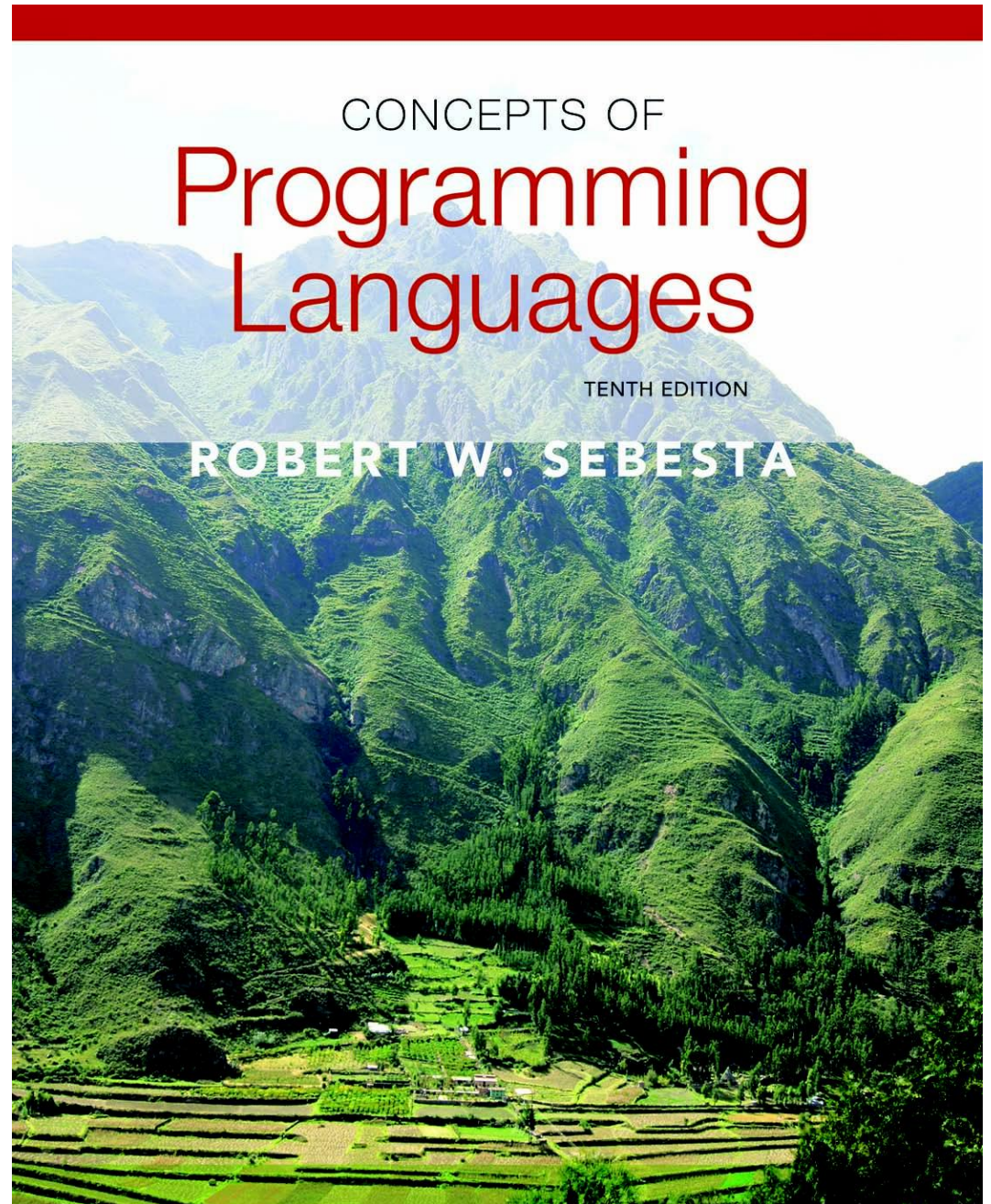


Chapter 4

Lexical and Syntax Analysis



Chapter 4 Topics

- Introduction
- Lexical Analysis
- The Parsing Problem
- **Recursive–Descent Parsing**
- **Bottom–Up Parsing**

Recursive–Descent Parsing

- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
- EBNF is ideally suited for being the basis for a recursive–descent parser, because EBNF minimizes the number of nonterminals

Recursive-Descent Parsing (continued)

- A grammar for simple expressions:

`<expr> → <term> { (+ | -) <term> }`

`<term> → <factor> { (* | /) <factor> }`

`<factor> → id | int_constant | (<expr>)`

Recursive–Descent Parsing (continued)

- Assume we have a lexical analyzer named `lex`, which puts the next token code in `nextToken`
- The coding process when there is only one RHS:
 - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error
 - For each nonterminal symbol in the RHS, call its associated parsing subprogram

Recursive-Descent Parsing (continued)

```
/* Function expr
   Parses strings in the language
   generated by the rule:
   <expr> → <term> {(+ | -) <term>}
*/

void expr() {

    /* Parse the first term */

    term();
    /* As long as the next token is + or -, call
       lex to get the next token and parse the
       next term */

    while (nextToken == ADD_OP ||
           nextToken == SUB_OP) {
        lex();
        term();
    }
}
```

Recursive–Descent Parsing (continued)

- This particular routine does not detect errors
- Convention: Every parsing routine leaves the next token in **nextToken**

Recursive–Descent Parsing (continued)

- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse
 - The correct RHS is chosen on the basis of the next token of input (the lookahead)
 - The next token is compared with the first token that can be generated by each RHS until a match is found
 - If no match is found, it is a syntax error

Recursive-Descent Parsing (continued)

```
/* term
Parses strings in the language generated by the rule:
<term> -> <factor> ((* | /) <factor>)
*/
void term() {

    /* Parse the first factor */
    factor();

    /* As long as the next token is * or /,
       next token and parse the next factor */
    while (nextToken == MULT_OP || nextToken == DIV_OP) {
        lex();
        factor();
    }
} /* End of function term */
```

Recursive-Descent Parsing (continued)

```
/* Function factor
   Parses strings in the language
   generated by the rule:
   <factor> -> id | (<expr>) */

void factor() {

    /* Determine which RHS */
    if (nextToken == ID_CODE || nextToken == INT_CODE)

        /* For the RHS id, just call lex */
        lex();

    /* If the RHS is (<expr>) - call lex to pass over the left parenthesis,
       call expr, and check for the right parenthesis */
    else if (nextToken == LP_CODE) {
        lex();
        expr();
        if (nextToken == RP_CODE)
            lex();
        else
            error();
    } /* End of else if (nextToken == ... */

    else error(); /* Neither RHS matches */
}
```

Recursive–Descent Parsing (continued)

– Trace of the lexical and syntax analyzers on `(sum + 47) / total`

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26 Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24 Next lexeme is /
Exit <factor>
```

```
Next token is: 11 Next lexeme is total
Enter <factor>
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

Recursive–Descent Parsing (continued)

- The LL Grammar Class
 - The Left Recursion Problem
 - If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top–down parser
 - A grammar can be modified to remove direct left recursion as follows:
For each nonterminal, A,
 1. Group the A–rules as $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
where none of the β 's begins with A
 2. Replace the original A–rules with
$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

Recursive–Descent Parsing (continued)

- The other characteristic of grammars that disallows top–down parsing is the lack of pairwise disjointness
 - The inability to determine the correct RHS on the basis of one token of lookahead
 - Def: $\text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}$
(If $\alpha \Rightarrow^* \epsilon$, ϵ is in $\text{FIRST}(\alpha)$)

Recursive–Descent Parsing (continued)

- Pairwise Disjointness Test:
 - For each nonterminal, A , in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that

$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \phi$$

- Example:

$A \rightarrow a \mid bB \mid cAb$

$A \rightarrow a \mid aB$

Recursive–Descent Parsing (continued)

- Left factoring can resolve the problem

Replace

$\langle \text{variable} \rangle \rightarrow \text{identifier} \mid \text{identifier} [\langle \text{expression} \rangle]$

with

$\langle \text{variable} \rangle \rightarrow \text{identifier} \langle \text{new} \rangle$

$\langle \text{new} \rangle \rightarrow \varepsilon \mid [\langle \text{expression} \rangle]$

or

$\langle \text{variable} \rangle \rightarrow \text{identifier} [[\langle \text{expression} \rangle]]$

(the outer brackets are metasympols of EBNF)

Bottom-up Parsing

- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

Bottom-up Parsing (continued)

- Intuition about handles:

- Def: β is the *handle* of the right sentential form

- $\gamma = \alpha\beta w$ if and only if $S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha\beta w$

- Def: β is a *phrase* of the right sentential form

- γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow + \alpha_1 \beta \alpha_2$

- Def: β is a *simple phrase* of the right sentential

- form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$

Bottom-up Parsing (continued)

- Intuition about handles (continued):
 - The handle of a right sentential form is its leftmost simple phrase
 - Given a parse tree, it is now easy to find the handle
 - Parsing can be thought of as handle pruning

Bottom-up Parsing (continued)

- Shift-Reduce Algorithms
 - Reduce is the action of replacing the handle on the top of the parse stack with its corresponding LHS
 - Shift is the action of moving the next token to the top of the parse stack

Bottom-up Parsing (continued)

- Advantages of LR parsers:
 - They will work for nearly all grammars that describe programming languages.
 - They work on a larger class of grammars than other bottom-up algorithms, but are as efficient as any other bottom-up parser.
 - They can detect syntax errors as soon as it is possible.
 - The LR class of grammars is a superset of the class parsable by LL parsers.

Bottom-up Parsing (continued)

- LR parsers must be constructed with a tool
- Knuth's insight: A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions
 - There are only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, on the parse stack

Bottom-up Parsing (continued)

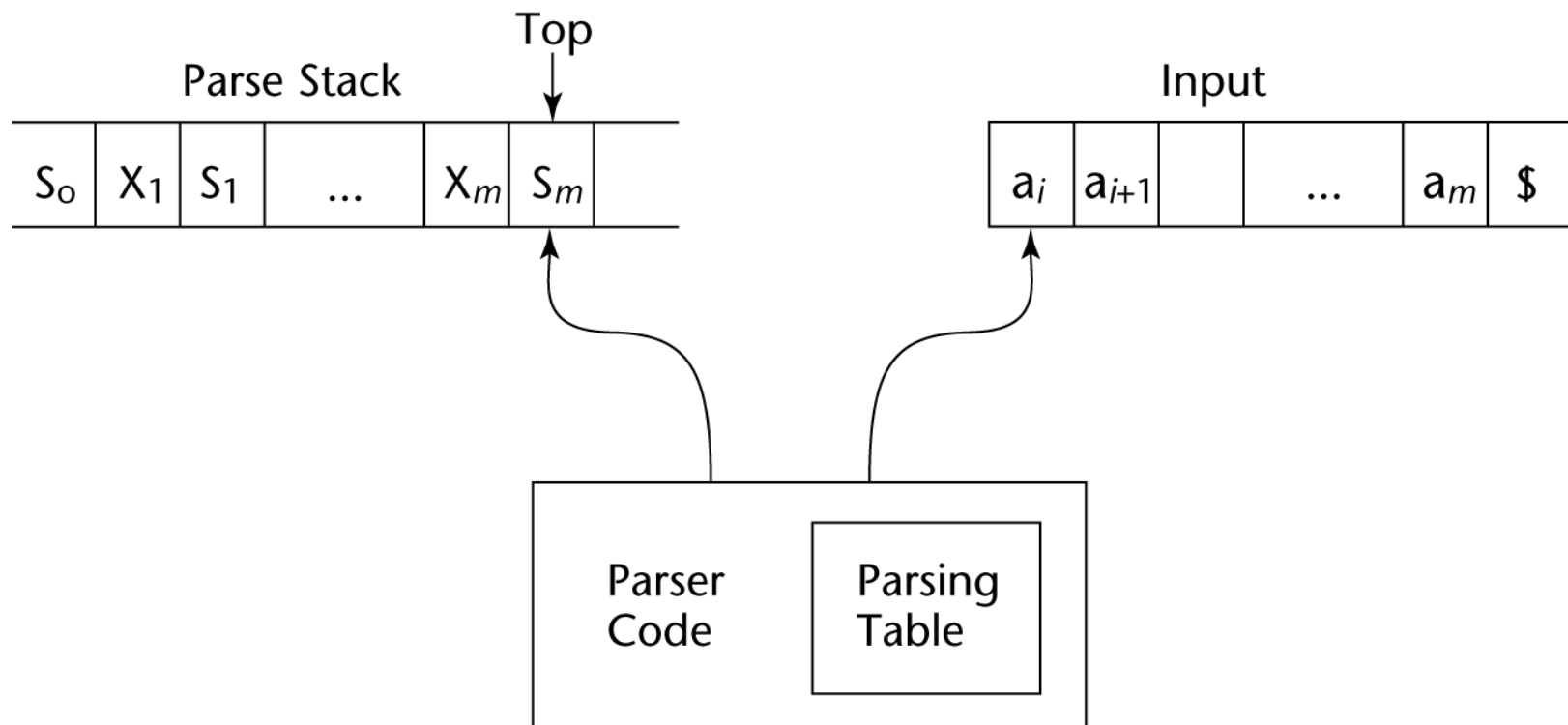
- An LR configuration stores the state of an LR parser

$(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m, a_i a_{i+1} \dots a_n \$)$

Bottom-up Parsing (continued)

- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
 - The ACTION table specifies the action of the parser, given the parser state and the next token
 - Rows are state names; columns are terminals
 - The GOTO table specifies which state to put on top of the parse stack after a reduction action is done
 - Rows are state names; columns are nonterminals

Structure of An LR Parser



Bottom-up Parsing (continued)

- Initial configuration: $(S_0, a_1 \dots a_n \$)$
- Parser actions:
 - For a Shift, the next symbol of input is pushed onto the stack, along with the state symbol that is part of the Shift specification in the Action table
 - For a Reduce, remove the handle from the stack, along with its state symbols. Push the LHS of the rule. Push the state symbol from the GOTO table, using the state symbol just below the new LHS in the stack and the LHS of the new rule as the row and column into the GOTO table

Bottom-up Parsing (continued)

- Parser actions (continued):
 - For an Accept, the parse is complete and no errors were found.
 - For an Error, the parser calls an error-handling routine.

LR Parsing Table

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Bottom-up Parsing (continued)

- A parser table can be generated from a given grammar with a tool, e.g., **yacc** or **bison**

Summary

- Syntax analysis is a common part of language implementation
- A lexical analyzer is a pattern matcher that isolates small-scale parts of a program
 - Detects syntax errors
 - Produces a parse tree
- A recursive-descent parser is an LL parser
 - EBNF
- Parsing problem for bottom-up parsers: find the substring of current sentential form
- The LR family of shift-reduce parsers is the most common bottom-up parsing approach