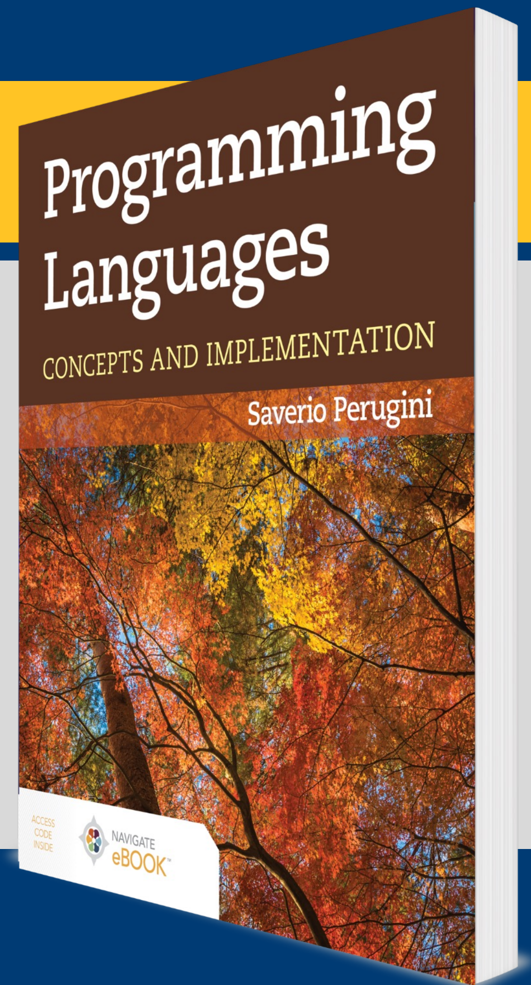
CHAPTER 3

Scanning and Parsing

Additional Material: Brian Hare, UMKC



Chapter 3: Scanning and Parsing

Although mathematical notation undoubtedly possesses parsing rules, they are rather loose, sometimes contradictory, and seldom clearly stated. . . .The proliferation of programming languages shows no more uniformity than mathematics. Nevertheless, programming languages do bring a different perspective. . . .Because of their application to a broad range of topics, their strict grammar, and their strict interpretation, programming languages can provide new insights into mathematical notation.

—Kenneth E. Iverson

Any implementation of a programming language involves scanning and parsing the source program into a representation that can be subsequently processed (i.e., interpreted or compiled or a combination of both).

Outline

- **3.1 Chapter Objectives**
- 3.2 Scanning
- 3.3 Parsing
- 3.4 Recursive-Descent Parsing
- 3.5 Bottom-up, Shift-Reduce Parsing and Parser Generators
- 3.6 PLY: Python Lex-Yacc
- 3.7 Top-down Vis-à-Vis Bottom-up Parsing
- 3.8 Thematic Takeaways

3.1 Chapter Objectives

- Establish an understanding of *scanning*.
- Establish an understanding of *parsing*.
- Introduce *top-down parsing*.
- Differentiate between *table-driven* and *recursive-descent* top-down parsers.
- Illustrate the natural relationship between a *context-free grammar* and a *recursive-descent parser*.
- Introduce *bottom-up, shift-reduce parsing*.
- Introduce parser-generation tools (e.g., `lex`/`yacc`).

Outline

- 3.1 Chapter Objectives
- **3.2 Scanning**
- 3.3 Parsing
- 3.4 Recursive-Descent Parsing
- 3.5 Bottom-up, Shift-Reduce Parsing and Parser Generators
- 3.6 PLY: Python Lex-Yacc
- 3.7 Top-down Vis-à-Vis Bottom-up Parsing
- 3.8 Thematic Takeaways

3.2 Scanning (1 of 2)

- The first step of *scanning* (also referred to as *lexical analysis*) is to parcel the characters (from the alphabet Σ) of the string representing the line of code into lexemes.
- Lexemes can be formally described by *regular expressions* and *regular grammars*.
- *Lexical analysis* is the process of determining if a string (typically of a programming language) is lexically valid—that is, if all of the lexical units of the string are lexemes.

3.2 Scanning (2 of 2)

- *Free-format languages* are languages where formatting has no effect on program structure—of course, other than use of some delimiter to determine where lexical units begin and end.
- Languages where formatting has an effect on program structure, and where lexemes must occur in predetermined areas, are called *fixed-format languages*.
- Other languages, including Python, Haskell, Miranda, and Ocaml, use *layout-based* syntactic grouping (i.e., indentation).

Table 3.1 Parceling Lexemes into Tokens in the Sentence `int i = 20;`

Lexeme	Token
`int`	reserved word
`i`	identifier
`=`	special symbol
`20`	constant
`;`	special symbol

Figure 3.1 Simplified View of Scanning and Parsing: The Front End

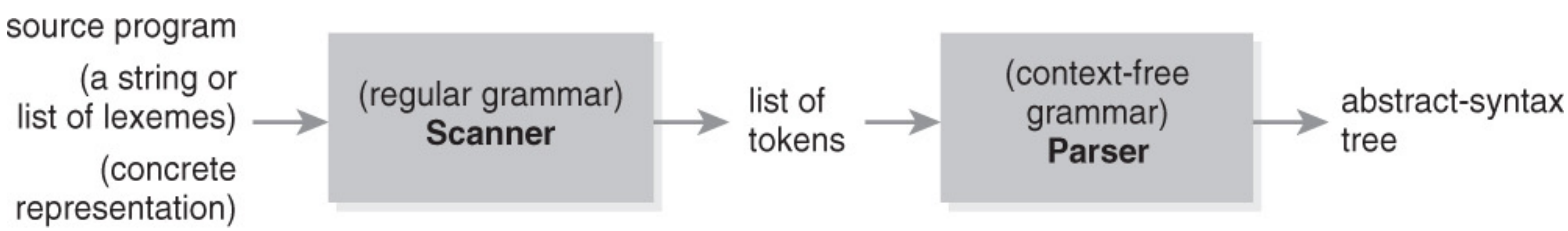


Figure 3.2 More Detailed View of Scanning and Parsing

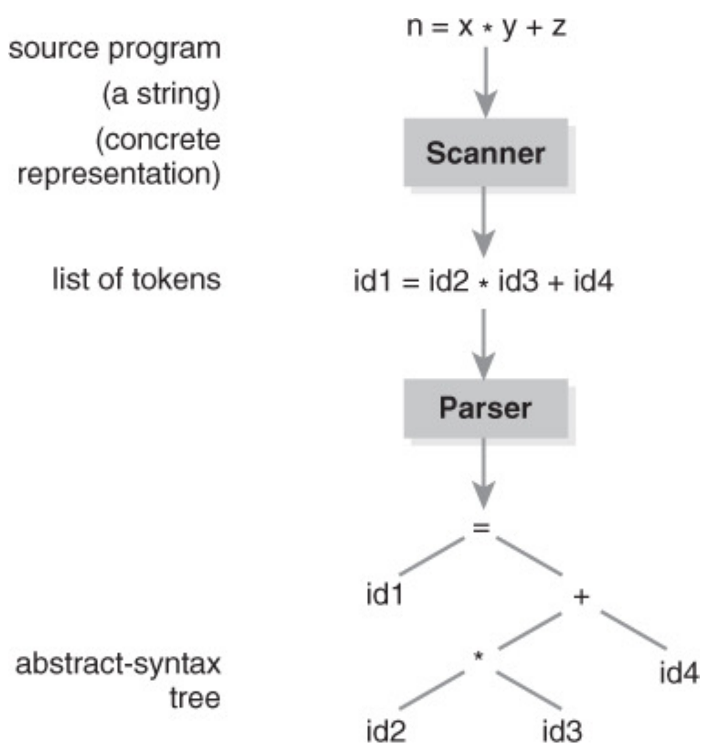
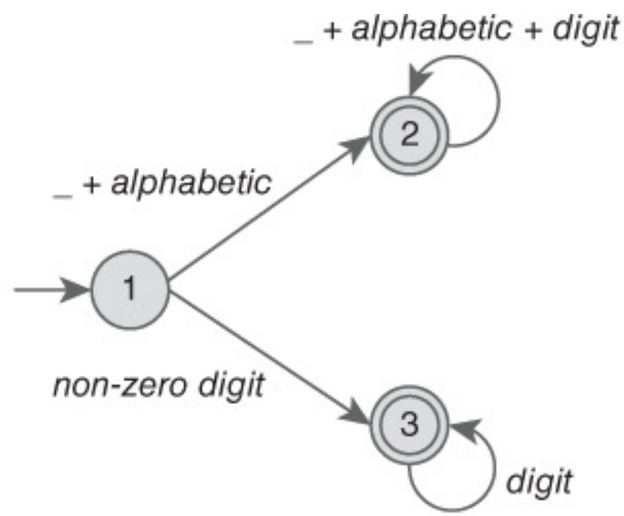


Figure 3.3 A Finite-State Automaton for a Legal Identifier and Positive Integer in C



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

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

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

Table 3.2 Two-Dimensional Array Modeling a Finite-State Automaton for a Legal Identifier and Positive Integer in C

	current state		
input character	**1**	**2**	**3**
_	2	2	*ERROR*
`a + b + ... + y + z`	2	2	*ERROR*
`A + B + ... + Y + Z`	2	2	*ERROR*
0	*ERROR*	2	3
`1 + 2 + ... + 8 + 9`	3	2	3

Outline

- 3.1 Chapter Objectives
- 3.2 Scanning
- **3.3 Parsing**
- 3.4 Recursive-Descent Parsing
- 3.5 Bottom-up, Shift-Reduce Parsing and Parser Generators
- 3.6 PLY: Python Lex-Yacc
- 3.7 Top-down Vis-à-Vis Bottom-up Parsing
- 3.8 Thematic Takeaways

3.3 Parsing

- *Parsing* (or *syntactic analysis*) is the process of determining whether a string is a sentence (in some language) and, if so, (typically) converting the *concrete representation* of it into an *abstract representation*, which generally facilitates the intended subsequent processing of it.
- A *concrete-syntax representation* of a program is typically a string (or a parse tree as shown in Chapter 2, where the terminals along the fringe of the tree from left-to-right constitute the input string).
- A *parse tree* and *abstract-syntax tree* are the syntactic analogs of a *lexeme* and *token* from *lexics*, respectively (Table 3.3). (See Section 9.5 for more details on abstract-syntax representations.)
- A *parser* (or *syntactic analyzer*) is the component of an interpreter or compiler that also typically converts the source program, once syntactically validated, into an abstract, or more easily manipulable, representation.

Table 3.3 (Concrete) Lexemes and Parse Trees Vis-à-Vis (Abstract) Tokens and Abstract-Syntax Trees, Respectively

		lexics		*syntax*	
concrete		lexeme	$\in$	parse tree	
$\downarrow$	*scanning* $\rightsquigarrow$	$\downarrow$		$\downarrow$	$\leftsquigarrow$ *parsing*
abstract		token	$\in$	abstract-syntax tree	

Outline

- 3.1 Chapter Objectives
- 3.2 Scanning
- 3.3 Parsing
- **3.4 Recursive-Descent Parsing**
- 3.5 Bottom-up, Shift-Reduce Parsing and Parser Generators
- 3.6 PLY: Python Lex-Yacc
- 3.7 Top-down Vis-à-Vis Bottom-up Parsing
- 3.8 Thematic Takeaways

3.4 Recursive-Descent Parsing

- 3.4.1 A Complete Recursive-Descent Parser
- 3.4.2 A Language Generator

3.4.1 A Complete Recursive-Descent Parser

Python code for a parser, with an embedded scanner, for a language of S-expressions with atoms `x`, `y`, and `z`

3.4.2 A Language Generator

A Python program that is a generator of sentences from the language of S-expressions with atoms `x`, `y`, and `z`

Table 3.4 Implementation Differences in Top-down Parsers: Table-Driven Vis à Vis Recursive-Descent

Type of Top-down Parser	Parse Table Used	Parse Stack Used
Table-driven	explicit 2-D array data structure	explicit stack object in program
Recursive-descent	implicit/embedded in the code	implicit call stack of program

Type of Top-down Parser	Construction Complexity	Program Readability	Program Efficiency
Table-driven	complex; use generator	less readable	efficient
Recursive-descent	uncomplex; write by hand	more readable	efficient

Outline

- 3.1 Chapter Objectives
- 3.2 Scanning
- 3.3 Parsing
- 3.4 Recursive-Descent Parsing
- **3.5 Bottom-up, Shift-Reduce Parsing and Parser Generators**
- 3.6 PLY: Python Lex-Yacc
- 3.7 Top-down Vis-à-Vis Bottom-up Parsing
- 3.8 Thematic Takeaways

3.5 Bottom-up, Shift-Reduce Parsing and Parser Generators

- A p*arser generator* is a program that accepts a syntactic specification of a language in the form of a grammar and automatically generates a parser from it.
- Parser generators are available for a wide variety of programming languages, including Python (PLY) and Scheme (SLLGEN). ANTLR (ANother Tool for Language Recognition) is a parser generator for a variety of target languages, including Java.

3.5 A Complete Example in `lexx` and `yacc`

`symexpr.l` and `symexpr.y`:

Specifications for `lex` and `yacc` (respectively) that generate a shift-reduce, bottom-up parser for the symbolic expression language presented earlier in this chapter

Outline

- 3.1 Chapter Objectives
- 3.2 Scanning
- 3.3 Parsing
- 3.4 Recursive-Descent Parsing
- 3.5 Bottom-up, Shift-Reduce Parsing and Parser Generators
- **3.6 PLY: Python Lex-Yacc**
- 3.7 Top-down Vis-à-Vis Bottom-up Parsing
- 3.8 Thematic Takeaways

3.6 PLY: Python Lex-Yacc

- 3.6.1 A Complete Example in PLY
- 3.6.2 Camille Scanner and Parser Generators in PLY

3.6.1 A Complete Example in PLY

The PLY analog of the `lex` and `yacc` specifications from Section 3.5 to generate a parser for the symbolic expression language.

3.6.2 Camille Scanner and Parser Generators in PLY

A PLY scanner specification for the tokens in the Camille language and a PLY parser specification for the Camille language defined by the grammar for a Camille version used in Chapter 11.

Outline

3.7 Top-Down Vis-à-Vis Bottom-up Parsing

- A hierarchy of parsers can be developed based on properties of grammars used in them (Table 3.5).
- Top-down and bottom-up parsers are classified as LL and LR parsers, respectively.
  - The first L indicates that both read the input string from Left-to-right.
  - The second character indicates the type of derivation the parser constructs:
    - Top-down parsers construct a Leftmost derivation.
    - Bottom-up parsers construct a Rightmost derivation.

Table 3.5 Top-down Vis à Vis Bottom-up Parsers

Description of Parser	Parser Type	Reads Input	Derivation Constructed	Requisite Grammar	Recursion in Rules
Bottom-up	**LR**	**L**eft-to-right	**R**ightmost	unambiguous	left-recursive†
Top-down	**LL**	**L**eft-to-right	**L**eftmost	unambiguous	right-recursive‡

(Key: ‡ = requisite; † = preferred.)

Table 3.6 LL Vis-à-Vis LR Grammars (Note: LL ⊂ LR.)

Grammar Type	Grammar Ambiguity	Recursion in Rules	Grammar Construction	Grammar Readability
LR	unambiguous	left- or right-recursive	less restrictive	reasonable readable
LL	unambiguous	right-recursive only	restrictive	readable

Outline

- 3.1 Chapter Objectives
- 3.2 Scanning
- 3.3 Parsing
- 3.4 Recursive-Descent Parsing
- 3.5 Bottom-up, Shift-Reduce Parsing and Parser Generators
- 3.6 PLY: Python Lex-Yacc
- 3.7 Top-down Vis-à-Vis Bottom-up Parsing
- **3.8 Thematic Takeaways**

3.8 Thematic Takeaways

- A seminal contribution to computer science is the discovery that grammars can be used as both *language-generation* devices and *language-recognition* devices.
- The structure of a recursive-descent parser follows naturally from the structure of a grammar, but the grammar must be in the proper form.