7

formal systems and programming languages: an introduction

The objectives of this chapter are to:

1. Provide an informal introduction to formal systems and grammars
2. Demonstrate the relevance of theoretical to practical work in the area of programming languages
3. Present formal systems and terminology that are commonly used in the literature
4. Present theoretical tools relevant to compiling techniques
5. Describe two methods for programming language specification: Backus-Naur Form and canonic systems
6. Present an example of research being conducted in this area and pose some theoretical questions that are being asked about programming languages and formal systems

This chapter presents a brief view of a complicated and varied subject; more complete treatments will be found in the references (Chapter 10).

Except for the description of Backus-Naur Form in section 7.5, and problem 1, this chapter may be omitted from presentations of systems programming with a purely practical orientation.

7.1 USES OF FORMAL SYSTEMS IN PROGRAMMING LANGUAGES

A formal system is an uninterpreted calculus or logistic system. It consists of an alphabet, a set of words called axioms, and a finite set of relations called rules of inference. Examples of formal systems are: set theory, boolean algebra, propositional and predicate calculus, Post systems, Euclid's plane geometry, Backus Normal Form, and Peano arithmetic. A formal system is uninterpreted in the sense that no meaning is formally attached to the symbols of the system; there is

for each of the above-mentioned systems a standard informal interpretation of the symbols, but other interpretations are generally valid so far as the systems themselves are concerned.

We generally construct formal systems in order to have formalized models of informal, intuitive notions. A formal model can be studied mathematically; and if the model is appropriate, the results may tell us much about the notions that it portrays.

Formal systems are becoming important in the design, implementation, and study of programming languages. Specifically, various sorts of formal systems are used for syntax specification, syntax-directed compilation, compiler verification, complexity studies, and analysis of the structure of languages.

7.1.1 Language Specification

Formal systems are used to define the form (the *syntax*) of a programming language. Such definition is important both to the user and to the implementer of the language. The user needs (for reference) a clear description of the language. The implementer is faced with the problems of transferability and maintenance. If the same language is to be implemented (transferred) on different machines, the legal strings of the language must be well-defined so that the user-level appearance is, so far as is practicable, invariant. Further, the implementer must be concerned with maintenance. Both the user and the maintainer of a compiler need an exact specification of the acceptable strings of the corresponding language.

7.1.2 Syntax-Directed Compilers

A syntax-directed compiler uses a data base containing the syntactical rules of a source language to *parse* (recognize - find the sequence of rules that generate the string) the source-language input. Formal systems are used as the data base. Because of the increase in the number of programming languages and machines, researchers have been looking into automatic generation of compilers. Their approach has been to have formal definitions of both the input or source language L_{source} and the output language L_{object}. The output of the compiler generator would be a translator $T: L_{source} \rightarrow L_{object}$.

A problem similar to that of automatic compiler-writing is the problem of generating test programs to validate a language processor. If the source language is formally defined, the generative techniques described in the next section provide a method for automatically producing such test programs. This can be useful in testing software since a computer can be more pedantic than a human tester.

7.1.3 Complexity Structure Studies

Formal systems are used to study the complexity of programming languages and of their compilers. Compiler writers and language designers want to know which features of language inordinately increase the complexity of a compiler's recognition phase. A compiler writer also wants a basis for evaluating the performance of a compiler. He would like to know the theoretically optimum level of performance (in terms of number of steps) that he could expect for the process of translation. After implementing a version of the compiler, he could compare its performance to the theoretical limit; if it was within some tolerance, he could forego further effort. But if the performance was two orders of magnitude worse than the theoretical bound, improvement would clearly be in order.

Analogies will be found in the work surrounding Shannon's information theory. Shannon determined a measure for information and applied it to the coding of information. The resulting theory gave bounds on efficiency in information coding and transmission. Bell Telephone Laboratories researchers devising coding techniques for transmission of information could compare the performance of their schemes with Shannon's bound, using it as a yardstick. Shannon's theory does not determine coding schemes. It merely provides a measure of how efficient they can be. Similarly, in complexity studies, the compiler writer would not be told what method to use, but merely whether a better method might exist.

7.1.4 Structure Analysis

Formal systems are used in attempts to prove the equivalence and the validity of programs. The work in proving the equivalence of programs is motivated by the prospect of global optimization. If there were an algorithm which recognized the equivalence of two different programs, the faster program could be used in place of the slower-running one.

A formal theory also provides a framework for analyzing and comparing various languages. It helps answer questions such as:

1. What are the basic language features?
2. What constructs can exist in the language? How can the features be combined to build new constructs?
3. What categories of problems can be programmed in the language?
4. What is the cost or difficulty of writing a program?

These key questions may be approached through formalization. The answers are also of interest in the field of machine design; the ideal computer should efficiently perform the operations corresponding to the basic features of a language.

7.2 FORMAL SPECIFICATION

7.2.1 Approaching a Formalism

Before going deeper into formalism it is useful to analyze some of the problems in formally defining a language and to look into the intuitive basis for the definitions. A language may be thought of as a set of sentences or formulae – strings of symbols – with well-defined structures and, usually, meaning. The rules specifying valid constructions of a language are its *syntax*: the syntax of a language describes its form. For example, when we say that x+2 is an expression, but x2+ is not, we are referring to the syntax of algebra. The assignment of a meaning, or *interpretation*, to symbols and formulae is the *semantics* of a language. When we say, for example, that the value of x+2 is the sum of the values of x and 2 – or that $2 \cdot x = x+x$ is true – we are referring to the usual semantics of algebra.

If all languages consisted of a finite number of legal sentences or formulae, syntactic definition would not pose a problem; it would suffice merely to list all legal sentences – a string of symbols would be a sentence only if it appeared in the list. The problem of definition exists because almost all languages of any utility contain an unlimited (or very large) number of valid sentences. It is not possible to store a list of all valid strings for infinite languages. But it is not necessary to store the list of sentences if any member of the list can be generated whenever it is needed, even though to generate all sentences may be endless. If an algorithm exists that will successively produce legal strings, an arbitrary string is in the language if it ever occurs in the list that is generated: if it is a legal string, it will be generated after some finite (but possibly long) time. Such a listing algorithm is called a *generative specification* of the language.

If the algorithm generates sentences in such an order that each new sentence is at least as long (has at least as many symbols) as the previous sentence, it is clearly possible to determine whether or not a given string is in the language. Whenever the algorithm begins to generate sentences longer than the string being tested, that string cannot be in the language unless it has already been generated. This type of algorithm enables us to decide after a finite number of steps whether or not a string is a legal sentence. If such a decision can be made in finite time for every string, the language is called *decidable*.

An alternative type of algorithm could be used to specify languages. In this second approach the string to be tested is fed as data into the algorithm. The algorithm then analyzes the input, performs whatever computation is necessary, and produces an answer indicating "yes, the string is legal," or "no, it is not

legal." This is called an *analytic* specification. A language with an analytic specification is decidable if the analyzer always stops after finitely many steps for every input string. Unfortunately, formal analytic specifications are often very difficult to derive; this chapter will deal primarily with generative specifications.

English is not suitable for defining languages formally because it is too vague and leads to ambiguous definitions. It is necessary to develop a formalism in which language definitions can be stated. This defining language is the *syntactic meta-language*. When we employ a language to talk about some language (itself or another) we shall call the latter the *object language* and the former the *meta-language*. A formal system is a meta-language. Symbols of the object language are called *terminal symbols*. Symbols of a meta-language that denote strings in the object language are called *nonterminals*. To formally define the meta-language would require a meta-meta-language; therefore, we hope that the meta-language is intuitively clear.

The first step of the definition process is to establish the *universe of discourse*. That is, it is necessary to specify the objects being discussed. The most elementary object is a symbol. Symbols are concatenated to form strings, which may or may not belong to the language.

Definition 1: An *alphabet* T is a finite set of symbols ("terminal symbols"). A formula (also called a string or a sentence) is a concatenation of symbols.

It is useful to have a notation for the class of all possible finite strings on an alphabet T. This is designated by T*. For any set U, U* represents the set of all possible finite concatenations of elements of the set U. Small Greek letters are used to denote strings. We commonly write λ to represent the "null" or "empty" string (i.e. the string which contains no elements).

Generally, a language does not include all possible strings on its alphabet. Only certain strings are valid formulae in the language. Thus:

Definition 2: A *language*, L, is a subset of the set of finite concatenations of symbols in an alphabet T. We write this $L \subset T^*$.

7.2.2 Development of Formal Specification

Let us turn for an example to English syntax. English is not just a collection of

groups of words – there is an underlying structure connecting the words. Given the sentence

"The student studies hard."

one can construct the sentence structure shown in Figure 7.1. In particular, one can separate the Noun Phrase (NP) from the Verb Phrase (VP) and then complete the analysis by subdividing these phrases into individual words. Since all sentences have some structure, it should be possible to generate this structure in small steps and thus build up to complex sentences. We can represent structure graphically with a syntax tree; branches from each node on the tree indicate its logical subdivisions.

For example, we might begin with the classification "sentence" and replace it by the pair NP and VP to construct one possible form that a sentence might have. It is notationally convenient to write this as

sentence → NP VP

It is clear from the context that NP represents the linguistic class "noun phrase." In Figure 7.1 we must distinguish between the string "NP", which may be replaced by the string "article", and the string "The", which cannot be replaced. We distinguish the names of classes from words in the language by placing the meta-brackets "⟨"and"⟩" around symbols used to represent linguistic classes. The first structure rule then becomes:

(1) ⟨sentence⟩ → ⟨NP⟩ ⟨VP⟩
(2) ⟨NP⟩ → ⟨article⟩ ⟨noun⟩

Similarly, a verb-phrase consists of a verb possibly modified by an adverb.

(3) ⟨VP⟩ → ⟨verb⟩
(4) ⟨VP⟩ → ⟨verb⟩ ⟨adverb⟩

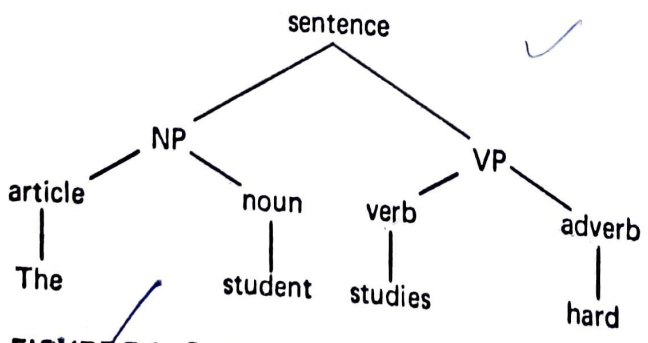

FIGURE 7.1 Syntax tree for "the student studies hard"

The way these structure transformations or rewriting rules are written allows a ⟨VP⟩ to have optionally one adverb while an ⟨NP⟩ must have exactly one adjective.

The last step is to list possible terminal-symbol replacements for the class representatives, ⟨article⟩, ⟨adverb⟩, ⟨noun⟩, and ⟨verb⟩:

(5) ⟨article⟩ → The
(6) ⟨noun⟩ → student
(7) ⟨verb⟩ → studies
(8) ⟨adverb⟩ → hard
(9) ⟨adverb⟩ → slowly

Using these rewriting rules, it is possible to build a sentence by replacing the linguistic class symbols with the structures that they represent. Figure 7.2 gives a derivation of "The student studies hard." By changing the last step and using

⟨adverb⟩ → slowly

it is possible to generate instead "The student studies slowly." The structure rewriting rules or *transformations* determine the form of the language generated. In our example, the language is a very small subset of English.

Such a system of rewriting rules constitutes an algorithm for generating sentences. By changing the symbols from words to phonemes and allowing for more complex structure transformations, one can describe much of English syntax in a similar manner. This scheme can handle most programming-language features as well. However, features that require more specification power will lead us to seek other, more discriminating methods of language specification.

7.3 FORMAL GRAMMARS

The above example indicated a way to formalize the process of generating the

Step	*Structure*	*Rule applied*
a)	⟨sentence⟩	
b)	⟨NP⟩ ⟨VP⟩	(1)
c)	⟨article⟩ ⟨noun⟩ ⟨VP⟩	(2)
d)	⟨article⟩ ⟨noun⟩ ⟨verb⟩ ⟨adverb⟩	(4)
e)	The ⟨noun⟩ ⟨verb⟩ ⟨adverb⟩	(5)
f)	The student ⟨verb⟩ ⟨adverb⟩	(6)
g)	The student studies ⟨adverb⟩	(7)
h)	The student studies hard	(8)

FIGURE 7.2 Steps in a derivation