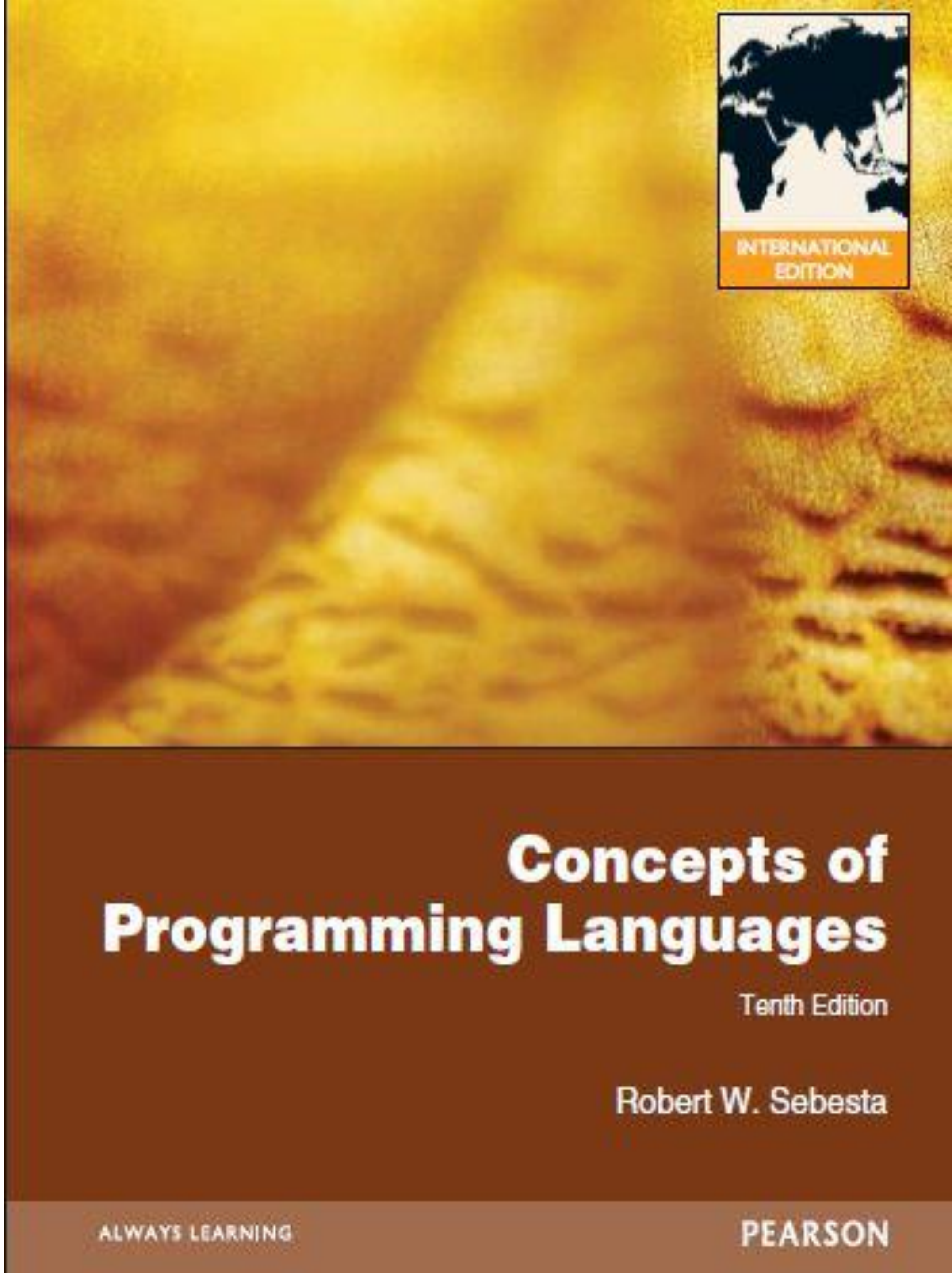
Programming Language

Instructor:
Min-Chun Hu
anita_hu@mail.ncku.edu.tw


Concepts of Programming Languages
Tenth Edition
Robert W. Sebesta
INTERNATIONAL EDITION
ALWAYS LEARNING
PEARSON

Lecture 3
Lexical and Syntax Analysis

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

Introduction

- Three approaches for implementing PLs:
  - Compilation:
    - Translates a program written in high-level programming language into machine code
    - E.g. C++ and COBOL
  - Pure interpretation :
    - Programs are interpreted in their original form by a software interpreter
    - Usually used for small systems in which execution efficiency is not crucial. (E.g. JavaScript, Python)
  - Hybrid implementation :
    - Translate programs written in high-level languages into intermediate forms, which are interpreted.
    - Result in much slower program execution than compiler systems.
    - Use Just-in-Time (JIT) compilers (e.g. JAVA、Microsoft .NET systems) to translate intermediate code to machine code.

Introduction

- Syntax analyzer, or parsers, are nearly always based on a formal description of the syntax of the programs (i.e. BNF, context-free grammers)
- Advantages of using BNF to describe syntax:
  - Provides a clear and concise syntax description
  - The parser can be based directly on the BNF
  - Parsers based on BNF are easy to maintain because of their modularity.

Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer*
    - Deals with small-scale language constructs, such as names and numerical literals
    - A finite automaton based on a regular grammar
  - A high-level part called a *syntax analyzer*, or parser
    - Deals with large-scale language constructs, such as expressions, statements, and program units
    - A push-down automaton based on a context-free grammar, or BNF)

Syntax Analysis

- Reasons to Separate Lexical and Syntax Analysis:
  - *Simplicity* – Less complex approaches can be used for lexical analysis; separating them simplifies the parser
  - *Efficiency* – Separation allows optimization of the lexical analyzer
  - *Portability* – Lexical analyzer reads input program files and often includes buffering of that input, hence it is somewhat platform dependent (not portable). The parser is always platform independent (portable).

Lexical Analysis

- A lexical analyzer is a pattern matcher for character strings
  - Find a substring of a given string of characters that matches a given character pattern
- A lexical analyzer is a “front-end” for the parser, which collects characters into logical groupings (*lexemes*) and assigns internal codes (*tokens*) to the groupings according to their structure.

Lexemes vs Tokens

```
result = oldsum - value / 100;
```

Lexemes	Tokens
result	IDENT
=	ASSIGN_OP
oldsum	IDENT
-	SUB_OP
value	IDENT
/	DIV_OP
100	INT_LIT
;	SEMICOLON

Lexical Analysis (Cont.)

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- Lexical-analysis process includes:
  - Skipping comments and white space outside lexemes
  - Insert lexemes for user-defined names into the symbol table, which is used by later phases of the compiler
  - Detect syntactic errors in tokens and report to the user
    - E.g. ill-formed floating-point literals

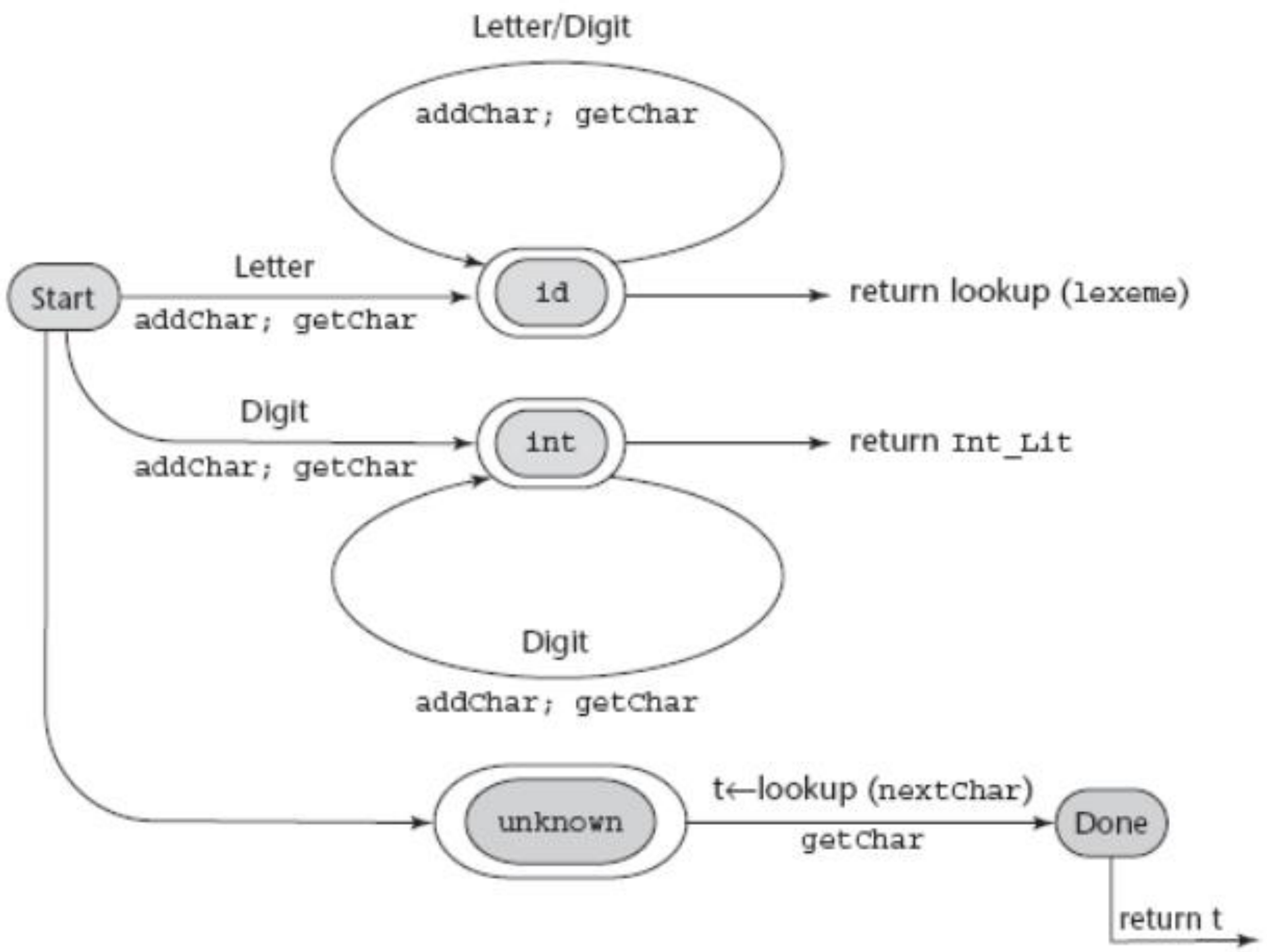
Lexical Analysis (Cont.)

- Three approaches to building a lexical analyzer:
  - Write a formal description of the tokens using a descriptive language related to regular expressions, and use a software tool that constructs a table-driven lexical analyzer from such a description. (e.g. lex for UNIX)
  - Design a state (transition) diagram that describes the token patterns of the language and write a program that implements the state diagram
  - Design a state (transition) diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram

State Diagram

- State diagrams are representations of a class of mathematical machines called finite automata.
- Finite automata can be designed to recognize members of regular languages.
- The tokens of a programming language are a regular language, and a lexical analyzer is a finite automaton.

State Diagram (Cont.)



A state diagram to recognize names, parentheses, and arithmetic operators

State Diagram Design

- A state diagram is a directed graph with nodes labeled with state names and arcs labeled with the input characters that cause the transitions among the states.
- A naïve state diagram would have a transition from every state on every character in the source language – such a diagram would be very large!

Lexical Analysis (Cont.)

- In many cases, transitions can be combined to simplify the state diagram
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent
    - Use a character class that includes all letters
  - When recognizing an integer literal, all digits are equivalent
    - Use a digit class

Lexical Analysis (Cont.)

- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word

Lexical Analysis (Cont.)

- Convenient utility subprograms:
  - `getChar`
    - Get the next character of input and put it in the global variable `nextChar`
    - Determine the class of the input character and put it in the global variable `charClass`
  - `addChar`
    - Put the character in `nextChar` into the string array `lexeme`
  - `getNonBlank`
    - Skip white space every time the analyzer is called
  - `lookup`
    - Determines whether the string in `lexeme` is a reserved word
    - Returns a token code

Lexical Analyzer

Implementation:

→ SHOW `front.c` (pp. 192–197)

- Following is the output of the lexical analyzer of `front.c` when used on `(sum + 47) / total`

```
Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is sum
Next token is: 21 Next lexeme is +
Next token is: 10 Next lexeme is 47
Next token is: 26 Next lexeme is )
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is total
Next token is: -1 Next lexeme is EOF
```

The Parsing Problem

- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly
  - Produce the parse tree, or at least a trace of the parse tree, for the syntactically correct program
- Two categories of parsers
  - *Top down* – produce the parse tree, beginning at the root
    - Order is that of a leftmost derivation
    - Traces or builds the parse tree in preorder
  - *Bottom up* – produce the parse tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation
- Useful parsers look only one token ahead in the input

Example

id_list→ id id_list_tail

id_list_tail→ , id id_list_tail

id_list_tail→ ;

- Top down?
- Bottom up?

Notations for grammar symbols

- Terminal symbols (lexemes)
  - (a,b,…)
- Nonterminal symbols (abstractions)
  - (A,B,…)
- Terminal or nonterminals
  - (W,X,Y,Z)
- Strings or terminals (sentence)
  - (w,x,y,z)
- Mixed strings (RHS of grammar rules)
  - $(\alpha, \beta, \delta, \gamma)$

The Parsing Problem (Cont.)

- Top-down Parsers
  - Given a sentential form, xAα , the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A

    Current sequential form : xAα

    A –rules: A → bB | cBb | a

    Next sentential form : xbBα, xcBbα or xaα
- The most common top-down parsing algorithms (LL parsers) based on BNF are implemented by:
  - coded implementation: Recursive descent parser
  - table driven implementation

The Parsing Problem (Cont.)

- Bottom-up parsers
  - Given a right sentential form, α, determine what substring of α is the RHS of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation

    S→aAc
    A→aA|b

    S=>aAc=>aaAc=>aabc

  - The process of finding the correct RHS (handle) to reduce is complicated since a given right sequential form may include more than one RHS from the grammar of the language to be parsed
  - The most common bottom-up parsing algorithms are in the LR family

The Parsing Problem (Cont.)

- The Complexity of Parsing
  - Parsers that work for any unambiguous grammar are complex and inefficient
    - $O(n^3)$, where n is the length of the input
  - Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time
    - $O(n)$, where n is the length of the input

Recursive-Descent Parsing

- Consists of a collection of subprograms, many of which are recursive. (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

<if_statement>→ if<logic_expr><statement>[else<statement>]

<ident_list>→ident {, ident}

Recursive-Descent Parsing (Cont.)

- 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 (Cont.)

- A grammar for simple arithmetic expressions that does not force any associativity rule:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | ( <expr> )
```

Recursive-Descent Parsing (Cont.)

```
/* 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 (Cont.)

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

Recursive-Descent Parsing (Cont.)

- 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 (Cont.)

```
/* 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 */
```

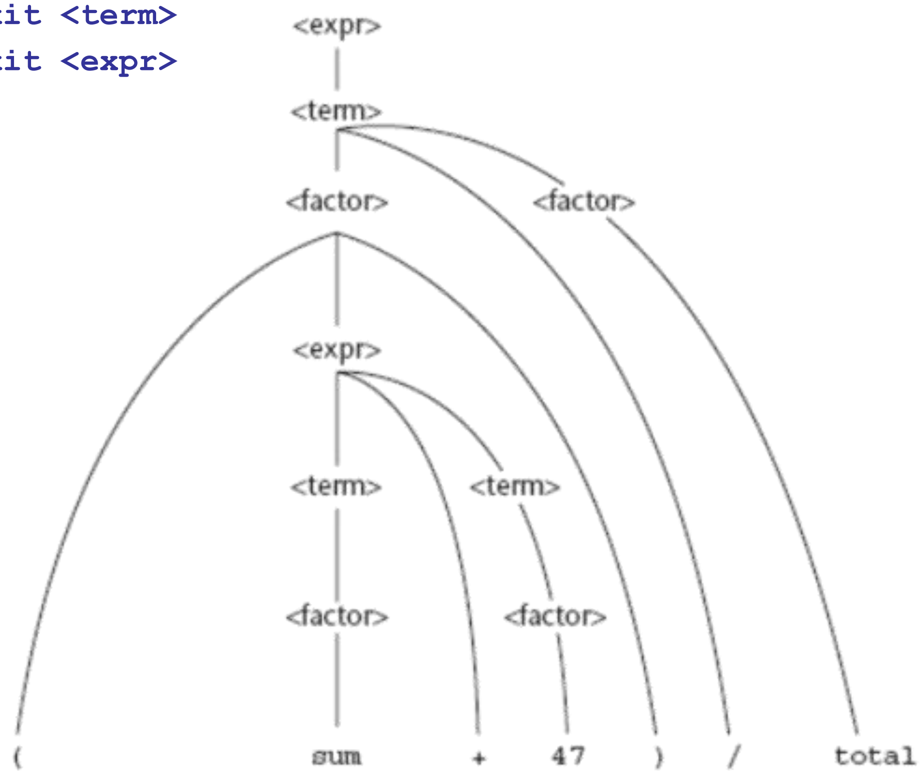
Recursive-Descent Parsing (Cont.)

```
/* 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 (Cont.)

- 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 (Cont.)

- The LL Grammar Class
  - The Left Recursion Problem: e.g. $A \rightarrow A+B$
    - If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parse
    - A grammar can be modified to remove direct left recursion as follows:

      For each nonterminal, A,

      1. Group the A-rules as $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
         where none of the β's begins with A
      2. Replace the original A-rules with
         $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$
         $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$

Example

E→ E+T | T

T→T*F | F

F→ (E) | id

- For the E rules, we have α_1=+T, β=T.

  Replace E rules with: E→TE', E'→+TE'|ϵ
- For the T rules, we have α_1=*F, β=F.

  Replace T rules with: T→FT', T'→*FT'|ϵ

Recursive-Descent Parsing (Cont.)

- 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 the next token of input
  - Def: FIRST(α) = $\{a \mid \alpha =>^* a\beta\}$

    (If $\alpha =>^* \varepsilon$, ε is in FIRST(α))
- 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

- $A \rightarrow aB \mid bAb \mid Bb$
  $B \rightarrow cB \mid d$
  FIRST sets for the RHSs of the A–rules are: {a} {b} {c,d}

➔ Disjoint !

- $A \rightarrow aB \mid BAb$
  $B \rightarrow aB \mid b$
  FIRST sets for the RHSs of the A–rules are: {a} {a,b}

➔ Not disjoint !

Recursive-Descent Parsing (Cont.)

- Left factoring can be used to modify a non-disjoint grammar to a disjoint one:
  - Replace

    <variable> → identifier | identifier [<expression>]

    with

    <variable> → identifier <new>

    <new> → ε | [<expression>]

    or

    <variable> → identifier [[<expression>]]

    (the outer brackets are metasymbols 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

Grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Rightmost derivation:

$E \Rightarrow \underline{E + T}$

$\Rightarrow E + \underline{T*F}$

$\Rightarrow E + T*\underline{id}$

$\Rightarrow E + \underline{F}*id$

$\Rightarrow E + \underline{id}*id$

$\Rightarrow \underline{T} + id*id$

$\Rightarrow \underline{F} + id*id$

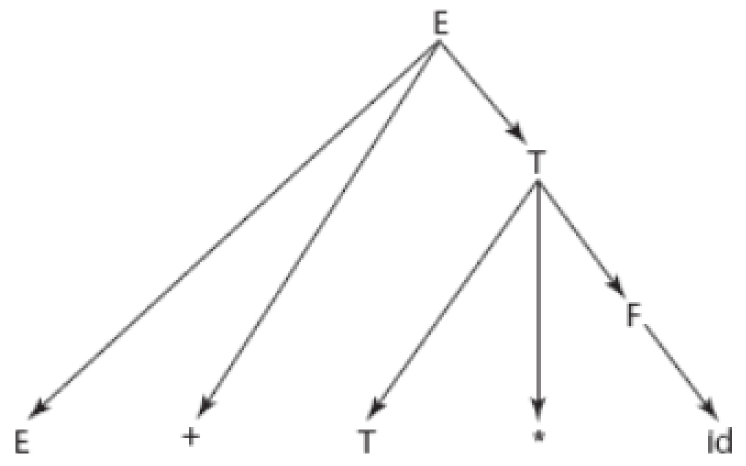
$\Rightarrow \underline{id} + id*id$

Bottom-up Parsing (Cont.)

- 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 (Cont.)

- 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 (Cont.)

- 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 (Cont.)

- 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.

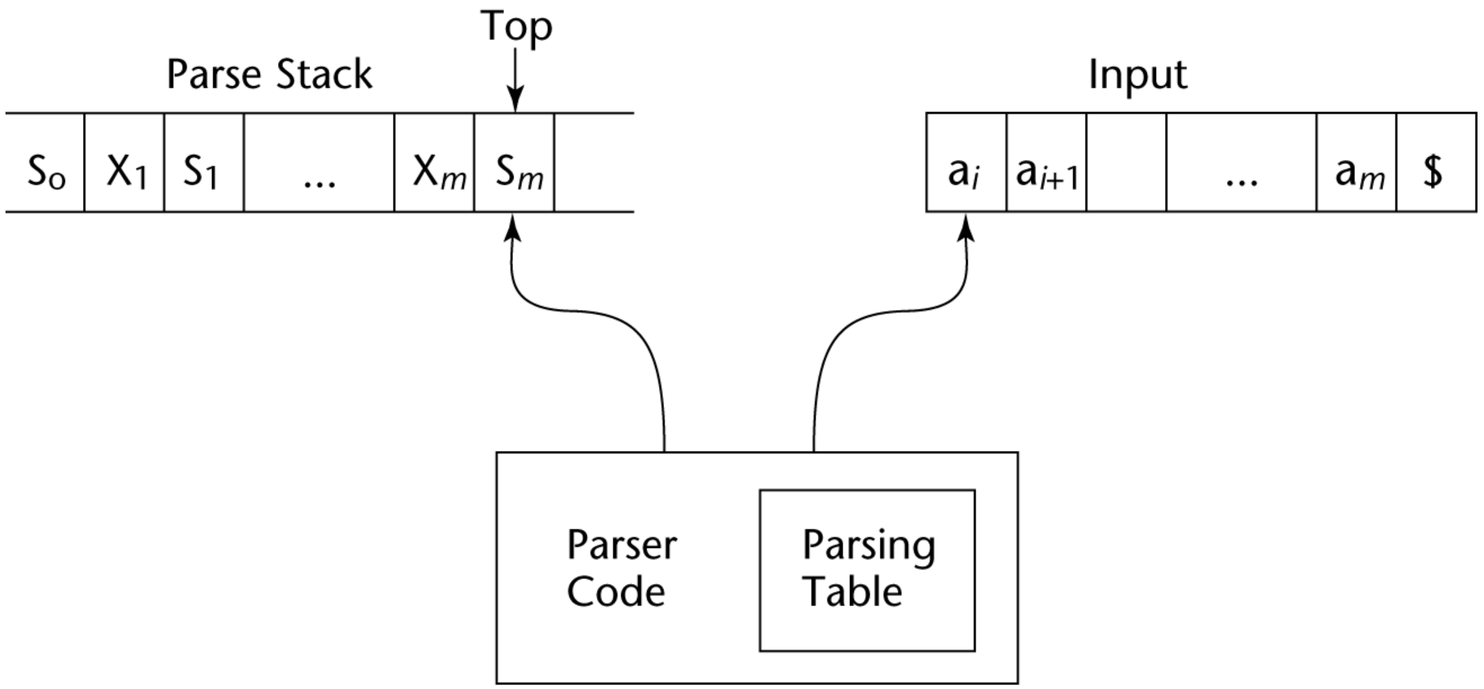
Bottom-up Parsing (Cont.)

- 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 (Cont.)

- An LR configuration stores the state of an LR parser

  $(S_0X_1S_1X_2S_2\ldots X_mS_m,\ a_ia_{i+1}\ldots a_n\$)$



Bottom-up Parsing (Cont.)

- 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

Bottom-up Parsing (Cont.)

- Initial configuration: (S_0, $a_1 \ldots a_n\$$)
- 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

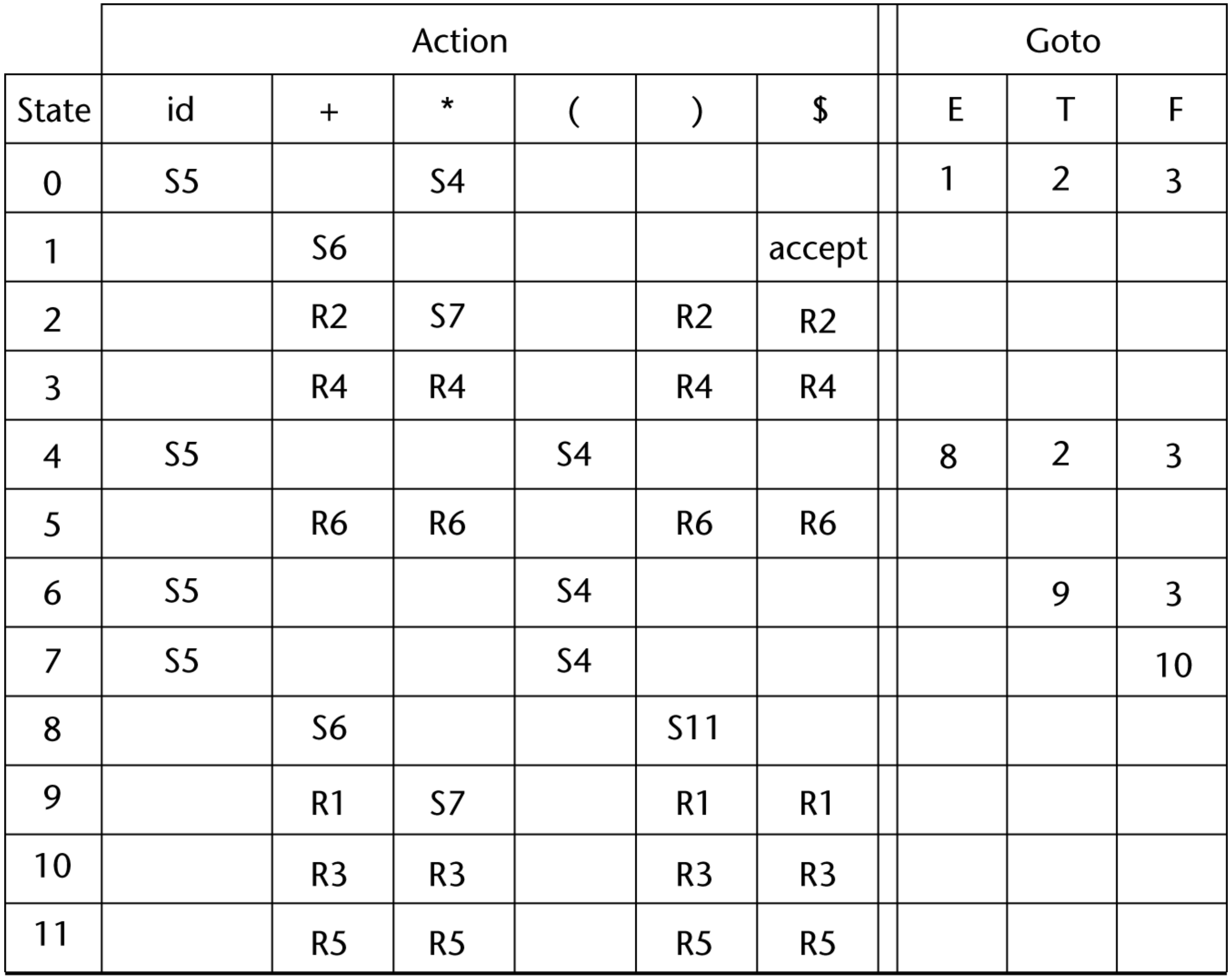
Bottom-up Parsing (Cont.)

- 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

Grammar:

1. E → E+T
2. E → T
3. T → T*F
4. T → F
5. F → (E)
6. F → id



	Action						Goto		
State	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 (Cont.)

- 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

Why not left recursive?

- Expression→Expression+Term
- Corresponding recursive descent parser:

```
function Expression()
{
    Expression();  match('+');  Term();
}
```

- Fall into infinite recursion when executed !