CHAPTER 2

Grammars

Grammar, which knows how to control even kings . . .
—Molière, *Les Femmes Savantes* (1672), Act II, scene vi

THIS chapter describes the context-free grammars used in this specification to define the lexical and syntactic structure of a program.

2.1 Context-Free Grammars

A *context-free grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a *language*, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

2.2 The Lexical Grammar

A *lexical grammar* for the Java programming language is given in (§3). This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol *Input* (§3.5), that describe how sequences of Unicode characters (§3.1) are translated into a sequence of input elements (§3.5).

These input elements, with white space (§3.6) and comments (§3.7) discarded, form the terminal symbols for the syntactic grammar for the Java programming language and are called *tokens* (§3.5). These tokens are the identifiers

(§3.8), keywords (§3.9), literals (§3.10), separators (§3.11), and operators (§3.12) of the Java programming language.

2.3 The Syntactic Grammar

The *syntactic grammar* for the Java programming language is given in Chapters 4, 6–10, 14, and 15. This grammar has tokens defined by the lexical grammar as its terminal symbols. It defines a set of productions, starting from the goal symbol *CompilationUnit* (§7.3), that describe how sequences of tokens can form syntactically correct programs.

2.4 Grammar Notation

Terminal symbols are shown in `fixed width` font in the productions of the lexical and syntactic grammars, and throughout this specification whenever the text is directly referring to such a terminal symbol. These are to appear in a program exactly as written.

Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example, the syntactic definition:

IfThenStatement:
    `if ( ` *Expression* `)` *Statement*

states that the nonterminal *IfThenStatement* represents the token `if`, followed by a left parenthesis token, followed by an *Expression*, followed by a right parenthesis token, followed by a *Statement*.

As another example, the syntactic definition:

ArgumentList:
    Argument
    ArgumentList **,** *Argument*

states that an *ArgumentList* may represent either a single *Argument* or an *ArgumentList*, followed by a comma, followed by an *Argument*. This definition of *ArgumentList* is *recursive*, that is to say, it is defined in terms of itself. The result is that an *ArgumentList* may contain any positive number of arguments. Such recursive definitions of nonterminals are common.

The subscripted suffix "*opt*", which may appear after a terminal or nonterminal, indicates an *optional symbol*. The alternative containing the optional symbol

actually specifies two right-hand sides, one that omits the optional element and one that includes it.

This means that:

BreakStatement:
`break` *Identifier*$_{opt}$ `;`

is a convenient abbreviation for:

BreakStatement:
`break ;`
`break` *Identifier* `;`

and that:

BasicForStatement:
`for (` *ForInit*$_{opt}$ `;` *Expression*$_{opt}$ `;` *ForUpdate*$_{opt}$ `)` *Statement*

is a convenient abbreviation for:

BasicForStatement:
`for ( ;` *Expression*$_{opt}$ `;` *ForUpdate*$_{opt}$ `)` *Statement*
`for (` *ForInit* `;` *Expression*$_{opt}$ `;` *ForUpdate*$_{opt}$ `)` *Statement*

which in turn is an abbreviation for:

BasicForStatement:
`for ( ; ;` *ForUpdate*$_{opt}$ `)` *Statement*
`for ( ;` *Expression* `;` *ForUpdate*$_{opt}$ `)` *Statement*
`for (` *ForInit* `; ;` *ForUpdate*$_{opt}$ `)` *Statement*
`for (` *ForInit* `;` *Expression* `;` *ForUpdate*$_{opt}$ `)` *Statement*

which in turn is an abbreviation for:

BasicForStatement:
`for ( ; ; )` *Statement*
`for ( ; ;` *ForUpdate* `)` *Statement*
`for ( ;` *Expression* `; )` *Statement*
`for ( ;` *Expression* `;` *ForUpdate* `)` *Statement*
`for (` *ForInit* `; ; )` *Statement*
`for (` *ForInit* `; ;` *ForUpdate* `)` *Statement*
`for (` *ForInit* `;` *Expression* `; )` *Statement*
`for (` *ForInit* `;` *Expression* `;` *ForUpdate* `)` *Statement*

so the nonterminal *BasicForStatement* actually has eight alternative right-hand sides.

A very long right-hand side may be continued on a second line by substantially indenting this second line, as in:

ConstructorDeclaration:
ConstructorModifiers$_{opt}$ *ConstructorDeclarator*
Throws$_{opt}$ *ConstructorBody*

which defines one right-hand side for the nonterminal *ConstructorDeclaration*.

When the words "one of" follow the colon in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition. For example, the lexical grammar contains the production:

ZeroToThree: one of
`0 1 2 3`

which is merely a convenient abbreviation for:

ZeroToThree:
`0`
`1`
`2`
`3`

When an alternative in a lexical production appears to be a token, it represents the sequence of characters that would make up such a token. Thus, the definition:

BooleanLiteral: one of
`true false`

in a lexical grammar production is shorthand for:

BooleanLiteral:
`t r u e`
`f a l s e`

The right-hand side of a lexical production may specify that certain expansions are not permitted by using the phrase "but not" and then indicating the expansions to be excluded, as in the productions for *InputCharacter* (§3.4) and *Identifier* (§3.8):

InputCharacter:
UnicodeInputCharacter but not CR or LF

Identifier:
IdentifierName but not a *Keyword* or *BooleanLiteral* or *NullLiteral*

Finally, a few nonterminal symbols are described by a descriptive phrase in roman type in cases where it would be impractical to list all the alternatives:

RawInputCharacter:
any Unicode character