches a beautiful <noun>
the woman watches a beautiful bicycle

```

Be aware that BNF is far from being sufficiently powerful to define the syntax of English or any other natural language. And we have not considered the consequences of semantics here. For example, this simple grammar can generate sentences with little sense such as the *bicycle* hears the book.

The power of BNF is better seen in a slightly more complicated example, shown in Fig. 2.3.

```

<expr> ::= <term> | <expr> + <term> | <expr> - <term>
<term> ::= <factor> | <term> * <factor> | <term> / <factor>
<factor> ::= <integer> | (<expr>)
<integer> ::= <digit> | <integer> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

Fig. 2.3 BNF for simple arithmetic expressions

These rules define the syntax of simple arithmetic expressions using integer values, the operators $+$, $-$, $*$ and $/$, together with parentheses for grouping. Expressions such as $2+3+4/5$ and $2+3*(44-567)$ can be generated via this set of BNF rules.

Several important points should be highlighted.

- There is no upper limit to the length of expressions that can be generated by these particular rules. This is because these rules make use of *recursion*, allowing rule expansion to continue for an arbitrary number of steps. Strict BNF has no mechanism for simple iteration and recursion is used instead. For example, $<\text{expr}>$ is defined in terms of itself.
- Consider the generation or *derivation* of the expression $1+2*3$ using this set of rules (we omit here the steps between $<\text{factor}>$ and a literal integer value).

```

<expr>
<expr> + <term>
<term> + <term>

```

```

<factor> + <term>
1 + <term>
1 + <term> * <factor>
1 + <factor> * <factor>
1 + 2 * <factor>
1 + 2 * 3

```

Note particularly that in the expansion $1 + \langle \text{term} \rangle$, there is an implication that the $2 * 3$ part of the expression is grouped as a single entity called a $\langle \text{term} \rangle$. The BNF rules can be used to support the idea of *operator precedence* where here the precedence of the $*$ operator is higher than the precedence of the $+$ operator—the multiplication is “done before” the addition. This of course coincides with the rules of traditional arithmetic and algebra. The phrasing of the BNF rules allow the specification of the precedence of operators. This notion will arise repeatedly in the techniques used for compiler construction.

- Similarly, the BNF rules can be used to express the *associativity* of operators. For example, consider the generation of the expression $1 + 2 + 3$. Here, the $1 + 2$ part of the expression is grouped as a single entity called a $\langle \text{term} \rangle$. The implication is, therefore, that the expression $1 + 2 + 3$ is interpreted as $(1 + 2) + 3$. The $+$ operator is *left-associative*.

If different associativity or precedence rules are required, then the BNF could be modified to express these different rules. It is perhaps surprising that such a simple metalanguage can do all this.

BNF has been used in the definition of many programming languages and over the years, many extensions to BNF have been proposed and subsequently used. There are several different variants of the low-level syntax of BNF-like metalanguages, but one variant became popular after its use in the ISO Pascal Standard [2]. This variant was called *Extended Backus–Naur Form (EBNF)*. It retains the basic principles of BNF but the syntactic detail is a little different. The BNF example above can be translated easily into EBNF as follows:

```

expr = term | expr "+" term | expr "-" term.
term = factor | term "*" factor | term "/" factor.
factor = integer | "(" expr ")".
integer = digit | integer digit.
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

```

The terminal symbols (the symbols that cannot be expanded any further) are all enclosed in double quotation marks, each production rule is terminated with a full stop and the token $=$ (rather than BNF's $::=$) separates the non-terminal token from its definition. The $<$ and $>$ brackets have disappeared. There are some other key additional features.

- Parentheses can be used to indicate grouping in the rule.
- There is a specific feature to indicate optionality in a rule: $[X]$ specifies zero or one instance of X , in other words specifying that X is optional.
- Repetition (not by recursion) is supported too: $\{X\}$ implies zero or more instances of X .

We can therefore write an essentially equivalent set of rules:

```

expr = term | expr ("+" | "-") term.
term = factor | term ("*" | "/") factor.
factor = integer | "("expr)".
integer = digit {digit}.
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

```

EBNF offers a more compact and probably clearer route towards the definition of the syntax of a programming language. It ties in well with the hand construction of syntax analysers as will be seen in Chap. 4.

Finally, a brief mention should be made of the use of *syntax diagrams* in the definition of programming languages. This approach was popularised in the original definition of Pascal [3] and uses a pictorial notation to represent the syntax of each of the non-terminal tokens of the language. This too proves to be an easily understood and compact notation.

2.2.1.1 Limitations

The formal syntax of most common programming languages, expressed in the form of BNF or equivalent, will almost certainly lack a specification for some of the rules for the construction of “well-formed” programs. This is likely to result from the fact that a BNF specification cannot express contextual constraints. For example, the programming language being defined may be such that all variables have to be declared appropriately before they are used. The C statement `i = 2;` is only valid in the context of a valid declaration for `i` such as `int i;`. This constraint does not appear in the usual syntax specification for C. We will see exactly why this is a problem caused by the limitations of metalanguages such as BNF a little later in this chapter.

It is tempting, therefore, to rethink the metalanguage used for language specification so that these additional rules can be incorporated somehow. Such metalanguages do exist; an example is the two-level grammar introduced in the definition of ALGOL 68 [4]. This approach essentially uses two distinct rule sets in two different metalanguages, the first set being used to generate the second (infinite) set which in turn is used to generate valid programs in ALGOL 68. This infinite set of production rules has the effect of allowing the specification of context-dependent constraints in the language.

This and similar approaches have not proved to be popular because of their complexity. The preferred approach is to stick with the simple context-free rules of BNF, or equivalent, and rely on other sets of rules, formal or informal, to define the additional constraints. As we will see, one of the key advantages of retaining this simple form of syntax specification is that the generation of the corresponding analysis phase of the compiler can be made very simple.

2.2.2 Semantics

Unfortunately, the specification of semantics is much harder than the specification of syntax. Formal approaches may be possible, based on mathematical formalisms. These definitions may prove to be long and complex, inaccessible to the casual user of the programming language. Formal semantics opens up the possibility of *proving* program correctness and removes the possibility of semantic ambiguity. Several approaches to the specification of programming language semantics have been developed—operational semantics, denotational semantics and axiomatic semantics, basing a formal description of semantics on the language’s syntax. There are many good textbooks in this area, for example see [5].

Attribute grammars can be used to help define aspects of the semantics of a programming language, allowing the specification of context-sensitive aspects by augmenting a context-free grammar. Here, grammar symbols are associated with *attributes*. These are values that can be passed to both parent and child of the grammar node in which the symbol appears. This approach allows the formal specification of the language’s operational semantics (how the program is interpreted as a sequence of computational steps), supporting semantic checks such as requiring the definition of a name before its use [6].

An alternative approach is to specify semantics somewhat more indirectly by producing a *reference implementation*. Here, a particular implementation is selected to define how all other implementations should behave. A program running on any of the implementations should behave as if it is running on the reference implementation. The simplicity of this approach is attractive. However, there are potential problems that may arise because of software or even hardware errors in that reference implementation which, strictly speaking, should be followed by all other implementations.

A third approach, and an approach used widely in the specification of popular languages, is to specify the semantics using a natural language. Here, text in a natural language such as English is used to describe the semantic rules of the programming language. Care is needed to avoid omission or ambiguity and to prevent the specification from becoming overly long. There is a real danger of assuming that the semantics of programming language constructs are “obvious”. This is far from being true—there are many examples of real programming language features that are often misinterpreted by the programmer and sometimes mis-implemented by the compiler writer.

This book takes the easy route and avoids issues concerned with the formal specification of semantics. There will be many semantics-related issues discussed, but using an informal (English language) notation. We, in common with many other compiler writers, follow this third approach for semantics specification.

2.3 Analysis of Programs

Before looking at practical approaches for the analysis phase of programming language translation, we have to cover just a little theory. We need a formal structure on which to base the process of analysis. It just cannot be done reliably in an ad hoc way. We need to look first at the idea of formal grammars and the notations associated with them. These grammars form the rock on which we can build code for programming language analysis.

2.3.1 Grammars

The term “grammar” has a wide range of definitions and nuances and it is hardly surprising that we need a tight and formal definition for it when used in the context of programming languages. The idea of a set of BNF rules to represent the grammar of a language has already been used in this chapter, but formally a little more is required.

The *grammar* (G) of a language is defined as a 4-tuple $G = (N, T, S, P)$ where:

- N is the finite set of non-terminal symbols.
- T is the finite set of terminal symbols (N and T are disjoint.)
- S is the *starting symbol*, $S \in N$. The starting symbol is the unique non-terminal symbol that is used as the starting point for the generation of all the strings of the language.
- P is the finite set of *production rules*. A production rule defines a string transformation and it has the general form $\alpha \rightarrow \beta$. This rule specifies that any occurrence of the string α in the string to be transformed is replaced by the string β .

There are constraints on the constitution of the strings α and β . If U is defined by $U = N \cup T$ (i.e. U is the set of all non-terminal and terminal symbols), then α has to be a member of the set of all non-empty strings that can be formed by the concatenation of members of U , and it has to contain at least one member of N . β has to be a member of the set of all strings that can be formed by the concatenation of members of U , including the empty string (i.e. $\beta \in U^*$).

Looking back at the BNF definition of the simple arithmetic expressions in Fig. 2.3, it is easy to see that this forms the basis of the formal grammar of the language. Here N is the set {expr, term, factor, integer, digit}, T is the set {+, -, *, /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, S is expr and P is simply the set of rules in Fig. 2.3, translating the BNF syntax into whatever syntax is used to represent the rules, traditionally $::=$ being replaced by \rightarrow . In practice, we can afford to be a bit sloppy in the definition and use of the term grammar since the specification of N , T and S are usually obvious from the BNF (or equivalent) production rules. There is a convention that the starting symbol is the non-terminal defined in the first production rule.

If BNF, EBNF or syntax diagrams are used to specify the production rules, all rules have a particular characteristic that the left-hand sides of productions are always single non-terminal symbols. This is certainly allowed by the rules defining a grammar, but this restriction gives these grammars certain important features which will be examined in Sect. 2.3.2.

A grammar gives the rules for deriving strings of characters conforming to the syntactic rules of the grammar. A *sentential form* is any string that can be derived from S , the starting symbol. And a *sentence* is a sentential form not containing any non-terminal symbols. A sentence is something final, it cannot be expanded any further. In the context of grammars for programming languages, a sentence is a complete program, containing just terminal symbols (i.e. the characters of the language).

2.3.2 Chomsky Hierarchy

Looking at the definition of a grammar in the last section, it is clear that the important and potentially problematic component is P , the set of production rules. A production rule has the form $\alpha \rightarrow \beta$, loosely translated as “anything can be transformed to anything” (although we have already stated some restrictions on the content of α and β). The key question then is to remove this generality by restricting the forms of the production rules to see whether less general rules can be useful for defining and analysing computer programming languages.

In the 1950s, Noam Chomsky produced a hierarchical classification of formal grammars which provides a framework for the definition of programming languages as well as for the analysis of programs written in these languages [7]. This hierarchy is made up of four levels, as follows:

- A *Chomsky type 0* or a *free grammar* or an *unrestricted grammar* contains productions of the form $\alpha \rightarrow \beta$. The restrictions on α and β are those already mentioned in the section above. This was our starting point in the definition of a grammar and as suggested, these grammars are not sufficiently restricted to be of any practical use for programming languages.
- A *Chomsky type 1* or a *context-sensitive grammar* has productions of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ where $\alpha, \beta, \gamma \in U^*$, γ is non-null and A is a single non-terminal symbol. The left context is α , the right context is β and in this particular context, A is transformed to γ .

This type of grammar turns out to have significant relevance to programming languages. The concept of context is central to programming language definition—“... this statement is only valid in the context of an appropriate declaration of i ...”, for example. However, in practice, these grammars do not turn out to be particularly helpful because defining the context-sensitive aspects of a programming language in terms of a type 1 grammar can turn into a nightmare of complexity. So we need to simplify further and specify context-sensitive aspects by resorting to other means, such as English language descriptions.

- A *Chomsky type 2* or a *context-free grammar* has productions of the form $A \rightarrow \gamma$ where A is a single non-terminal symbol. These productions correspond directly to BNF rules. In general, if BNF can be used in the definition of a language, then that language is no more complex than Chomsky type 2. These grammars are central to the definition and analysis of the majority of programming languages. Despite the simplicity of the productions they are capable of defining powerful and complex program syntax. Chapters 4 and 5, in the discussion of syntax analysis, are based on this grammar type. Programming languages are generally defined using type 2 grammars and these grammars are used directly in the production of code for program analysis.
- A *Chomsky type 3* or a *regular grammar* or a *finite-state grammar* puts further restrictions on the form of the productions. Here, all productions are of the form $A \rightarrow a$ or $A \rightarrow aB$ where A and B are non-terminal symbols and a is a terminal symbol. These grammars turn out to be far too restrictive for defining the syntax of conventional programming languages, but do have a key place in the specification of the syntax of the basic lexical tokens dealt with by the lexical analysis phase of a compiler (see Chap. 3). They are the grammars for languages that can be specified using *regular expressions* and programs in these languages can be recognised using a *finite-state machine*.

These grammar types form a hierarchy, such that all type 3 languages are also type 2, 1 and 0, all type 2 languages are also type 1 and 0 and all type 1 languages are also type 0.

2.3.3 Parsing

Suppose we have a set of BNF (or equivalent) production rules defining the grammar of a programming language. We have already seen how by expanding these rules, programs can be generated. This is a simple process. We can use these grammar rules as a reference while writing programs in that language to help ensure that what is written is syntactically correct. Because the BNF specification lacks the power to define the context-sensitive aspects of the language, we will need additional advice about, for example, making sure that names are declared, that types have to match, and so on. This collection of information serves to define the programming language and should offer enough to allow the writing of *syntactically correct programs*.

The reverse process of going from a program to some data structure representing the structure and details of the program, also checking that the program is indeed syntactically correct, is unfortunately much harder. This is the process of program analysis or *parsing* and is one of the key tasks performed by a compiler.

Why is parsing so much harder? Consider a simple example based on the grammar presented in Fig. 2.3 and on its subsequent use to generate the expression $1+2*3$. Let us try using this grammar to work backwards from the expression $1+2*3$ to the starting symbol $\langle \text{expr} \rangle$. We know that this should be possible because it can be

achieved by simply reversing the steps used in its generation. Again, to simplify, we ignore the steps between a literal integer value and `<factor>`.

<code>1 + 2 * 3</code>	
<code><factor> + 2 * 3</code>	(using <code><factor> → 1</code>)
<code><term> + 2 * 3</code>	(using <code><term> → <factor></code>)
<code><expr> + 2 * 3</code>	(using <code><expr> → <term></code>)
<code><expr> + <factor> * 3</code>	(using <code><factor> → 2</code>)
<code><expr> + <term> * 3</code>	(using <code><term> → <factor></code>)
<code><expr> * 3</code>	(using <code><expr> → <expr> + <term></code>)
<code><expr> * <factor></code>	(using <code><factor> → 3</code>)
<code><expr> * <term></code>	(using <code><term> → <factor></code>)
<code><expr> * <expr></code>	(using <code><expr> → <term></code>)

At this stage we seem to be stuck, implying that `1+2*3` is syntactically incorrect. What has gone wrong?

The process of parsing repeatedly matches substrings with the right-hand sides of productions, replacing the matched substrings with the corresponding production's left-hand side. Problems arise when there is more than one substring that can be matched or *reduced* at any stage. It turns out that the choice of substring to be matched is important. By trying this parsing process again using a different set of reductions, we get a different result.

<code>1 + 2 * 3</code>	
<code>1 + 2 * <factor></code>	(using <code><factor> → 3</code>)
<code>1 + <factor> * <factor></code>	(using <code><factor> → 2</code>)
<code>1 + <term> * <factor></code>	(using <code><term> → <factor></code>)
<code>1 + <term></code>	(using <code><term> → <term> * <factor></code>)
<code><factor> + <term></code>	(using <code><factor> → 1</code>)
<code><term> + <term></code>	(using <code><term> → <factor></code>)
<code><expr> + <term></code>	(using <code><expr> → <term></code>)
<code><expr></code>	(using <code><expr> → <expr> + <term></code>)

In this case, we end with the starting symbol so that the parse has succeeded.

But how do we determine the proper set of reductions? It may be that we can reach the starting symbol via several different sets of reduction operations. In this case, we conclude that the grammar is *ambiguous* and it needs repair, either by altering the set of productions or (not so desirable) by adding additional descriptive explanation to indicate which particular set of reductions is correct. Choosing the set of reductions to be applied on a sentence is the central issue in parsing. In Chaps. 4 and 5 algorithms are proposed for tackling this problem.

2.3.3.1 The Output of the Parser

The parser obtains a stream of tokens, from the lexical analyser in a conventional compiler, and matches them with the tokens in the production rules. As well as indicating whether the input to the parser forms a syntactically correct sentence, the parser must also generate a data structure reflecting the syntactic structure of the input. This can then be passed on to later stages of the compiler.

This data structure is traditionally a tree. The *parse tree* is constructed as the parser performs its sequence of reductions and the form of the parse tree directly reflects the syntactic specification of the language. The root node of the parse tree corresponds to the starting symbol of the grammar. For example, the tree generated by running the parser on the $1+2*3$ example could take the general form shown in Fig. 2.4a.

This form of tree accurately reflects the formal syntactic definition of the language, and much of the tree may turn out to be redundant. Therefore it may be adequate to generate a tree closer to the form shown in Fig. 2.4b. This is an *abstract syntax tree* where not every detail of the sequence of reductions performed by the parser is reflected in the tree. Nevertheless, the data in the tree is sufficient for later stages of compilation.

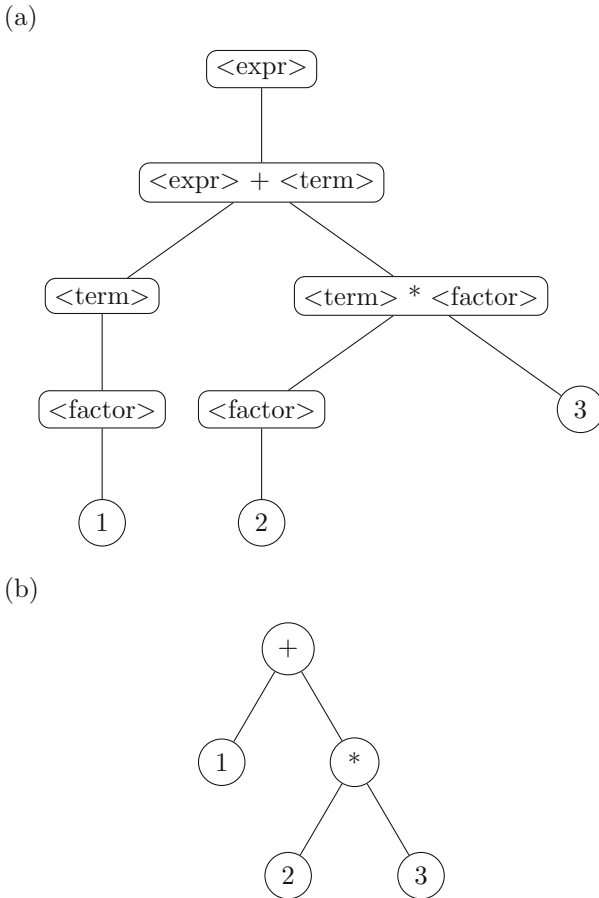


Fig. 2.4 Syntactic structure of the expression $1 + 2 * 3$

These trees and the algorithms used for their construction will be examined in detail in later chapters.

2.3.3.2 Parsing Strategies

There are two broad approaches for the construction of an algorithm for parsing. Most parsers can be classified as being either *top-down parsers* or *bottom-up parsers*. The parsing process takes the string to be parsed and repeatedly matches substrings with the right-hand sides of productions, replacing those substrings with the corresponding left-hand sides. If we start off with a syntactically correct sentence, the parsing process should transform the sentence to the starting symbol via a sequence of sentential forms. We have already seen that the choice of which reductions are made is important and a correctly written parser gets that choice right. Practical parsers rarely take this approach of repeatedly manipulating a potentially very long character string, but the principle applies.

The *top-down parser* starts with the starting symbol of the grammar and hence with the root of the parse tree. Its goal is to match the input available with the definition of the starting symbol. So if the starting symbol S is defined as $S \rightarrow AB$, the goal of recognising S will be achieved by recognising an instance of an A followed by recognising an instance of a B . Similarly, if S is defined as $S \rightarrow A|B$, then the goal of recognising S will be achieved by recognising an instance of an A or by recognising an instance of a B . The subgoals of recognising A and B are then dealt with according to subsequent rules in the grammar. When the right-hand side of a production that is being matched with the input contains *terminal* symbols, these symbols can be matched with the input string. If the matching fails, then the parsing process fails too. But if the matching succeeds then the process continues until, hopefully, all characters in the input have been matched, at which point the parse succeeds. It is hard to visualise how this top-down process corresponds to the process described above of parsing using repeated reductions on the original and then transformed input string, but the top-down parser *is* making repeated reductions, the order and choice being controlled by the structure of the set of productions. Chapter 4 examines this whole process in detail.

The *bottom-up parser* perhaps reflects a more obvious way of thinking about parsing, where, instead of starting with the starting symbol, we start with the input string, choose a substring to be matched with the right-hand side of a production, replace that substring with the corresponding left-hand side, and repeat until just the starting symbol remains (indicating success) or until no valid reduction can be performed (indicating failure). The parse tree is being constructed upwards from the leaves, finally reaching the starting symbol at the root. The key problem here is of course one of determining which reductions to apply and in which order. Again, we return to this issue in Chap. 4.

2.4 Compiler and Interpreter Structure

Having looked at some of the issues of programming language definition and analysis, we have to step back and examine the overall structure of the programming language translation process. The implementation of a programming language is potentially a huge software project. The GNU Compiler Collection (GCC) now contains well over 10 million lines of source code. This is, admittedly, an extreme example, but it does illustrate the need for good software engineering principles for compiler or interpreter projects. In order to start thinking about such a project, it is essential to consider the structure of a compiler or an interpreter in terms of a collection of logically separate modules so that a large task can be viewed as a collection of somewhat simpler tasks.

We start with the trivial view of a compiler or interpreter shown in Fig. 2.1. In the case of a compiler, the translator is generating code to run on a real or virtual machine. In the case of an interpreter, the translator is generating code which is interpreted by the interpreter program. There is no profound difference between these two approaches (a compiler can generate code that is interpreted by the hardware) and hence some of the internal structures of compilers and interpreters can be similar. In this section, we discuss specifically the modular structure of a compiler generating code for a real machine.

The first subdivision, we can make is to consider the compiler as having to perform two distinct tasks, as shown in Fig. 2.5. The analysis phase and the synthesis phase are often referred to as the *front-end* and the *back-end* respectively.

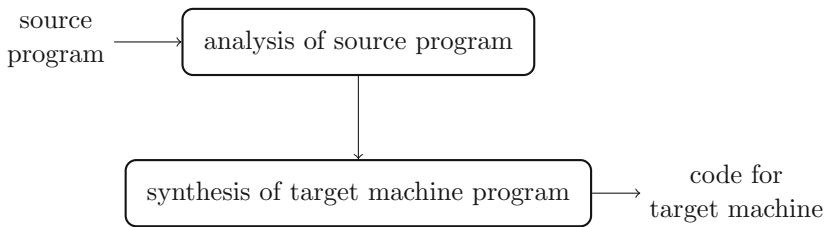


Fig. 2.5 The analysis/synthesis view of compilation

This subdivision, although simple, has major consequences for the design of compilers. The interface between these two phases is some intermediate language, loosely “mid-way” between the source and target languages. If this intermediate language is designed with care, it may then be possible to structure the compiler so that the analysis phase becomes *target machine independent* and the synthesis phase becomes *source language independent*. This, in theory, allows great potential savings in implementation effort in developing new compilers. If a compiler structured in this way needs to be retargeted to a new machine architecture, then only the synthesis phase needs to be modified or rewritten. The analysis phase should not need to be touched. Similarly, if the compiler needs to be modified to compile a different source language (targeting the same machine), then only the analysis phase needs

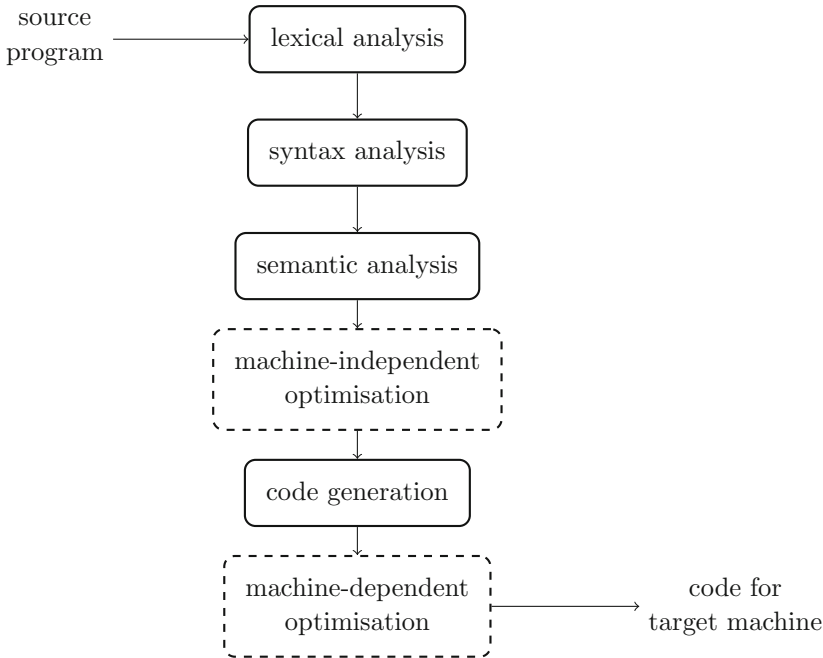


Fig. 2.6 Phases of compilation

to be modified or rewritten. We look at this issue from the view of the intermediate representation in Chap. 6.

But the subdivision into phases needs to be taken further to be of any use in the practical construction of a compiler. A traditional view of compiler structuring, referred to repeatedly in this book, is shown in Figure 2.6.

The lexical analysis, syntax analysis, semantic analysis and the machine-independent optimisation phases together form the front-end of the compiler and the code generation and machine-dependent phases form the back-end. These phases all have specific and distinct roles. And to support the design of these individual modules, the interfaces between them have to be defined with care.

2.4.1 Lexical Analysis

This first phase of compilation reads the characters of the source program and groups them together into a stream of *lexical tokens*. Each lexical token is a basic syntactic component of the programming language being processed. These are tokens such as numbers, identifiers, punctuation, operators, strings, reserved words and so on. Comments can be ignored unless the language defines special syntactic components

encoded in comments that may be needed later in compilation. White space (spaces, tabs, newlines, etc.) may be ignored except, again, where they have some syntactic significance (where spacing indicates block structure, where white space may occur in character strings and so on).

For example, this C program fragment:

```
sum = 0;
for (i=0; i<=99; i++) sum += a[i];    /* sum array */
```

will be read by the lexical analyser and it would generate this stream of tokens:

```
sum (identifier), =, 0 (integer constant), ;, for (reserved word), (, i (identifier), =, 0
(integer constant), ;, i (identifier), <=, 99 (integer constant), ;, i (identifier), ++, ),
sum (identifier), +=, a (identifier), [, i (identifier), ], ;
```

The syntax of these basic lexical tokens is usually simple, and the expectation is that the syntax can be specified formally in terms of a Chomsky type 3 grammar (i.e. in terms of regular expressions). This considerably simplifies the coding of the lexical analyser.

The output of the lexical analyser is a stream of tokens, passed to the syntax analyser. The interface could be such that the lexical analyser tokenises the entire input file and then passes the whole list of tokens to the syntax analyser. Alternatively, the tokens could be passed on to the syntax analyser one at a time, when demanded by the syntax analyser.

2.4.2 *Syntax Analysis*

The syntax analyser groups and structures the lexical tokens according to the syntax rules of the programming language. It performs the parsing process, as outlined above in Sect. 2.3.3, repeatedly grouping together components by performing reductions according to the production rules. Assuming that the sequence of tokens is syntactically correct, the parse should succeed. If the sequence is not syntactically correct, then the syntax analyser should report an error and then perform some appropriate recovery action.

The syntax analyser constructs a data structure representing the syntactic structure of the input. This is usually based on some form of tree where the nodes represent syntactic components defined by the grammar. This is the parse tree or abstract syntax tree. This data structure should contain or link to all the information needed by later phases of compilation. So, for example, a node corresponding to the occurrence of a constant value in the original program should contain or link to information defining that constant such as its type, value and so on.

It is clear that the lexical and syntax analysers are doing similar things. They are both grouping together characters or tokens into larger syntactic units. So there is an issue about whether a particular syntactic structure should be recognised by the lexical analyser or by the syntax analyser. The traditional approach, and it is

an approach that works well, is to recognise the simpler structures in the lexical analyser, specifically those that can be expressed in terms of a Chomsky type 3 grammar. Syntactic structures specified by a type 2 or more complex grammar are then left for resolution by the syntax analyser. In theory, the syntax analyser could deal with the lexical tokens using a type 2 grammar parsing approach, but this would add significantly to the complexity of the syntax analyser. Furthermore, by leaving the lexical analyser to deal with these tokens improves compiler efficiency because simpler and faster type 3 parsing techniques can be used.

2.4.3 *Semantic Analysis*

The analysis phase is not quite complete even after the syntax tree has been constructed. Traditional type 2 grammars used to build the syntax analyser cannot deal directly with contextual issues such as type checking, declaration and scopes of names, choice of overloaded operators and so on. This is the role of the semantic analysis phase. Traversing the tree, inserting and checking type information is done here. Typed languages may require that all or almost all the nodes in the tree be labelled with a data type. Complexity is increased when the language being compiled allows user-defined types. Rules for type compatibility have to be applied here too. For example, does the language allow an integer value to be assigned to a real (floating point) variable?

A second task of the semantic analysis phase is to *flatten* the parse tree to produce some form of *intermediate code*. The nature of this code is discussed in Chap. 6. It should be straightforward to generate this intermediate code by traversing the tree. The type information is preserved so that the intermediate code is functionally equivalent to the original source program. This intermediate representation can be regarded as the machine code for a carefully designed virtual machine.

2.4.4 *Machine-Independent Optimisation*

Generating intermediate code by simple tree traversal will yield code with opportunities for improvement. This optimisation phase, together with the optimisation performed during and after code generation, can make a dramatic difference to the quality of the code generated by the compiler. In Fig. 2.6 these phases are enclosed in dashed lines, indicating that they are optional. The compiler will still work without them, but the quality of generated code may be poor.

Post-semantic analysis is a good stage of compilation for code optimisation. The intermediate representation will have been designed with optimisation in mind and many optimisation techniques can be applied. These result, in some cases, in dramatic performance improvements. Removal of redundant code, function inlining, loop unrolling, dependence and flow analysis and so on can all be done here. This is

a target machine-independent phase because the optimisations being performed are making no assumptions about the low-level design of the target machine.

The output of this phase is a representation of the program being compiled in an intermediate form. It is likely, but not essential, that the input and the output of this phase are expressed in the same intermediate representation.

2.4.5 Code Generation

At this point, attention shifts away from the nature of the source language and moves towards the design and characteristics of the target machine. The code generation phase reads the (optimised) intermediate code and outputs functionally equivalent target machine instructions. This is easy to specify but the implementation requires the handling of complex detail.

The code generator has to select appropriate machine instructions, decide how the target machine registers are to be used, deal with a storage allocation scheme for all the variables and structures needed by the program as it runs, generate code to interface with libraries and the operating system. This is all being done in the context of the need to generate high-quality code.

2.4.6 Machine-Dependent Optimisation

At least partially incorporated into the code generation phase is the process of optimisation specifically geared towards the characteristics of the target machine. Use of special-purpose instructions and addressing modes, making use of target machine parallelism, using the target machine registers effectively and so on can make a significant impact on the quality of the target code. There are some optimisations that are best done as code is actually generated, whereas there are other techniques that are best run as a separate pass over the generated code.

There are some general issues concerning optimisation. First, the term “optimisation” is used in the compiler context in a somewhat unconventional way. It is not taken to mean “generate the *best* code possible”, but instead it implies “generate *better* code”. Furthermore, the aims of optimisation need to be clear when developing the compiler. Is the aim of optimisation to generate code that will run *fast* on the target machine? Or is it to generate *compact* code (maybe more appropriate for embedded systems)? Or is it some combination of the two? Is the aim to minimise the power consumption as the code runs? Managing these tradeoffs may be difficult and the aims should be clear as the optimisation phases are being developed.

The final output of the compiler is a program that can be run on the target machine, maybe after some further processing. The output may be some form of object file requiring processing by a linker or loader before it can actually run or maybe an

assembly language file requiring processing by an assembler to produce loadable target code.

2.4.7 Symbol Tables

The names (identifiers, symbols) used in the source program need to be stored during the compilation process. Further information relating to types, scope, declarations, values or locations and other source language-dependent features will need to be stored too. This information allows the compiler, for example, to ensure that variables are appropriately declared, perform type checking, generate appropriate intermediate code instructions and include symbolic names in the code generator output to allow symbolic debugging at runtime. Therefore, the symbol table in a typical compiler is a complex data structure, supporting efficient name lookup, accessible by any of the compilation phases.

Symbols may be inserted into the symbol table by the lexical analyser, but it may be better to perform this task in the syntax analyser where more context information is known. The syntax analyser can distinguish between the declaration and the use of a name and this is important when accessing the symbol table. The semantic analysis phase makes heavy use of the symbol table, and it may generate intermediate code that implicitly includes enough of the symbol table information to allow the code generation and optimisation phases to be free of the need to access the symbol table.

2.4.8 Implementation Issues

Finally, there are many practical issues to consider in designing the implementation plan of a compiler. In which programming language should the compiler or interpreter be written? What sort of testing strategy should be adopted? Are there techniques to simplify the implementation process? Are there good software tools to use? Can we make use of software that is already freely available by incorporating it in the compiler?

It may be that more software than just the compiler or interpreter needs to be written. Is there a need for a runtime debugger, linker or loader? Are there program development tools needed to be integrated with the compiler? Do we need a runtime system providing an interface between the running compiled program and the operating system and/or hardware? The task can easily get out of control but there are many standard implementation routes, some of which are examined in Chap. 9.

2.5 Conclusions and Further Reading

This chapter has shown that the design of the implementation of a programming language is by no means a trivial task but has also shown that the task may have become tractable by imposing a solid structure on which an implementation can be built. Good planning is vital in this sort of project especially if a team is involved. Starting from accurate definitions is essential. Formalising and automating the process makes the production of a reliable implementation so much more straightforward.

Compilers and interpreters are now very rarely constructed from scratch. Making use of already available software may well be crucial to make the project feasible. The separation of the compilation process into clearly defined modules and the use of standardised or pre-existing interfaces makes this process of modular construction very much easier.

This may be completely obvious, but it is worth stating nevertheless. It pays to start off with an excellent, in-depth knowledge of both the source language and the target machine. Conversely, writing a compiler may be one of the best ways to learn a programming language and a target machine!

The documents formally defining programming languages form a valuable resource for anyone involved in the task of language implementation. Early language definitions, such as the FORTRAN standard [8], are well worth examining to see how far we have come in both programming language design and also in techniques for programming language definition. The ALGOL 60 definition [1] is also a key historical document, particularly for its use of BNF. It is also worth taking a look at the definition of ALGOL 68 [4], again a key historical document but it clearly shows the importance of having an *accessible* language definition. The definitions of most more modern languages are easily found on the web, some simple and others of astonishing complexity.

Routes towards programming language implementation are sometimes complex. This is usually and paradoxically the case because of the need to reduce the amount of programming effort required. Understanding the stages required in such implementations is often difficult and *T-diagrams* were introduced as a simple visualisation mechanism [9, 10]. Deciding on whether to compile or interpret is a key question and picking out the language features (such as reflection in Java) that push towards an interpreted implementation is a helpful task.

A great deal has been written about the design of virtual machines (for example, see [11]) and documents easily accessible via the web provide designs for general-purpose and domain-specific virtual machines. The *Java Virtual Machine* (see [12]) is perhaps the most famous virtual machine and has been used in the implementations of a wide range of programming languages.

This is not a textbook about the more formal aspects of grammars and parsing. There are so many high-quality published resources in this area. A classic text is [13] and much useful background information is contained in [14, 15].

Finally, an excellent source of design information is the source code and documentation of existing compilers and interpreters. For example, the GNU GCC project is

well documented and the compiler source code is freely available. A good indication of the functionality offered by the compiler is given by the range of options available when running the compiler.

Exercises

- 2.1. The Java Virtual Machine has been used as a route to the implementation of many programming languages. Produce a list of some of these languages and implementations. Why was the JVM often chosen? Are there programming language features that do not map well onto the JVM?
- 2.2. Suppose you had to write a program to count the number of `if` statements used in a C program. Explain why the obvious approach of counting the number of matches with the character string “if” may produce the wrong answer. Why can the use of lexical analyser techniques help? Are aspects of syntax analysis required too?
- 2.3. Write a program to read grammar productions expressed in BNF or EBNF and generate random sentences from the grammar. To make the sentences more interesting it may help to be able to alter the relative probabilities of choice where alternatives are specified by the grammar.
Try this program on the grammar of a real programming language and if possible put it through a compiler for that language. Did you expect it to compile?
- 2.4. Write the syntax of BNF in EBNF (and the other way round).
- 2.5. Produce the grammar for a simple language specifying conventional arithmetic expressions involving integer constants, brackets and the four binary operators `+`, `-`, `*` and `/`. Make sure that expressions of the form $1+2*3$ are interpreted correctly by the grammar. Then extend the grammar to allow the unary operators `+` and `-`. Make sure that the grammar correctly interprets expressions of the form $-1-2$. Maybe produce an implementation, although this will be *much* easier once material in later chapters has been read!
- 2.6. Produce a grammar for simple arithmetic expressions with unconventional rules of precedence so that, for example, the expression $3*2+1$ is interpreted as $3*(2+1)$.
- 2.7. Check how your grammar interprets expressions of the form $1-2-3$. Change the grammar to make all four operators right-associative so that the value of $1-2-3$ is 2 rather than -4.

References

1. Naur P (1960) Report on the algorithmic language ALGOL 60. Commun ACM 3(5):299–314
2. Jensen K, Wirth N (1985) Pascal user manual and report – ISO Pascal standard, 3rd edn. Springer, New York
3. Jensen K, Wirth N (1975) The Pascal user manual and report, 2nd edn. Springer, New York

4. van Wijngaarden A, Mailloux BJ, Peck JEL, Coster CHA, Sintzoff M, Lindsey CH, Meertens LGLT, Fisker RG (1975) Revised report on the algorithmic language ALGOL 68. *Acta Inform* 5:1–236
5. Winskel G (1993) *The formal semantics of programming languages*. The MIT Press, Cambridge
6. Cooper KD, Torczon L (2011) *Engineering a compiler*, 2nd edn. Morgan Kaufmann, San Francisco
7. Chomsky N (1956) Three models for the description of language. *IRE Trans Inf Theory* 2:113–124
8. United States of America Standards Institute, New York (1966) USA Standard FORTRAN – USAS X3.9-1966
9. McKeeman WM, Horning JJ, Wortman DB (1970) *A compiler generator*. Prentice Hall, Englewood Cliffs
10. Terry PD (1986) *Programming language translation: a practical approach*. International computer science series. Addison-Wesley Publishing Company, Reading
11. Wilhelm R, Seidl H (2010) *Compiler design: virtual machines*. Springer, Berlin
12. Lindholm T, Yellin F (1997) *The Java virtual machine specification*. The Java series. Addison-Wesley, Reading
13. Hopcroft JE, Ullman JD (1979) *Introduction to automata theory, languages and computation*. Addison-Wesley Publishing Company, Reading
14. Aho AV, Lam MS, Sethi R, Ullman JD (2007) *Compilers - principles, techniques and tools*, 2nd edn. Pearson Education, Upper Saddle River
15. Mogensen TÆ (2011) *Introduction to compiler design*. Undergraduate topics in computer science. Springer, Berlin



<http://www.springer.com/978-3-319-52787-1>

A Practical Approach to Compiler Construction

Watson, D.

2017, XV, 254 p. 26 illus., Softcover

ISBN: 978-3-319-52787-1