National University
Of Computer and Emerging Sciences

Theory of Programming Languages

Lexical and Syntax Analyzer

Sajid Anwer

Department of Computer Science,
FAST-NUCES, CFD Campus

Lecture Outline

- Lexical and syntax analysis fundamentals
- Lexical Analysis
- Syntax Analysis (Parsing)
- Recursive-Descent Parsing
- Bottom-Up Parsing

Lexical and Syntax Analysis Fundamentals

- Programming language *implementation approaches* include:
  - » Implementation
  - » Interpretation
  - » Hybrid
  - » Just-in-Time (JIT)

- Language implementation systems must *analyze source code*, regardless of the specific implementation approach

- Nearly all syntax analysis is based on *a formal description* of the syntax of the source language

Lexical and Syntax Analysis Fundamentals

- The *syntax analysis portion* of a language processor nearly always consists of two parts:
  - A *low-level part* called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar).
  - A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF).
- 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*

Lexical and Syntax Analysis Fundamentals

- 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* - parts of the lexical analyzer may not be portable, but the parser always is portable, *why?*

National University
Of Computer and Emerging Sciences

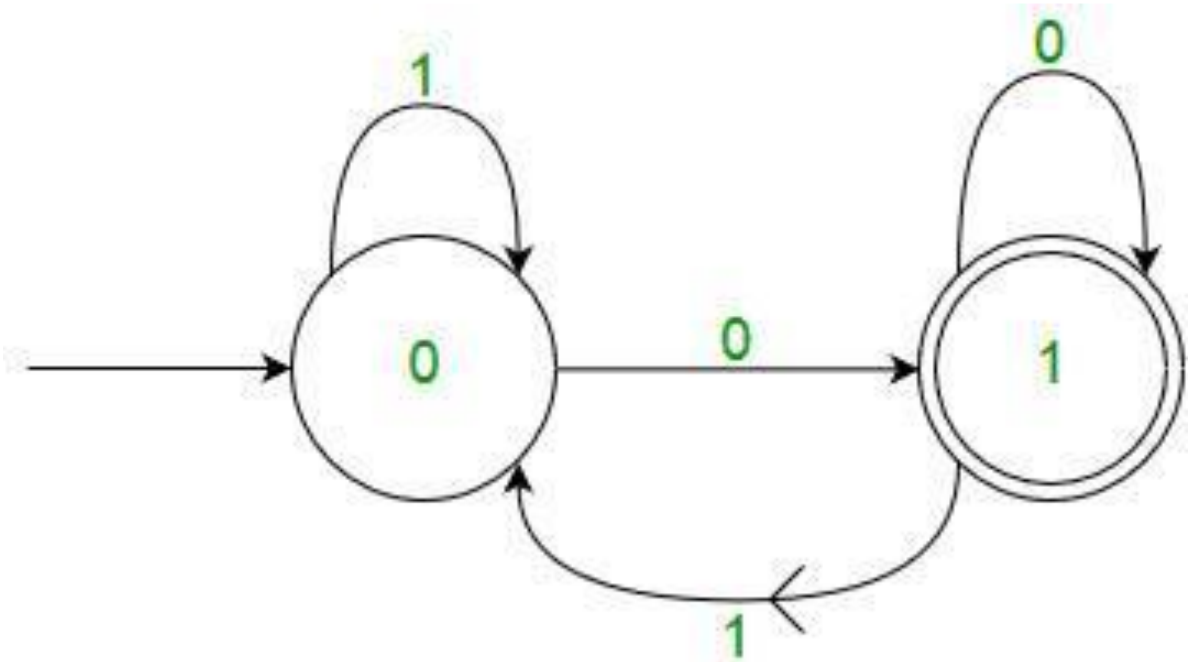
Lexical Analysis

- A lexical analyzer is a *pattern matcher* for character strings
- A lexical analyzer is a *"front-end"* for the parser
- Identifies substrings of the source program that belong together - *lexemes*
  - » Lexemes match a character pattern, which is *associated with a lexical category* called a *token*
  - » `sum` is a lexeme; its token may be `IDENT`
- The lexical analyzer is *usually a function* that is called by the parser when it needs the *next token*

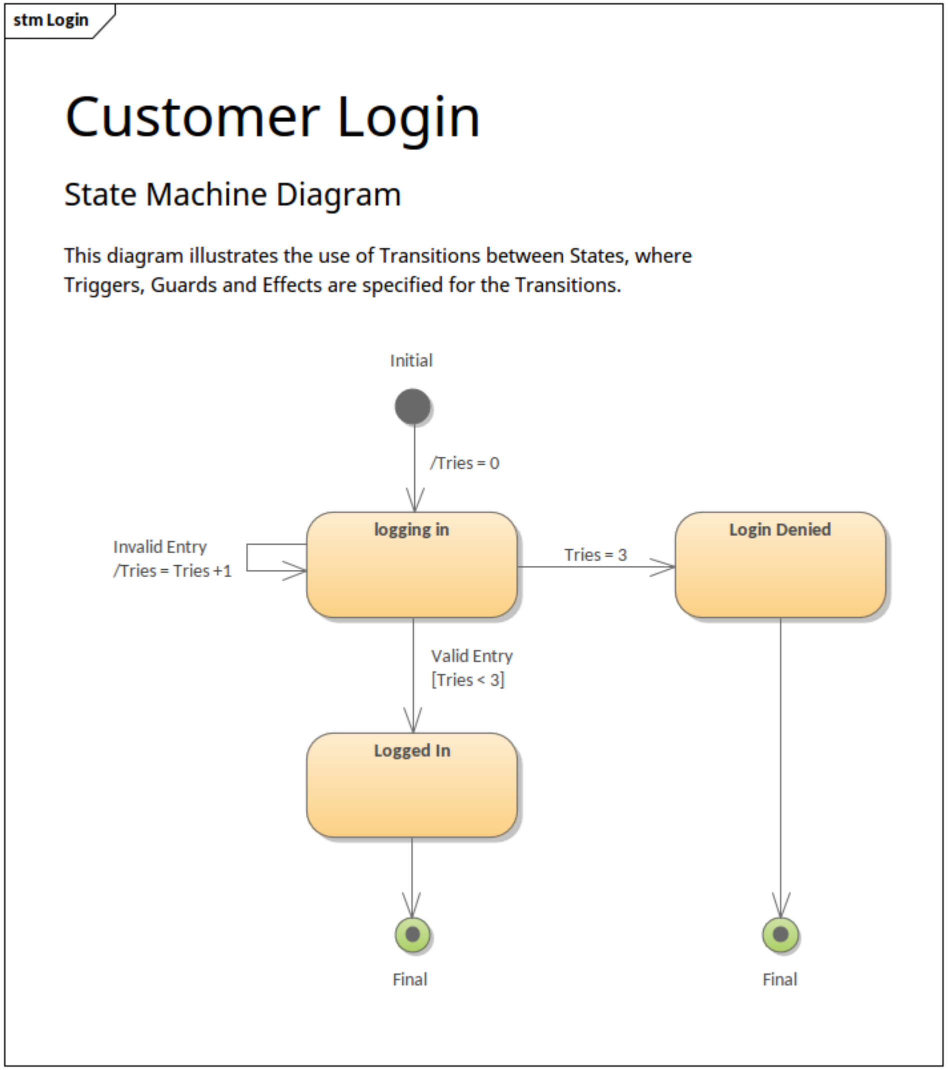
Lexical Analysis (continued)

- Three approaches to building a lexical analyzer:
  - » Write a *formal description* of the tokens and use a software tool that *constructs* a table-driven lexical analyzer from such a description
  - » Design a *state diagram* that describes the tokens and write a program that implements the state diagram
  - » Design a state diagram that describes the tokens and hand-construct a *table-driven implementation* of the state diagram

Lexical Analysis (continued)



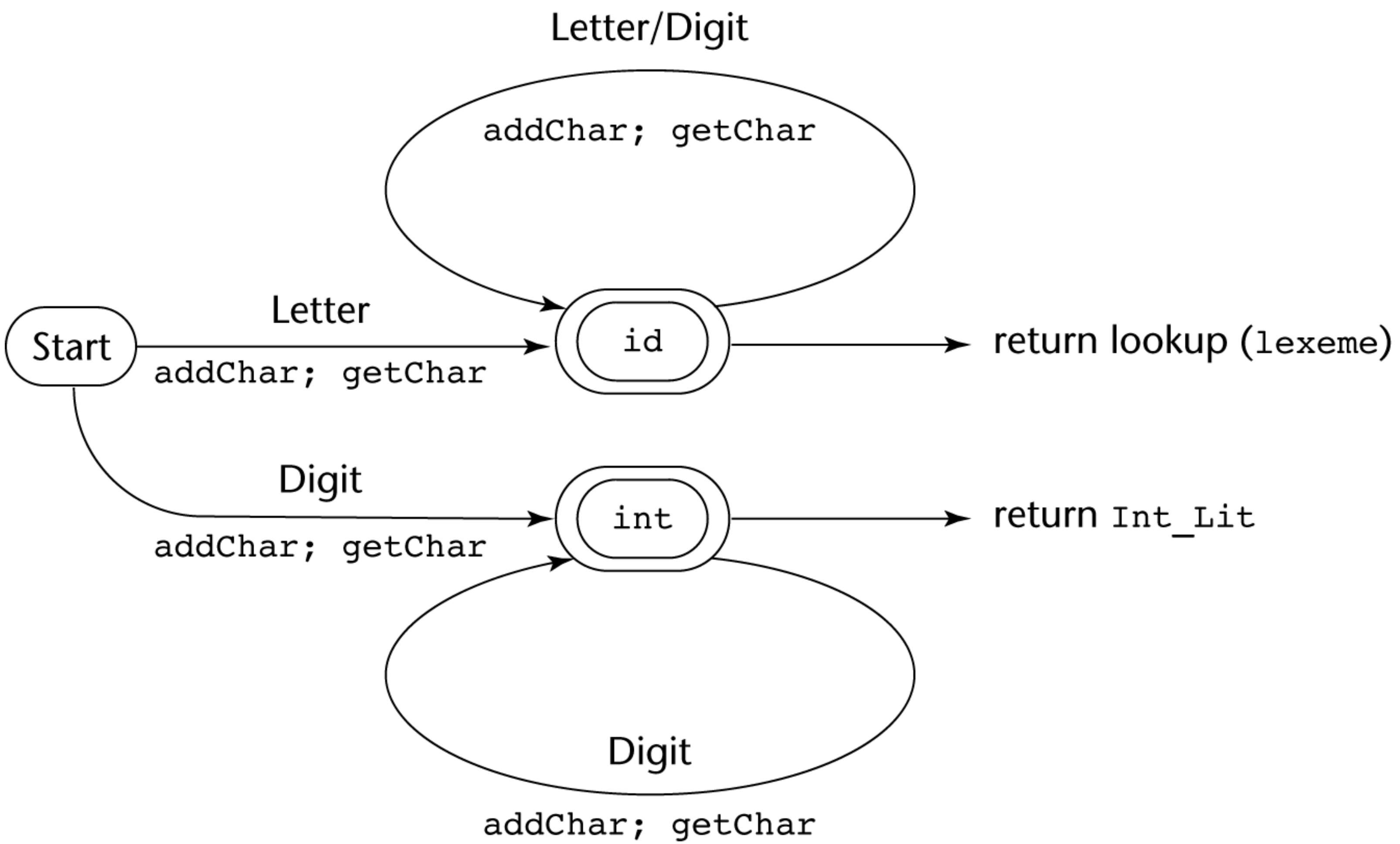
State Diagram Design



State Diagram Design

- 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
- 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
- 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

State Diagram



National University Of Computer and Emerging Sciences

Lexical Analyzer

Implementation:

→ SHOW `front.c` (pp. 172-177)

- 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
```

Syntax Analysis - Parsing

- 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 program

Syntax Analysis - Parsing

- 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

Syntax Analysis - Parsing

- Top-down Parsers
  - » Given a sentential form, $xA\alpha$, 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

- The most common top-down parsing algorithms:
  - » Recursive descent - a coded implementation
  - » LL parsers - table driven implementation

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)

- 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)

- A grammar for simple expressions:

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

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)

- 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>
```

National University
Of Computer and Emerging Sciences

Recursive-Descent Parsing (continued)