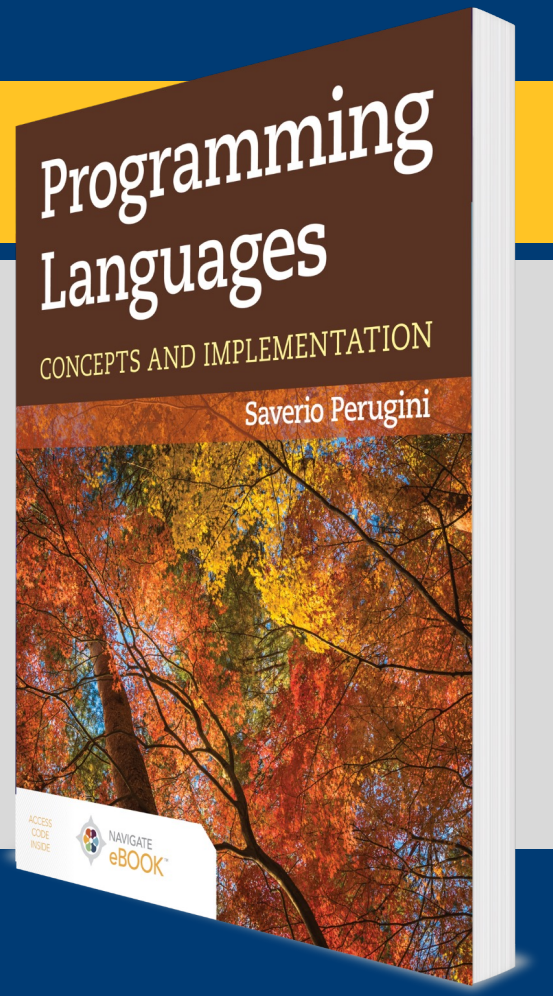
CHAPTER 2

Formal Languages and Grammars

Additional Material: Brian Hare, UMKC



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.

Outline

- **2.1 Chapter Objectives**
- 2.2 Introduction to Formal Languages
- 2.3 Regular Expressions and Regular Languages
- 2.4 Grammars and Backus–Naur Form
- 2.5 Context-Free Languages and Grammars
- 2.6 Language Generation: Sentence Derivations
- 2.7 Language Recognition: Parsing
- 2.8 Syntactic Ambiguity
- 2.9 Grammar Disambiguation
- 2.10 Extended Backus–Naur Form
- 2.11 Context-Sensitivity and Semantics
- 2.12 Thematic Takeaways
- 2.13 Chapter Summary

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.
- 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

- What is a formal language?
  - Set of sentences (strings) over some alphabet
  - Legal strings = sentences
- How do we define a formal language?
  - Grammar (define syntax of a language)
  - Any more
- Syntax and semantics
  - *Syntax* refers to *structure* or *form* of language.
  - *Semantics* refers to *meaning* of language.
- Previously, both syntax and semantics:
  - Used to be described intuitively
  - Now, well-defined, formal systems are available
- There are finite and infinite languages.
  - Is C an infinite language?
  - Most interesting languages are infinite.

Progressive Types of Sentence Validity

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

Progressive Types of Program Expression 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;`	√	√	√

Outline

- 2.1 Chapter Objectives
- 2.2 Introduction to Formal Languages
- **2.3 Regular Expressions and Regular Languages**
- 2.4 Grammars and Backus–Naur Form
- 2.5 Context-Free Languages and Grammars
- 2.6 Language Generation: Sentence Derivations
- 2.7 Language Recognition: Parsing
- 2.8 Syntactic Ambiguity
- 2.9 Grammar Disambiguation
- 2.10 Extended Backus–Naur Form
- 2.11 Context-Sensitivity and Semantics
- 2.12 Thematic Takeaways
- 2.13 Chapter Summary

2.3.1 Regular Expressions

Lexemes can be formally described by *regular grammars.*

- + (or)
- * (0 or more of previous character)
- Parentheses used for precedence

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

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	$L(r_1 + r_2) = L(r_1) \cup L(r_2)$

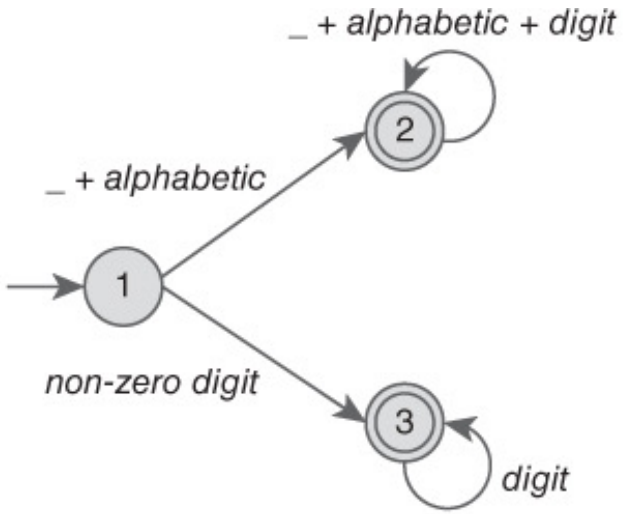
Table 2.4 Regular Expression Examples (Σ_{RE} = {ε, ∅, *, +, (,)}.)

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}

2.3.2 Finite-State Automata

- A regular expression intensionally *denotes* (the sentences of) a regular language.
- A *finite-state automaton* (FSA) is a model of computation used to recognize whether a string is a sentence in a particular language.
- We can think of an automaton as a simplified computer (Figure 2.1).
  - A computational mechanism that can *decide* whether a string is a sentence in a particular language—the set-membership problem

Finite State Automaton to Accept Positive Integers and Identifiers in C



alphabetic = a + b + ... + y + z + A + B + ... + Y + Z

non-zero digit = 1 + 2 + ... + 8 + 9

digit = 0 + 1 + ... + 8 + 9

2.3.3 Regular Languages

- *Regular grammars* (also called *linear grammars*) are generative devices for *regular languages.*
- *Regular grammars* define *regular languages.*
- Any finite language is regular.
- Sentences from regular languages are recognized using *finite state automata* (FSA).

Outline

- 2.1 Chapter Objectives
- 2.2 Introduction to Formal Languages
- 2.3 Regular Expressions and Regular Languages
- **2.4 Grammars and Backus–Naur Form**
- 2.5 Context-Free Languages and Grammars
- 2.6 Language Generation: Sentence Derivations
- 2.7 Language Recognition: Parsing
- 2.8 Syntactic Ambiguity
- 2.9 Grammar Disambiguation
- 2.10 Extended Backus–Naur Form
- 2.11 Context-Sensitivity and Semantics
- 2.12 Thematic Takeaways
- 2.13 Chapter Summary

2.4 Grammars and Backus–Naur Form (1 of 2)

- 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 in BNF for the language denoted by the $a^\star$ regular expression:

$$S \rightarrow \mathrm{a}\, S$$
$$S \rightarrow \epsilon$$

- The formal definition of a *grammar* is $G = (V, \Sigma, P, S)$, where
  - V is a set of *non-terminal symbols (e.g., {S}).*
  - Σ is an *alphabet* (e.g., $\Sigma = \{\mathrm{a}\}$).
  - 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 $x \neq \epsilon$
  - S is the *start symbol* and $S \in V$
- V is called the *non-terminal* alphabet, while Σ is the *terminal* alphabet.
- String of symbols from Σ are called *terminals*—the atomic lexical units of a program, called *lexemes*.
- The example grammar is defined formally as $G = (\{S\}, \{a\}, \{S \rightarrow aS, S \rightarrow \epsilon\}, S)$.

2.4 Grammars and Backus–Naur Form (2 of 2)

- Stream of tokens must conform to a grammar (must be arranged in a particular order).
- Grammars define how sentences are constructed.
- Defined using a metalanguage notation called Backus–Naur form (BNF)
  - John Backus at IBM for Algol 58
  - Peter Naur for Algol 60
  - Noam Chomsky
- By applying the production rules, beginning with the start symbol, a grammar can be used to *generate* a sentence from the language it defines.
- The following is a *derivation* of the sentence aaaa:

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

2.4.1 Regular Grammars (left- and right-linear grammars)

- A grammar is a *regular grammar* if and only if every production rule is in one of the following two forms:

$$X \rightarrow zY$$
$$X \rightarrow z$$

where $X \in V$, $Y \in V$, and $z \in \Sigma^*$.

- 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*:

$$X \rightarrow Yz$$
$$X \rightarrow z$$

- Left-linear grammars also generate regular languages.

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

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

Outline

- 2.1 Chapter Objectives
- 2.2 Introduction to Formal Languages
- 2.3 Regular Expressions and Regular Languages
- 2.4 Grammars and Backus–Naur Form
- **2.5 Context-Free Languages and Grammars**
- 2.6 Language Generation: Sentence Derivations
- 2.7 Language Recognition: Parsing
- 2.8 Syntactic Ambiguity
- 2.9 Grammar Disambiguation
- 2.10 Extended Backus–Naur Form
- 2.11 Context-Sensitivity and Semantics
- 2.12 Thematic Takeaways
- 2.13 Chapter Summary

2.5 Context-Free Languages and Grammars

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

Outline

- 2.1 Chapter Objectives
- 2.2 Introduction to Formal Languages
- 2.3 Regular Expressions and Regular Languages
- 2.4 Grammars and Backus–Naur Form
- 2.5 Context-Free Languages and Grammars
- **2.6 Language Generation: Sentence Derivations**
- 2.7 Language Recognition: Parsing
- 2.8 Syntactic Ambiguity
- 2.9 Grammar Disambiguation
- 2.10 Extended Backus–Naur Form
- 2.11 Context-Sensitivity and Semantics
- 2.12 Thematic Takeaways
- 2.13 Chapter Summary

2.6 Language Generation: Sentence Derivations (1 of 3)

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

Elements:

- Grammar = set of production rules
- Start symbol (<*sentence*>)
- Non-terminals (e.g., <*article*>, <*noun*>, <*verb*>, <*adverb*>)
- Terminals (e.g., a, an, the, apple, rose, umbrella, is, appears, here, there)

2.6 Language Generation: Sentence Derivations (2 of 3)

- What can we use grammars for?
- Language generation
  - Apply the rules in a top-down fashion
  - Construct a derivation
- Deriving sentences from the above grammar; derive “the apple is there" (=> means ``derive”)

$$
\begin{aligned}
\langle sentence \rangle &\Rightarrow \langle article \rangle \langle noun \rangle \langle verb \rangle \langle adverb \rangle . && (r_1)\\
&\Rightarrow \langle article \rangle \langle noun \rangle \langle verb \rangle \text{ there.} && (r_{11})\\
&\Rightarrow \langle article \rangle \langle noun \rangle \text{ is there.} && (r_8)\\
&\Rightarrow \langle article \rangle \text{ apple is there.} && (r_5)\\
&\Rightarrow \text{the apple is there.} && (r_4)
\end{aligned}
$$

Grammar for a Four-Function Calculator

Grammar for a simple arithmetic expressions for a simple four-function calculator:

(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

2.6 Language Generation: Sentence Derivations (3 of 3)

- There are *leftmost* and *rightmost* derivations.
- Some derivations are neither leftmost nor rightmost.
- Sample derivations of 132:

Leftmost derivation:

<expr>	⇒	<number>	(r_8)
	⇒	<number><digit>	(r_9)
	⇒	<number><digit><digit>	(r_9)
	⇒	<digit><digit><digit>	(r_{10})
	⇒	1 <digit><digit>	(r_{11})
	⇒	13 <digit>	(r_{11})
	⇒	132	(r_{11})

Rightmost derivation:

<expr>	⇒	<number>	(r_8)
	⇒	<number><digit>	(r_9)
	⇒	<number> 2	(r_{11})
	⇒	<number><digit> 2	(r_9)
	⇒	<number> 32	(r_{11})
	⇒	<digit> 32	(r_{10})
	⇒	132	(r_{11})

Outline

- 2.1 Chapter Objectives
- 2.2 Introduction to Formal Languages
- 2.3 Regular Expressions and Regular Languages
- 2.4 Grammars and Backus–Naur Form
- 2.5 Context-Free Languages and Grammars
- 2.6 Language Generation: Sentence Derivations
- **2.7 Language Recognition: Parsing**
- 2.8 Syntactic Ambiguity
- 2.9 Grammar Disambiguation
- 2.10 Extended Backus–Naur Form
- 2.11 Context-Sensitivity and Semantics
- 2.12 Thematic Takeaways
- 2.13 Chapter Summary

2.7 Language Recognition: Parsing

- Is a grammar a generative device or recognition device?
- Language recognition; do the reverse

  Generation: grammar → sentence

  Recognition: sentence → grammar

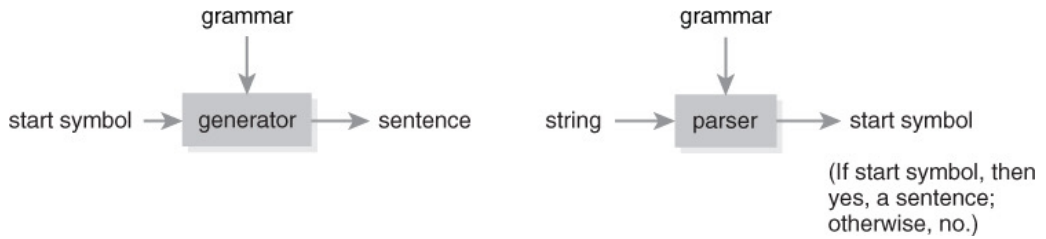


Figure 2.2 The dual nature of grammars as generative and recognition devices.

2.7 Language Generation and Recognition

Let's parse x + y * z (do the reverse):

. x + y ⋆ z	(shift)
x . + y ⋆ z	(reduce r_6)
<*id*> . + y⋆ z	(reduce r_5)
<*expr*> . + y ⋆ z	(shift)
<*expr*> + . y ⋆ z	(shift)
<*expr*> + y . ⋆ z	(reduce r_6)
<*expr*> + <*id*> . ⋆ z	(reduce r_5)
<*expr*> + <*expr*> . ⋆ z	(shift; why not reduce r_1 here instead?)
<*expr*> + <*expr*> ⋆ . z	(shift)
<*expr*> + <*expr*> ⋆ z .	(reduce r_6)
<*expr*> + <*expr*> ⋆ <*id*> .	(reduce r_5)
<*expr*> + <*expr*> ⋆ <*expr*> .	(reduce r_3; emit multiplication)
<*expr*> + <*expr*> .	(reduce r_1; emit addition)
<*expr*> .	(start symbol; this is a sentence)

- . (dot) denotes the top of the stack.
- The right-hand side is called the *handle.*
- This process is called *bottom-up* or *shift-reduce* parsing.

Outline

- 2.1 Chapter Objectives
- 2.2 Introduction to Formal Languages
- 2.3 Regular Expressions and Regular Languages
- 2.4 Grammars and Backus–Naur Form
- 2.5 Context-Free Languages and Grammars
- 2.6 Language Generation: Sentence Derivations
- 2.7 Language Recognition: Parsing
- **2.8 Syntactic Ambiguity**
- 2.9 Grammar Disambiguation
- 2.10 Extended Backus–Naur Form
- 2.11 Context-Sensitivity and Semantics
- 2.12 Thematic Takeaways
- 2.13 Chapter Summary

Table 2.6 Formal Grammars Vis-à-Vis BNF Grammars

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.

Shift-Reduce and Reduce-Reduce Conflicts

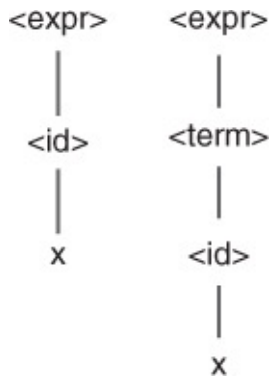
- We would like part of the meaning (or semantics) to be determined from the grammar (or syntax).
- The previous parse exhibits a *shift-reduce* conflict.
  - If we shift, multiplication will have higher precedence (desired).
  - If we reduce, addition will have higher precedence (undesired).
- There is also a *reduce-reduce* conflict (the above parse does not have one); consider the following:

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

Let's parse x

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

Parse Trees for the Expression x:



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.
- A parse tree is fully expanded.

Figure 2.3 Parse Trees for x + y * z

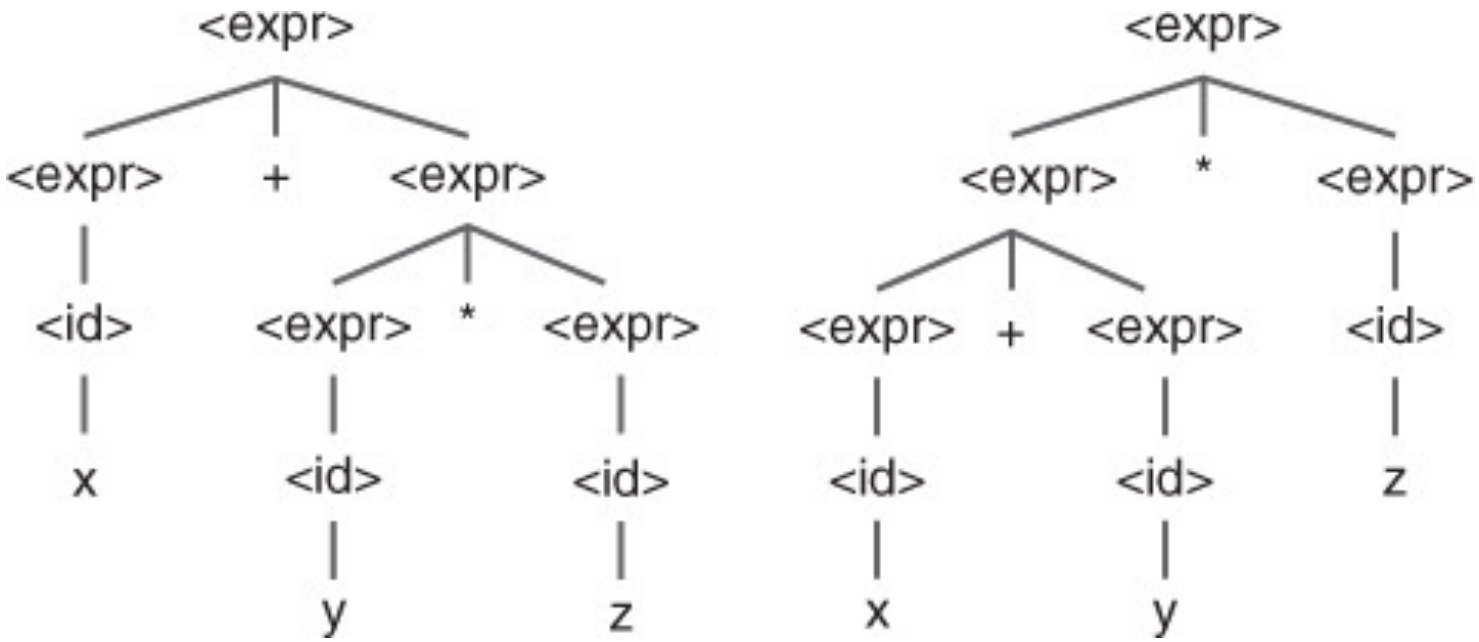


Figure 2.4 Parse Trees for x

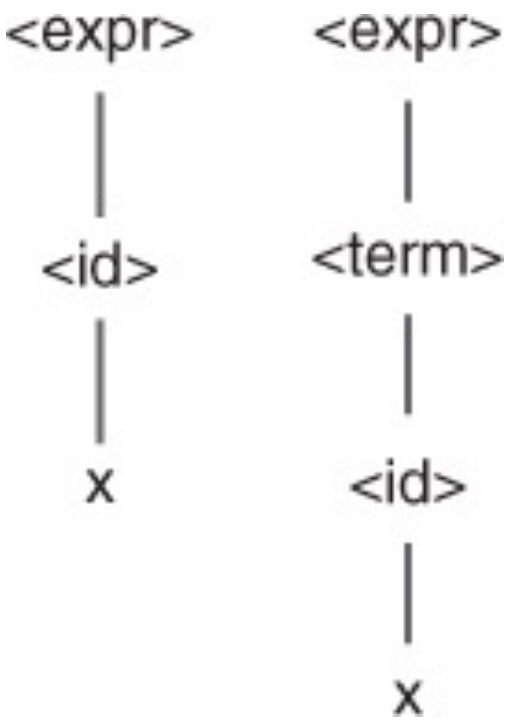


Figure 2.5 Parse Tree for the Expression 132

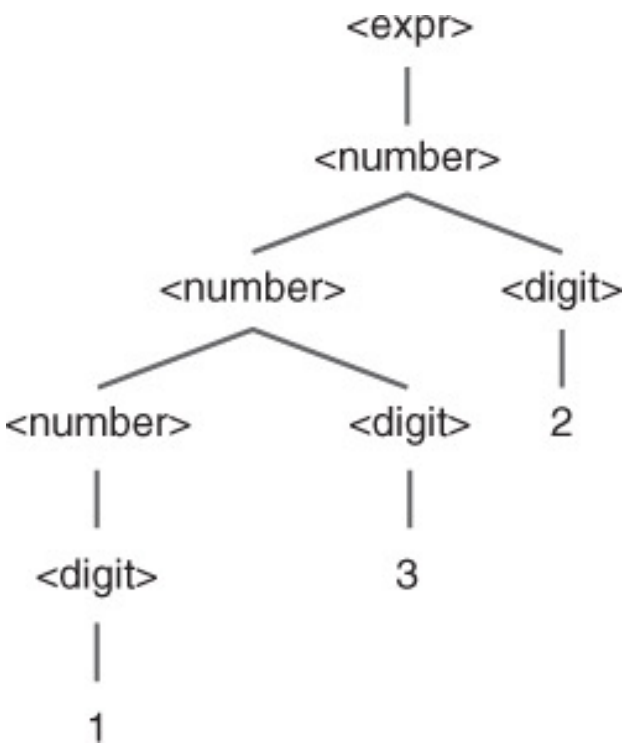


Table 2.7 The Dual Use of Grammars: For Generation (Constructing a Derivation) and Recognition (Constructing a Parse Tree)

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 2.8 Effect of Ambiguity on Semantics

If a sentence from a language has more than one parse tree, then the grammar for the language is *ambiguous.*

- Last three rows of Table 2.8 (shown here) make the grammar ambiguous.

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

Figure 2.6 Parse Trees for the Expression 1 + 3 + 2

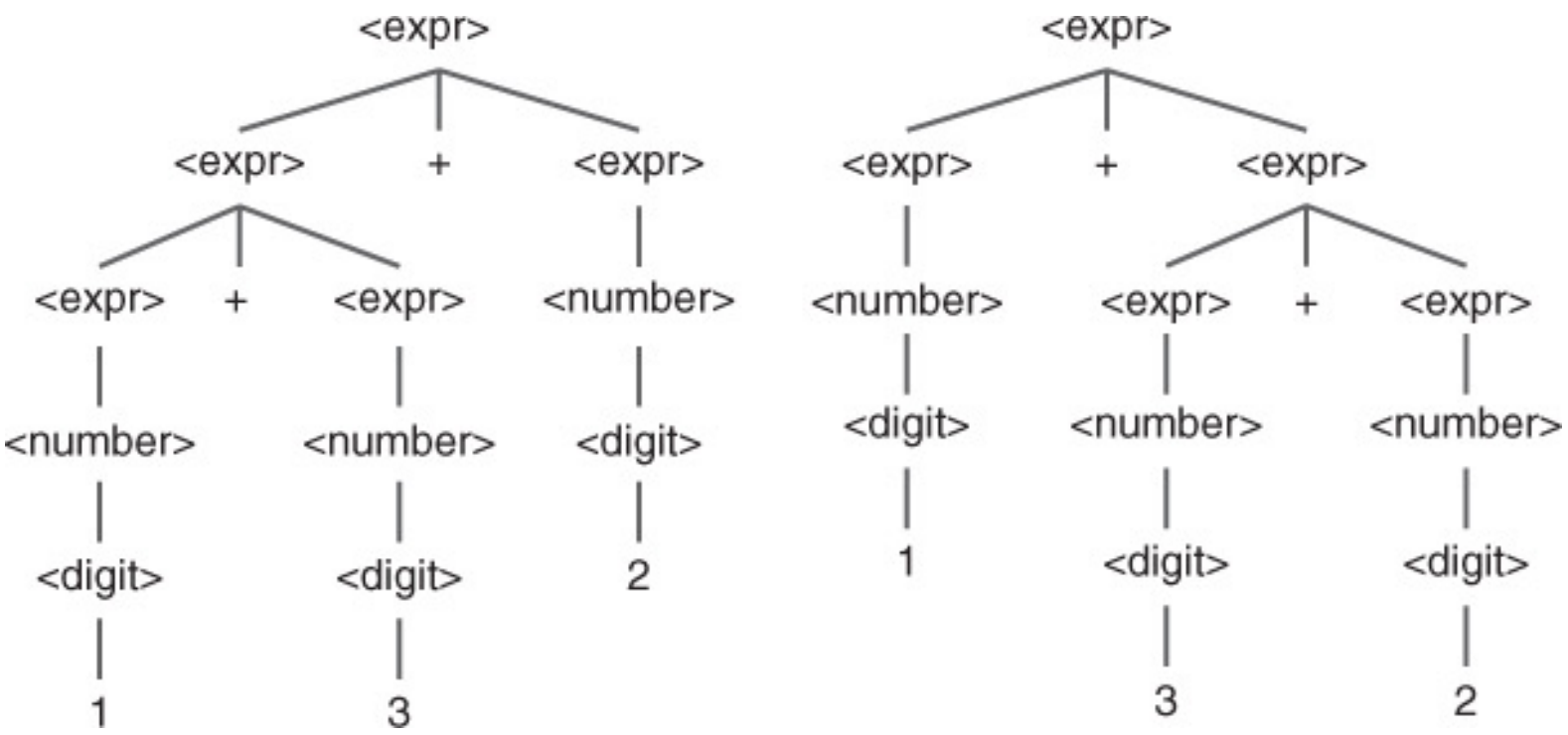


Figure 2.7 Parse Trees for the Expression 1 + 3 * 2

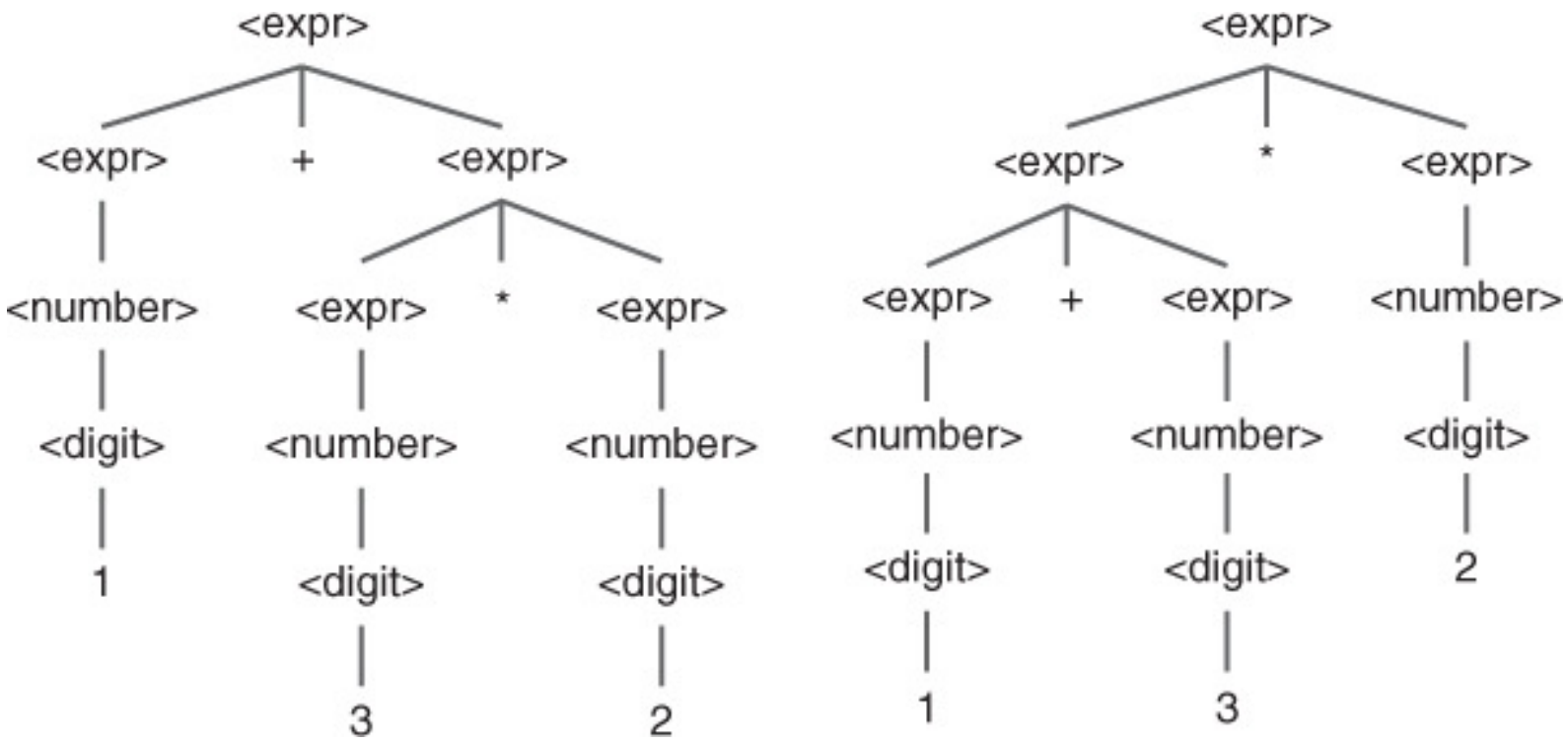


Figure 2.8 Parse Trees for the Expression 6 - 3 - 2

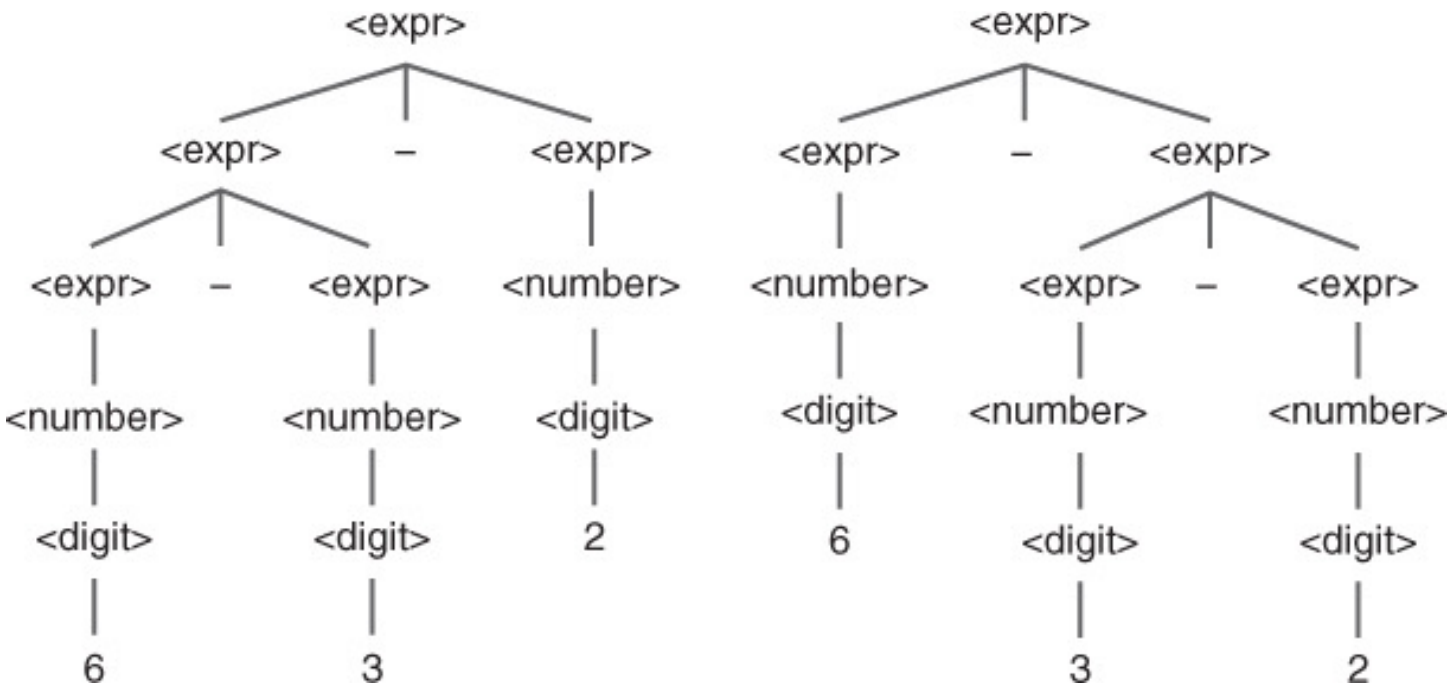


Table 2.9 Syntactic Ambiguity Vis-à-Vis Semantic Ambiguity

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 2.11 Interplay Between and Interdependence of Types of Ambiguity

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

How to Prove a Grammar Is Ambiguous

1. Generate an expression from the grammar and show the expression.
2. Give two parse trees "using the grammar" for that expression.

Notes:

- The expression must come from the grammar.
- A parse tree is fully expanded; it has no leaves that are non-terminals; they are all terminals.
- Collected leaves in each parse tree must constitute the expression.
- You cannot change the grammar while building the parse trees.
- You cannot change the expression while building the parse trees.

2.9 Grammar Disambiguation

- Desideratum: syntax imply semantics
  - Precedence
  - Associativity
- What does “have higher precedence’’ mean?
  - Occurs *lower* in the parse tree because expressions are evaluated bottom-up.
- Solution: either
  - State a disambiguating rule (order of precedence)
    - e.g., * has higher precedence than + (most languages, except APL)
    - Often implemented as “if more than 1 rule applies, prefer the one listed first in the grammar.” This breaks several rules of mathematics applied to grammars, but works.
  - Revise grammar to disambiguate it
    - Not always possible
    - Some grammars are inherently ambiguous.
    - For example, the “nearest preceding if” rule for if-else clauses

2.9.1 Operator Precedence

- Worked out solution for x + y * z

(r_1)	<*expr*> ::= <*expr*> + <*expr*>
(r_2)	<*expr*> ::= <*expr*> - <*expr*>
(r_3)	<*expr*> ::= <*term*>
(r_4)	<*term*> ::= <*term*> * <*term*>
(r_5)	<*term*> ::= <*term*> / <*term*>
(r_7)	<*term*> ::= (<*expr*>)
(r_8)	<*term*> ::= <*number*>

- Is this still ambiguous? Yes. Why?
- How can we disambiguate it?

Grammar revision:

Introduce new steps (non-terminals) in the non-terminal cascade so that multiplications are always lower than additions in the parse tree.

2.9.2 Associativity of Operators (1 of 2)

- Associativity
  - Comes into play when dealing with operators with same precedence
    - 6-3-2 = (6-3)-2 = 1 (left associative)
    - - - - 6 = -(-(-6))) = -6 (right associative)
  - Matters when adding floating-point numbers, or
  - With an operator such as subtraction (e.g., 6-3-2)
  - Which operators in C are right-associative?
- Overcoming ambiguity of associativity
  - Left-recursive rules lead to left associativity.
  - Right-recursive rules leads to right associativity.

2.9.2 Associativity of Operators (2 of 2)

- Grammar still ambiguous for 1 + 3 + 2 and 6 - 3 - 2?
- Let's fix it and make it left-associative

(r_1) <*expr*> ::= <*expr*> + <*term*>
(r_2) <*expr*> ::= <*expr*> - <*term*>
(r_3) <*expr*> ::= <*term*>
(r_4) <*term*> ::= <*term*> * <*factor*>
(r_5) <*term*> ::= <*term*> / <*factor*>
(r_7) <*factor*> ::= (<*expr*>)
(r_8) <*factor*> ::= <*number*>

- Theme: add another level of indirection by introducing a new non-terminal
  - Notice rules get lengthy
  - Why do we prefer a small rule set?

2.9.3 The Classical Dangling `else` Problem (1 of 2)

To which `if` does the `else` associate?

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

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

Source of problem: an ambiguous grammar

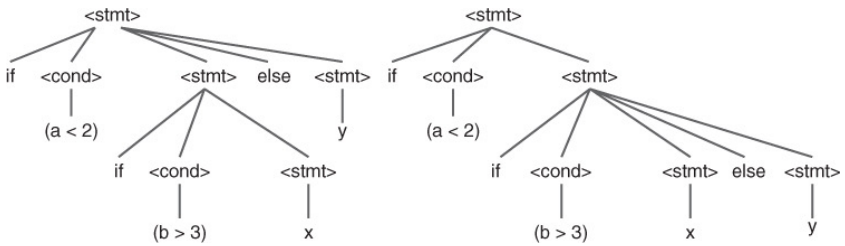
```
<stmt>  ::=  if <cond><stmt>
<stmt>  ::=  if <cond><stmt> else <stmt>
```

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

2.9.3 The Classical Dangling `else` Problem (2 of 2)

Parse trees for the sentence `if (a < 2) if (b > 3) x else y`.

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



(left) Parse tree for an if–if–else construction. (right) Parse tree for an if–if–else construction.

Outline

- 2.1 Chapter Objectives
- 2.2 Introduction to Formal Languages
- 2.3 Regular Expressions and Regular Languages
- 2.4 Grammars and Backus–Naur Form
- 2.5 Context-Free Languages and Grammars
- 2.6 Language Generation: Sentence Derivations
- 2.7 Language Recognition: Parsing
- 2.8 Syntactic Ambiguity
- 2.9 Grammar Disambiguation
- **2.10 Extended Backus–Naur Form**
- 2.11 Context-Sensitivity and Semantics
- 2.12 Thematic Takeaways
- 2.13 Chapter Summary

2.10 Extended Backus–Naur Form (EBNF)

EBNF adds includes the following syntactic extensions to BNF.

- | means ``alternation.''
- [*x*] means ``*x* is is optional'.'
- {*x*}* means ``zero or more of *x.*''
- {*x*}+ means ``one or more of *x*''
- {*x*}*(*c*) means ``zero or more of *x*s separated by *c*s.''
- {*x*}+(*c*) means ``one or more of *x*s separated by *c*s.''

EBNF is often more convenient, but gains us no new expressive power; anything in EBNF can be written in 'ordinary' BNF

Outline

- 2.1 Chapter Objectives
- 2.2 Introduction to Formal Languages
- 2.3 Regular Expressions and Regular Languages
- 2.4 Grammars and Backus–Naur Form
- 2.5 Context-Free Languages and Grammars
- 2.6 Language Generation: Sentence Derivations
- 2.7 Language Recognition: Parsing
- 2.8 Syntactic Ambiguity
- 2.9 Grammar Disambiguation
- 2.10 Extended Backus–Naur Form
- **2.11 Context-Sensitivity and Semantics**
- 2.12 Thematic Takeaways
- 2.13 Chapter Summary

2.11 Context-Sensitivity and Semantics

- What is an example of something that is context-sensitive? Or something that is not context-free?
  - First letter of a sentence must be capitalized.
    - Augustine is a saint.
    - The saint is Augustine.
- An example context-sensitive grammar (CSG) for this:

$$\begin{aligned} \langle sentence \rangle &\rightarrow \langle start \rangle \langle article \rangle \langle noun \rangle \langle verb \rangle \langle adverb \rangle . \\ \langle start \rangle \langle article \rangle &\rightarrow \text{A} \mid \text{An} \mid \text{The} \\ \langle article \rangle &\rightarrow \text{a} \mid \text{an} \mid \text{the} \end{aligned}$$

- Other examples of context:
  - A variable must be declared before it is used.
  - * operator in C

Semantics

- Static semantics
  - Declaring a variable before its usage
  - Type compatibility
  - Can use attribute grammars

- Dynamic semantics (in order from least to most formal)
  - Operational (interpreter implementation; Chapters 10–12)
  - Denotational
  - Axiomatic

Outline

- 2.1 Chapter Objectives
- 2.2 Introduction to Formal Languages
- 2.3 Regular Expressions and Regular Languages
- 2.4 Grammars and Backus–Naur Form
- 2.5 Context-Free Languages and Grammars
- 2.6 Language Generation: Sentence Derivations
- 2.7 Language Recognition: Parsing
- 2.8 Syntactic Ambiguity
- 2.9 Grammar Disambiguation
- 2.10 Extended Backus–Naur Form
- 2.11 Context-Sensitivity and Semantics
- **2.12 Thematic Takeaways**
- 2.13 Chapter Summary

2.12 Thematic Takeaways

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

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.)

Constructs and Capabilities

- Regular grammars can capture rules for a valid identifier and integer in C or Java.
- Context-free grammars can capture rules for a valid mathematical expression in C or Java.
- Neither can capture fact that a variable must be declared before it is used.
- Can model/push semantic properties like precedence and associativity into a context-free grammar

The Chomsky Hierarchy

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