# Chapter 4

# Lexical and Syntax Analysis

# Chapter 4 Topics

- Introduction
- Lexical Analysis

# Introduction

- 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 (BNF)

# Syntax Analysis

- 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

# 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

# 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`

# Example

Consider the following example of an assignment statement:

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

Following are the tokens and lexemes of this statement:

| *Token* | *Lexeme* |
|---|---|
| IDENT | result |
| ASSIGN_OP | = |
| IDENT | oldsum |
| SUB_OP | - |
| IDENT | value |
| DIV_OP | / |
| INT_LIT | 100 |
| SEMICOLON | ; |

# Lexical Analysis (continued)

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- Three approaches to building a lexical analyzer:
  - Write a formal description of the tokens patterns of the language using a descriptive language related to regular expressions, 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

# State Diagram Design

- A state transition diagram, or just state diagram, is a directed graph.
- The nodes of a state diagram are labeled with state names. The arcs are labeled with the input characters that cause the transitions among the states. An arc may also include actions the lexical analyzer must perform when the transition is taken.
- 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 (continued)

- 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
- Because variable names can include digits, the transition from the node following the first character of a name can use a single transition on LETTER or DIGIT to continue collecting the characters of a name.

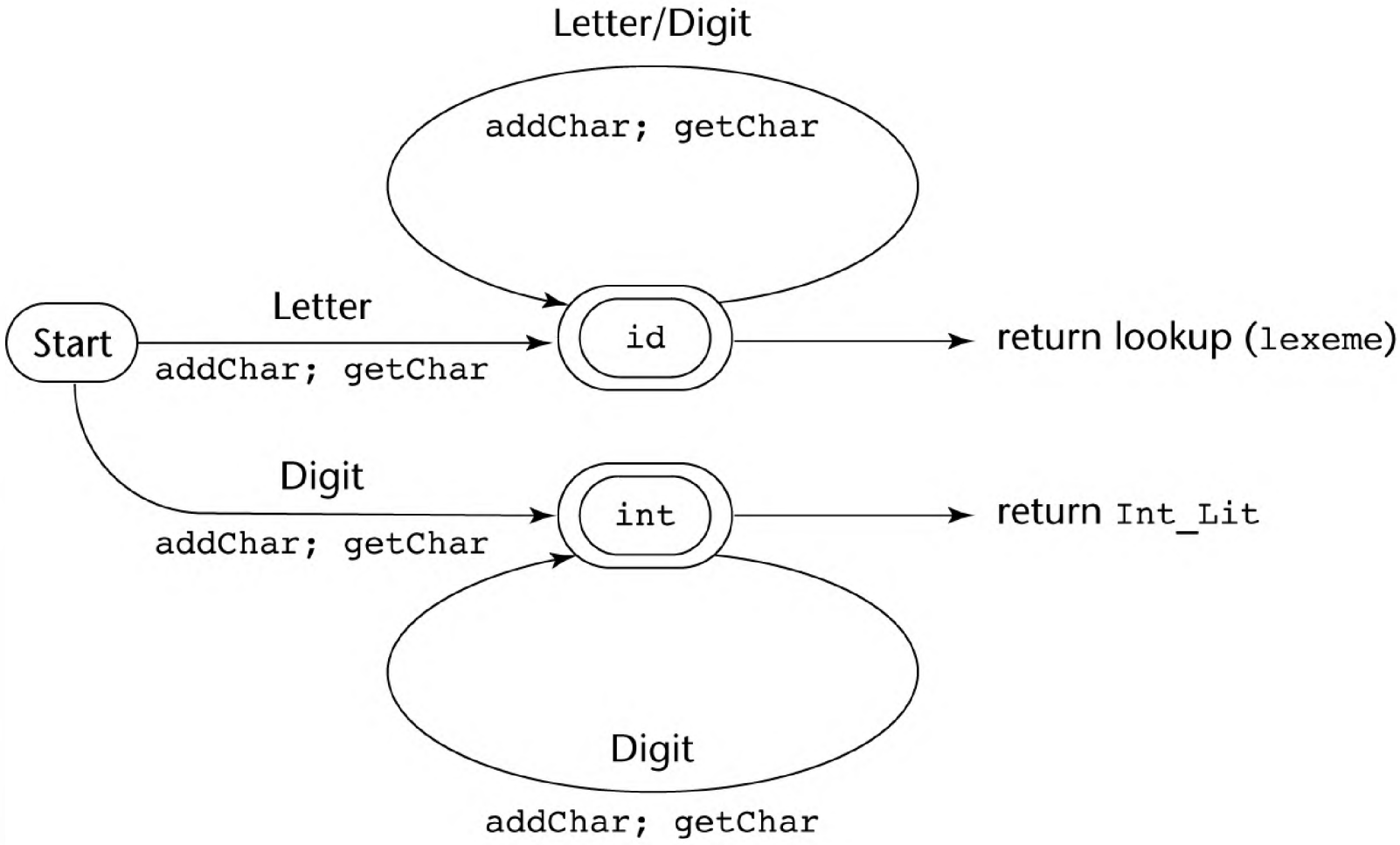
# Lexical Analysis (continued)

- 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 (continued)

- Convenient utility subprograms:
  - **`getChar`** – gets the next character of input, puts it in the global variable **`nextChar`**, determines its class and puts the class in the global variable **`charClass`**
  - **`addChar`** – puts the character from **`nextChar`** into the place the lexeme (string array) is being accumulated, **`lexeme`**
  - **`lookup`** – determines whether the string in **`lexeme`** is a reserved word (returns a code)

# State Diagram



# 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
```

# 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
- A syntax analyzer (parser)
  - Detects syntax errors
  - Produces a parse tree