Chapter 2

Formal Languages and Grammars

> [If] one combines the words "to write-while-not-writing": for then it means, that he has the power to write and not to write at once; whereas if one does not combine them, it means that when he is not writing he has the power to write.
>
> — Aristotle, *Sophistical Refutations*, Book I, Part 4

> Never odd or even

> Is it crazy how saying sentences backwards creates backwards sentences saying how crazy it is

IN this chapter, we discuss the constructs (e.g., regular expressions and context-free grammars) for defining programming languages and explore their capabilities and limitations. Regular expressions can denote the lexemes of programming languages (e.g., an identifier), but not the higher-order syntactic structures (e.g., expressions and statements) of programming languages. In other words, regular expressions can denote identifiers and other lexemes while context-free grammars can capture the rules for a valid expression or statement. Neither can capture the rule that a variable must be declared before it is used. Context-free grammars are integral to both the definition and implementation of programming languages.

2.1 Chapter Objectives

- Introduce *syntax* and *semantics*.
- Describe formal methods for defining the syntax of a programming language.
- Establish an understanding of *regular languages*, *expressions*, and *grammars*.
- Discuss the use of *Backus–Naur Form* to define grammars.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

- Establish an understanding of *context-free languages* and *grammars*.
- Introduce the role of *context* in programming languages and the challenges in modeling context.

2.2 Introduction to Formal Languages

An *alphabet* is a finite set of symbols denoted by Σ. A *string* is a combination of symbols, also called characters, over an alphabet. For instance, strings over the alphabet Σ = {a, b, c} include a, aa, aaa, bb, aba, and abc. The *empty string* (i.e., a string of zero characters) is represented as ϵ. The *Kleene closure operator* of an alphabet (i.e., $\Sigma^\star$) represents the set of all possible strings that can be constructed through zero or more concatenations of characters from the alphabet. Thus, the set of all possible strings from the alphabet Σ = {a, b, c} is $\Sigma^\star$. While Σ is always finite, $\Sigma^\star$ is always infinite and always contains ϵ. The strings in $\Sigma^\star$ are *candidate sentences*.

A *formal language* is a set of strings. Specifically, a formal language L is a subset of $\Sigma^\star$, where each string from $\Sigma^\star$ in L is called a *sentence*. Thus, a formal language is a set of sentences. For instance, {a, aa, aaa, bb, aba, abc} is a formal language. There are finite and infinite languages. Finite languages have a finite number of sentences. The language described previously is a finite language (i.e., it has six sentences), whereas the Scheme programming language is an infinite language. Most interesting languages are infinite.

Determining whether a string s from $\Sigma^\star$ is in L (i.e., whether the candidate sentence s is a valid sentence) depends on the complexity of L. For instance, determining if a string s from $\Sigma^\star$ is in the language of all three-character strings is simpler than determining if s is in the language of *palindromes* (i.e., strings that read the same both forward and backward; e.g., dad, eye, or noon). Thus, determining if a string is a sentence is a set-membership problem.

Recall that *syntax* refers to the *structure or form* of language and *semantics* refers to the *meaning* of language. Formal notational systems are available to define the syntax and semantics of formal languages. This chapter is concerned with establishing an understanding of those formal systems and how they are used to define the syntax of programming languages. Armed with an understanding of the theory of formal language definition mechanisms and methods, we can turn to practice and study how those devices can be used to recognize a valid program prior to interpretation or compilation in Chapter 3.

There are three progressive types of sentence validity. A sentence is *lexically* valid if all the words of the sentence are valid. A sentence is *syntactically* valid if it is lexically valid and the ordering of the words is valid. A sentence is *semantically* valid if it is lexically and syntactically valid and has a valid meaning.

Consider the sentences in Table 2.1. The first candidate sentence is not lexically valid because "saintt" is not a word; therefore, the sentence cannot be syntactically or semantically valid. The second candidate is lexically valid because all of its words are valid, but it is not syntactically valid because the arrangement of those words does not conform to the subject–verb–article–object structure of English sentences; thus, it cannot be semantically valid. The third candidate is

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Candidate Sentence	Lexically Valid	Syntactically Valid	Semantically Valid
Augustine is a saintt.	×	×	×
Saint Augustine is a.	√	×	×
Saint is a Augustine.	√	√	×
Augustine is a saint.	√	√	√

Table 2.1 Progressive Types of Sentence Validity

Candidate Expression	Lexically Valid	Syntactically Valid	Semantically Valid
`= intt + 3 y x;`	×	×	×
`= int + 3 y x;`	√	×	×
`int 3 = y + x;`	√	√	×
`int y = x + 3;`	√	√	√

Table 2.2 Progressive Types of Program Expression Validity

lexically valid because all of its words are valid and syntactically valid because the arrangement of those words conforms to the subject–verb–article–object structure of English sentences, but it is not semantically valid because the sentence does not make sense. The fourth candidate sentence is lexically, syntactically, and semantically valid. Notice that these types of sentence validity are progressive. Once a candidate sentence fails any test for validity, it automatically fails a more stringent test for validity. In other words, if a candidate sentence does not even have valid words, those words can never be arranged correctly. Similarly, if the words of a candidate sentence are not arranged correctly, that sentence can never make semantic sense. For instance, the second sentence in Table 2.1 is not syntactically valid so it can never be semantically valid.

Recall that validating a string as a sentence is a set-membership problem. We saw previously that the first step to determining if a string of words, where a word is a string of non-whitespace characters, is a sentence is to determine if each individual word is a sentence (in a simpler language). Only after the validity of every individual word in the entire string is established can we examine whether the words are arranged in a proper order according to the particular language in which this particular, entire string is a candidate sentence. Notice that these steps are similar to the steps an interpreter or compiler must execute to determine the validity of a program (i.e., to determine if the program has any syntax errors). Table 2.2 illustrates these steps of determining program expression validity. Next, we examine those steps through a formal lens.

2.3 Regular Expressions and Regular Languages

2.3.1 Regular Expressions

Since languages can be infinite, we need a concise, yet formal method of describing languages. A *regular expression* is a pattern represented as a string that concisely

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Regular Expression	Denotes	Language
Atomic Regular Expressions		
x	the single character x	$L(x) = x$
ϵ	empty string	$L(\epsilon) = \epsilon$
$\emptyset$	empty set	$L(\emptyset) = \{\}$
Compound Regular Expressions		
$(r^\star)$	zero or more of r_1	$L((r)^\star) = L(r)^\star$
$(r_1 r_2)$	concatenation of r_1 and r_2	$L(r_1 r_2) = L(r_1)L(r_2)$
$(r_1 + r_2)$	either r_1 or r_2	$L(r_1 + r_2) = L(r_1) \cup L(r_2)$

Table 2.3 Regular Expressions (Key: $x \in \Sigma$.)

and formally denotes the strings of a language. A regular expression is itself a string in a language, albeit a *metalanguage*—a language used to describe a language. Thus, regular expressions have their own alphabet and syntax, not to be confused with the alphabet and syntax of the language that a regular expression is used to define.

Table 2.3 presents the six primitive constructs from which any regular expression can be constructed. These constructs are factored into three *primitive* regular expressions (i.e., x, ϵ, and $\emptyset$) and three *compound* regular expressions (constructed with the $\star$, concatenation, and + operators). Thus, some characters in the alphabet of regular expressions are special and called *metacharacters* [e.g., ϵ, $\emptyset$, $\star$, +, (, and)].[1] In particular, $\Sigma_{RE} = \{\epsilon, \emptyset, \star, +, (,)\}$. We have already encountered the $\star$ (or Kleene closure) operator as applied to a set of symbols (or alphabet). Here, it is applied to a regular expression r, where $r^\star$ denotes zero or more occurrences of r. For instance, the regular expression opus* defines the language {opu, opus, opuss, opusss, ...}. The regular expression (ab)* denotes the language {ϵ, ab, abab, ababab, ...}. In both cases, the set of sentences, and therefore the language, are infinite. In short, a regular expression denotes a set of strings (i.e., the sentences of the language that the regular expression denotes).

The + operator is used to construct a compound regular expression from two subexpressions, where the language denoted by the compound expression contains the strings from the union of the sets denoted by the two subexpressions. For instance, the regular expression "the + Java + programming + language" denotes the language {the, Java + programming, language}. Similarly,

$$\text{opus}(1+2+3+4+5+6+7+8+9)(0+1+2+3+4+5+6+7+8+9)^\star$$

denotes the language

{opus1, opus2, ..., opus9, opus10, opus11, ..., opus98, opus99}

1. Sometimes some of the characters in the set of metacharacters are also in the alphabet of the language being defined (i.e., $\Sigma_{RE} \cap \Sigma \neq \emptyset$). In these cases, there must be a way to disambiguate the meaning of the overloaded character. For example, a \ is used in UNIX to *escape* the special meaning of the metacharacter following it.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

and

$$(0+1+\ldots+8+9)(0+1+\ldots+8+9)(0+1+\ldots+8+9)\text{–}(0+1+\ldots+8+9)(0+1+\ldots+8+9)\ \text{–}(0+1+\ldots+8+9)(0+1+\ldots+8+9)(0+1+\ldots+8+9)(0+1+\ldots+8+9)$$

which denotes the language of Social Security numbers.

Table 2.4 presents a set of compound regular expressions with the associated language that each denotes. Parentheses in compound regular expressions are used for grouping subexpressions. In the absence of parentheses, highest to lowest precedence proceeds in a top-down manner, as shown in Table 2.3 (e.g., $\star$ has the highest precedence and $+$ has the lowest precedence).

An enumeration of the elements of a set of sentences defines a formal language *extensionally*, while a regular expression defines a formal language *intensionally*.

A regular expression is a *denotational* construct for a (certain type of) formal language. In other words, a regular expression denotes sentences from the language it represents. For example, the regular expression opus* denotes the language {opu, opus, opuss, opusss, ...}.

Regular expressions are implemented in a variety of UNIX tools (e.g., `grep`, `sed`, and `awk`). Most programming languages implement regular expressions

Regular Expression	Denotes	Regular Language
abc	the string abc	{abc}
a + b + c	any one character in the set {a, b, c}	{a, b, c}
a + e + i + o + u	any one character in the set {a, e, i, o, u}	{a, e, i, o, u}
ϵ + a	"a" or the empty string	{ϵ, a}
a(b + c)	"a" followed by any character in the set {b, c}	{ab, ac}
ab + cd	any one string in the set {ab, cd}	{ab, cd}
a(b + c)d	"a" followed by any character in the set {b, c} followed by "d"	{abd, acd}
a*	"a" zero or more times	{ϵ, a, aa, aaa, ...}
aa*	"a" one or more times	{a, aa, aaa, ...}
aaaa*	"a" three or more times	{aaa, aaaa, aaaaa, ...}
aaaaaaaa	"a" exactly eight times	{aaaaaaaa}
a + aa + aaa + aaaa + aaaaa	"a" between one and five times	{a, aa, aaa, aaaa, aaaaa}
aaa + aaaa + aaaaa + aaaaaa	"a" between three and six times	{aaa, aaaa, aaaaa, aaaaaa}

Table 2.4 Examples of Regular Expression ($\Sigma_{\mathsf{re}} = \{\epsilon, \emptyset, \star, +, (,)\}$.)

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

either natively in the case of scripting languages (e.g., Perl and Tcl) or through a library or package (e.g., Python, Java, Go).[2]

2.3.2 Finite-State Automata

Recall that a regular expression intensionally *denotes* (the sentences of) a regular language. Now we turn to a computational mechanism that can *decide* whether a string is a sentence in a particular language—the set-membership problem mentioned previously. A *finite-state automaton* (FSA) is a model of computation used to recognize whether a string is a sentence in a particular language. Figure 2.1 presents a finite-state automaton[3] that recognizes sentences in the language denoted by the regular expression

$$(1{+}2{+}\cdots{+}8{+}9)(0{+}1{+}2{+}\cdots{+}8{+}9)^\star +$$
$$(_{+}a{+}b{+}\cdots{+}y{+}z{+}A{+}B{+}\cdots{+}Y{+}Z)(_{+}a{+}b{+}\cdots{+}y{+}z{+}A{+}B{+}\cdots{+}Y{+}Z{+}0{+}1{+}\cdots{+}8{+}9)^\star$$

which describes positive integers and legal identifiers in the C programming language.

We can think of an automaton as a simplified computer (Figure 2.1) that, when given a string (i.e., candidate sentence) as input, outputs either yes or no to indicate whether the input string is in the particular language that the machine has been

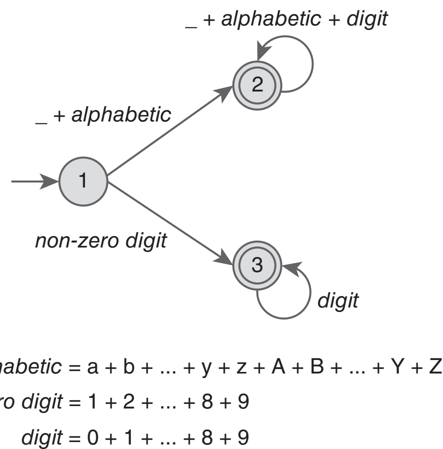

Figure 2.1 A finite-state automaton for a legal identifier and positive integer in the C programming language.

2. The set of metacharacters available to construct regular expressions in most programming languages and UNIX tools has evolved over the years beyond syntactic sugar (for formal regular expressions) and can be used to denote non-regular languages. For instance, the `grep` regular expression `\([a-z]\)\([a-z]\)[a-z]\2\1` matches the language of palindromes of five-character, lowercase letters—a non-regular language.

3. More precisely, this finite-state automaton is a *nondeterministic finite automaton* or NFA. However, the FSA in Figure 2.1 is not a formally a FSA because it has only three transitions, but it should have one for each individual input character that moves the automaton from one state to another. For instance, there should be nine transitions between states 1 and 3—one for each non-zero digit.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

constructed to recognize. In particular, if after running the entire string through the machine one character a time, the automaton is left in an accepting state (i.e., one represented by a double circle, such as states 2 and 3 in Figure 2.1), the string is a sentence. If after running the string through the machine, the machine is left in a non-accepting state (i.e., one represented by a single circle, such as state 1 in Figure 2.1), the string is not a sentence. Formally, a FSA *decides* a language.

2.3.3 Regular Languages

A *regular language* is a formal language that can be denoted by a regular expression and recognized by a finite-state automaton. A regular language is the most restrictive type of formal language. A regular expression is a *denotational* construct for a regular language. In other words, a regular expression denotes sentences from the language it represents. For example, the regular expression opus* denotes the regular language {opu, opus, opuss, opusss, ...}.

If a language is finite, it can be denoted by a regular expression. This regular expression is constructed by enumerating each element of the finite set of sentences in the language with intervening + metacharacters. For example, the finite language {a, b, c} is denoted by the regular expression a + b + c. Thus, all finite languages are regular, but the reverse is not true.

In summary, a regular language (which is the most restrictive type of formal language) is denoted by a regular expression and is recognized by a finite-state automaton (which is the simplest model of computation).

Conceptual Exercises for Section 2.3

Exercise 2.3.1 Give a *regular expression* that defines a language whose sentences are the set of all strings of alphabetic (in any case) and numeric characters that are permissible as login IDs for a computer account, where the first character must be a letter and the string must contain at least one character, but no more than eight.

Exercise 2.3.2 Give a *regular expression* that denotes the language of five-digit zip codes (e.g., 45469) with an optional four-digit extension (e.g., 45469-0280).

Exercise 2.3.3 Give a *regular expression* to denote the language of phrases of exactly three words separated by whitespace, where a word is any string of non-whitespace characters and whitespace is any string of spaces or tabs. In your expression, represent a single space character as □ and a single tab character as →. Among the set of sentences that your regular expression denotes are the three underlined substrings in the following string: A room with a view.

Exercise 2.3.4 Give a *regular expression* that denotes the language of decimals representing ASCII characters (i.e., integers between between 0–127, without leading 0s for any integer except 0 itself). Thus, the strings 0, 2, 25, and 127 are in the language, but 00, 02, 000, 025, and 255 are not.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Exercise 2.3.5 Give a *regular expression* for the language of zero or more nested, matched parentheses, where every opening and closing parenthesis has a match of the other type, with the matching opening parentheses appearing before the matching closing parentheses in the sentence, but where the parentheses are never nested more than three levels deep (i.e., no character in the string is ever within more than three levels of nesting). To avoid confusion between parentheses in the string and parentheses used for grouping in the regular expression, use the "l" and "r" characters to denote left (i.e., opening) and right (i.e., closing) parentheses in the string, respectively.

Exercise 2.3.6 Since all finite languages are regular, we can construct an FSA for any finite language. Describe how an FSA for a finite language can be constructed.

2.4 Grammars and Backus–Naur Form

Grammars are yet another way to define languages. A *formal grammar* is used to define a formal language. The following is a formal grammar defined for the language denoted by the $a^\star$ regular expression:

$$\begin{array}{lcl} S & \rightarrow & \text{a}\, S \\ S & \rightarrow & \epsilon \end{array}$$

The formal definition of a *grammar* is $G = (V, \Sigma, P, S)$, where

- V is a set of *non-terminal* symbols (e.g., $\{S\}$ in the grammar shown here).
- Σ is an alphabet (e.g., $\Sigma = \{\text{a}\}$).
- P is a finite set of *production rules*, each of the form $x \rightarrow y$, where x and y are strings over $\Sigma \cup V$ and $x \neq \epsilon$ (or, alternatively, P is a finite relation $P : V \rightarrow (V \cup \Sigma)^\star$ (e.g., each line in the example grammar is a production rule).
- S is the *start symbol* and $S \in V$ (e.g., S).

V is called the *non-terminal* alphabet, while Σ is the *terminal* alphabet, and $V \cap \Sigma = \emptyset$. In other words, strings of symbols from Σ are called *terminals*. Formally, for each terminal t, $t \in \Sigma^\star$ (e.g., "a" in the example grammar is the only terminal). We can think of terminals as the atomic lexical units of a program, called *lexemes*. The example grammar is defined formally as $G = (\{S\}, \{a\}, \{S \rightarrow aS, S \rightarrow \epsilon\}, S)$.

Notice that a grammar is a metalanguage, or a language that describes a language. Moreover, like regular expressions, grammars have their own syntax—again, not to be confused with the syntax of the languages they are used to define. Thus, grammars themselves are defined using a metalanguage—a language for defining a language, which, in this case, could itself be called a metalanguage—a language for defining a language defines a language! A metalanguage for defining grammars is called *Backus–Naur Form* (BNF). BNF takes its name from the last names of John **B**ackus, who developed the notation and used it to define the syntax of ALGOL 58 at IBM, and Peter **N**aur, who later extended the notation and used it for ALGOL 60 (Section 2.10). The example grammar G is in BNF.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

By applying the production rules, beginning with the start symbol, a grammar can be used to *generate* a sentence from the language it defines. For instance, the following is a *derivation* of the sentence aaaa:

$$S \overset{r_1}{\Rightarrow} \text{a}S \overset{r_1}{\Rightarrow} \text{aa}S \overset{r_1}{\Rightarrow} \text{aaa}S \overset{r_1}{\Rightarrow} \text{aaaa}S \overset{r_2}{\Rightarrow} \text{aaaa}$$

Note that every application of a production rule involves replacing the non-terminal on the left-hand side of the rule with the entire right-hand side of the rule. The semantics of the symbol $\Rightarrow$ is "derives" and the symbol indicates a one-step derivation relation. The r_n annotation over each $\Rightarrow$ symbol indicates which production rule is used in the substitution. The $\Rightarrow^\star$ symbol indicates a zero-or-more-step derivation relation. Thus, $S \Rightarrow^\star$ aaaa.

A formal grammar is a *generative* construct for a formal language. In other words, a grammar generates sentences from the language it defines. Formally, if $G = (V, \Sigma, S, P)$, then the language generated by G is $L(G) = \{x \mid x \in \Sigma^\star$ and $S \Rightarrow^\star x\}$. A grammar for the language denoted by the regular expression opus* is $(\{S, W\}, \{\text{o}, \text{p}, \text{u}, \text{s}\}, \{S \rightarrow \text{opu}W, W \rightarrow \text{s}W, W \rightarrow \epsilon\})$, which generates the language {opu, opus, opuss, . . . }.

2.4.1 Regular Grammars

Linguist Noam Chomsky formalized a set of grammars in the late 1950s—unintentionally making a seminal contribution to computer science. Chomsky's work resulted in the *Chomsky hierarchy*, which is a progressive classification of formal grammars used to describe the syntax of languages.

Level 1 of the hierarchy defines a type of formal grammar, called a *regular grammar*, which is most appropriate for describing the lexemes of programming languages (e.g., keywords in C such as `int` and `float`). The complete set of lexemes of a language is referred to as a *lexicon* (or *lexis*). A grammar is a *regular grammar* if and only if every production rule is in one of the following two forms:

$$\begin{aligned} X &\rightarrow zY \\ X &\rightarrow z \end{aligned}$$

where $X \in V$, $Y \in V$, and $z \in \Sigma^\star$. A grammar whose production rules conform to these patterns is called a *right-linear grammar*. Grammars whose production rules conform to the following pattern are called *left-linear grammars*:

$$\begin{aligned} X &\rightarrow Yz \\ X &\rightarrow z \end{aligned}$$

Left-linear grammars also generate regular languages. Notice the one-for-one replacement of a non-terminal for a non-terminal in V in the rules of a right- or left-linear grammar. Thus, a regular grammar is also referred to as a *linear grammar*. Regular grammars define a class of languages known as *regular languages*.

A regular grammar is a *generative* device for a regular language. In other words, it generates sentences from the regular language it defines. However, a grammar does not have to be regular to generate a regular language. We leave it as an

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Regular expressions	*denote*	regular languages.
Regular grammars	*generate*	regular languages.
Finite-state automata	*recognize*	regular languages.
All three	*define*	regular languages.

Table 2.5 Relationship of Regular Expressions, Regular Grammars, and Finite-State Automata to Regular Languages

exercise to define a non-regular grammar that defines a regular language (i.e., one that can be denoted by a regular expression; Conceptual Exercise 2.10.7).

In summary, a regular language (which is the most restrictive type of formal language) is:

- *denoted* by a regular expression,
- *recognized* by a finite-state automaton (which is the simplest model of computation), and
- *generated* by a regular grammar.

See Table 2.5.

Regular expressions, regular grammars, and finite-state automata are equivalent in their power to denote, generate, and recognize regular languages. In other words, there does not exist a regular language that could be *denoted* with a regular expression that could not be *decided* by a FSA or *generated* by a regular grammar. Mechanical techniques can be used to convert from one of these three models of a regular language to any of the other two.

An enumeration of the elements of a set of sentences defines a regular language *extensionally*, while a regular expression, finite-state automata, and regular grammar each define a regular language *intensionally*.

Some formal languages are not regular. Moreover, grammars, in addition to being language-generation devices, can be used (like an FSA) as language-recognition devices. We return to this theme of the dual nature of grammars while discussing context-free grammars in the next section.

2.5 Context-Free Languages and Grammars

There is a limit on the expressivity of regular expressions and regular grammars. In other words, some languages cannot be defined by a regular expression or a regular grammar. As a result, there are also computational limits on the sentence-recognition capabilities of finite-state automata. Consider the language L of balanced parentheses, whose sentences are strings of nested parentheses with the same number of opening parentheses in the first half of the string as closing parentheses in the second half of the string: $L = \{(^n)^n \mid n \geqslant 0 \text{ and } \Sigma = \{(,)\}\}$. The strings () and (((()))) are balanced and, therefore, sentences in this language; conversely, the strings (, ()), and ((()) are unbalanced and not in the language. In formal language theory, a language of strings of balanced parentheses is called

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

a *Dyck language*. A Dyck language cannot be defined by a regular expression. Alternatively, consider the language L of binary *palindromes*—binary numbers that read the same forward as backward: $L = \{xx^r \mid x \in \{0, 1\}^\star\}$, where x^r means "a reversed copy of x." The strings 00, 11, 101, 010, 1111, and 001100 are in the language, but 01, 10, 1000, and 1101 are not. We cannot construct either a regular expression or a regular grammar to define these languages. In other words, neither a regular expression nor a regular grammar has the expressive capability to model these languages.

What capability is absent from regular expressions or regular grammars that renders them unusable for defining these languages? Consider how we might implement a computer program to recognize strings of balanced parentheses. We could use a stack data structure to match each opening parenthesis with a closing parenthesis. Whenever we encounter an open parenthesis, we push it onto the stack; whenever we see a closing parenthesis, we pop from the stack. If the stack is empty when all the characters in the string are consumed, then the parentheses in the string are balanced and the string is a sentence; otherwise, it is not. The utility of a stack (formally, a *pushdown automata*) for this purpose implies that we need some form of *unbounded* memory to the match parentheses in the candidate string (i.e., to keep track of the number of unclosed open parentheses unknown a priori). Recall that the F in FSA stands for *finite*.

While regular expressions can denote the lexemes (e.g., identifiers) of programming languages, they cannot model syntactic structures nested arbitrarily deep that involve *balanced pairs* of lexemes (e.g., matched curly braces or `begin`/`end` keyword pairs identifying blocks of code; or parentheses in mathematical expressions), which are ubiquitous in programming languages. In other words, a sequence of lexemes in a program must be arranged in a particular order, and that order cannot be captured by a regular expression or a regular grammar. Regular expressions are expressive enough to denote the lexemes of programming languages, but not the higher-order syntactic structures (e.g., expressions and statements) of programming languages. Therefore, we must turn our attention to formal grammars with greater expressive capabilities than regular grammars if we need to define more sophisticated formal languages, including, in particular, programming languages.

Level 2 of the Chomsky hierarchy defines a type of formal grammar, called a *context-free grammar*, which is most appropriate for defining (and, as we see later, implementing) programming languages. Like the production rules of a regular grammar, the productions of a context-free grammar must conform to a particular pattern, but that pattern is less restrictive than the pattern to which regular grammars must adhere. The productions of a context-free grammar may have only one non-terminal on the left-hand side. Formally, a grammar is a *context-free grammar* if and only if every production rule is in the following form:

$$X \rightarrow \gamma$$

where $X \in V$ and $\gamma \in (\Sigma \cup V)^\star$, there is only one non-terminal on the left-hand side of any rule, and X can be replaced with γ anywhere. Notice that since this

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

definition is less restrictive than that of a regular grammar, every regular grammar is also a context-free grammar, but the reverse is not true.

Context-free grammars define a class of formal languages called *context-free languages*. The concept of balanced pairs of syntactic entities—the essence of a Dyck language—is at the heart of context-free languages. This single syntactic feature (and its variations) distinguishes regular languages from context-free languages, and the capability of expressing balanced pairs is the essence of a context-free grammars.

2.6 Language Generation: Sentence Derivations

Consider the following a context-free grammar defined in BNF for simple English sentences:

(r_1)	<*sentence*>	→	<*article*> <*noun*> <*verb*> <*adverb*>.
(r_2)	<*article*>	→	a
(r_3)	<*article*>	→	an
(r_4)	<*article*>	→	the
(r_5)	<*noun*>	→	apple
(r_6)	<*noun*>	→	rose
(r_7)	<*noun*>	→	umbrella
(r_8)	<*verb*>	→	is
(r_9)	<*verb*>	→	appears
(r_{10})	<*adverb*>	→	here
(r_{11})	<*adverb*>	→	there

As briefly shown here, grammars are used to generate sentences from the language they define. Beginning with the start symbol and repeatedly applying the production rules until the string contains no non-terminals results in a *derivation*—a sequence of applications of the production rules of a grammar beginning with the start symbol and ending with a sentence (i.e., a string of all terminals arranged according to the rules of the grammar). For example, consider deriving the sentence "the apple is there." from the preceding grammar. The r_n parenthesized annotation on the right-hand side of each application indicates which production rule was used in the substitution:

<*sentence*>	⇒	<*article*><*noun*><*verb*><*adverb*> .	(r_1)
	⇒	<*article*> <*noun*> <*verb*> there.	(r_{11})
	⇒	<*article*> <*noun*> is there.	(r_8)
	⇒	<*article*> apple is there.	(r_5)
	⇒	the apple is there.	(r_4)

The result (on the right-hand side of the ⇒ symbol) of each step is a string containing terminals and non-terminals that is called a *sentential form*. A *sentence* is a sentential form containing only terminals.

Peter Naur extended BNF for ALGOL 60 to make the definition of the production rules in a grammar more concise. While we discuss the details of

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

the extension, called *Extended Backus–Naur Form* (EBNF), later (in Section 2.10), we cover one element of the extension, alternation, here since we use it in the following examples. Alternation allows us to consolidate various production rules whose left-hand sides match into a single rule whose right-hand side consists of the right-hand sides of each of the individual rules separated by the | symbol. Therefore, alternation is syntactic sugar, in that any grammar using it can be rewritten without it. *Syntatic sugar* is a term coined by Peter Landin that refers to special, typically terse syntax in a language that serves only as a convenient method for expressing syntactic structures that are traditionally represented in the language through uniform and often long-winded syntax. With alternation, we can define the preceding grammar, which contains 11 production rules with only 5 rules:

(r_1) <sentence> → <article> <noun> <verb> <adverb>.
(r_2) <article> → a | an | the
(r_3) <noun> → apple | rose | umbrella
(r_4) <verb> → is | appears
(r_5) <adverb> → here | there

To differentiate non-terminals from terminals, especially when using grammars to describe programming languages, we place non-terminal symbols within the symbols < > by convention.[4]

Consider the following context-free grammar for arithmetic expressions for a simple four-function calculator with three available identifiers:

(r_1) <expr> ::= <expr> + <expr>
(r_2) <expr> ::= <expr> − <expr>
(r_3) <expr> ::= <expr> ⋆ <expr>
(r_4) <expr> ::= <expr> / <expr>
(r_5) <expr> ::= <id>
(r_6) <id> ::= x | y | z
(r_7) <expr> ::= (<expr>)
(r_8) <expr> ::= <number>
(r_9) <number> ::= <number> <digit>
(r_{10}) <number> ::= <digit>
(r_{11}) <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A derivation is called *leftmost* if the leftmost non-terminal is always replaced first in each step. The following is a leftmost derivation of 132:

<expr> ⇒ <number> (r_8)
⇒ <number><digit> (r_9)
⇒ <number><digit><digit> (r_9)

4. Interestingly, Chomsky and Backus/Naur developed their notion for defining grammars independently. Thus, the two notions have some minor differences: Chomsky used uppercase letters for non-terminals, the → symbol in production rules, and ϵ as the empty string; Backus/Naur used words in any case enclosed in <> symbols, ::=, and <empty>, respectively.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

$$\begin{array}{lll} \Rightarrow & \langle \mathit{digit}\rangle\langle \mathit{digit}\rangle\langle \mathit{digit}\rangle & (r_{10}) \\ \Rightarrow & 1\ \langle \mathit{digit}\rangle\langle \mathit{digit}\rangle & (r_{11}) \\ \Rightarrow & 13\ \langle \mathit{digit}\rangle & (r_{11}) \\ \Rightarrow & 132 & (r_{11}) \end{array}$$

A derivation is called *rightmost* if the rightmost non-terminal is always replaced first in each step. The following is a rightmost derivation of 132:

$$\begin{array}{llll} \langle \mathit{expr}\rangle & \Rightarrow & \langle \mathit{number}\rangle & (r_8) \\ & \Rightarrow & \langle \mathit{number}\rangle\langle \mathit{digit}\rangle & (r_9) \\ & \Rightarrow & \langle \mathit{number}\rangle\ 2 & (r_{11}) \\ & \Rightarrow & \langle \mathit{number}\rangle\langle \mathit{digit}\rangle\ 2 & (r_9) \\ & \Rightarrow & \langle \mathit{number}\rangle\ 32 & (r_{11}) \\ & \Rightarrow & \langle \mathit{digit}\rangle\ 32 & (r_{10}) \\ & \Rightarrow & 132 & (r_{11}) \end{array}$$

Some derivations, such as the next two derivations, are neither leftmost nor rightmost:

$$\begin{array}{llll} \langle \mathit{expr}\rangle & \Rightarrow & \langle \mathit{number}\rangle & (r_8) \\ & \Rightarrow & \langle \mathit{number}\rangle\langle \mathit{digit}\rangle & (r_9) \\ & \Rightarrow & \langle \mathit{number}\rangle\langle \mathit{digit}\rangle\langle \mathit{digit}\rangle & (r_9) \\ & \Rightarrow & \langle \mathit{number}\rangle\langle \mathit{digit}\rangle\ 2 & (r_{11}) \\ & \Rightarrow & \langle \mathit{number}\rangle\ 32 & (r_{11}) \\ & \Rightarrow & \langle \mathit{digit}\rangle\ 32 & (r_{10}) \\ & \Rightarrow & 132 & (r_{11}) \end{array}$$

$$\begin{array}{llll} \langle \mathit{expr}\rangle & \Rightarrow & \langle \mathit{number}\rangle & (r_8) \\ & \Rightarrow & \langle \mathit{number}\rangle\langle \mathit{digit}\rangle & (r_9) \\ & \Rightarrow & \langle \mathit{number}\rangle\langle \mathit{digit}\rangle\langle \mathit{digit}\rangle & (r_9) \\ & \Rightarrow & \langle \mathit{number}\rangle\ 3\ \langle \mathit{digit}\rangle & (r_{11}) \\ & \Rightarrow & \langle \mathit{digit}\rangle\ 3\ \langle \mathit{digit}\rangle & (r_{10}) \\ & \Rightarrow & 13\ \langle \mathit{digit}\rangle & (r_{11}) \\ & \Rightarrow & 132 & (r_{11}) \end{array}$$

The following is a rightmost derivation of $x + y \star z$:

$$\begin{array}{llll} \langle \mathit{expr}\rangle & \Rightarrow & \langle \mathit{expr}\rangle + \langle \mathit{expr}\rangle & (r_1) \\ & \Rightarrow & \langle \mathit{expr}\rangle + \langle \mathit{expr}\rangle \star \langle \mathit{expr}\rangle & (r_3) \\ & \Rightarrow & \langle \mathit{expr}\rangle + \langle \mathit{expr}\rangle \star \langle \mathit{id}\rangle & (r_5) \\ & \Rightarrow & \langle \mathit{expr}\rangle + \langle \mathit{expr}\rangle \star z & (r_6) \\ & \Rightarrow & \langle \mathit{expr}\rangle + \langle \mathit{id}\rangle \star z & (r_5) \\ & \Rightarrow & \langle \mathit{expr}\rangle + y \star z & (r_6) \\ & \Rightarrow & \langle \mathit{id}\rangle + y \star z & (r_5) \\ & \Rightarrow & x + y \star z & (r_6) \end{array}$$

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

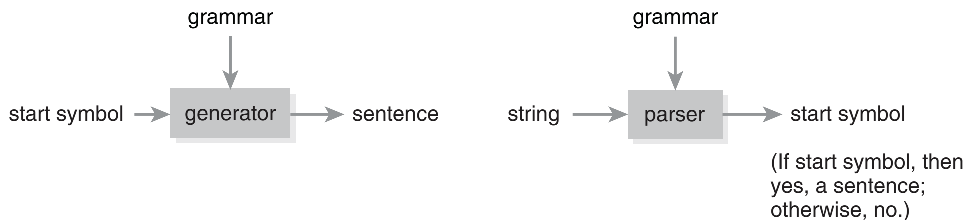

Figure 2.2 The dual nature of grammars as generative and recognition devices. (left) A language generator that accepts a grammar and a start symbol and generates a sentence from the language defined by the grammar. (right) A language parser that accepts a grammar and a string and determines if the string is in the language.

2.7 Language Recognition: Parsing

In the prior subsection we used context-free grammars as language generation devices to construct derivations. We can also implement a computer program to construct derivations; that is, to randomly choose the rules used to substitute non-terminals. That sentence-generator program takes a grammar as input and outputs a random sentence in the language defined by that grammar (see the left side of Figure 2.2). One of the seminal discoveries in computer science is that grammars can (like finite-state automata) also be used for language *recognition*—the reverse of generation. Thus, we can implement a computer program to accept a candidate string as input and construct a rightmost derivation in reverse to determine whether the input string is a sentence in the language defined by the grammar (see the right side of Figure 2.2). That computer program is called a *parser* and the process of constructing the derivation is called *parsing*—the topic of Chapter 3. If in constructing the rightmost derivation in reverse we return to the start symbol when the input string is expired, then the string is a sentence; otherwise, it is not.

Language *generation*:	start symbol	$\longrightarrow$	sentence
Language *recognition*:	sentence	$\longrightarrow$	start symbol

A *generator* applies the production rules of a grammar *forward*. A *parser* applies the rules *backward*.[5]

Consider parsing the string $x + y \star z$. In the following parse, . denotes "top of the stack":

1	. x + y ⋆ z	(shift)
2	x . + y ⋆ z	(reduce r_6)
3	<*id*> . + y⋆ z	(reduce r_5)
4	<*expr*> . + y ⋆ z	(shift)
5	<*expr*> + . y ⋆ z	(shift)
6	<*expr*> + y . ⋆ z	(reduce r_6)

5. Another class of parsers applies production rules in a top-down fashion (Section 3.4).

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

7	<*expr*> + <*id*> . ⋆ z	(reduce r_5)
8	<*expr*> + <*expr*> . ⋆ z	(shift; why not reduce r_1 here instead?)
9	<*expr*> + <*expr*> ⋆ . z	(shift)
10	<*expr*> + <*expr*> ⋆ z .	(reduce r_6)
11	<*expr*> + <*expr*> ⋆ <*id*> .	(reduce r_5)
12	<*expr*> + <*expr*> ⋆ <*expr*> .	(reduce r_3; emit multiplication)
13	<*expr*> + <*expr*> .	(reduce r_1; emit addition)
14	<*expr*> .	(start symbol; this is a sentence)

The left-hand side of the . represents a stack and the right-hand side of the . (i.e., the top of the stack) represents the remainder of the string to be parsed, called the *handle*. At each step, either shift or reduce. To determine which to do, examine the stack. If the items at the top of the stack match the right-hand side of any production rule, replace those items with the non-terminal on the left-hand side of that rule. This is known as *reducing*. If the items at the top of the stack do not match the right-hand side of any production rule, shift the next lexeme on the right-hand side of the . to the stack. If the stack contains only the start symbol when the input string is entirely consumed (i.e., shifted), then the string is a sentence; otherwise, it is not.

This process is called *shift-reduce* or *bottom-up* parsing because it starts with the string or, in other words, the terminals, and works back through the non-terminals to the start symbol. A bottom-up parse of an input string constructs a rightmost derivation of the string in reverse (i.e., bottom-up). For instance, notice that reading the lines of the rightmost derivation in Section 2.6 in reverse (i.e., from the bottom line up to the top line) corresponds to the shift-reduce parsing method discussed here. In particular, the production rules in the preceding shift-reduce parse of the string x + y ⋆ z are applied in reverse order as those in the rightmost derivation of the same string in Section 2.6. Later, in Chapter 3, we contrast this method of parsing with *top-down* or *recursive-descent* parsing. The preceding parse proves that x + y ⋆ z is a sentence.

2.8 Syntactic Ambiguity

The following parse, although different from that in Section 2.7, proves precisely the same result—that the string is a sentence.

1	. x + y ⋆ z	(shift)
2	x . + y ⋆ z	(reduce r_6)
3	<*id*> . + y ⋆ z	(reduce r_5)
4	<*expr*> . + y ⋆ z	(shift)
5	<*expr*> + . y ⋆ z	(shift)
6	<*expr*> + y . ⋆ z	(reduce r_6)
7	<*expr*> + <*id*> . ⋆ z	(reduce r_5)
8	<*expr*> + <*expr*> . ⋆ z	(reduce r_1; emit addition; why not shift here instead?)

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

A formal grammar defines only the *syntax* of a formal language. A BNF grammar defines the *syntax* of a programming language, and some of its *semantics* as well.

Table 2.6 Formal Grammars Vis-à-Vis BNF Grammars

9	<*expr*> . $\star$ z	(shift)
10	<*expr*> $\star$. z	(shift)
11	<*expr*> $\star$ z .	(reduce r_6)
12	<*expr*> $\star$ <*id*> .	(reduce r_5)
13	<*expr*> $\star$ <*expr*> .	(reduce r_3; emit multiplication)
14	<*expr*> .	(start symbol; this is a sentence)

Which of these two parses is preferred? How can we evaluate which is preferred? On what criteria should we evaluate them? The short answer to these questions is: It does not matter. The objective of language recognition and parsing is to determine if the input string is a sentence (i.e., does its structure conform to the grammar). Both of these parses meet that objective; thus, with respect to syntax, they both equally meet the objective. Here, we are only concerned with the syntactic validity of the string, not whether it makes sense (i.e., semantic validity). Parsing deals with syntax rather than semantics.

However, parsers often address issues of semantics with techniques originally intended only for addressing syntactic validity. One reason for this is that, unfortunately, unlike for syntax, we do not have formal models of semantics that are easily implemented in a computer system. Another reason is that addressing semantics while parsing can obviate the need to make multiple passes through the input string. While formal systems help us reason about concepts such as syntax and semantics, programming language systems implemented based on these formalisms must address practical issues such as efficiency. (Certain types of parsers require the production rules of the grammar of the language of the sentences they parse to be in a particular form, even though the same language can be defined using production rules in multiple forms. We discuss this concept in Chapter 3.) Therefore, although this approach is considered impure from a formal perspective, sometimes we address syntax and semantics at the same time (Table 2.6).

2.8.1 Modeling Some Semantics in Syntax

One way to gently introduce semantics into syntax is to think of syntax implying semantics as a desideratum. In other words, the form of an expression or command (i.e., its syntax) should provide some clue as to its meaning (i.e., semantics). A complaint against UNIX systems vis-à-vis systems with graphical user interfaces is that the form (i.e., syntax) of a UNIX command does not imply the meaning (i.e., semantics) of the command (e.g., `ls`, `ps`, and `grep` vis-à-vis `date` and `whoami`). The idea of integrating semantics into syntax may not seem so foreign a concept. For instance, we are taught in introductory computer programming courses to use

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

identifier names that imply the meaning of the variable to which they refer (e.g., `rate` and `index` vis-à-vis `x` and `y`).

Here we would like to infuse semantics into parsing in an identifiable way. Specifically, we would like to evaluate the expression while parsing it. This helps us avoid making unnecessary passes over the string if it is a sentence. Again, it is important to realize we are shifting from the realm of syntactic validity into interpretation. The two should not be confused, as they serve different purposes. Determining if a string is a sentence is completely independent of evaluating it for a return value. We often subconsciously impart semantics onto an expression such as x + y ⋆ z because without any mention of meaning we presume it is a mathematical expression. However, it is simply a string conforming to a syntax (i.e., form) and can have any interpretation or meaning we impart to it. Indeed, the meaning of the expression x + y ⋆ z could be a list of five elements.

Thus, in evaluating an expression while parsing it, we are imparting knowledge of how to interpret the expression (i.e., semantics). Here, we interpret these sentences as standard mathematical expressions. However, to evaluate these mathematical expressions, we must adopt even more semantics beyond the simple interpretation of them as mathematical expressions. If they are mathematical expressions, to evaluate them we must determine which operators have *precedence* over each other [i.e., is x + y⋆ z interpreted as (x + y) ⋆ z or x + (y ⋆ z)] as well as the order in which each operator *associates* [i.e., is 6 − 3 − 2 interpreted as (6 − 3) − 2 or 6 − (3 − 2)?]. *Precedence* deals with the order of distinct operators (e.g., ⋆ computes before +), while *associativity* deals with the order of operators with the same precedence (e.g., − associates left-to-right).

Formally, a binary operator ⊕ on a set S is *associative* if $(a \oplus b) \oplus c = a \oplus (b \oplus c)\ \forall a, b, c \in S$. Intuitively, *associativity* means that the value of an expression containing more than one instance of a single binary associative operator is independent of evaluation order as long as the sequence of the operands is unchanged. In other words, parentheses are unnecessary and rearranging the parentheses in such an expression does not change its value.

Notice that both parses of the expression x + y ⋆ z are the same until line 8, where a decision must be made to shift or reduce. The first parse shifts while the second reduces. Both lead to successful parses. However, if we evaluate the expression while parsing it, each parse leads to different results. One way to evaluate a mathematical expression while parsing it is to emit the mathematical operation when reducing. For instance, in step 12 of the first parse, when we reduce <expr> ⋆ <expr> to <expr>, we can compute y ⋆ z. Similarly, in step 13 of that same parse, when we reduce <expr> + <expr> to <expr>, we can compute x + <the result computed in step 12>. This interpretation [i.e., x + (y ⋆ z)] is desired because in mathematics multiplication has higher precedence than addition. Now consider the second parse. In step 8 of that parse, when we (prematurely) reduce <expr> + <expr> to <expr>, we compute x + y. Then in step 13, when we reduce <expr> ⋆ <expr> to <expr>, we compute <the result computed in step 8> ⋆ z. This interpretation [i.e., (x + y) ⋆ z] is obviously not desired. If we shift at step 8, multiplication has higher precedence

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

than addition (desired). If we reduce at step 8, addition has higher precedence than multiplication (undesired). Therefore, we prefer the first parse. These two parses exhibit a *shift-reduce conflict*. If we shift at step 8, then multiplication has higher precedence than addition (which is the desired semantics). If we reduce at step 8, then addition has higher precedence (which is the undesired semantics).

The possibility of a *reduce-reduce* conflict also exists. Consider the following grammar:

(r_1)	<expr>	::=	<term>
(r_2)	<expr>	::=	<id>
(r_3)	<term>	::=	<id>
(r_4)	<id>	::=	x | y | z

and a bottom-up parse of the expression x:

. x	(shift)
x .	(reduce r_4)
<id> .	(reduce r_2 or r_3 here?)

2.8.2 Parse Trees

The underlying source of shift-reduce and reduce-reduce conflicts is an *ambiguous grammar*. A grammar is *ambiguous* if there exists a sentence that can be parsed in more than one way. A parse of a sentence can be graphically represented using a parse tree. A *parse tree* is a tree whose root is the start symbol of the grammar, non-leaf vertices are non-terminals, and leaves are terminals, where the structure of the tree represents the conformity of the sentence to the grammar. A parse tree is fully expanded. Specifically, it has no leaves that are non-terminals and all of its leaves are terminals that, when collected from left to right, constitute the expression whose parse it represents. Thus, a grammar is *ambiguous* if we can construct more than one parse tree for the same sentence from the language defined by the grammar. Figure 2.3 gives parse trees for the expression x + y $\star$ z derived from the four-function calculator grammar in Section 2.6. The left tree represents the first parse and the right tree represents the second parse. The existence of these trees proves that the grammar is ambiguous. The last grammar in Section 2.8.1 is also

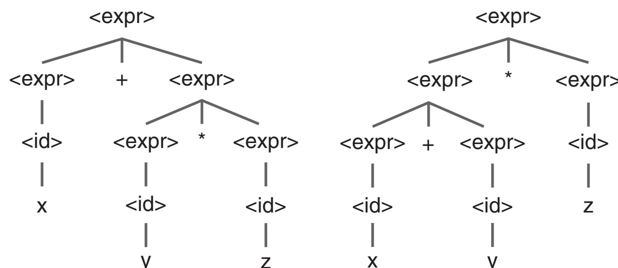

Figure 2.3 Two parse trees for the expression x + y $\star$ z.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

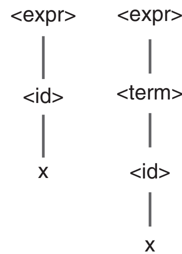

Figure 2.4 Parse trees for the expression x.

ambiguous; a proof of ambiguity exists in Figure 2.4, which contains two parse trees for the expression x.

Ambiguity is a term used to describe a grammar, whereas a shift-reduce conflict and a reduce-reduce conflict are phrases used to describe a particular parse. However, each concept is a different side of the same coin. If a grammar is ambiguous, a bottom-up parse of a sentence in the language the grammar defines will exhibit either a shift-reduce or reduce-reduce conflict, and vice versa.

Thus, proving a grammar is ambiguous is a straightforward process. All we need to do is build two parse trees for the same expression. Much more difficult, by comparison, is proving that a grammar is unambiguous.

It is important to note that a parse tree is not a derivation, or vice versa. A derivation illustrates how to *generate* a sentence. A parse tree illustrates the opposite—how to *recognize* a sentence. However, both prove a sentence is in a language (Table 2.7). Moreover, while multiple derivations of a sentence (as illustrated in Section 2.6) are not a problem, having multiple parse trees for a sentence is a problem—not from a recognition standpoint, but rather from an interpretation (i.e., meaning) perspective. Consider Table 2.8, which contains four sentences from the four-function calculator grammar in Section 2.6. While the

A derivation	*generates*	a sentence in a formal language.
A parse tree	*recognizes*	a sentence in a formal language.
Both	*prove*	a sentence is in a formal language.

Table 2.7 The Dual Use of Grammars: For *Generation* (Constructing a Derivation) and *Recognition* (Constructing a Parse Tree)

Sentence	Derivation(s)	Parse Tree(s)	Semantics
132	multiple	one	one: 132
1 + 3 + 2	multiple	multiple	one: 6
1 + 3 * 2	multiple	multiple	multiple: 7 or 8
6 - 3 - 2	multiple	multiple	multiple: 1 or 5

Table 2.8 Effect of Ambiguity on Semantics

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

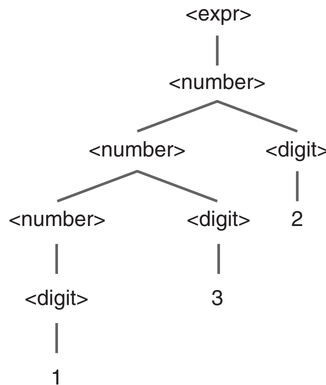

Figure 2.5 Parse tree for the expression 132.

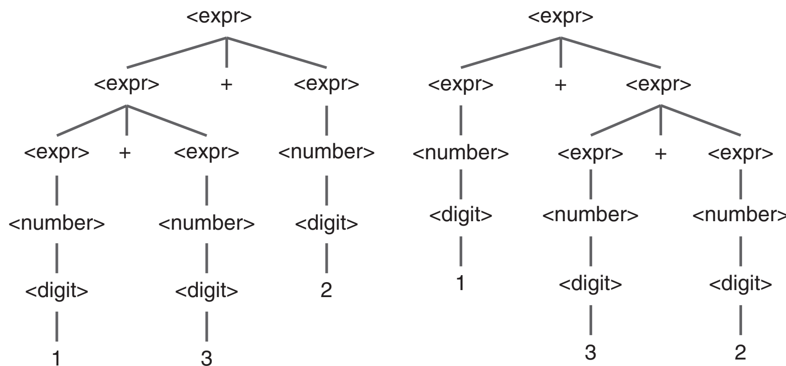

Figure 2.6 Parse trees for the expression $1 + 3 + 2$.

first sentence 132 has multiple derivations, it has only one parse tree (Figure 2.5) and, therefore, only one meaning. The second expression, $1 + 3 + 2$, in contrast, has multiple derivations and multiple parse trees. However, those parse trees (Figure 2.6) all convey the same meaning (i.e., 6). The third expression, $1 + 3 \star 2$, also has multiple derivations and parse trees (Figure 2.7). However, its parse trees each convey a different meaning (i.e., 7 or 8). Similarly, the fourth expression, $6 - 3 - 2$, has multiple derivations and parse trees (Figure 2.8), and those parse trees each have different interpretations (i.e., 1 or 5). The last three rows of Table 2.8 show the grammar to be ambiguous even though the ambiguity manifested in the expression $1 + 3 + 2$ is of no consequence to interpretation. The third expression demonstrates the need for rules establishing precedence among operators, and the fourth expression illustrates the need for rules establishing how each operator associates (left-to-right or right-to-left).

Bear in mind, that we are addressing semantics using a formalism intended for syntax. We are addressing semantics using formalisms and techniques reserved for syntax primarily because we do not have easily implementable methods

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

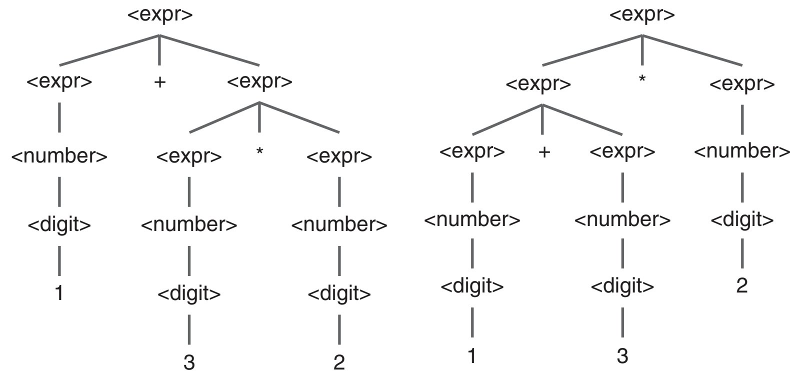

Figure 2.7 Parse trees for the expression $1 + 3 \star 2$.

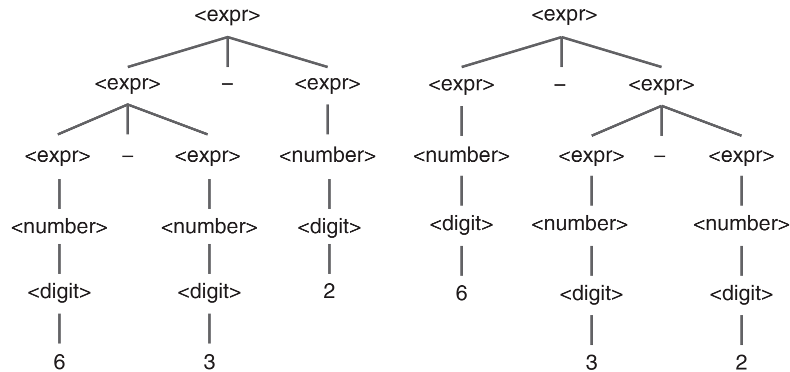

Figure 2.8 Parse trees for the expression $6 - 3 - 2$.

for dealing with context, which is necessary to effectively address semantics, in computer systems. By definition, *context-free* grammars are not intended to model *context*. However, the semantics we address through syntactic means—namely, precedence and associativity—are not dependent on context. In other words, multiplication does not have higher precedence than addition in some contexts and vice versa in others (though it could, since we are defining the language[6]). Similarly, subtraction does not associate left-to-right in some contexts and right-to-left in others. Therefore, all we need to do is make a decision for each and implement the decision.

Typically semantic rules such as precedence and associativity are specified in English (in the absence of formalisms to encode semantics easily and succinctly) in the programming manual of a particular programming language (e.g., $\star$ has higher precedence than $+$ and $-$ associates left-to-right). Thus, English is one way to specify semantic rules. However, English itself is ambiguous. Therefore, when the ambiguity—in the formal language, not English—is not dependent on context, as

6. In the programming language APL, addition has higher precedence than multiplication.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

in the case here with precedence and associativity, we can modify the grammar so that the ambiguity is removed, making the meaning (or semantics) determinable from the grammar (syntax). When ambiguity is dependent on context, grammar disambiguation to force one interpretation is not possible because you actually want more than one interpretation, though only one per context. For instance, the English sentence "Time flies like an arrow" can be parsed multiple ways. It can be parsed to indicate that there are creatures called "time flies," which really like arrows (i.e., <*adjective*> <*noun*> <*verb*> <*article*> <*noun*>), or metaphorically (i.e., <*noun*> <*verb*> <*preposition*> <*article*> <*noun*>). English is a language with an ambiguous grammar. How can we determine intended meaning? We need the surrounding context provided by the sentences before and after this sentence. Consider parsing the sentence "Mary saw the man on the mountain with a telescope.", which also has multiple interpretations corresponding to the different parses of it. This sentence has *syntactic ambiguity*, meaning that the same sentence can be diagrammed (or parsed) in multiple ways (i.e., it has multiple syntactic structures). "They are moving pictures." and "The duke yet lives that Henry shall depose."[7] are other examples of sentences with multiple interpretations.

English sentences can also exhibit *semantic ambiguity*, where there is only one syntactic structure (i.e., parse), but the individual words can be interpreted differently. An underlying source of these ambiguities is the presence of *polysemes*—a word with one spelling and pronunciation, but different meanings (e.g., book, flies, or rush). Polysemes are the opposite of *synonyms*—different words with one meaning (e.g., peaceful and serene). Polysemes that are different parts of speech (e.g., book, flies, or rush) can cause syntactic ambiguity, whereas polysemes that are the same part of speech (e.g., mouse) can cause semantic ambiguity. Note that not all sentences with syntactic ambiguity contain a polyseme (e.g., "They are moving pictures."). For summaries of these concepts, see Tables 2.9 and 2.10.

Similarly, in programming languages, the source of a semantic ambiguity is not always a syntactic ambiguity. For instance, consider the expression `(Integer)-a` on line 5 of the following Java program:

```
1  class SemanticAmbiguity {
2     public static void main(String args[]) {
3        int a = 1;
4        int Integer = 5;
5        int b = (Integer)-a;
6        System.out.println(b); // prints 4, not -1
7        b = (Integer)(-a);
8        System.out.println(b); // prints -1, not 4
9     }
10 }
```

The expression `(Integer)-a` (line 5) has only one parse tree given the grammar of a four-function calculator presented this section (assuming `Integer` is an <*id*>) and, therefore, is syntactically unambiguous. However, that expression has multiple interpretations in Java: (1) as a *subtraction*—the variable `Integer`

7. *Henry VI* by William Shakespeare.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Concept	Syntactic Structure(s)	Meaning	Example
Syntactic ambiguity	multiple	multiple	They are moving pictures.
Semantic ambiguity	one	multiple	The mouse was right on my computer.

Table 2.9 Syntactic Ambiguity Vis-à-Vis Semantic Ambiguity

Term	Spelling	Pronunciation	Meaning	Example(s)
Polysemes	same	same	different	book, flies, or rush
Homonyms				
Homophones	different	same	different	knight/night
Homographs	same	different	different	close or wind
Synonyms	different	different	same	peaceful/serene

Table 2.10 Polysemes, Homonyms, and Synonyms

minus the variable `a`, which is 4, and (2) as a *type cast*—type casting the value `-a` (or `-1`) to a value of type `Integer`, which is `-1`. Table 2.11 contains sentences from both natural and programming languages with various types of ambiguity, and demonstrates the interplay between those types. For example, a sentence without syntactic ambiguity can have semantic ambiguity; and a sentence without semantic ambiguity can have syntactic ambiguity.

We have two options for dealing with an ambiguous grammar, but both have disadvantages. First, we can state disambiguation rules in English (i.e., attach notes to the grammar), which means we do not have to alter (i.e., lengthen) the grammar, but this comes at the expense of being less formal (by the use of English). Alternatively, we can disambiguate the grammar by revising it, which is a more formal approach than the use of English, but this inflates the number of production rules in the grammar. Disambiguating a grammar is not always possible. The existence of context-free languages for which no unambiguous context-free grammar exists has been proven (in 1961 with *Parikh's theorem*). These languages are called *inherently ambiguous languages*.

Sentence	Ambiguity		
	Lexical	Syntactic	Semantic
flies	√	√	√
Time flies like an arrow.	√	√	√
They are moving pictures.	×	×	√
`*`	√	√	√
`1+3+2`	×	√	×
`1+3*2`	√	√	√
`(Integer)-a`	×	×	√

Table 2.11 Interplay Between and Interdependence of Types of Ambiguity

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

2.9 Grammar Disambiguation

Here, "having higher precedence" means "occurring lower in the parse tree" because expressions are evaluated bottom-up. In general, grammar disambiguation involves introducing additional non-terminals to prevent a sentence from being parsed multiple ways. To remove the ambiguity caused by (the lack of) operator precedence, we introduce new steps (i.e., non-terminals) in the non-terminal cascade so that multiplications are always lower than additions in the parse tree. Recall that we desire part of the meaning (or semantics) to be determined from the grammar (or syntax).

2.9.1 Operator Precedence

Consider the following updated grammar, which addresses precedence:

<*expr*> ::= <*expr*> + <*expr*>
<*expr*> ::= <*expr*> − <*expr*>
<*expr*> ::= <*term*>
<*term*> ::= <*term*> ⋆ <*term*>
<*term*> ::= <*term*> / <*term*>
<*term*> ::= (<*expr*>)
<*term*> ::= (<*id*>)
<*term*> ::= <*number*>

With this grammar it is no longer possible to construct two parse trees for the expression x + y ⋆ z. The expression x + y ⋆ z, by virtue of being parsed using this revised grammar, will always be interpreted as x + (y ⋆ z). However, while the example grammar addresses the issue of precedence, it remains ambiguous because it is still possible to use it to construct two parse trees for the expression $6 - 3 - 2$ since it does not address associativity (Figure 2.8). Recall that associativity comes into play when dealing with operators with the same precedence. Subtraction is left-associative [e.g., $6 - 3 - 2 = (6 - 3) - 2 = 1$], while unary minus is right-associative [e.g., $- - -6 = -(-(-6))$]. Associativity is mute with certain operators, including addition [e.g., $1 + 3 + 2 = (1 + 3) + 2 = 1 + (3 + 2) = 6$], but significant with others, including subtraction and unary minus. Theoretically, addition associates either left or right with the same result. However, when addition over floating-point numbers is implemented in a computer system, associativity is significant because left- and right-associativity can lead to different results. Thus, the grammar is still ambiguous for the sentences $1 + 3 + 2$ and $6 - 3 - 2$, although the former does not cause problems because both parses result in the same interpretation.

2.9.2 Associativity of Operators

Consider the following updated grammar, which addresses precedence *and* associativity:

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

<*expr*> ::= <*expr*> + <*term*>
<*expr*> ::= <*expr*> − <*term*>
<*expr*> ::= <*term*>
<*term*> ::= <*term*> ⋆ <*factor*>
<*term*> ::= <*term*> / <*factor*>
<*term*> ::= <*factor*>
<*factor*> ::= (<*expr*>)
<*factor*> ::= (<*id*>)
<*factor*> ::= <*number*>

In disambiguating the grammar for associativity, we follow the same thematic process as we used earlier: Obviate multiple parse trees by adding another level of indirection through the introduction of a new non-terminal. If we want an operator to be left-associative, then we write the production rule for that operator in a left-recursive manner because left-recursion leads to left-associativity. Similarly, if we want an operator to be right-associative, then we write the production rule for that operator in a right-recursive manner because right-recursion results in right-associativity. Since subtraction is a left-associative operator, we write the production rule as <*expr*> ::= <*expr*> − <*term*> (i.e., left-recursive) rather than <*expr*> ::= <*term*> − <*expr*> (i.e., right-recursive). The same holds for division. Since addition and multiplication are non-associative operators, we write the production rules dealing with those operators in a left-recursive manner for consistency. Therefore, the final non-ambiguous grammar is that shown previously.

2.9.3 The Classical Dangling `else` Problem

The dangling `else` problem is a classical example of grammar ambiguity in programming languages: In the absence of curly braces for disambiguation, when we have an `if–else` statement such as `if` <*expr*$_1$> `if` <*expr*$_2$> <*stmt*$_1$> `else` <*stmt*$_2$>, the `if` to which the `else` is associated is ambiguous. In other words, without a semantic rule, the statement can be interpreted in the following two ways:

```
if expr1
   if expr2
      stmt1
   else
      stmt2
```

```
if expr1
   if expr2
      stmt1
else
   stmt2
```

Indentation is used to indicate to which `if` the `else` is *intended* to be associated. Of course, in free-form languages, indentation has no bearing on program semantics.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

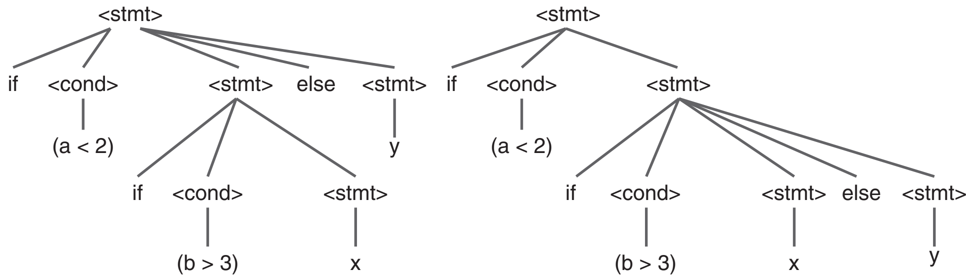

Figure 2.9 Parse trees for the sentence `if (a < 2) if (b > 3) x else y`. (left) Parse tree for an `if`–(`if`)–`else` construction. (right) Parse tree for an `if`–(`if`–`else`) construction.

In C, the semantic rule is that an `else` associates with the closest unmatched `if` and, therefore, the first interpretation is used.

Consider the following grammar for generating `if`–`else` statements:

<*stmt*> ::= `if` <*cond*><*stmt*>
<*stmt*> ::= `if` <*cond*><*stmt*> `else` <*stmt*>

Using this grammar, we can generate the following statement (save for the comment):

```
if (a < 2)
   if (b > 3)
      x = 4;
   else /* associates with which if above ? */
      y = 5;
```

for which we can construct two parse trees (Figure 2.9) proving that the grammar is ambiguous. Again, since formal methods for modeling semantics are not easily implementable, we need to revise the grammar (i.e., syntax) to imply the desired meaning (i.e., semantics). We can do that by disambiguating this grammar so that it is capable of generating `if` sentences that can only be parsed to imply that any `else` associates with the nearest unmatched `if` (i.e., parse trees of the form shown on the right side of Figure 2.9). We leave it as an exercise to develop an unambiguous grammar to solve the dangling `else` problem (Conceptual Exercise 2.10.25).

Notice that while semantics (e.g., precedence and associativity) can sometimes be reasonably modeled using context-free grammars, which are devices for modeling the *syntactic structure* of language, context-free grammars can always be used to model the *lexical structure* (or *lexics*) of language, since any regular language can be modeled by a context-free grammar. For instance, embedded into the first grammar of a four-function calculator presented in this section is the lexics of the numbers:

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

<*number*> ::= <*number*> <*digit*>
<*number*> ::= <*digit*>
<*digit*> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Thus, in the four-function calculator grammar containing these productions, the token structure (of numbers) and the syntactic structure of the expressions are inseparable. Alternatively, we could have used the regular expression $(0+1+\cdots+8+9)(0+1+\cdots+8+9)^\star$ to define the lexics and used a simpler rule in the context-free grammar:

<*number*> ::= 0 | 1 | 2 | 3 | ... | 2^{31}-2 | 2^{31}-1

2.10 Extended Backus–Naur Form

Extended Backus–Naur Form (EBNF) includes the following syntactic extensions to BNF.

- | means "alternation."
- $[x]$ means "x is optional."
- $\{x\}^*$ means "zero or more of x."
- $\{x\}^+$ means "one or more of x."
- $\{x\}^{*(c)}$ means "zero or more of x separated by cs."
- $\{x\}^{+(c)}$ means "one or more of x separated by cs."

Note that we have already encountered the extension to BNF for alternation (using |). Consider the following context-free grammar defined in BNF:

<*symbol-expr*> ::= x
<*symbol-expr*> ::= y
<*symbol-expr*> ::= z
<*symbol-expr*> ::= (<*s-list*>)
<*s-list*> ::= <*s-list*>, <*symbol-expr*>
<*s-list*> ::= <*symbol-expr*>

which can be used to derive the following sentences: x, (x, y, z), ((x)), and (((x)), ((y), (z))). We can reexpress this grammar in EBNF using alternation as follows:

<*symbol-expr*> ::= x | y | z | (<*s-list*>)
<*s-list*> ::= <*s-list*>, <*symbol-expr*>|<*symbol-expr*>

We can express r_2 more concisely using the extension for an optional item:

<*symbol-expr*> ::= x | y | z | (<*s-list*>)
<*s-list*> ::= [<*s-list*>,] <*symbol-expr*>

As another example, consider the following grammar defined in BNF:

<*arglist*> ::= <*arg*>, <*arg*>
<*arg*> ::= <*arglist*>

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

It can be rewritten in EBNF as a single rule:

$$<\mathit{arglist}> \quad ::= \quad <\mathit{arg}>, <\mathit{arg}> \{, <\mathit{arg}>\}^*$$

and can be simplified further as

$$<\mathit{arglist}> \quad ::= \quad <\mathit{arg}>, <\mathit{arg}> \{<\mathit{arg}>\}^{*(,)}$$

or expressed alternatively as

$$<\mathit{arglist}> \quad ::= \quad <\mathit{arg}>, \{<\mathit{arg}>\}^{+(,)}$$

These extensions are intended for ease of grammar definition. Any grammar defined in EBNF can be expressed in BNF. Thus, these shortcuts are simply syntactic sugar. In summary, a context-free language (which is a type of formal language) is generated by a context-free grammar (which is a type of formal grammar) and recognized by a pushdown automaton (which is a model of computation).

Conceptual Exercises for Sections 2.4–2.10

Exercise 2.10.1 Define a *regular grammar* in BNF for the language of Conceptual Exercise 2.3.1.

Exercise 2.10.2 Define a *regular grammar* in EBNF for the language of Conceptual Exercise 2.3.1.

Exercise 2.10.3 Define a *regular grammar* in BNF for the language of Conceptual Exercise 2.3.3.

Exercise 2.10.4 Define a *regular grammar* in EBNF for the language of Conceptual Exercise 2.3.3.

Exercise 2.10.5 Define a *regular grammar* in BNF for the language of Conceptual Exercise 2.3.4.

Exercise 2.10.6 Define a *regular grammar* in EBNF for the language of Conceptual Exercise 2.3.4.

Exercise 2.10.7 Define a grammar G, where G is not regular but defines a *regular language* (i.e., one that can be denoted by a *regular expression*).

Exercise 2.10.8 Express the *regular expression* hw(1+2+...+8+9)(0+1+2+...+8+9)* as a *regular grammar*.

Exercise 2.10.9 Express the *regular expression* hw(1+2+...+8+9)(0+1+2+...+8+9)* as a *context-free grammar*.

Exercise 2.10.10 Notice that the grammar of a four-function calculator presented in Section 2.6 is capable of generating numbers containing one or more leading

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

0s (e.g., 001 and 0001931), which four-function calculators are typically unable to produce. Revise this grammar so that it is unable to generate numbers with leading zeros, save for 0 itself.

Exercise 2.10.11 Reduce the number of production rules in the grammar of a four-function calculator presented in Section 2.6. In particular, consolidate rules r_1–r_4 into two rules by adding a new non-terminal <*operator*>.

Exercise 2.10.12 Describe in English, as precisely as possible, the language defined by the following grammar:

$$\begin{array}{lcl} T & \rightarrow & \text{ab} \mid \text{ba} \\ T & \rightarrow & \text{ab}T \mid \text{ba}T \\ T & \rightarrow & \text{a}T\text{b} \mid \text{b}T\text{a} \\ T & \rightarrow & \text{a}T\text{b}T \mid \text{b}T\text{a}T \end{array}$$

where T is a non-terminal and a and b are terminals.

Exercise 2.10.13 Prove that the grammar in Conceptual Exercise 2.10.12 is *ambiguous*.

Exercise 2.10.14 Consider the following grammar in EBNF:

$$\begin{array}{lcl} \langle expr \rangle & ::= & \langle expr \rangle + \langle expr \rangle \mid \langle term \rangle \\ \langle term \rangle & ::= & \langle term \rangle \star \langle term \rangle \mid \langle expr \rangle \mid \text{id} \end{array}$$

where <*expr*> and <*term*> are non-terminals and +, $\star$, and id are terminals.

(a) Prove that this grammar is *ambiguous*.
(b) Modify this grammar so that it is *unambiguous*.
(c) Define an *unambiguous* version of this grammar containing *only two non-terminals*.

Exercise 2.10.15 Prove that the following grammar defined in EBNF is *ambiguous*:

$$\begin{array}{llcl} (r_1) & \langle symbol\text{-}expr \rangle & ::= & \text{x} \mid \text{y} \mid \text{z} \mid (\langle s\text{-}list \rangle) \\ (r_2) & \langle s\text{-}list \rangle & ::= & [\langle s\text{-}list \rangle,]\ \langle symbol\text{-}expr \rangle \\ (r_3) & \langle s\text{-}list \rangle & ::= & [\langle symbol\text{-}expr \rangle,]\ \langle symbol\text{-}expr \rangle \end{array}$$

where <*symbol-expr*> and <*s-list*> are non-terminals; x, y, z, (, and) are terminals; and <*symbol-expr*> is the start symbol.

Exercise 2.10.16 Does removing rule r_3 from the grammar in Conceptual Exercise 2.10.15 eliminate the ambiguity from the grammar? If not, prove that the grammar with r_3 removed is still ambiguous.

Exercise 2.10.17 Define a grammar for a language L consisting of strings that have n copies of the letter a followed by the same number of copies of the letter b, where $n > 0$. Formally, $L = \{a^n b^n \mid n > 0 \text{ and } \Sigma = \{a, b\}\}$, where x^n means "n copies of

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

x." For instance, the strings ab, aaaabbbb, and aaaaaaaabbbbbbbb are sentences in the language, but the strings a, abb, ba, and aaabb are not. Is this language *regular*? Explain.

Exercise 2.10.18 Define an *unambiguous*, *context-free grammar* for a language L of *palindromes* of binary numbers. A *palindrome* is a string that reads the same forward as backward. For example, the strings 0, 1, 00, 11, 101, and 100101001 are palindromes, while the strings 10, 01, and 10101010 are not. The empty string ϵ is not in this language. Formally, $L = \{xx^r \mid x \in \{0, 1\}^\star\}$, where x^r means "a reversed copy of x."

Exercise 2.10.19 Matching syntactic entities (e.g., parentheses, brackets, or braces) is an important aspect of many programming languages. Define a *context-free grammar* capable of generating only balanced strings of (nested or flat) matched parentheses. The empty string ϵ is not in this language. For instance, the strings (), ()(), (()), (()())(), and ((()())()) are sentences in this language, while the strings)(,)(),)()(, (()(), ())((, and ((()()) are not. Note that *not* all strings with the same number of open and close parentheses are in this language. For example, the strings)(and)()(are not sentences in this language. State whether your grammar is ambiguous and, if it is ambiguous, prove it.

Exercise 2.10.20 Define an *unambiguous*, *context-free grammar* for the language of Exercise 2.10.19.

Exercise 2.10.21 Define a *context-free grammar* for a language L of binary numbers that contain the same number of 0s and 1s. Formally, $L = \{x \mid x \in \{0, 1\}^\star$ and the number of 0s in x equals the number of 1s in $x\}$. For instance, the strings 01, 10, 0110, 1010, 011000100111, and 000001111011 are sentences in the language, while the strings 0, 1, 00, 11, 1111000, 01100010011, and 00000111011 are not. The empty string ϵ is not in this language. Indicate whether your grammar is ambiguous and, if it is ambiguous, prove it.

Exercise 2.10.22 Solve Exercise 2.10.21 with an *unambiguous* grammar.

Exercise 2.10.23 Rewrite the grammar in Section 2.9.3 in EBNF.

Exercise 2.10.24 The following grammar for `if–else` statements has been proposed to eliminate the dangling `else` ambiguity (Aho, Sethi, and Ullman 1999, Exercise 4.5, p. 268):

```
<stmt>         ::= if <expr> <stmt>|<matched_stmt>
<matched_stmt> ::= if <expr> <matched_stmt> else <stmt>
<matched_stmt> ::= <other>
```

where the non-terminal <*other*> generates some non-`if` statement such as a `print` statement. Prove that this grammar is still *ambiguous*.

Exercise 2.10.25 Define an *unambiguous* grammar to remedy the dangling `else` problem (Section 2.9.3).

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Exercise 2.10.26 Surprisingly enough, the abilities of programmers have historically had little influence on programming language design and implementation, despite programmers being the primary users of programming languages! For instance, the ability to nest comments is helpful when a programmer desires to comment out a section of code that may already contain a comment. However, the designers of C decided to forbid nesting comments. That is, comments *cannot* nest in C. As a consequence, the following code is not syntactically valid in C:

```
/* the following function contains a bug;
   I'll just comment it out for now.
void f() {
   /* an integer x */
   int x;
   ...
}
 */
```

Why did the designers of C decide to forbid nesting comments?

Exercise 2.10.27 Give a specific example of *semantics* in programming languages not mentioned in this chapter.

Exercise 2.10.28 Can a language whose sentences are all sets from an *infinite* universe of items be defined with a *context-free grammar*? Explain.

Exercise 2.10.29 Can a language whose sentences are all sets from a *finite* universe of items be defined with a *context-free grammar*? Explain.

Exercise 2.10.30 Consider the language L of binary strings where the first half of the string is identical to the second half (i.e., all sentences have even length). For instance, the strings 11, 0000, 0101, 1010, 010010, 101101, and 11111111, are sentences in the language, but the strings 0110 and 1100 are not. Formally, $L = \{xx \mid x \in \{0, 1\}^*\}$. Is this language *context-free*? If so, give a *context-free grammar* for it. If not, state why not.

2.11 Context-Sensitivity and Semantics

Context-free grammars, by definition, cannot represent context in language. A classical example of context-sensitivity in English is "the first letter of a sentence must be capitalized." A *context-sensitive grammar*[8] for this property of English sentences is:

<*sentence*> → <*start*><*article*><*noun*><*verb*><*adverb*>.
<*start*><*article*> → A | An | The
<*article*> → a | an | the

8. Note that the use of the words *-free* and *-sensitive* in the names of formal grammars is inconsistent. The *-free* in context-free grammar indicates what such a grammar is *unable* to model—namely, context. In contrast, the *-sensitive* in context-sensitive grammar indicates what such a grammar *can* model.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

In a context-sensitive grammar, the left-hand side of a production rule is not limited to one non-terminal, as is the case in context-free grammars. In this example, the production rule "<article>→ A | An | The" only applies in the context of <start> to the left of <article>; that is, the non-terminal <start> provides the context for the application of the rule.

The pattern to which the production rules of a context-sensitive grammar must adhere are less restrictive than that of a context-free grammar. The productions of a context-sensitive grammar may have more than one non-terminal on the left-hand side. Formally, a grammar is a *context-sensitive grammar* if and only if every production rule is in the form:

$$\alpha X \beta \rightarrow \alpha \gamma \beta$$

where $X \in V$ and $\alpha, \beta, \gamma \in (\Sigma \cup V)^{\star}$, and X can be replaced with γ only in the context of α to its left and β to its right. The strings α and β may be empty in the productions of a context-sensitive grammar, but $\gamma \neq \epsilon$. However, the rule $S \rightarrow \epsilon$ is permitted as long as S does not appear on the right-hand side of any production.

Context and semantics are often confused. Recall that semantics deals with the meaning of a sentence. Context can be used to *validate* or *discern* the meaning of a sentence. Context can be used in two ways:

- **Determine semantic validity**. A classical example of context-sensitivity in programming languages is "a variable must be declared before it is used." For instance, while the following C program is syntactically valid, context reveals that it is not semantically valid because the variable `y` is referenced, but never declared:

```
int main() {
   int x;
   y = 1;
}
```

  Even if all referenced variables are declared, context may still be necessary to identify type mismatches. For instance, consider the following C++ program:

```
1  int main() {
2     int x;
3     bool y;
4
5     x = 1;
6     y = false;
7     x = y;
8  }
```

  Again, while this program is syntactically correct, it is not semantically valid because of the assignment of the value of a variable of one type to a variable of a different type (line 6). We need methods of *static semantics* (i.e., before run-time) to address this problem. We can generate semantically invalid programs from a context-free grammar because the production rules of a context-free grammar always apply, regardless of the context in which

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

a non-terminal on the left-hand side appears; hence, the rules are called context-free.

- **Disambiguate semantic validity**. Another example of context-sensitivity in programming languages is the `*` operator in C. Its meaning is dependent upon the context in which it is used. It can be used (1) as the multiplication operator (e.g., `x*3`); (2) as the pointer dereferencing operator (e.g., `*ptr`); and (3) in the declaration of pointer types (e.g., `int* ptr`). Without context, the semantics of the expression `x* y` are ambiguous. If we see the declarations `int x=1, y=2;` immediately preceding this expression, the meaning of the `*` is multiplication. However, if the statement `typedef int x;` precedes the expression `x* y`, it declares a pointer to an `int`.

Formalisms, including context-sensitive grammars, for dealing with these and other issues of semantics in programming languages are not easily implementable. Context-free grammars lend themselves naturally to the implementation of parsers (as we see in Chapter 3); context-sensitive grammars do not and, therefore, are not helpful in parser implementation. Thus, while C, Python, and Scheme are context-sensitive languages, the parser for them is implemented using a context-free grammar.

A practical approach to modeling context in programming languages is to infuse context, where practically possible, into a context-free grammar—that is, to include additional production rules to help (brute-)force the syntax to imply the semantics.[9] This approach involves designing the context-free production rules in such a way that they cannot generate a semantically invalid program. We used this approach previously to enforce proper operator precedence and associativity.

Applying this approach to capture more sophisticated semantic rules, including the requirement that variables must be declared prior to use, leads to an inordinately large number of production rules; consequently it is often unreasonable and impractical. For instance, consider the determination of whether a collection of items is a set (i.e., an unordered collection without duplicates). That determination requires context. In particular, to determine if an element disqualifies the collection from being a set, we must examine the other items in the collection (i.e., the context). If the universe from which the items in the collection are drawn is finite, we can simply enumerate all possible sets from that universe. Such an enumeration results in not only a context-free grammar, but also a regular grammar. However, that approach can involve a large number of production rules. A device called an *attribute grammar* is an extension to a context-free grammar that helps bridge the gap between content-free and context-sensitive grammars, while being practical for use in language implementation (Section 2.14).

While we encounter semantics of programming languages throughout this text, we briefly comment on formal semantics here. There are two types of

9. Both approaches—use of context-sensitive grammar and use of a context-free grammar with many rules modeling the context—model context in a purely *syntactic* way (i.e., without ascribing meaning to the language). For instance, with a context-sensitive grammar or a context-free grammar with many rules to enforce semantic rules for C, it is impossible to generate a program referencing an undeclared variable, and a program referencing an undeclared variable would be *syntactically* invalid.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

semantics: static and dynamic. In general, in computing, these terms mean before and during run-time, respectively. An example of *static semantics* is the detection of the use of an undeclared variable or a type incompatibility (e.g., `int x = "this is not an int";`). Attribute grammars can be used for static semantics.

There are three approaches to *dynamic semantics*: operational, denotational, and axiomatic. Operational semantics involves discerning the meaning of a programming construct by exploring the effects of running a program using it. Since an interpreter for a programming language, through its implementation, implicitly specifies the semantics of the language it interprets, running a program through an interpreter is an avenue to explore the operational semantics of the expressions and statements within the program. (Building interpreters for programming languages with a variety of constructs and features is the primary focus of Chapters 10–12.) Consider the English sentence "I chose wisely" which is in the past tense. If we replace the word "chose" with "chos," the sentence has a lexics error because the substring "chos" is not lexically valid. However, if we replace the word "chose" with "choose," the sentence is lexically, syntactically, and semantically valid, but in the present tense. Thus, the semantics of the sentence are valid, but unintended. Such a semantic error, like a run-time error in a program, is difficult to a detect.

Conceptual Exercises for Section 2.11

Exercise 2.11.1 Give an example of a property in programming languages (other than any of those given in the text) that is *context-sensitive* or, in other words, an example property that is *not* context-free.

Exercise 2.11.2 A *context-sensitive grammar* can express context that a *context-free grammar* cannot model. State what a *context-free grammar* can express that a *regular grammar* cannot model.

Exercise 2.11.3 We stated in this section that sometimes we can infuse context into a *context-free grammar* (often by adding more production rules) even though a context-free grammar has no provisions for representing context. Express the *context-sensitive grammar* given in Section 2.11 enforcing the capitalization of the first character of an English sentence using a *context-free grammar*.

Exercise 2.11.4 Define a *context-free grammar* for the language whose sentences correspond to sets of the elements a, b, and c. For instance, the sentences $\{a\}$, $\{a, b\}$, $\{a, b, c\}$ are in the language, but the sentences $\{a, a\}$, $\{b, a, b\}$, and $\{a, b, c, a\}$ are not.

2.12 Thematic Takeaways

- The identifiers and numbers in programming languages can be described by a *regular grammar*.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

- The nested expressions and blocks in programming languages can be described by a *context-free grammar*.
- Neither a regular nor a context-free grammar can describe the rule that a variable must be declared before it is used.
- Grammars are language recognition devices as well as language generative devices.
- An *ambiguous* grammar poses a problem for language recognition.
- Two parse trees for the same sentence from a language are sufficient to prove that the grammar for the language is *ambiguous*.
- Semantic properties, including precedence and associativity, can be modeled in a context-free grammar.

2.13 Chapter Summary

This chapter addresses *constructs* (e.g., regular expressions, grammars, automata) for defining (i.e., denoting, generating, and recognizing, respectively) languages and the *capabilities* (or limitations) of those constructs in relation to programming languages (Table 2.12). A *regular expression* denotes a set of strings—that is, the sentences of the language that the regular expression denotes. *Regular expressions* and *regular grammars* can capture the rules for a valid identifier in a programming language. More generally, regular expressions can model the *lexics* (i.e., lexical structure) of a programming language. *Context-free grammars* can capture the concept of balanced entities nested arbitrarily deep (e.g., parentheses, brackets, curly braces) whose use is pervasive in the syntactic structures (e.g., mathematical expression, `if–else` blocks) of programming languages. More generally, context-free grammars can model the *syntax* (i.e., syntactic structure) of a programming language. (Formally, context-free grammars are expressive enough to define formal languages that require an unbounded amount of memory used in a restricted way [i.e., LIFO] to recognize sentences in those languages.) If a sentence from a language has more than one parse tree, then the grammar for the language is *ambiguous*. Neither regular grammars nor context-free grammars can capture

Formal Language/ Grammar	Modeling Capability	Example Language	PL Analog	PL Code Example
Regular	lexemes	$L(a^\star b^\star)$	tokens (ids, #s)	`index1;17.76`
Context-free	balanced pairs	$\{a^n b^n \mid n \geqslant 0\}$	nested expressions/ blocks	`(a*(b+c));if/else`
Context-free	palindromes	$\{xx^r \mid x \in \{a, b\}^\star\}$	—	—
Context-sensitive	one-to-one mapping	$\{xx \mid x \in \{a, b\}^\star\}$	variable declarations and references	`int a; a=1;`
Context-sensitive	context	$\{a^n b^n c^n \mid n \geqslant 0\}$	—	—

Table 2.12 Formal Grammar Capabilities Vis-à-Vis Programming Language Constructs (Key: PL = programming language.)

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Type	**Formal Language**	(defined/generated by) **Formal Grammar**	(recognized by) **Automaton** (model of computation)	(constraints on) **Production Rules**
Type-3	regular language	regular grammar	deterministic finite automaton	$X \rightarrow zY \mid z$ or $X \rightarrow Yz \mid z$
Type-2	context-free language	context-free grammar	pushdown automaton	$X \rightarrow \gamma$
Type-1	context-sensitive language	context-sensitive grammar	linear-bounded automaton	$\alpha X \beta \rightarrow \alpha\gamma\beta$
Type-0	recursively enumerable language	unrestricted grammar	Turing machine	$\alpha \rightarrow \beta$

Table 2.13 Summary of Formal Languages and Grammars, and Models of Computation

the rule that a variable must be declared before it is used. However, we can model some semantic properties, including operator precedence and associativity, with a context-free grammar. Thus, not all formal grammars have the same expressive power; likewise, not all automata have the same power to decide if a string is a sentence in a language. (The corollary is that there are limits to computation.) While most programming languages are context-sensitive (because variables often must be declared before they are used), context-free grammars are the theoretical basis for the syntax of programming languages (in both language definition and implementation, as we see in Chapters 3 and 4).

Table 2.13 summarizes each of the progressive four types of formal grammars in the Chomsky Hierarchy; the class of formal language each grammar generates; the type of automaton that recognizes each member of each class of those formal languages; and the constraints on the production rules of the grammars. Regular and context-free grammars are fundamental topics in the study of the formal languages. In our course of study, they are useful for both describing the syntax of and parsing programming languages. In particular, regular and context-free grammars are essential ingredients in scanners and parsers, respectively, which are discussed in Chapter 3.

2.14 Notes and Further Reading

We refer readers to Webber (2008) for a practical, more detailed discussion of formal languages, grammars, and automata theory.

John Backus and Peter Naur are the recipients of the 1977 and 2005 ACM A. M. Turing Awards, respectively, in part, for their contributions to language design (through Fortran and ALGOL 60, respectively) and their contributions of formal methods for the specification of programming languages.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Attribute grammars are a formalism contributed by Donald Knuth, which can be used to capture semantics in a practical way; these grammars are context-free grammars annotated with semantics rules and checks. Knuth is the recipient of the 1974 ACM A. M. Turing Award for contributions to programming language design, including attribute grammars, and to "the art of computer programming"—communicated through his monograph titled *The Art of Computer Programming*.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.