

Chapter 2

Defining Program Syntax

2.1 Introduction

The simplest questions about a programming language are often questions of form. What does an expression, a statement, or a function definition look like? How are comments delimited? Where do the semicolons go? Questions like these are questions of programming-language *syntax*.

The *syntax* of a programming language is the part of the language definition that says how programs look: their form and structure.

The more difficult questions are often questions of behavior. What does a given expression, statement, or function do? How does it work? What can go wrong when it runs? Questions like these are questions of programming-language *semantics*.

The *semantics* of a programming language is the part of the language definition that says what programs do: their behavior and meaning.

This chapter introduces the formal grammar used to define programming-language syntax. (One of the formal techniques for defining programming-language semantics is introduced in Chapter 23.) By the end of the chapter, you should be able to understand these grammars and to write simple ones for yourself.

2.2 A Grammar Example for English

Let's start with an example of a grammar for a familiar language: English. An *article* can be the word *a* or *the*. We use the symbol $\langle A \rangle$ for *article* and express our definition this way:

$$\langle A \rangle ::= a \mid the$$

A *noun* can be the word *dog*, *cat*, or *rat*:

$$\langle N \rangle ::= dog \mid cat \mid rat$$

A *noun phrase* is an article followed by a noun:

$$\langle NP \rangle ::= \langle A \rangle \langle N \rangle$$

A *verb* can be the word *loves*, *hates*, or *eats*:

$$\langle V \rangle ::= loves \mid hates \mid eats$$

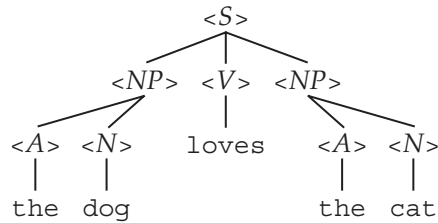
A *sentence* is a noun phrase, followed by a verb, followed by another noun phrase:

$$\langle S \rangle ::= \langle NP \rangle \langle V \rangle \langle NP \rangle$$

Put them all together and you have a grammar that defines a small subset of unpunctuated English:

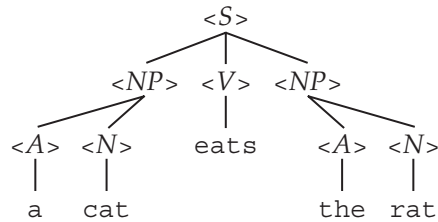
$$\begin{aligned} \langle S \rangle &::= \langle NP \rangle \langle V \rangle \langle NP \rangle \\ \langle NP \rangle &::= \langle A \rangle \langle N \rangle \\ \langle V \rangle &::= loves \mid hates \mid eats \\ \langle N \rangle &::= dog \mid cat \mid rat \\ \langle A \rangle &::= a \mid the \end{aligned}$$

How does such a grammar define a language? Think of the grammar as a set of rules that say how to build a tree. Put $\langle S \rangle$ at the root of the tree, and the grammar tells how children can be added at any point (node) in the tree. Such a tree is called a *parse tree*. It is a convention that parse trees are drawn growing downward, with the root at the top, like this:



The children of each node in a parse tree must follow the forms specifically allowed by the grammar. For example, an $\langle NP \rangle$ node must have the children $\langle A \rangle$ and $\langle N \rangle$, because the grammar includes only one rule for $\langle NP \rangle$, namely $\langle NP \rangle ::= \langle A \rangle \langle N \rangle$. Check the tree above; you will see that every node with a child or children follows one of the rules of the grammar.

By reading the fringe of the tree from left to right, you get a sentence in the language defined by the grammar. For the parse tree above, it is the sentence “the dog loves the cat.” Here is a different parse tree following the same grammar. The sentence “a cat eats the rat” is also in the language.



The language defined by a grammar is the set of all strings that can be formed as the fringes of parse trees generated by the grammar.

2.3 A Grammar Example for a Programming Language

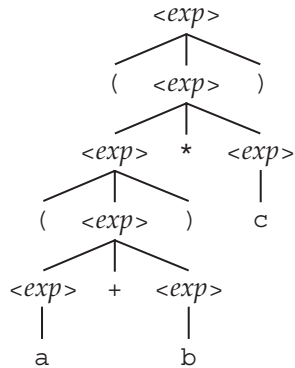
Here is an example of a grammar for a simple language of expressions with three variables.

$\langle exp \rangle ::= \langle exp \rangle + \langle exp \rangle \mid \langle exp \rangle * \langle exp \rangle \mid (\langle exp \rangle) \mid a \mid b \mid c$

The grammar says that an expression can be the sum of two expressions, the product of two expressions, an expression enclosed in parentheses, or one of the variables a , b , or c . Thus, our language includes expressions such as these:

a
 a + b
 a + b * c
 ((a + b) * c)

Here is a parse tree for that last expression, ((a + b) * c):

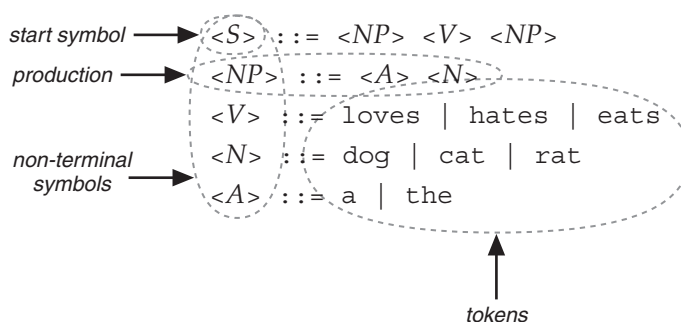


Unlike our first grammar, this one defines an infinite language; expressions can be arbitrarily long. It is a recursive grammar—an *<exp>* node can occur as the descendant of another *<exp>* node in the parse tree.

Finding a parse tree for a given string (with respect to a given grammar) is called *parsing* the string. Language systems must parse every program they run. There are many interesting algorithms for efficient parsing, but we will not go into them here. Sometimes a grammar allows several different parse trees for the same string. In Chapter 3 we will see why this is a problem and what to do about it. For now, our goal is simply to define the language. As long as the grammar generates at least one parse tree for a given string of tokens, that string is in the language.

2.4 A Definition of Grammars: Backus-Naur Form

Now that you have seen two concrete examples, let's go back and define all the parts of a grammar. A grammar has four important parts: the set of *tokens*, the set of *non-terminal symbols*, the set of *productions*, and a particular non-terminal symbol called the *start symbol*. Here are the four parts in our English language grammar:



The *tokens* are the smallest units of syntax. They are the strings and symbols that we choose not to think of as consisting of smaller parts. In our example we choose to think of the word *cat* as a single token, just as in programming languages one chooses to think of keywords (like *if*), names (like *fred*), and operators (like *!=*) as single tokens.

The *non-terminal symbols* are strings enclosed in angle brackets, such as $\langle NP \rangle$. The non-terminal symbols of a grammar often correspond to different kinds of language constructs. English has constructs like sentences and noun phrases; programming languages have constructs like statements and expressions. The grammar designates one of the non-terminal symbols as the root of the parse tree: the *start symbol*.

A *production* consists of a left-hand side, the separator $::=$, and a right-hand side. The left-hand side of a production is a single non-terminal symbol; the right-hand side is a sequence of one or more things, each of which can be either a token or a non-terminal symbol. A production gives one possible way of building a parse tree; it permits the non-terminal symbol on the left-hand side to have the symbols on the right-hand side, in order, as its children in a parse tree.

Productions with the same non-terminal symbol on the left-hand side may be written in an abbreviated form: the left-hand side, the separator $::=$, and the right-hand sides separated by the special symbol $|$. This abbreviated form, using $|$ to separate the right-hand sides, is for convenience only. It is equivalent to the full form, in which each production is written out separately. For example, here again is the grammar for a simple language of expressions:

$$\begin{aligned} \langle exp \rangle ::= & \langle exp \rangle + \langle exp \rangle \mid \langle exp \rangle * \langle exp \rangle \mid (\langle exp \rangle) \\ & \mid a \mid b \mid c \end{aligned}$$

Here it is written in a different way, without the $|$ notation:

```

<exp> ::= <exp> + <exp>
<exp> ::= <exp> * <exp>
<exp> ::= ( <exp> )
<exp> ::= a
<exp> ::= b
<exp> ::= c

```

One final detail about grammars: the special non-terminal symbol *<empty>* is sometimes used where the grammar needs to generate an empty string—a string of no tokens. For instance, an *if-then* statement with an optional *else* part might be defined like this:

```

<if-stmt> ::= if <expr> then <stmt> <else-part>
<else-part> ::= else <stmt> | <empty>

```

This form for writing grammars was developed by John Backus and Peter Naur around 1960. Their reports on the development of the Algol language (Algol 60) used such grammars to describe the syntax. Their notation for grammars is now called Backus-Naur Form (BNF). The grammar examples so far have used BNF, and they will continue to do so throughout the book.

2.5 Writing Grammars

Writing a grammar is a bit like writing a program; it uses some of the same mental muscles. A program is a finite, structured, mechanical thing that specifies a potentially infinite collection of runtime behaviors. To write a program, you have to be able to imagine how the code you are crafting will unfold when it executes. Similarly, a grammar is a finite, structured, mechanical thing that specifies a potentially infinite language. To write a grammar, you have to be able to imagine how the productions you are crafting will unfold when parse trees are built. The BNF syntax is simple enough, but it takes practice to write BNF definitions for programming-language constructs. Don't worry—learning to write grammars is not as difficult as learning to program!

The most important advice for writing grammars is this: *divide and conquer*. In writing grammars, as in programming, it is very important to break problems down into simpler subproblems. Let's take an example from Java: the Java statements that define local variables. Here are three:

```

float a;
boolean a,b,c;
int a=1, b, c=1+2;

```

These statements consist of a type name, followed by a list of one or more variable names being declared. The variable names are separated by commas. The statements end with semicolons. Each of the variable names may be followed by an equal sign and an expression; this syntax is used to specify a value for initializing the variable.

Our goal is to make a BNF grammar to define the language of these Java statements. We will use *<var-dec>* as the start symbol. Our very first step is to divide the problem into smaller pieces. The major components are the type name, the list of variables, and the final semicolon. The type name and the list of variables need further elaboration, so we will use non-terminal symbols for them. This gives a production:

$$\langle var-dec \rangle ::= \langle type-name \rangle \langle declarator-list \rangle ;$$

For type names, we can just list the primitive types of Java. (For a full Java grammar, you would have to allow class names, interface names, and array types here as well, but we will skip that part.)

$$\begin{aligned} \langle type-name \rangle ::= & \text{boolean} \mid \text{byte} \mid \text{short} \mid \text{int} \\ & \mid \text{long} \mid \text{char} \mid \text{float} \mid \text{double} \end{aligned}$$

Now we are left with the problem of defining the declarator list. This is still rather complicated, so we will divide the problem again. A declarator list is a list of one or more declarators, separated by commas. We can write this as the following:

$$\begin{aligned} \langle declarator-list \rangle ::= & \langle declarator \rangle \\ & \mid \langle declarator \rangle , \langle declarator-list \rangle \end{aligned}$$

This recursive rule says that a declarator list is either a single declarator or a declarator, followed by a comma, followed by a (smaller) declarator list. Notice how this always gives at least one declarator and gives commas between declarators.

That leaves us with the smaller problem of defining a declarator. It is a variable name followed, optionally, by an equal sign and an expression. We can write the following:

$$\langle declarator \rangle ::= \langle variable-name \rangle \mid \langle variable-name \rangle = \langle expr \rangle$$

For a full Java grammar, you would have to allow pairs of square brackets after the variable name, as one way of declaring arrays, and you would have to allow array initializers on the right-hand side of the equal sign. In addition, of course, we still

need parts of the grammar to define legal variable names and legal expressions, but we will end this example here.

One more important piece of advice about writing a grammar: test it as you would test a program. In our Java example, we started with a list of statements in the language. We could now test the grammar to make sure these statements could be parsed and to make sure illegal examples could not be parsed. Don't forget this step when you are doing the grammar-writing exercises at the end of the chapter.

2.6 Lexical Structure and Phrase Structure

The grammars seen so far have defined a language in terms of tokens, like names, keywords, and operators. Nevertheless, a program is usually stored not as a sequence of such tokens, but as a simple text file. A text file is just one long sequence of characters—letters, numbers, spaces, tabs, end-of-line markers, and so on. How should such a sequence of characters be divided into tokens?

A grammar whose tokens are not individual characters, but meaningful chunks like names, keywords, and operators, is incomplete. It defines the *phrase structure* of the language by showing how to construct parse trees with tokens at the leaves, but it does not define the *lexical structure* of the language by showing how to divide the program text into these tokens.

It is possible to combine the two parts by writing a single grammar whose tokens are individual characters. This is almost never done, because such grammars are horribly ugly and hard to read. For example, most modern programming languages allow white space—spaces, tabs, end-of-line markers, and so on—between tokens. A definition of such a language, going all the way down to the level of individual characters, would have to mention this white space in virtually every production. Our definition for the `if-then` statement would end up looking like this:

```
<if-stmt> ::= if <white-space> <expr> <white-space>
              then <white-space>
                  <stmt> <white-space> <else-part>
<else-part> ::= else <white-space> <stmt> | <empty>
```

To make matters worse, white space is required between some tokens (most languages treat `then p` differently from `thenp`), but not between others (most languages treat `a+b` the same as `a + b`). Don't forget comments too; in most modern languages, comments can occur in the middle of a line, like white space. A

grammar could specify this character-by-character level of detail, but it would be very tedious to read and to write.

Most modern language definitions specify the lexical structure and the phrase structure separately. Sometimes the lexical structure is defined informally, since it is usually much simpler than the phrase structure, but more often the lexical structure and the phrase structure are specified by two separate grammars. A token-level grammar (specifying the phrase structure) defines a program as a sequence of tokens. A character-level grammar (specifying the lexical structure) defines a text file as a sequence of program elements like tokens and white space. A character-level grammar might look like this:

```

<program-file> ::= <end-of-file> | <element> <program-file>
<element> ::= <token> | <one-white-space> | <comment>
<one-white-space> ::= <space> | <tab> | <end-of-line>
<token> ::= <name> | <operator> | <constant> | ...

```

(This is an incomplete example, of course. You would go on to define all of these non-terminal symbols, down to the character level.) As you can see, there are no phrase-level language constructs, no expressions or statements; the grammar just specifies how a file is to be divided into a sequence of tokens and other elements.

These two separate parts of the syntax definition are usually reflected in the implementation of language systems. A component called the *scanner* (or *lexer*) reads an input file and converts it to a stream of tokens, discarding the white space and comments. Then another component called the *parser* reads the stream of tokens and forms the parse tree.

We can distill a piece of wisdom from the history of programming-language design and implementation: separating lexical structure from phrase structure is a good idea. It makes language definitions easier to write, it makes language systems easier to implement, and it even makes languages easier for people to read. Some early languages had features that made the separation of lexical structure from phrase structure very difficult. For example, some early languages, including dialects of Fortran and Algol, allow spaces anywhere—even in the middle of a keyword. Some languages, including dialects of Fortran and PL/I, allow variable names to be the same as keywords. These features make compiling these languages considerably trickier, since scanning and parsing cannot be cleanly separated.

Regarding older languages, one historical aspect of lexical structure should be mentioned. The end-of-line markers in a program file are usually treated as white space in modern languages. They have no more significance than a space or a tab. They could be replaced with spaces, making the file one very long line, without

causing any problems other than the headache someone would get from trying to read it.

Since the end-of-line markers have no special significance, it follows that the column numbers have no special significance either. The fact that a character occurs in the sixth column rather than the seventh makes no difference in most modern languages. In some older languages, column positions are very significant. Languages that were popular when punched cards were still widely used often have a *fixed-format* lexical structure. This means that some columns in each line have special significance. In earlier dialects of Fortran, for example, there was one statement per card (we would now say “one statement per line”), and the first few columns were reserved for the statement label. Cobol also used a fixed-format lexical structure, as did Basic. The first languages that abandoned this column-oriented approach were advertised as being *free-format*. Today, since almost all modern languages are free-format (including the modern dialects of Fortran, Cobol, and Basic), few people bother to make this distinction.

2.7 Other Grammar Forms

There are many different ways of writing a grammar. They all capture the same basic ideas, but with different notational conventions. This section describes several different notations for grammars.

BNF

BNF (Backus-Naur Form) has many minor variations. Some people use = or → instead of : : =. Some people use a distinct typeface for terminal symbols, as we have done in our examples. Some people leave out the angle brackets, relying on the typeface to show the difference between tokens and non-terminal symbols. Some people use single quotation marks around tokens. This is an especially good idea when the token is the same as one of the BNF special characters <, >, |, or : : =. (These special characters are called *metasymbols* of the grammar; they are part of the language of the definition, not part of the language being defined.)

EBNF

A few more metasymbols can be added to BNF to help with common patterns of language definition. For example, [,] , { , } , (, and) might be added:

- [*something*] in the right-hand side of a production means that the *something* inside is optional.

- $\{ something \}$ in the right-hand side of a production means that the *something* inside can be repeated any number of times (zero or more).
- Parentheses are used to group things on the right-hand side so that $|$, $[]$, and $\{ \}$ can be used in the same production unambiguously.

With these new metasymbols, some common patterns can be defined more simply than with plain BNF. For example, an `if-then` statement with an optional `else` part might be defined like this:

$$\langle if\text{-}stmt \rangle ::= \text{if } \langle expr \rangle \text{ then } \langle stmt \rangle \text{ [else } \langle stmt \rangle \text{]}$$

Remember that the square brackets are not part of the language being defined; they are metasymbols that make the `else` part optional. A list of zero or more statements, each ending with a semicolon, might have a definition like this:

$$\langle stmt\text{-}list \rangle ::= \{ \langle stmt \rangle ; \}$$

Again, the curly brackets are not part of the language being defined; they are metasymbols that allow the $\langle stmt \rangle ;$ part to repeat zero or more times. A list of zero or more things, each of which can be either a statement or a declaration and each ending with a semicolon, might have a definition like this:

$$\langle thing\text{-}list \rangle ::= \{ (\langle stmt \rangle \mid \langle declaration \rangle) ; \}$$

The parentheses are metasymbols; they make it clear that the `;` token is not part of the choice permitted by the $|$ metasymbol.

Plain BNF can define all these patterns easily enough, but extra metasymbols like those above make the grammar easier to read. Much like the $|$ metasymbol from plain BNF, the new metasymbols allow you to express a collection of productions succinctly.

Any grammar syntax that extends BNF in this way is called an EBNF (extended BNF). Many variations have been used. Here is an excerpt from the definition for Java. Although the notation looks completely different, you can still see the EBNF ideas. The subscript *opt*, in particular, works like our square brackets, indicating an optional part of the syntax.

While Statement:

$$\text{while } (\textit{Expression}) \textit{Statement}$$

Do Statement:

$$\text{do } \textit{Statement} \text{ while } (\textit{Expression}) ;$$

For Statement:

$$\text{for } (\textit{ForInit}_{\text{opt}} ; \textit{Expression}_{\text{opt}} ; \textit{ForUpdate}_{\text{opt}}) \textit{Statement}$$

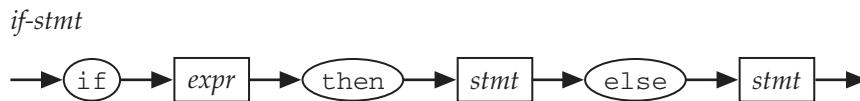
Syntax Diagrams

Another way to express grammars is graphically, using *syntax diagrams*. A syntax diagram uses a directional graph to show the productions for each non-terminal symbol. Any path through the graph gives a legal way to add children to the parse tree.

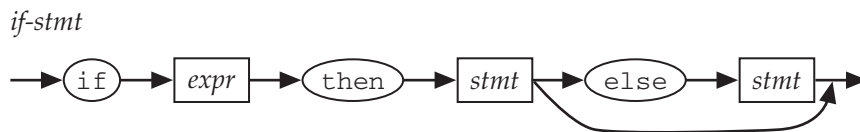
For simple BNF-style productions, the corresponding syntax diagram is just a chain of boxes (for non-terminal symbols) and ovals (for terminal symbols). For example, a production like this:

$$\langle \text{if-stmt} \rangle ::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$$

would be represented as a syntax diagram like this:



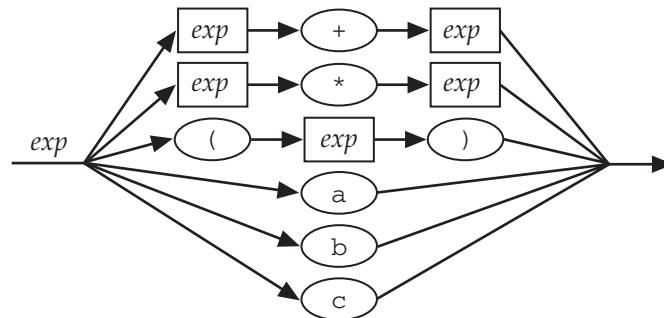
For EBNF productions that specify an optional part, the syntax diagram shows a path to bypass that part. For example, the `else` part could be made optional like this:



When there are multiple productions for a non-terminal symbol, the syntax diagram uses branching. For example, consider the earlier sample grammar for expressions:

$$\begin{aligned} \langle \text{exp} \rangle ::= & \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle) \\ & \mid a \mid b \mid c \end{aligned}$$

These six productions could be captured in a syntax diagram like this:

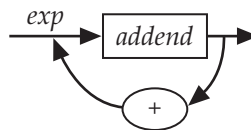


You can see why syntax diagrams are sometimes called *railroad diagrams*.

A syntax diagram can also have loops. These are a bit like the curly brackets of EBNF, since the loop can be repeated zero or more times in a path through the diagram. For example, this EBNF production specifies one or more addends separated by plus signs:

$$\langle \text{exp} \rangle ::= \langle \text{addend} \rangle \{ + \langle \text{addend} \rangle \}$$

This syntax diagram specifies the same thing:



In casual use, syntax diagrams are easy to read. They are often used in elementary, tutorial descriptions of programming languages. One drawback of syntax diagrams is that they make it hard to say exactly what the resulting parse tree looks like. (We will see in Chapter 3 why it is important to know exactly what the parse tree looks like.) Another drawback of syntax diagrams is that it is difficult to make them machine readable. There are software tools that automatically generate the code for the part of a language system that parses the language. Such tools need a machine-readable grammar as input, not a syntax diagram.

Formal, Context-free Grammars

A branch of theoretical computer science called *formal languages* studies formal grammars. Books and university courses on formal languages use notation that is slightly different than what has been shown so far, like this:

$$\begin{aligned} S &\rightarrow aSb \mid X \\ X &\rightarrow cX \mid \epsilon \end{aligned}$$

In formal languages, grammars of the kind we have been seeing are called *context-free grammars*. They are “context-free” because the children of a node in the parse tree depend only on that node’s non-terminal symbol; they do not depend on the context of neighboring nodes in the tree. Other kinds of grammars include *regular grammars*, which are less expressive, and *context-sensitive grammars*, which are more expressive.¹

■ ■ ■ 2.8 ■ ■ ■ Conclusion

The notations used for programming-language grammars have many variations, but they are largely cosmetic. The underlying ideas are the same. Grammars are used to define the syntax of programming languages, both at the level of the lexical structure—the division of the program text into meaningful tokens—and at the level of the phrase structure—the organization of tokens into a parse tree showing how the meaningful structures of a program (its expressions, statements, declarations, and so on) are organized.

There is a powerful connection between theory and practice here. The division between lexical structure and phrase structure is reflected in the implementation of language systems. In fact, if the grammars are in just the right form, they can be fed into parser-generators, which automatically generate those parts of the language system that scan and parse the language. When you write a grammar, you have several potential audiences: novice users, who just want to find out what legal programs look like; advanced users and language-system implementers, who need an exact, detailed definition to work from; and automatic tools, which derive other programs from your grammar automatically.

Perhaps even more important is the positive influence grammars have on language design. You can see this influence when you compare pre-BNF languages (like early Fortran) with post-BNF languages. A language with a simple, readable, short grammar has a simple, memorable phrase structure. That makes the language easier to learn and use.

1. Usually, the expressive power of regular grammars is just right for defining the lexical structure of a programming language, and the expressive power of context-free grammars is just right for defining the phrase structure.

Exercises

Exercise 1 Give a BNF grammar for each of the languages below. For example, a correct answer for “the set of all strings consisting of zero or more concatenated copies of the string *ab*” would be this grammar:

$$\langle S \rangle ::= ab \langle S \rangle \mid \langle \text{empty} \rangle$$

There are often many correct answers.

- a. The set of all strings consisting of zero or more *as*.
- b. The set of all strings consisting of an uppercase letter followed by zero or more additional characters, each of which is either an uppercase letter or one of the digits 0 through 9.
- c. The set of all strings consisting of one or more *as*.
- d. The set of all strings consisting of one or more digits. (Each digit is one of the characters 0 through 9.)
- e. The set of all strings consisting of zero or more *as* with a semicolon after each one.
- f. The set of all strings consisting of the keyword *begin*, followed by zero or more statements with a semicolon after each one, followed by the keyword *end*. Use the non-terminal $\langle \text{statement} \rangle$ for statements, and do not give productions for it.
- g. The set of all strings consisting of one or more *as* with a semicolon after each one.
- h. The set of all strings consisting of the keyword *begin*, followed by one or more statements with a semicolon after each one, followed by the keyword *end*. Use the non-terminal $\langle \text{statement} \rangle$ for statements, and do not give productions for it.
- i. The set of all strings consisting of one or more *as*, with a comma between each *a* and the next. (There should be no comma before the first or after the last.)
- j. The set of all strings consisting of an open bracket (the symbol $[$) followed by a list of one or more digits separated by commas, followed by a closing bracket (the symbol $]$).
- k. The set of all strings consisting of zero or more *as*, with a comma between each *a* and the next. (There should be no comma before the first or after the last.)
- l. The set of all strings consisting of an open bracket (the symbol $[$) followed by a list of zero or more digits separated by commas, followed by a closing bracket (the symbol $]$).

Exercise 2 Give an EBNF grammar for each of the languages of Exercise 1. Use the EBNF extensions wherever possible to simplify the grammars.

Exercise 3 Give a syntax diagram for each of the languages of Exercise 1. Use branching and loops in your syntax diagrams to make them as clear as possible.

Exercise 4 Consider the earlier simple grammar for expressions:

$$\begin{aligned} \langle \text{exp} \rangle ::= & \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle) \\ & \mid a \mid b \mid c \end{aligned}$$

Suppose the lexical structure of the language allows any number of spaces to occur anywhere in the expression. Give a BNF grammar that defines this explicitly, at the character level, using one grammar to capture both the phrase structure and the lexical structure. For example, your grammar should generate both $(a+b)$ and $(a + b)$. Use a single-quoted space, ' ', to indicate the space character in your grammar.

Further Reading

If you are interested in algorithms for efficient parsing, try any book on compiler construction. This one is such a classic that it even has a nickname; because of the illustration on its cover, it is called the “Dragon Book.”

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Boston, MA: Addison-Wesley, 1985.

The earlier quotation from the Java language definition was taken from

Gosling, James, Bill Joy, and Guy Steele. *The Java™ Language Specification*. Boston, MA: Addison-Wesley, 1996.

There have been occasional (unsuccessful) attempts to standardize the form of grammars for programming languages. This paper is an interesting historical example:

Wirth, Niklaus. “What can we do about the unnecessary diversity of notation for syntactic definitions?” *Communications of the ACM*, November 1977.

A recent attempt is the ISO standard for EBNF: *ISO/IEC 14977:1996*, which may be purchased from the ISO online. (The final document for this standard is not freely available online, though you may find drafts of it at various sites.) In spite of the existence of this standard, most language definitions continue to use ad hoc EBNFs.