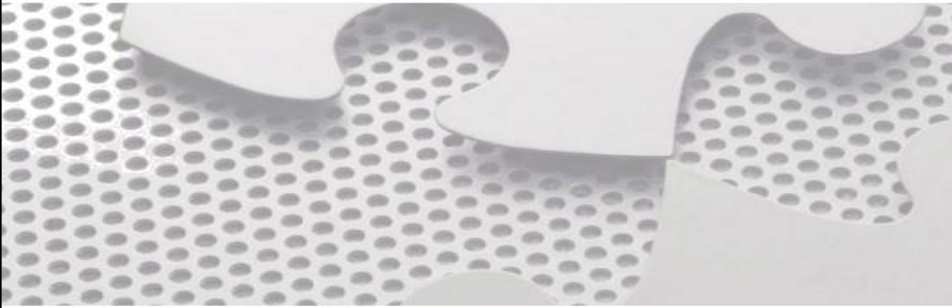


# Programming Languages Third Edition



## *Chapter 6* *Syntax*

### Objectives

- Understand the lexical structure of programming languages
- Understand context-free grammars and BNFs
- Become familiar with parse trees and abstract syntax trees
- Understand ambiguity, associativity, and precedence
- Learn to use EBNFs and syntax diagrams

## Objectives (cont'd.)

- Become familiar with parsing techniques and tools
- Understand lexics vs. syntax vs. semantics
- Build a syntax analyzer for TinyAda

## Introduction

- Syntax is the structure of a language
- 1950: Noam Chomsky developed the idea of context-free grammars
- John Backus and Peter Naur developed a notational system for describing these grammars, now called **Backus-Naur forms**, or **BNFs**
  - First used to describe the syntax of Algol60
- Every modern computer scientist needs to know how to read, interpret, and apply BNF descriptions of language syntax

## Introduction (cont'd.)

- Three variations of BNF:
  - Original BNF
  - Extended BNF (EBNF)
  - Syntax diagrams

## Lexical Structure of Programming Languages

- **Lexical structure:** the structure of the **tokens**, or words, of a language
  - Related to, but different than, the syntactic structure
- **Scanning** phase: the phase in which a translator collects sequences of characters from the input program and forms them into tokens
- **Parsing** phase: the phase in which the translator processes the tokens, determining the program's syntactic structure

## Lexical Structure of Programming Languages (cont'd.)

- Tokens generally fall into several categories:
  - **Reserved words** (or **keywords**)
  - **Literals** or **constants**
  - **Special symbols**, such as “,” “<=”, or “+”
  - **Identifiers**
- **Predefined identifiers**: identifiers that have been given an initial meaning for all programs in the language but are capable of redirection
- **Principle of longest substring**: process of collecting the longest possible string of nonblank characters

Programming Languages, Third Edition

7

## Lexical Structure of Programming Languages (cont'd.)

- **Token delimiters** (or **white space**): formatting that affects the way tokens are recognized
- Indentation can be used to determine structure
- **Free-format** language: one in which format has no effect on program structure other than satisfying the principle of longest substring
- **Fixed format** language: one in which all tokens must occur in prespecified locations on the page
- Tokens can be formally described by **regular expressions**

Programming Languages, Third Edition

8

## Lexical Structure of Programming Languages (cont'd.)

- Three basic patterns of characters in regular expressions:
  - Concatenation: done by sequencing the items
  - Repetition: indicated by an asterisk after the item to be repeated
  - Choice, or selection: indicated by a vertical bar between items to be selected
- [ ] with a hyphen indicate a range of characters
- ? indicates an optional item
- Period indicates any character

## Lexical Structure of Programming Languages (cont'd.)

- Examples:
  - Integer constants of one or more digits  
`[0-9]+`
  - Unsigned floating-point literals  
`[0-9]+(\.[0-9]+)?`
- Most modern text editors use regular expressions in text searches
- Utilities such as `lex` can automatically turn a regular expression description of a language's tokens into a scanner

## Lexical Structure of Programming Languages (cont'd.)

- Simple scanner input:  
\* + ( ) 42 # 345
- Produces this output:

```
TT_TIMES
TT_PLUS
TT_LPAREN
TT_RPAREN
TT_NUMBER: 42
TT_ERROR: #
TT_NUMBER: 345
TT_EOL
```

## Context-Free Grammars and BNFs

- Example: simple grammar

```
(1) sentence → noun-phrase verb-phrase .
(2) noun-phrase → article noun
(3) article → a | the
(4) noun → girl | dog
(5) verb-phrase → verb noun-phrase
(6) verb → sees | pets
```

**Figure 6.2** A grammar for simple english sentences

- → separates left and right sides
- | indicates a choice

## Context-Free Grammars and BNFs (cont'd.)

- **Metasymbols:** symbols used to describe the grammar rules
- Some notations use angle brackets and pure text metasymbols
  - Example:  $\langle \text{sentence} \rangle ::= \langle \text{noun-phrase} \rangle \langle \text{verb-phrase} \rangle \text{“.”}$
- **Derivation:** the process of building in a language by beginning with the **start symbol** and replacing left-hand sides by choices of right-hand sides in the rules

## Context-Free Grammars and BNFs (cont'd.)

```
sentence ⇒ noun-phrase verb-phrase . (rule 1)
           ⇒ article noun verb-phrase . (rule 2)
           ⇒ the noun verb-phrase . (rule 3)
           ⇒ the girl verb-phrase . (rule 4)
           ⇒ the girl verb noun-phrase . (rule 5)
           ⇒ the girl sees noun-phrase . (rule 6)
           ⇒ the girl sees article noun . (rule 2)
           ⇒ the girl sees a noun . (rule 3)
           ⇒ the girl sees a dog . (rule 4)
```

**Figure 6.3** A derivation using the grammar of Figure 6.2

## Context-Free Grammars and BNFs (cont'd.)

- Some problems with this simple grammar:
  - A legal sentence does not necessarily make sense
  - Positional properties (such as capitalization at the beginning of the sentence) are not represented
  - Grammar does not specify whether spaces are needed
  - Grammar does not specify input format or termination symbol

## Context-Free Grammars and BNFs (cont'd.)

- **Context-free grammar:** consists of a series of grammar rules
- Each rule has a single phrase structure name on the left, then a  $\rightarrow$  metasymbol, followed by a sequence of symbols or other phrase structure names on the right
- **Nonterminals:** names for phrase structures, since they are broken down into further phrase structures
- **Terminals:** words or token symbols that cannot be broken down further



## Context-Free Grammars and BNFs (cont'd.)

- **Productions:** another name for grammar rules
  - Typically there are as many productions in a context-free grammar as there are nonterminals
- **Backus-Naur form:** uses only the metasymbols “→” and “|”
- **Start symbol:** a nonterminal representing the entire top-level phrase being defined
- **Language of the grammar:** defined by a context-free grammar

## Context-Free Grammars and BNFs (cont'd.)

- A grammar is **context-free** when nonterminals appear singly on the left sides of productions
  - There is no **context** under which only certain replacements can occur
- Anything not expressible using context-free grammars is a semantic, not a syntactic, issue
- BNF form of language syntax makes it easier to write translators
- Parsing stage can be automated

## Context-Free Grammars and BNFs (cont'd.)

- Rules can express recursion

```
expr → expr + expr | expr * expr | ( expr ) | number  
number → number digit | digit  
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

**Figure 6.4** A simple integer arithmetic expression grammar

```
number ⇒ number digit  
⇒ number digit digit  
⇒ digit digit digit  
⇒ 2 digit digit  
⇒ 23 digit  
⇒ 234
```

**Figure 6.5** A derivation for the *number* 234 using the grammar of Figure 6.4

## Context-Free Grammars and BNFs (cont'd.)

```
translation-unit → external-declaration  
| translation-unit external-declaration  
  
external-declaration → function-definition | declaration  
  
function-definition → declaration-specifiers declarator  
| declaration-specifiers declarator compound-statement  
| declarator declaration-list compound-statement  
| declarator compound-statement  
  
declaration → declaration-specifiers ';' ;  
| declaration-specifiers init-declarator-list ';' ;  
  
init-declarator-list → init-declarator
```

**Figure 6.6** Partial BNFs for C (adapted from Kernighan and Ritchie [1988]) (*continues*)

## Context-Free Grammars and BNFs (cont'd.)

```
| init-declarator-list ' , ' init-declarator  
  
init-declarator → declarator | declarator '=' initializer  
declarator → pointer direct-declarator | direct-declarator  
  
pointer → '*' type-qualifier-list pointer | '*' type-qualifier-list  
          | '*' pointer | '*'  
  
direct-declarator → ID  
                  | '(' declarator ')'  
                  | direct_declarator '[' constant_expression ']'  
                  | direct_declarator '(' parameter_type_list ')'  
                  | direct_declarator '(' identifier_list ')'  
                  | direct_declarator '(' ')'  
                  ...  
                  ...
```

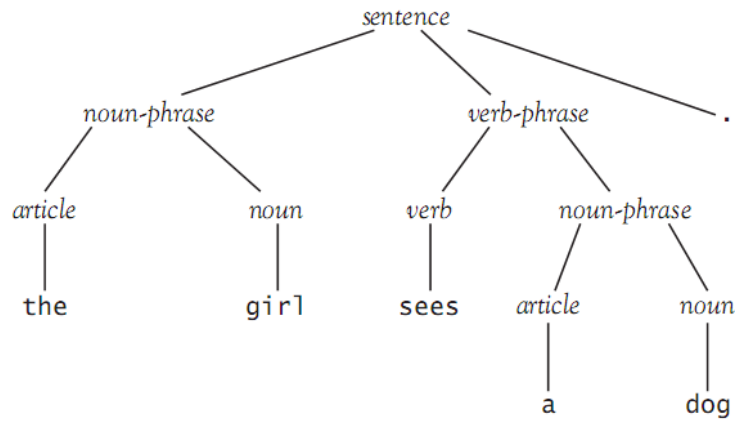
**Figure 6.6** Partial BNFs for C (adapted from Kernighan and Ritchie [1988])  
Programming Languages, Third Edition

21

## Parse Trees and Abstract Syntax Trees

- Syntax establishes structure, not meaning
  - But meaning is related to syntax
- **Syntax-directed semantics:** process of associating the semantics of a construct to its syntactic structure
  - Must construct the syntax so that it reflects the semantics to be attached later
- **Parse tree:** graphical depiction of the replacement process in a derivation

## Parse Trees and Abstract Syntax Trees (cont'd.)

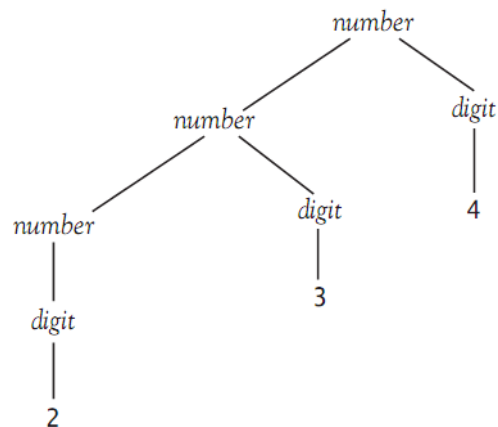


**Figure 6.7:** Parse tree for the sentence "the girl sees a dog."

Programming Languages, Third Edition

23

## Parse Trees and Abstract Syntax Trees (cont'd.)



**Figure 6.8:** Parse tree for the number 234

Programming Languages, Third Edition

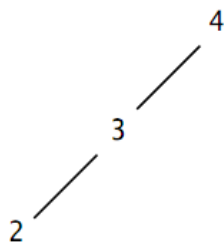
24

## Parse Trees and Abstract Syntax Trees (cont'd.)

- Nodes that have at least one child are labeled with nonterminals
- Leaves (nodes with no children) are labeled with terminals
- The structure of a parse tree is completely specified by the grammar rules of the language and a derivation of the sequence of terminals
- All terminals and nonterminals in a derivation are included in the parse tree

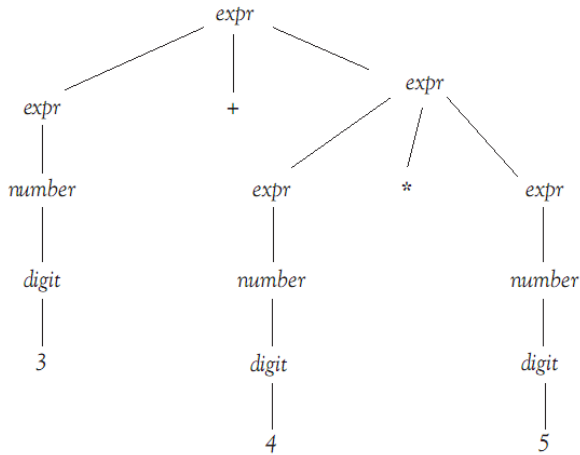
## Parse Trees and Abstract Syntax Trees (cont'd.)

- Not all terminals and nonterminals are needed to determine completely the syntactic structure of an expression or sentence



**Figure 6.9:** Parse tree for determining structure of the number 234

Complete parse tree



Condensed parse tree

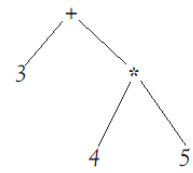


Figure 6.10: Condensing parse tree for  $3 + 4 * 5$

## Parse Trees and Abstract Syntax Trees (cont'd.)

- **Abstract syntax trees** (or **syntax trees**): trees that abstract the essential structure of the parse tree
  - Do away with terminals that are redundant
- Example:

$if\text{-statement} \rightarrow if ( expression ) statement \text{ else } statement$

Parse tree	Abstract syntax tree

Figure 6.11: Parse tree and abstract syntax tree for grammar rule  $if\text{-statement} \rightarrow if ( expression ) statement \text{ else } statement$

## Parse Trees and Abstract Syntax Trees (cont'd.)

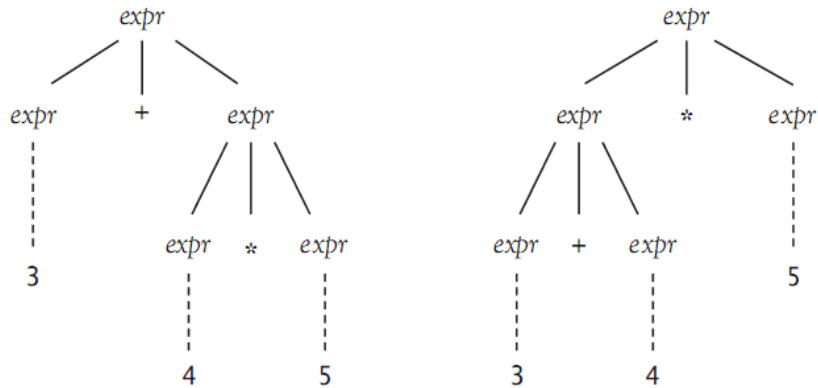
- Can write out rules for abstract syntax similar to BNF rules, but they are of less interest to a programmer
- Abstract syntax is important to a language designer and translator writer
- **Concrete syntax**: ordinary syntax

## Ambiguity, Associativity, and Precedence

- Two different derivations can lead to the same parse tree or to different parse trees
- **Ambiguous** grammar: one for which two distinct parse or syntax trees are possible
- Example: derivation for 234 given earlier

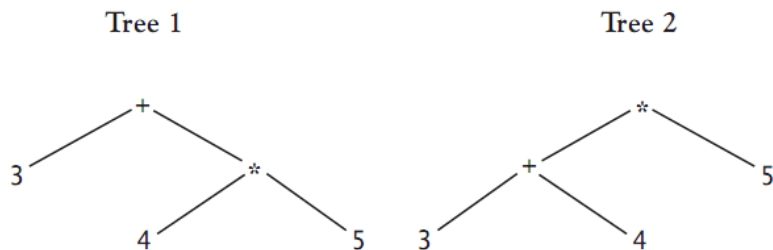
*number* ⇒ *number digit*  
⇒ *number 4*  
⇒ *number digit 4*  
⇒ *number 34*  
...

## Ambiguity, Associativity, and Precedence (cont'd.)



**Figure 6.12:** Two parse trees for  $3 + 4 * 5$

## Ambiguity, Associativity, and Precedence (cont'd.)



**Figure 6.13** Two abstract syntax trees for  $3 + 4 * 5$ , indicating the ambiguity of the grammar of Figure 6.4



## Ambiguity, Associativity, and Precedence (cont'd.)

- Certain special derivations that are constructed in a special order can only correspond to unique parse trees
- **Leftmost derivation:** the leftmost remaining nonterminal is singled out for replacement at each step
  - Each parse tree has a unique leftmost derivation
- Ambiguity of a grammar can be tested by searching for two different leftmost derivations

## Ambiguity, Associativity, and Precedence (cont'd.)

**Leftmost Derivation 1**  
(Corresponding to  
Tree 1 of Figure 6.13)

---

$expr \Rightarrow expr + expr$   
 $\Rightarrow number + expr$   
 $\Rightarrow digit + expr$   
 $\Rightarrow 3 + expr$   
 $\Rightarrow 3 + expr * expr$   
 $\Rightarrow 3 + number * expr$   
 $\Rightarrow \dots$  (etc.)

---

**Leftmost Derivation 2**  
(Corresponding to  
Tree 2 of Figure 6.13)

---

$expr \Rightarrow expr * expr$   
 $\Rightarrow expr + expr * expr$   
 $\Rightarrow number + expr * expr$   
 $\Rightarrow \dots$  (etc.)

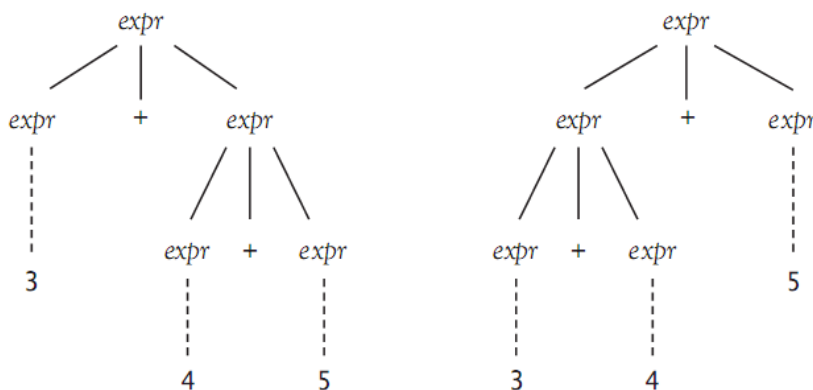
---

**Figure 6.14** Two leftmost derivations for  $3 + 4 * 5$ , indicating the ambiguity of the grammar of Figure 6.4

## Ambiguity, Associativity, and Precedence (cont'd.)

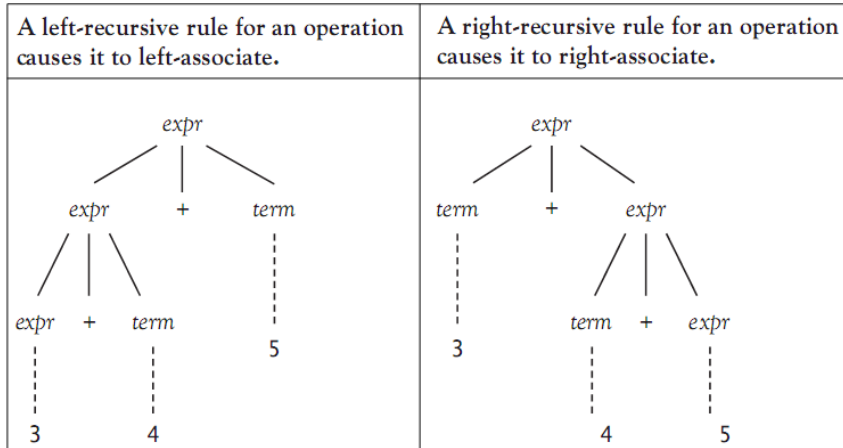
- Ambiguous grammars present difficulties
  - Must either revise them to remove ambiguity or state a **disambiguating rule**
- Usual way to revise the grammar is to write a new grammar rule called a term that establishes a precedence cascade
- Can replace  $expr \rightarrow expr + expr$ 
  - With either  $expr \rightarrow expr + term$  or  $expr \rightarrow term + expr$
- First rule is **left-recursive**; second rule is **right-recursive**

## Ambiguity, Associativity, and Precedence (cont'd.)



**Figure 6.15:** Addition as either right- or left-associative

## Ambiguity, Associativity, and Precedence (cont'd.)



**Figure 6.16:** Parse trees showing results of left- and right-recursive rules  
Programming Languages, Third Edition

37

## Ambiguity, Associativity, and Precedence (cont'd.)

```

expr → expr + term | term
term → term * factor | factor
factor → ( expr ) | number
number → number digit | digit
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    
```

**Figure 6.17** Revised grammar for simple integer arithmetic expressions

## EBNFs and Syntax Diagrams

- **Extended Backus-Naur form (or EBNF):**  
introduces new notation to handle common issues
- Use curly braces to indicate 0 or more repetitions
  - Assumes that any operator involved in a curly bracket repetition is left-associative
  - Example:  $number \rightarrow digit \{ digit \}$
- Use square brackets to indicate optional parts
  - Example:  
 $if\text{-}statement \rightarrow if ( expression ) statement [ else statement ]$

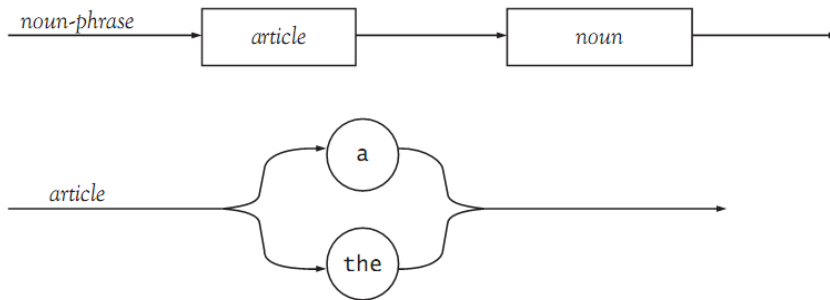
## EBNFs and Syntax Diagrams (cont'd.)

```
expr → term { + term }  
term → factor { * factor }  
factor → ( expr ) | number  
number → digit { digit }  
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

**Figure 6.18** EBNF rules for simple integer arithmetic expressions

## EBNFs and Syntax Diagrams (cont'd.)

- **Syntax diagram:** indicates the sequence of terminals and nonterminals encountered in the right-hand side of the rule



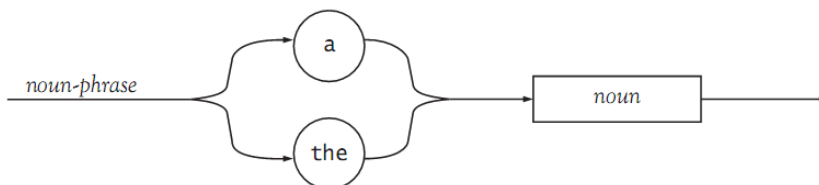
**Figure 6.19:** Syntax diagrams for *noun-phrase* and *article* of the simple English grammar presented in Section 6.2

Programming Languages, Third Edition

41

## EBNFs and Syntax Diagrams (cont'd.)

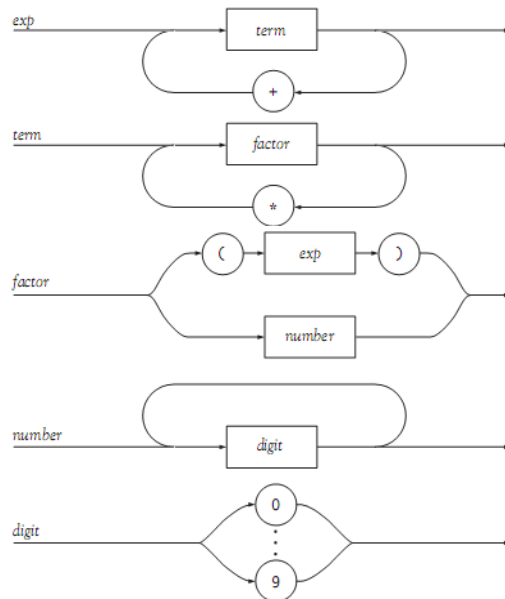
- Use circles or ovals for terminals, and squares or rectangles for nonterminals
  - Connect them with lines and arrows indicating appropriate sequencing
- Can condense several rules into one diagram
- Use loops to indicate repetition



**Figure 6.20:** Condensed version of diagrams shown in Figure 6.19

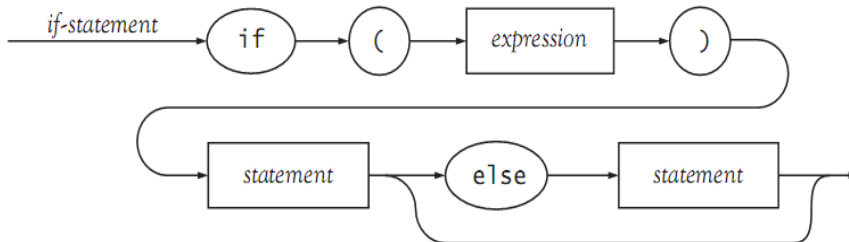
Programming Languages, Third Edition

42



**Figure 6.21:** Syntax diagrams for a simple integer expression grammar

## EBNFs and Syntax Diagrams (cont'd.)



**Figure 6.22** Syntax diagram for the *if-statement* in C

## Parsing Techniques and Tools

- A grammar written in BNF, EBNF, or syntax diagrams describes the strings of tokens that are syntactically legal
  - It also describes how a parser must act to parse correctly
- **Recognizer**: accepts or rejects strings based on whether they are legal strings in the language
- **Bottom-up parser**: constructs derivations and parse trees from the leaves to the roots
  - Matches an input with right side of a rule and **reduces** it to the nonterminal on the left

## Parsing Techniques and Tools (cont'd.)

- Bottom-up parsers are also called shift-reduce parsers
  - They shift tokens onto a stack prior to reducing strings to nonterminals
- **Top-down parser**: expands nonterminals to match incoming tokens and directly construct a derivation
- **Parser generator**: a program that automates top-down or bottom-up parsing
- Bottom-up parsing is the preferred method for parser generators (also called **compiler compilers**)

## Parsing Techniques and Tools (cont'd.)

- **Recursive-descent parsing:** turns nonterminals into a group of mutually recursive procedures based on the right-hand sides of the BNFs
  - Tokens are matched directly with input tokens as constructed by a scanner
  - Nonterminals are interpreted as calls to the procedures corresponding to the nonterminals

## Parsing Techniques and Tools (cont'd.)

```
void sentence(){
    nounPhrase();
    verbPhrase();
}

void nounPhrase(){
    article();
    noun();
}

void article(){
    if (token == "a") match("a", "a expected");
    else if (token == "the") match("the", "the expected");
    else error("article expected");
}
```



## Parsing Techniques and Tools (cont'd.)

- Left-recursive rules may present problems
  - Example:  $expr \rightarrow expr + term \mid term$
  - May cause an infinite recursive loop
  - No way to decide which of the two choices to take until a + is seen
- The EBNF description expresses the recursion as a loop:  $expr \rightarrow term \{ + term \}$
- Thus, curly brackets in EBNF represent **left recursion removal** by the use of a loop

## Parsing Techniques and Tools (cont'd.)

- Code for a right-recursive rule such as:  
 $expr \rightarrow term @ expr \mid term$
- This corresponds to the use of square brackets in EBNF:  
 $expr \rightarrow term [ @ expr ]$ 
  - This process is called **left-factoring**
- In both left-recursive and left-factoring situations, EBNF rules or syntax diagrams correspond naturally to the code of a recursive-descent parser

## Parsing Techniques and Tools (cont'd.)

- **Single-symbol lookahead:** using a single token to direct a parse
- **Predictive parser:** a parser that commits itself to a particular action based only on the lookahead
- Grammar must satisfy certain conditions to make this decision-making process work
  - Parser must be able to distinguish between choices in a rule
  - For an optional part, no token beginning the optional part can also come after the optional part

## Parsing Techniques and Tools (cont'd.)

- YACC: a widely used parser generator
  - Freeware version is called Bison
  - Generates a C program that uses a bottom-up algorithm to parse the grammar
- YACC generates a procedure `yyparse` from the grammar, which must be called from a main procedure
- YACC assumes that tokens are recognized by a scanner procedure called `yylex`, which must be provided

## Lexics vs. Syntax vs. Semantics

- Specific details of formatting, such as white-space conventions, are left to the scanner
  - Need to be stated as lexical conventions separate from the grammar
- Also desirable to allow a scanner to recognize structures such as literals, constants, and identifiers
  - Faster and simpler and reduces the size of the parser
- Must rewrite the grammar to express the use of a token rather than a nonterminal representation

## Lexics vs. Syntax vs. Semantics (cont'd.)

- Example: a number should be a token

```
expr → term { + term }  
term → factor { * factor }  
factor → ( expr ) | NUMBER
```

**Figure 6.26** Numbers as tokens in simple integer arithmetic

- Uppercase indicates it is a token whose structure is determined by the scanner
- **Lexics:** the lexical structure of a programming language

## Lexics vs. Syntax vs. Semantics (cont'd.)

- Some rules are context-sensitive and cannot be written as context-free rules
- Examples:
  - Declaration before use for variables
  - No redeclaration of identifiers within a procedure
- These are semantic properties of a language
- Another conflict occurs between **predefined identifiers** and **reserved words**
  - Reserved words cannot be used as identifiers
  - Predefined identifiers can be redefined in a program

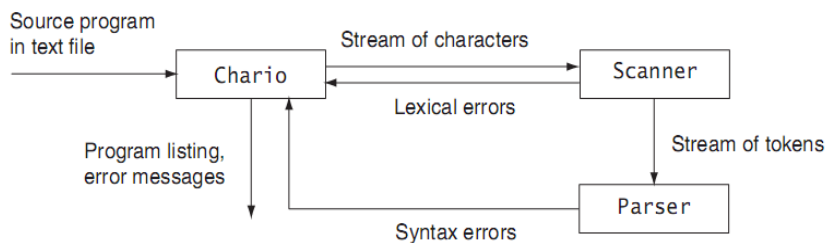
## Lexics vs. Syntax vs. Semantics (cont'd.)

- Syntax and semantics can become interdependent when semantic information is required to distinguish ambiguous parsing situations

## Case Study: Building a Syntax Analyzer for TinyAda

- **TinyAda**: a small language that illustrates the syntactic features of many high-level languages
- TinyAda includes several kinds of declarations, statements, and expressions
- Rules for declarations, statements, and expressions are indirectly recursive, allowed for nested declarations, statements, and expressions
- **Parsing shell**: applies the grammar rules to check whether tokens are of the correct types
  - Later, we will add mechanisms for semantic analysis

## Case Study: Building a Syntax Analyzer for TinyAda (cont'd.)



**Figure 6.29** Data flow in the TinyAda syntax analyzer