Chapter 4

Lexical and Syntax Analysis

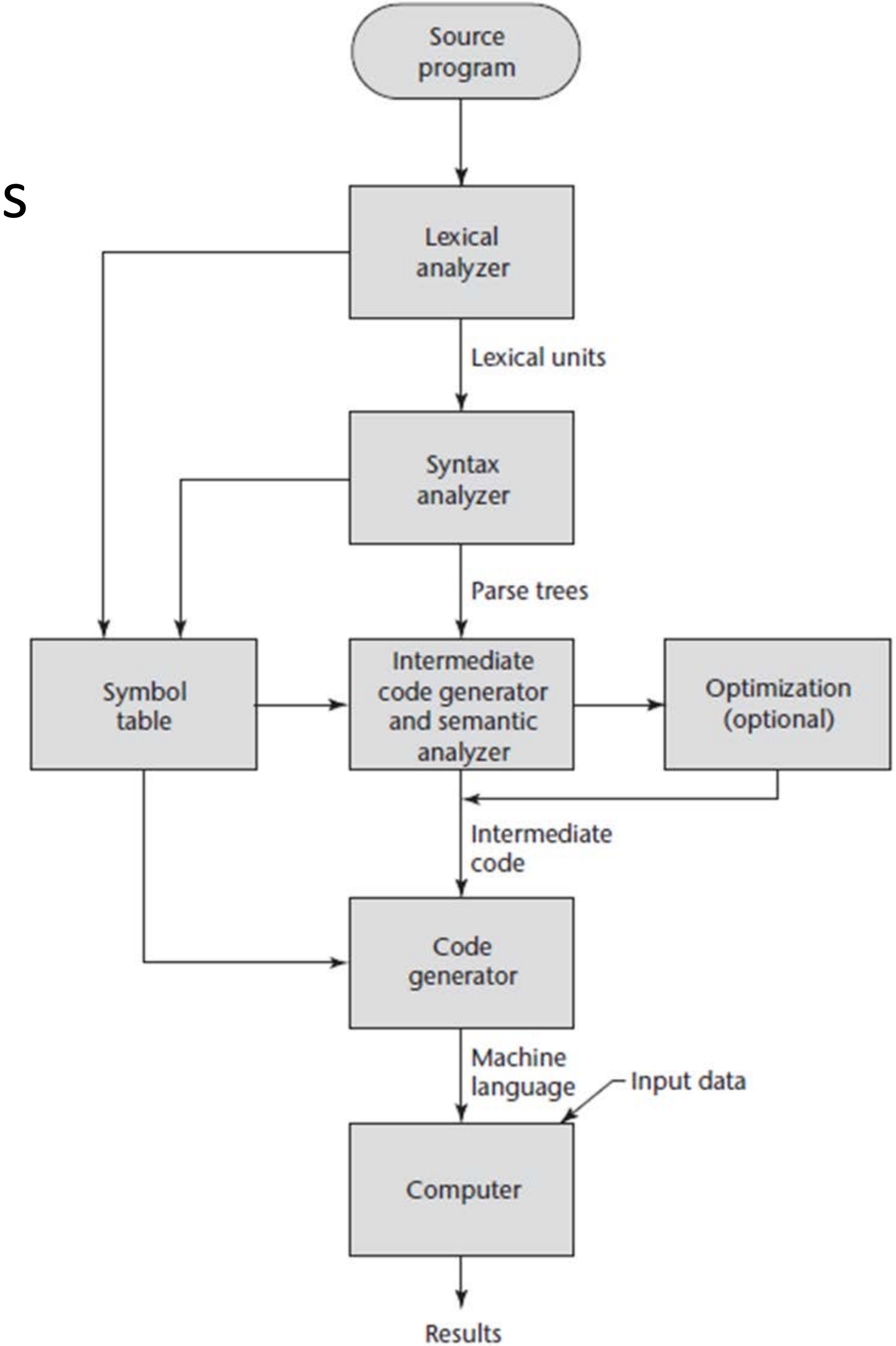
Topics

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

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)

Recall:
The Compilation Process



Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - *Lexical analyzer*, or *Scanner*: a low-level part
    - mathematically, a *Finite Automaton* based on a *Regular Grammar* – CSci 435.
  - *Syntax analyzer*, or *Parser*: a high-level part
    - mathematically, a *Push-Down Finite Automaton* based on a *Context-Free Grammar* in BNF – CSci 435.

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.

Why 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 that requires a significant portion of total compilation time.
- *Portability*:
  - parts of the lexical analyzer may not be portable,

    but the parser always is portable;

    the parser can be platform-independent.

Lexical Analysis (LA) = Scanner

- A *lexical analyzer* is a *pattern matcher* for character strings.
  - a *front-end* for the parser.
  - It 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*.
  - E.g.) `result = oldsum – value / 100;`

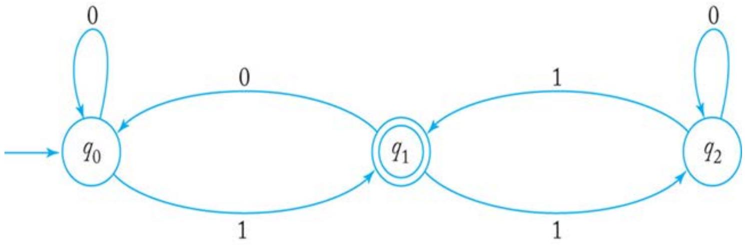
Lexeme	Token
result, oldsum, value	IDENT
=	ASSIGN_OP
–	SUB_OP
/	DIV_OP
100	INT_LIT

Lexical Analysis (cont.)

- Lexical analyzer: a function that is called by the parser when it needs the next token.
- Three approaches to build a lexical analyzer:
  - Write a *formal description of the tokens* (in *regular expression*) and use a software tool that constructs a *table-driven lexical analyzer* from such a description.
  - Design a *state (transition) diagram* that describes the tokens and write a *program* that implements the state diagram.
  - Design a *state (transition) diagram* that describes the tokens and *hand-construct a table-driven* implementation of the state diagram.

State (Transition) Diagram Design

- a directed graph (to represent a transition function)
- 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!



- Representation of transitions of computing machines, Finite Automata (FA).
- FA is designed to recognize a *regular language* generated by a *regular grammar.*
- The *tokens* of prog. language are a regular language.
- A lexical analyzer is a Finite Automaton.

State (Transition) Diagram Design

- From CSci 435: A *Finite Automata (FA)* is defined by the quintuple $M = (Q, \Sigma, \delta, q_0, F)$

  where

  Q : a *finite* set of *states*

  Σ : a set of symbols, called the *input alphabet*

  $\delta : Q \times \Sigma \rightarrow Q \quad \forall q \in Q$ -- a *transition function*

  $q_0 \in Q$: the *initial state*

  $F (\subseteq Q)$: a set of the *final states*.

- Example: Consider the FA:

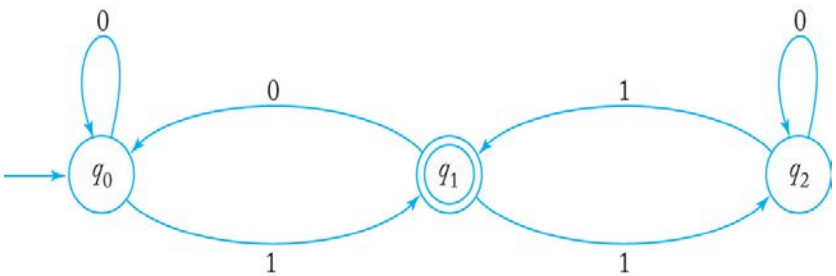
  $Q = \{ q_0, q_1, q_2 \}$, $\Sigma = \{ 0, 1 \}$, $F = \{ q_1 \}$

  where the transition function is given by

  $\{ \delta(q_0, 0) = q_0, \ \delta(q_0, 1) = q_1,$

  $\delta(q_1, 0) = q_0, \ \delta(q_1, 1) = q_2,$

  $\delta(q2, 0) = q_2, \ \delta(q_2, 1) = q_1 \}.$

  Language recognized by FA

  $= 0^*1 (00^*1 + 10^*1)^*$ in regular expression.

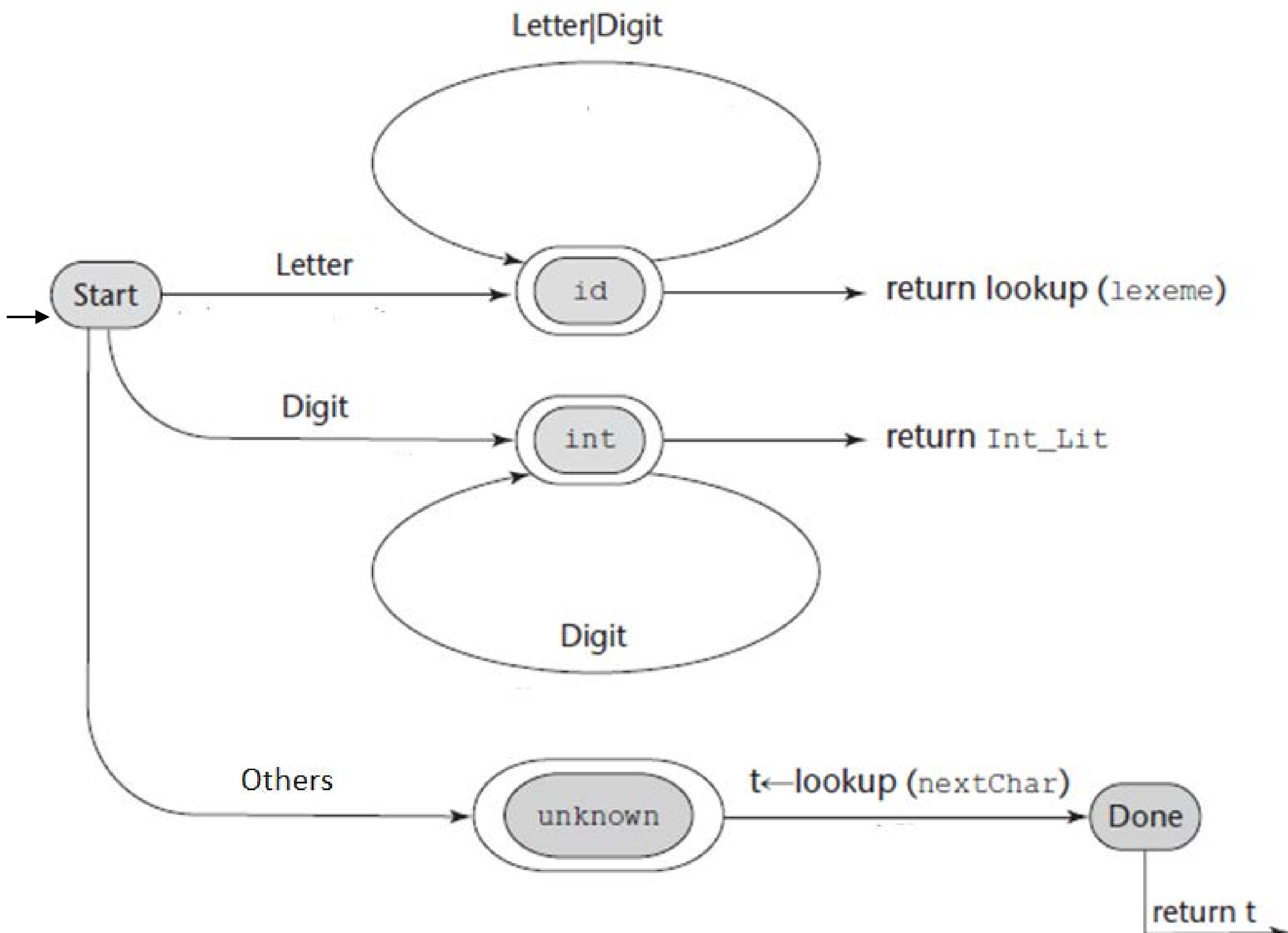
Lexical Analysis (cont.)

- 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:
      {A, B, …, Z, *a*, *b*, …, *z*}
  - When recognizing an integer literal,
    all digits are equivalent - use a *digit class*: {0, 1, 2, …, 9}
- 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*.

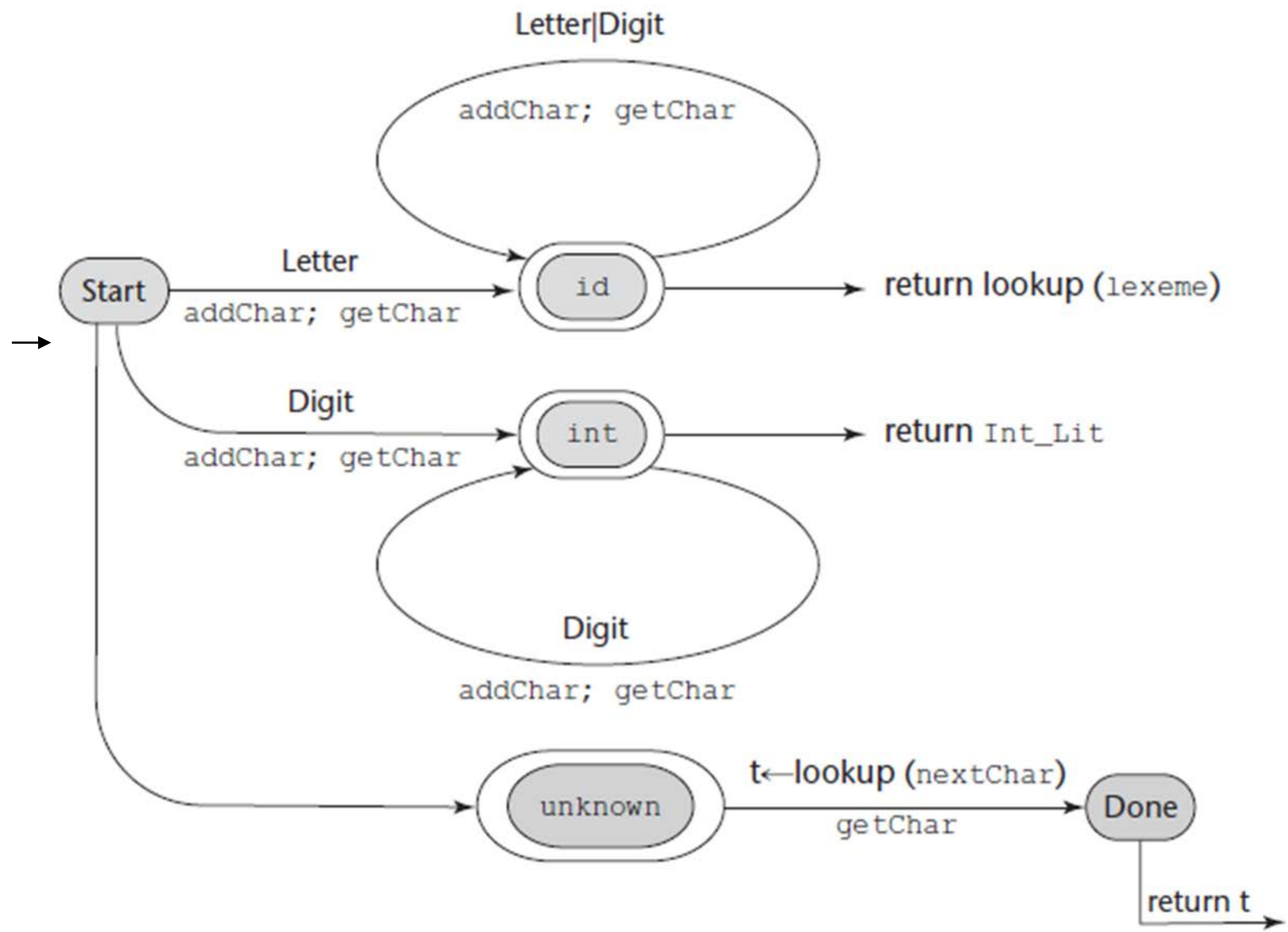
Lexical Analysis (cont.)

- Utility subprograms:
  - **`getChar`:**
    - gets the next character of input,
      puts it in (global variable) **`nextChar`**,
      determines its *class* and puts the class in **`charClass`**
      e.g.) Letter, Digit
  - **`addChar`:**
    - puts the character from **`nextChar`** into
      the place the lexeme is being accumulated, **`lexeme.`**
    - `Lexeme` is implemented as a character string or an array,
  - **`lookup`:**
    - determines the token code
      - for a single-character token or lexeme
      - whether the string in **`lexeme`** is a reserved word

State transition Diagram



where the **id**, **int**, and unknown are the final states to classify a token, i.e. a class of lexeme.



`id`: Letter·(Letter + Digit)* where '+' expresses '|' (i.e. OR)
`Int`: Digit·Digit* in the regular expression.

front.c

```
/* Function declarations */
void  addChar();
void  getChar();
void  getNonBlank();
int   lex();
```

```
/* Character classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99
```

```
/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
```

```
Main
        getChar ();
        do {
                lex ()
        }  while ( nextToken ≠ EOF );

lex () → nextToken
        // identify identifiers/integer_literal/unknown characters  and classify its token
        switch (charClass) {
                case Letter | Digit :
                        addChar ();          // form a lexeme
                        getChar ();
                        while ( ) {
                            addChar( );
                            getChar( );
                        }
                        nextToken = **    // based on the result of case
                case Unknown :
                        lookup (nextChar)   // for unknown characters (e.g. (, ), operators)
        }
        return nextToken;

lookup (ch)   → nextToken
        //  classify a token for an unknown character
        addChar ();
        nextToken = ***
        return nextToken;
```

```
/* main driver */
main() {

/* Open the input data file and process its contents */
   if  ((in_fp = fopen("front.in", "r")) == NULL)
     printf("ERROR - cannot open front.in \n");
   else  {
     getChar();
     do   {
       lex();
     }  while  (nextToken! = EOF);
   }
}
```

Overall structure of Lexical Analyzer

Details: slide-#20

```
Main
        getChar ();
        do {
                lex ()
        }   while ( nextToken ≠ EOF );

lex () → nextToken
        // identify identifiers/integer_literal/unknown characters  and classify its token
        switch (charClass) {
                case Letter | Digit :
                    addChar ();         // form a lexeme
                    getChar ();
                    while (  ) {
                        addChar( );
                        getChar( );
                    }
                    nextToken = **    // based on the result of case
                case Unknown :
                    lookup (nextChar)   // for unknown characters (e.g. (, ), operators)
        }
        return nextToken;

lookup (ch)    → nextToken
        //  classify a token for an unknown character
        addChar ();
        nextToken =  ***
        return nextToken;
```

```
/* getChar - a function to get the next character of
             input and determine its character class */

 void getChar() {
   if  ((nextChar = getc(in_fp)) ≠ EOF) {
     if  (isalpha(nextChar))
      charClass = LETTER;
     else if  (isdigit(nextChar))
           charClass = DIGIT;
          else  charClass = UNKNOWN;
   }
   else
     charClass = EOF;
}
```

```
/* addChar - a function to add nextChar to lexeme */
void  addChar() {
  if  (lexLen <= 98) {
    lexeme[lexLen++] = nextChar;
    lexeme[lexLen] = 0;
  }
  else
    printf("Error - lexeme is too long \n");
}
```

```
/* getNonBlank - a function to call getChar until it
                 returns a non-whitespace character */
void getNonBlank() {
  while  (isspace(nextChar))
    getChar();
}
/
```

```
int  lookup(char  ch) {
  switch  (ch) {
    case  '(':
      addChar();
      nextToken = LEFT_PAREN;
      break;

    case  ')':
      addChar();
      nextToken = RIGHT_PAREN;
      break;

    case  '+':
      addChar();
      nextToken = ADD_OP;
      break;

    case  '-':
      addChar();
      nextToken = SUB_OP;
      break;

    case  '*':
      addChar();
      nextToken = MULT_OP;
      break;

    case  '/':
      addChar();
      nextToken = DIV_OP;
      break;

    default:
      addChar();
      nextToken = EOF;
      break;
  }
  return  nextToken;
}
```

```
/* lex - a simple lexical analyzer for arithmetic
         expressions */
int  lex() {
  lexLen = 0;
  getNonBlank();
  switch  (charClass) {

 /* Parse identifiers */
     case  LETTER:
       addChar();
       getChar();
       while  (charClass == LETTER || charClass == DIGIT) {
         addChar();
         getChar();
       }
     nextToken = IDENT;
     break;

 /* Parse integer literals */
     case  DIGIT:
       addChar();
       getChar();
       while  (charClass == DIGIT) {
        addChar();
        getChar();
    }
    nextToken = INT_LIT;
    break;

 /* Parentheses and operators */
     case  UNKNOWN:
       lookup(nextChar);
       getChar();
       break;

 /* EOF */
     case  EOF:
       nextToken = EOF;
       lexeme[0] = 'E';
       lexeme[1] = 'O';
       lexeme[2] = 'F';
       lexeme[3] = 0;
       break;
  } /* End of switch */
  printf("Next token is: %d, Next lexeme is %s\n",
          nextToken, lexeme);
  return  nextToken;
} /* End of function lex */
```

Lexical Analyzer (LA)

- LA locates the next lexeme in the input, determine its associated token code, and returns (lexeme, its token) to the Syntax Analyzer (i.e. Parser).

- Implementation:

→ front.c (pg. 166-170)

– the output of the lexical analyzer for `(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
```

→ 25112110262411-1

```
/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
```

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 the current sentential form.
- The *LR family of shift-reduce parsers* is the most common bottom-up parsing approach.