Lexical and Syntax Analysis

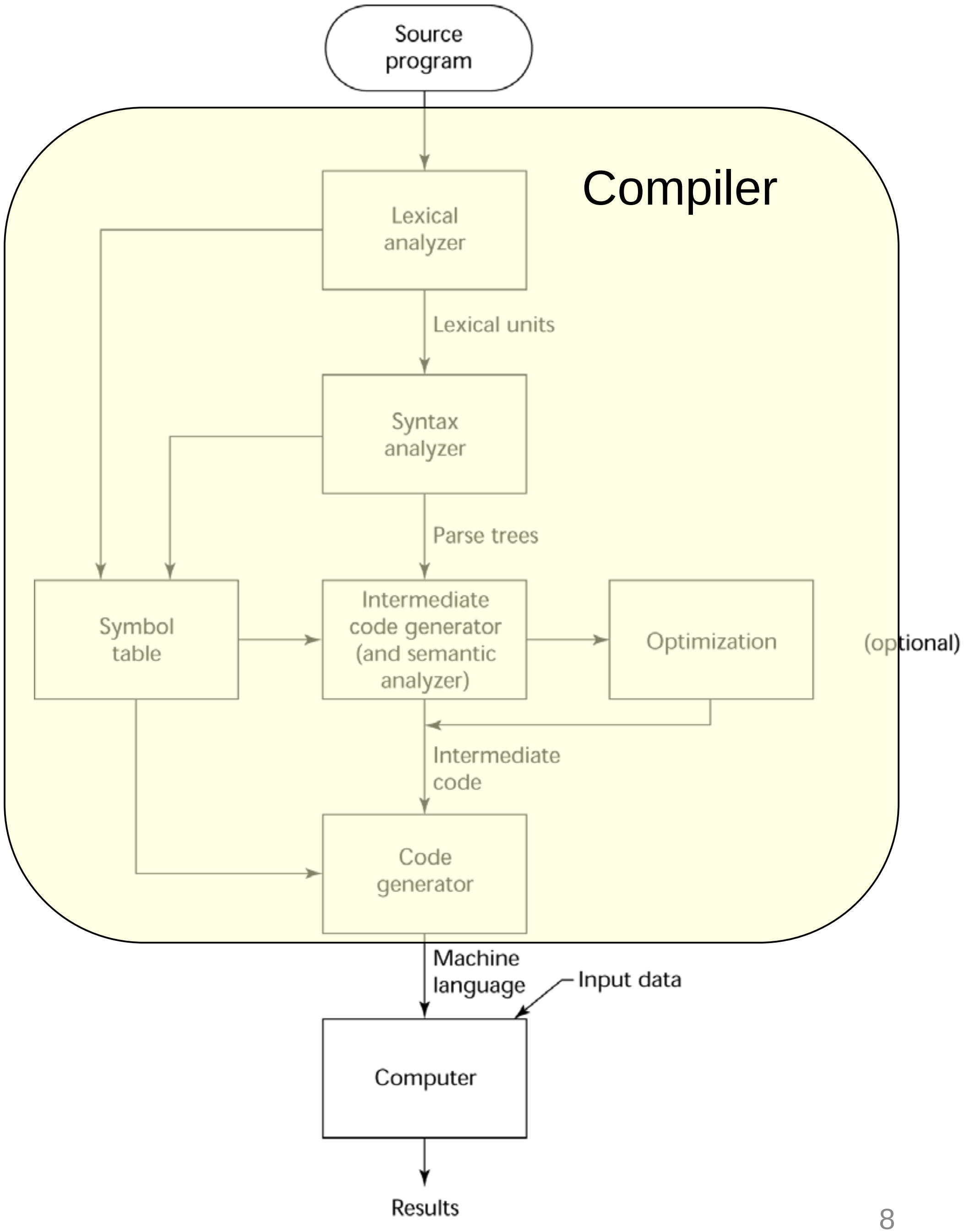
Chapter 4

Chapter 4 Topics

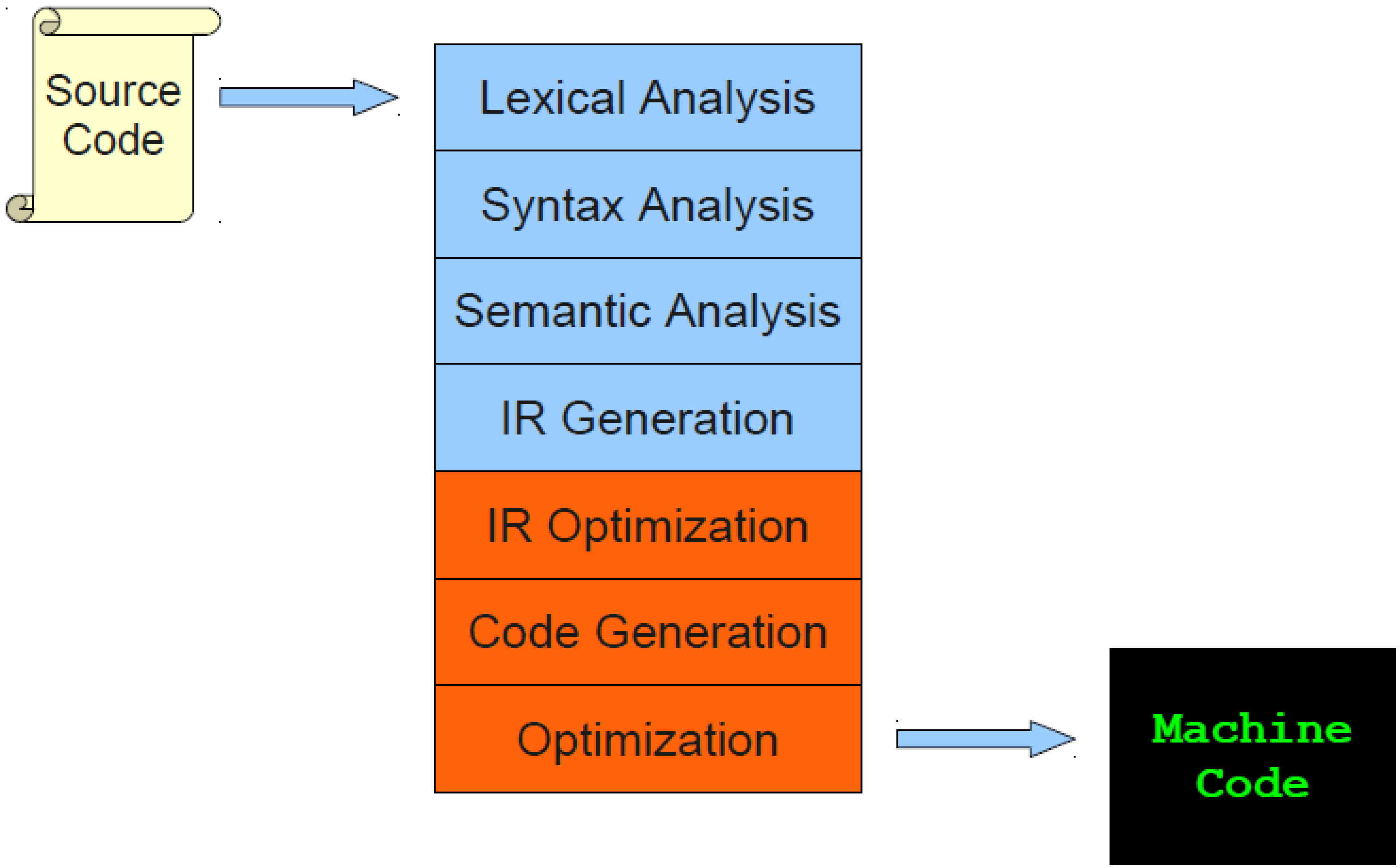
- Introduction
- The Structure of a Modern Compiler
- Lexical Analysis
- The Parsing Problem

The big picture

Compilation Process



The Structure of a Modern Compiler



IR: Intermediate Representation

```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

Lexical Analysis

Syntax Analysis

Semantic Analysis

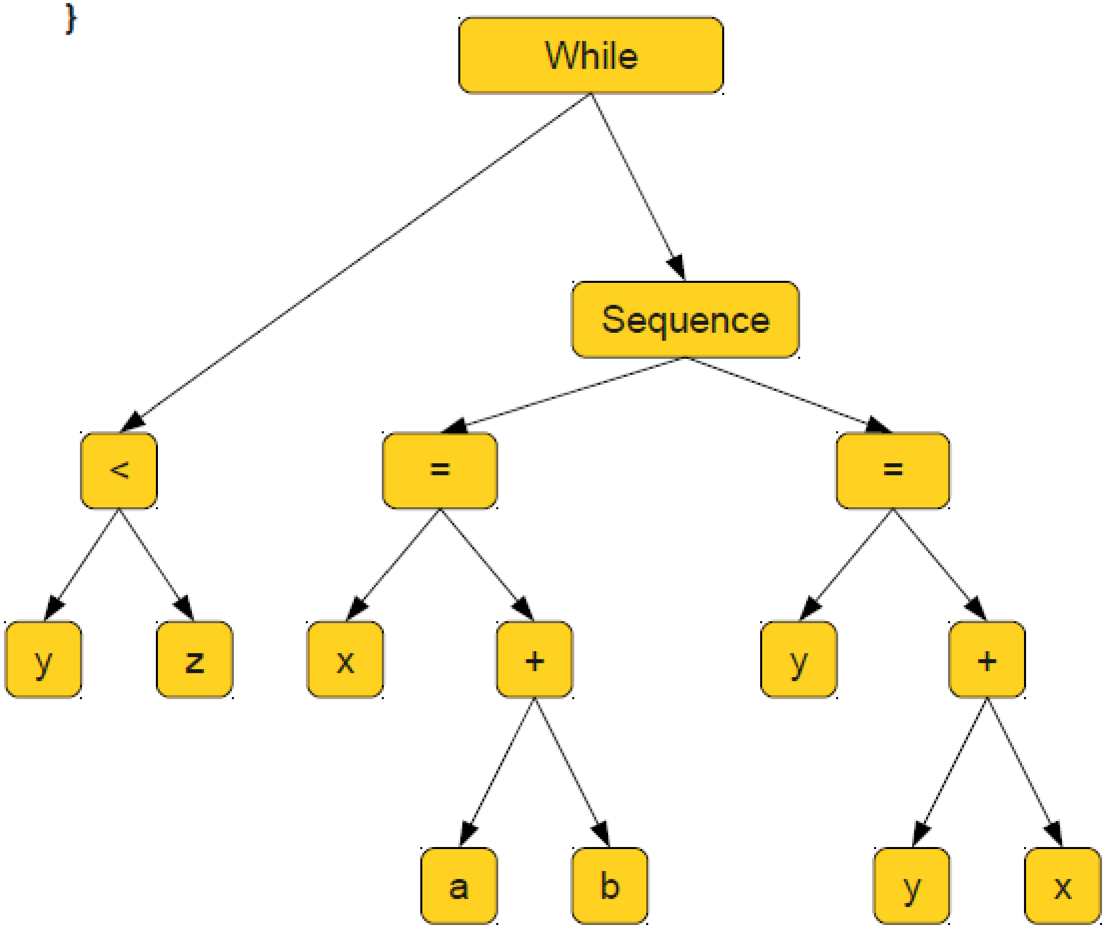
IR Generation

IR Optimization

Code Generation

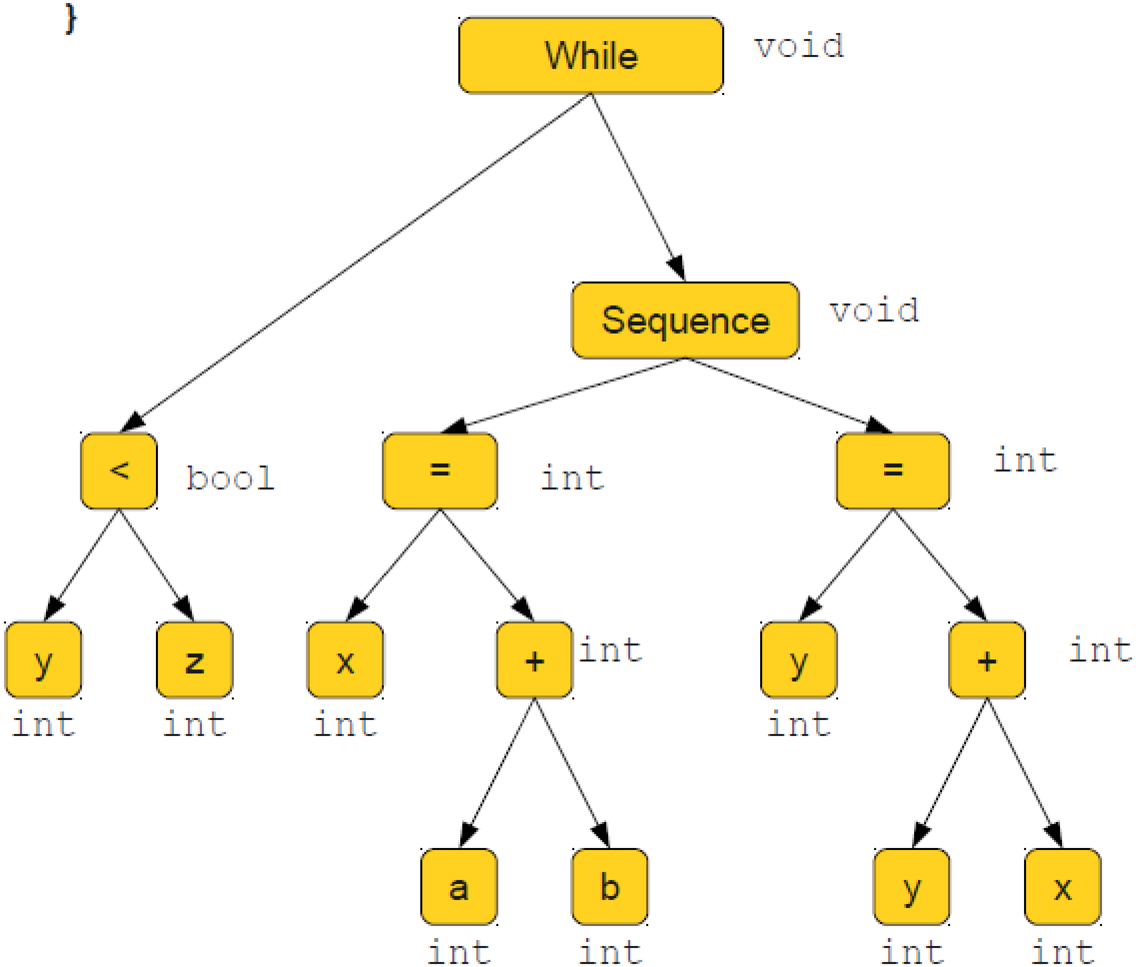
Optimization

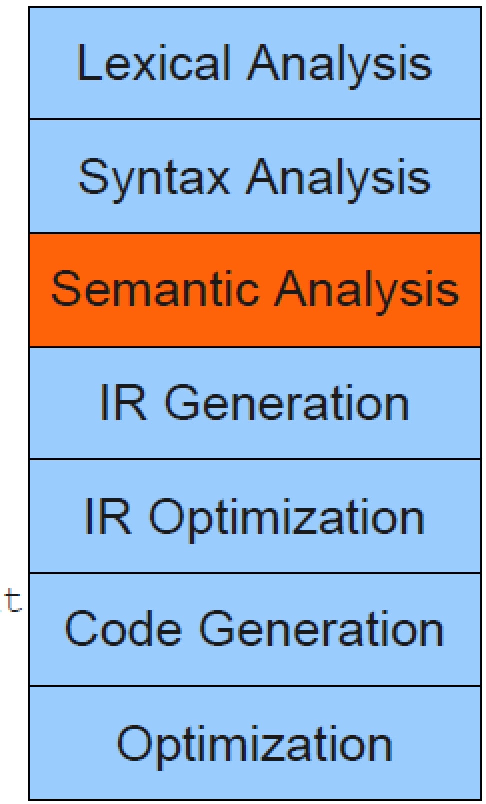
```
while (y < z) {
     int x = a + b;
     y += x;
}
```



- Lexical Analysis
- **Syntax Analysis**
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

```
while (y < z) {
    int x = a + b;
    y += x;
}
```





```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
Loop: x   = a + b
      y   = x + y
      _t1 = y < z
      if _t1 goto Loop
```

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- **IR Generation**
- IR Optimization
- Code Generation
- Optimization

```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
      x   = a + b
Loop: y   = x + y
      _t1 = y < z
      if _t1 goto Loop
```

Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Generation

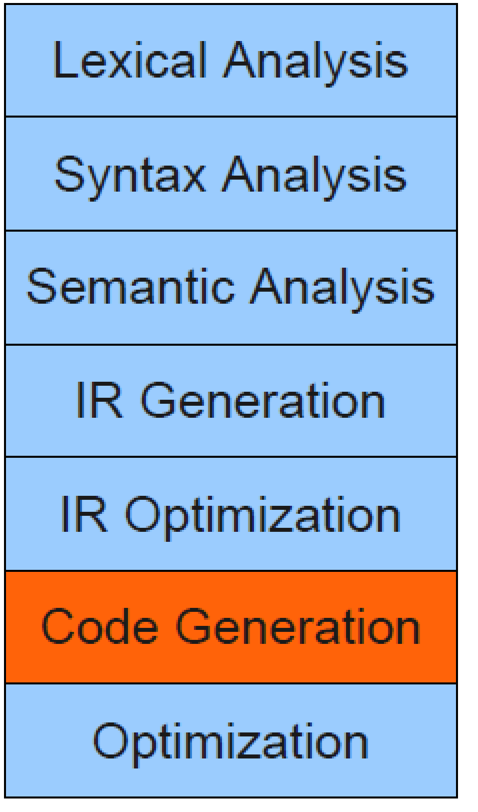
IR Optimization

Code Generation

Optimization

```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
      add $1, $2, $3
Loop: add $4, $1, $4
      slt $6, $4, $5
      beq $6, loop
```

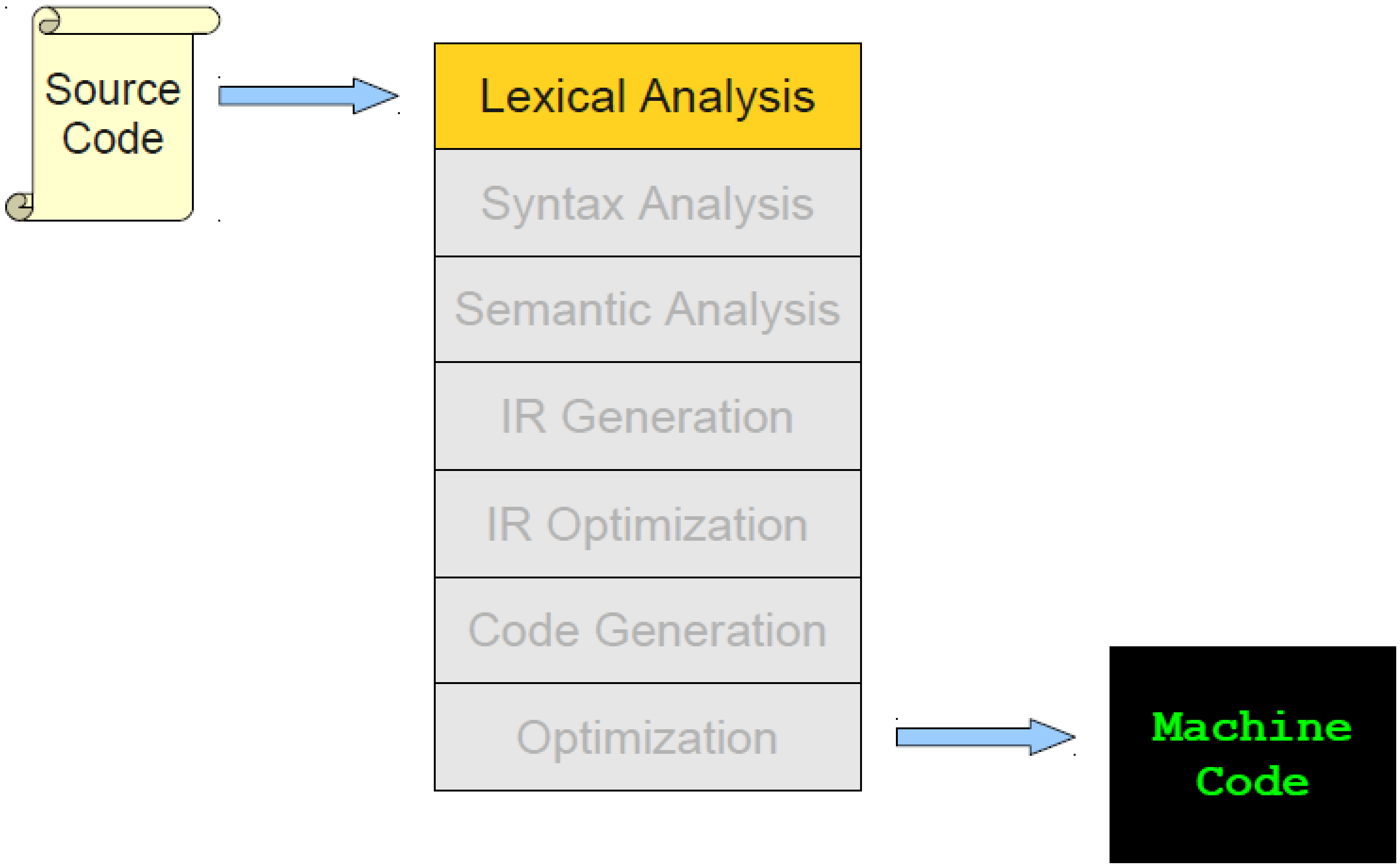


```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
       add $1, $2, $3
Loop:  add $4, $1, $4
       blt $4  $5, loop
```

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- **Optimization**

Next →


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Definitions

- A *lexeme* is the lowest level syntactic unit of a language (e.g. the, boy)
- A *token* is a category of lexemes (e.g., identifier)

```
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	;

Introduction

For all practical purposes, we can think of the source code as a very long string of characters from some alphabet.

Lexical analyzers collects characters into logical groupings (lexemes) and assigns internal codes (tokens) to the grouping.

As an analogy, we can think of the lexical analyzer as a machine that chops up source code into the individual “words”.

What lexical analyzers do

- The “front-end” for the parser
- A pattern matcher for character strings
- Identifies substrings of the source program that belong together => lexemes
  - Example: sum = B –5;

Lexeme	**Token**	
sum	ID	(identifier)
=	ASSIGN_OP	
B	ID	
-	SUBTRACT_OP	
5	INT_LIT	(integer literal)
;	SEMICOLON	

What lexical analyzers do/2

- **Functions:**
  - ⇨ Extract lexemes from a given input string and produce the corresponding tokens, while skipping comments and blanks
  - ⇨ Insert lexemes for user-defined names into symbol table, which is used by later phases of the compiler
  - ⇨ Detect syntactic errors in tokens and report such errors to user

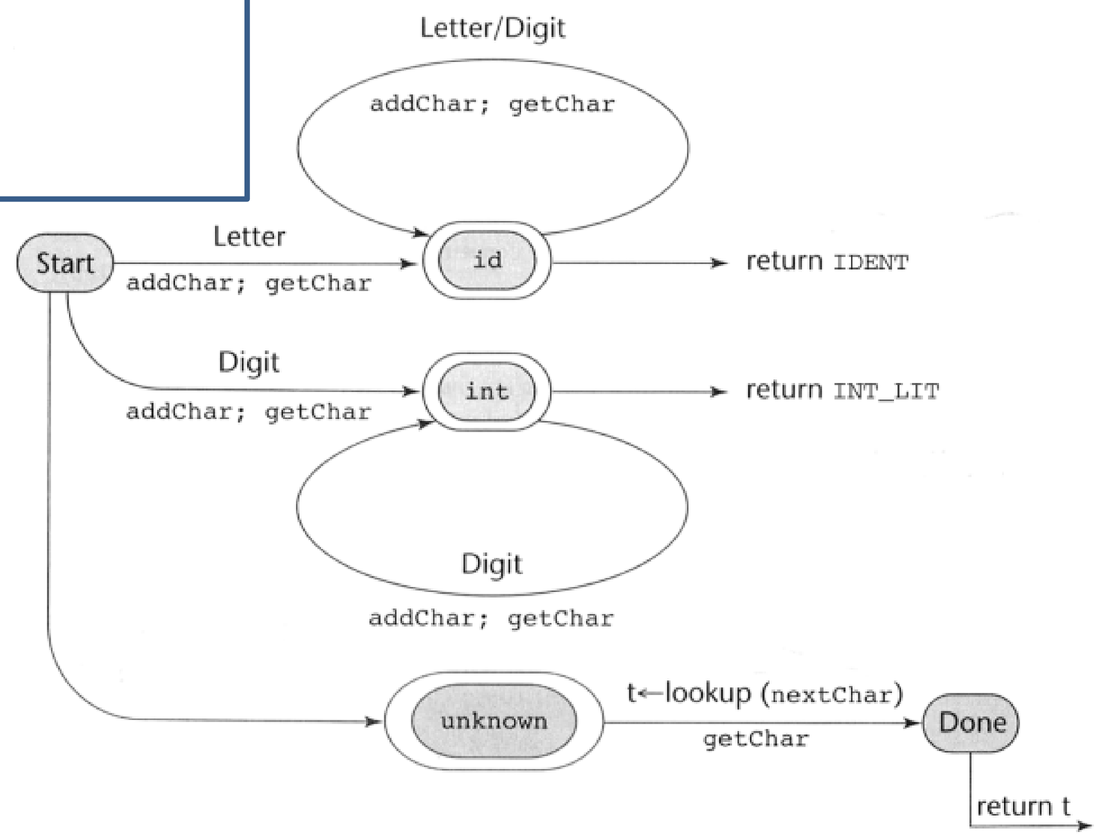
How to build a lexical analyzer?

- Three ways (p175)
- We will take the second approach:
- Formal regular grammar => software package that generates the lexical analyzer.
- **State transition diagrams that describe token types => write a program to implement the diagram**
- State transition diagrams that describe token types => table driven approach to implement the diagram.

State Transition Diagram: Example

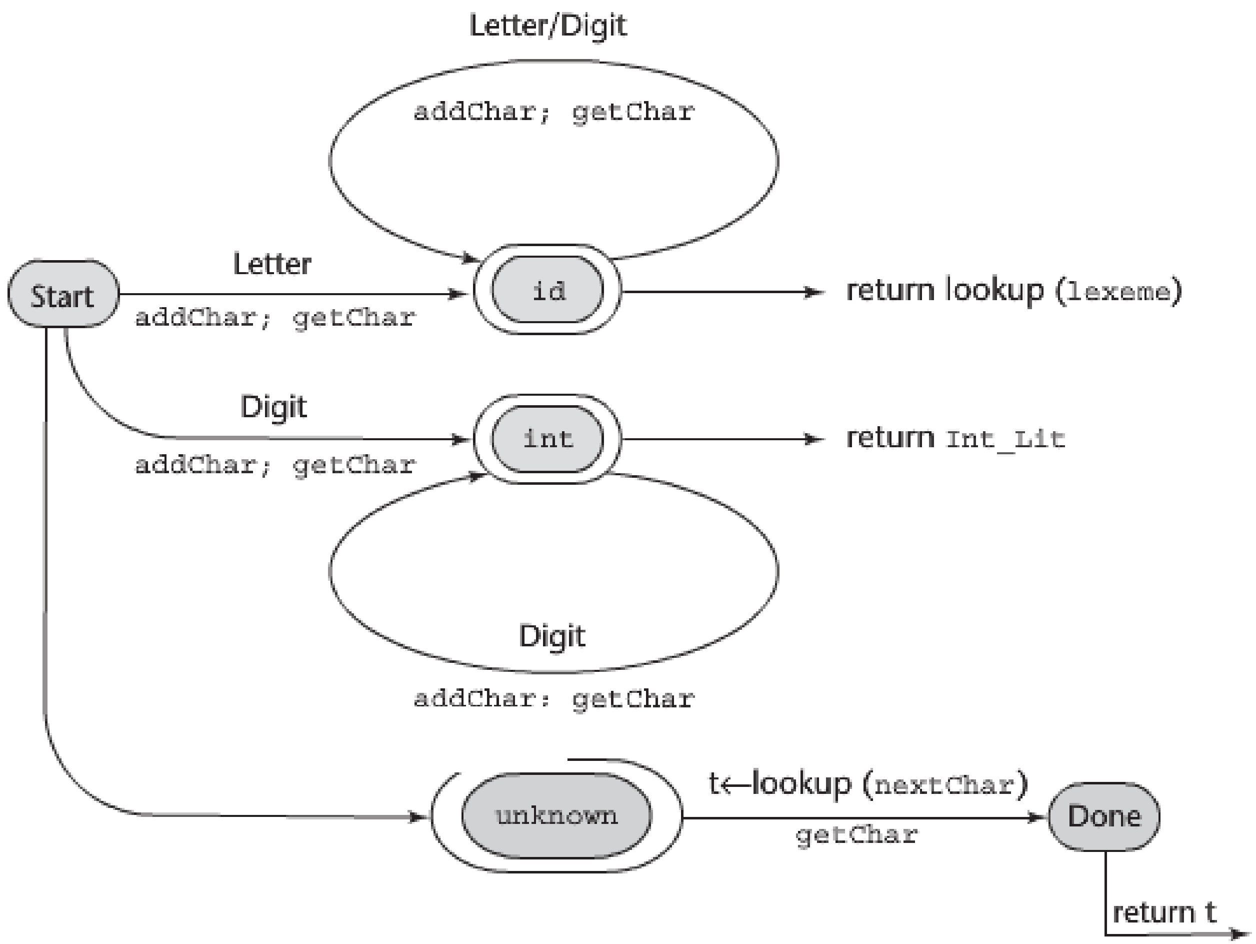
<id> → <letter>{<letter>|<int>} **Tokens in CFG**
<int> → <digit>{<digit>}
<letter> → A | B | C |…| Z | a | b | …| z
<digit> → 0 | 1 | 2 | … | 9
<special>->*|/|+|-|(|)

Id -> letter(letter|digit)*
int->digit digit*
digit->[0-9]
letter->[a-zA-Z] **Tokens in**
special->[*/+-()] **Regular grammar**



Suppose we need a lexer that recognizes only names, integer literals, parentheses, and arithmetic operators.

State Transition Diagram: Example



Suppose we need a lexer that recognizes only names, reserved words, integer literals, parentheses, and arithmetic operators.

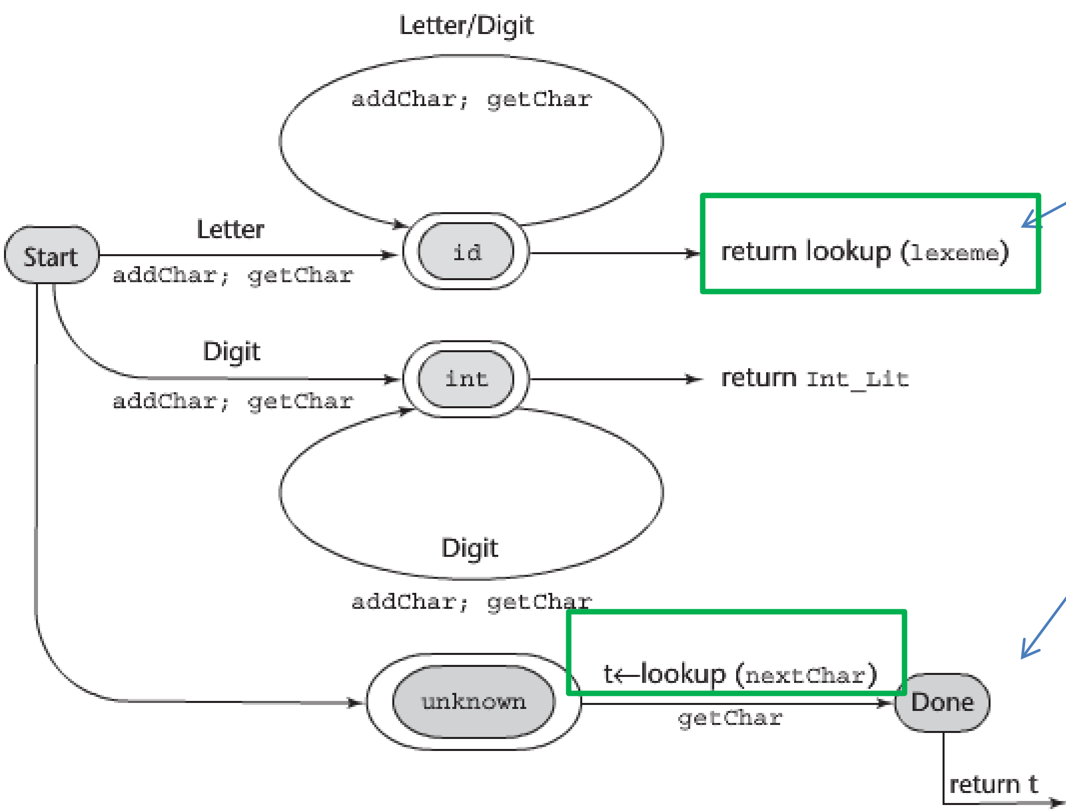
Example/3

Need to distinguish reserved words from identifiers

e.g., reserved words: main and int

identifiers: sum and B

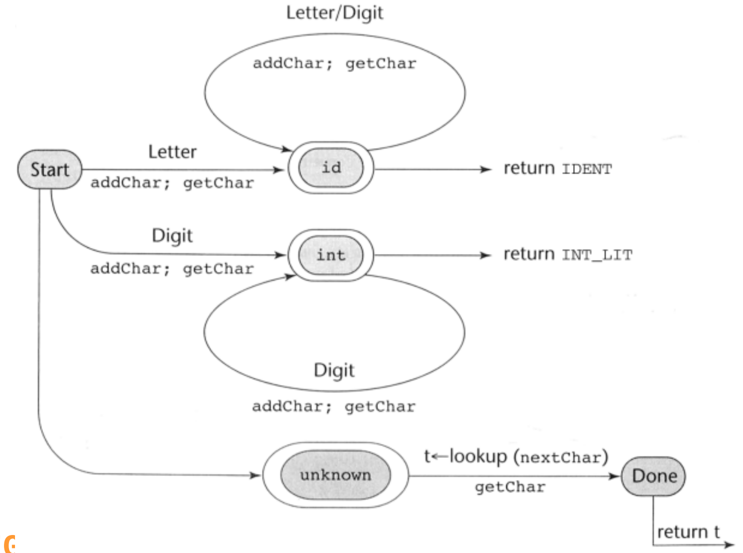
Use a table lookup to determine whether a possible identifier is in fact a reserved word



To determine whether name is a reserved word (not in the book Implementation.)

To determine whether a character is a parenthesis, or arithmetic operator

Example :Implementation/3




```
int lex() {
    lexLen = 0;
     getNonBlank();
     switch (charClass) {
       case LETTER:
            addChar();
            getChar();
            while (charClass == LETTER || charClass == DIG
               addChar();
               getChar(); }
               nextToken = IDENT;
               break;
        case DIGIT:
            addChar();
            getChar();
             while (charClass == DIGIT) {
                addChar();
                getChar(); }
              nextToken = INT_LIT;
              break;
        case UNKNOWN:
            lookup(nextChar);
            getChar();
            break
```

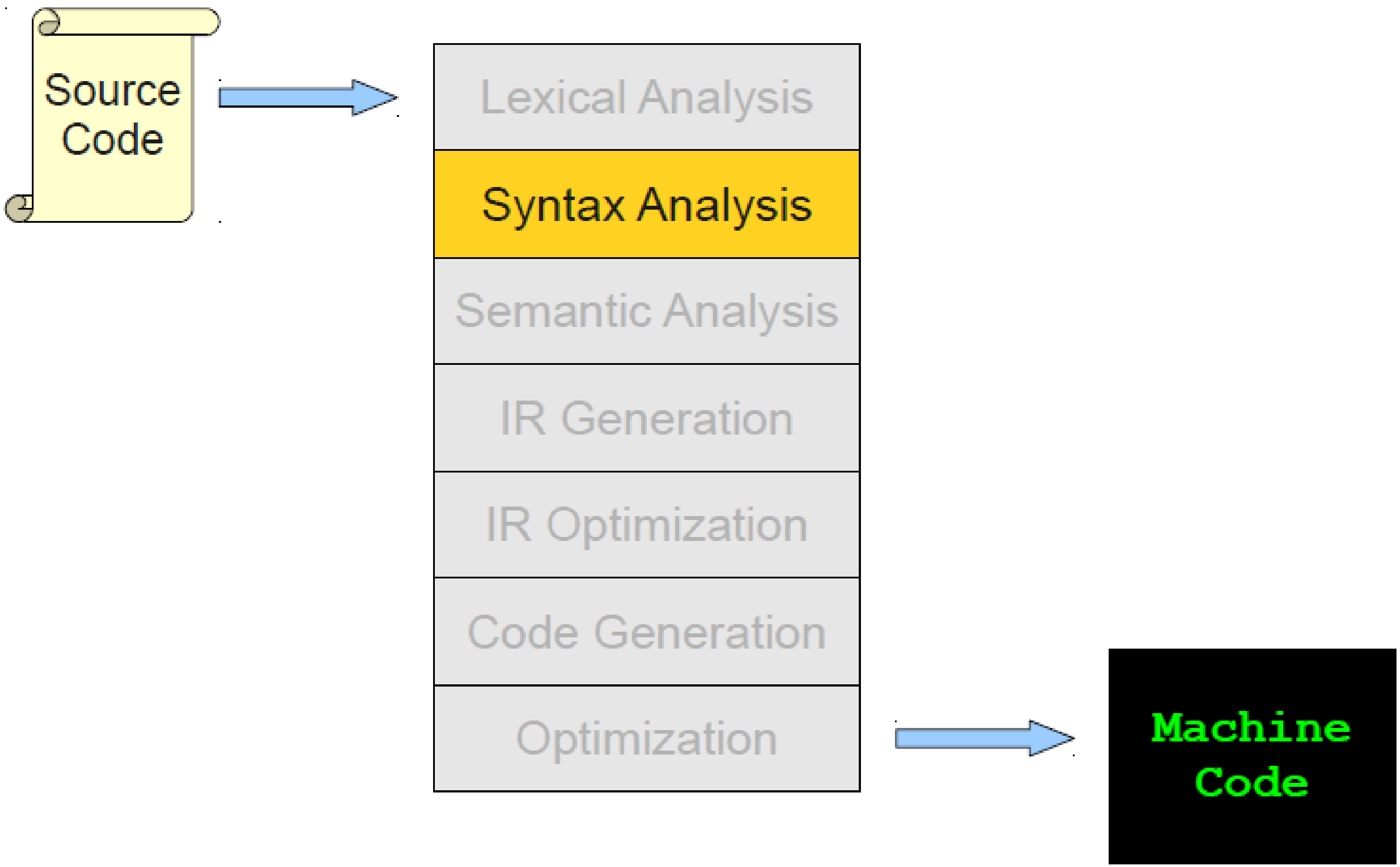
* This does not have logic for checking reserved words

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

```c
void getChar()
{
    nextChar = getc(in_fp);
   // printf("next char %c\t", nextChar);
    if (nextChar != EOF)
    {
        if (isalpha(nextChar))
            charClass = LETTER;

        else if (isdigit(nextChar))
            charClass = NUMBER;

        else
            charClass = UNKNOWN;
    }
    else
        charClass = EOF;
//printf("char class %d\t", charClass);

}
```

```c
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;
```

Where We Are



Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Introduction

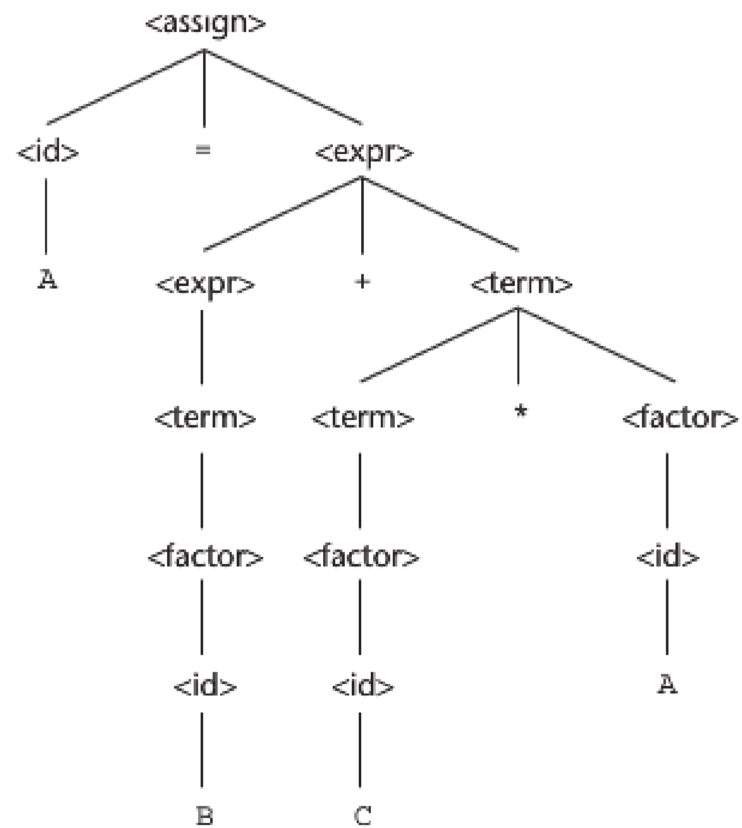
- The syntax analyzer receives the source code in the form of tokens from the lexical analyzer and performs syntax analysis, which determines **the structure** of the program.

- Typically, the result generated by the syntax analyzer is a parse tree.

An illustration

Figure 3.3

The unique parse tree for `A = B + C * A` using an unambiguous grammar



<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

Lexer vs. Parser

Phase	*Input*	*Output*
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree

What do parsers do?

- Goals of a parser:
  - Find all syntax errors
  - Produce parse trees for input program
- Two categories of parsers:
  - **Top down**
    - produces the parse tree, beginning at the root
  - **Bottom up**
    - produces the parse tree, beginning at the leaves

Top-down parsers

- Launch the process with the start symbol and apply the productions until you arrive at the desired string.
  - Builds a parse tree in preorder
  - Corresponds leftmost derivation
  - LL parser

$S \to aAc$

$A \to aA \mid b$

S => aAc => aaAc => aabc

With this grammar

```
S  ->  AB
A  ->  aA | ε
B  ->  b | bB
```

Here is a top-down parse of aaab

S	
⟹ AB	S -> AB
⟹ aAB	A -> aA
⟹ aaAB	A -> aA
⟹ aaaAB	A -> aA
⟹ aaaεB	A -> ε
⟹ aaab	B -> b

Top-down Parsers Again

From deviation aspect

- 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
    - x: all terminals
    - A: the leftmost nonterminal in the sentential form
    - α: combination of terminal and non-terminal

(Predicative) Top-down Parsers

From deviation aspect

- A nonterminal has more than one RHS
- 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

<expr> $\rightarrow$ <term> {('+' | '-') <term>}

A -> bB | cBb | a

(Predicative)Recursive Descent Parser

- A top-down parser implementation
  - Based on directly on BNF/EBNF
- Consists of a collection of subprograms
  - A recursive descent parser has a subprogram for each non-terminal symbol
    - The responsibility of the subprogram: parse sentences that can be generated by that nonterminal

Example

<expr> → <term> {(‘+’ | ’-’) <term>}

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

<factor> → id | ‘(‘<expr>‘)’

```
void expr()
{...}

void term()
{...}

void factor()
{...}

void error()
{...}
```

-- “id” here is not a terminal, but it stands for the identifier token

<factor> → <id> | ‘(‘<expr>‘)’

-- language literals are enclosed with ‘ ‘

-- For each terminal symbol/token 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

Example/2

<expr> → <term> {('+' | '-') <term>}

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

<factor> → id | '('<expr>')'

```
void expr() {
   printf("Enter <expr>\n");
   term();
   while (nextToken ==PLUS_CODE
   ||nextToken == MINUS_CODE)
   {
        lex();
        term();
   }
   printf("Exit <expr>\n");
}
```

- lex() is the lexical analyzer function. It gets the next lexeme and puts its token code in the global variable nextToken
- All subprograms are written with the convention that each one leaves the next token of input in nextToken

Example/3

<expr> → <term> {('+' | '-'') <term>}

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

<factor> → id | '('<expr>')'

```
void term() {
    printf("Enter <term>\n");
   /* parse the first factor */
   factor();
    while (nextToken == MUL_CODE ||
        nextToken == DIV_CODE)
        {
          lex();
         factor();
        }
    printf("Exit <term>\n");
 }
```

<expr> $\rightarrow$ <term> {('+' | '-'') <term>}

<term> $\rightarrow$ <factor> {('*' | '/')<factor>}

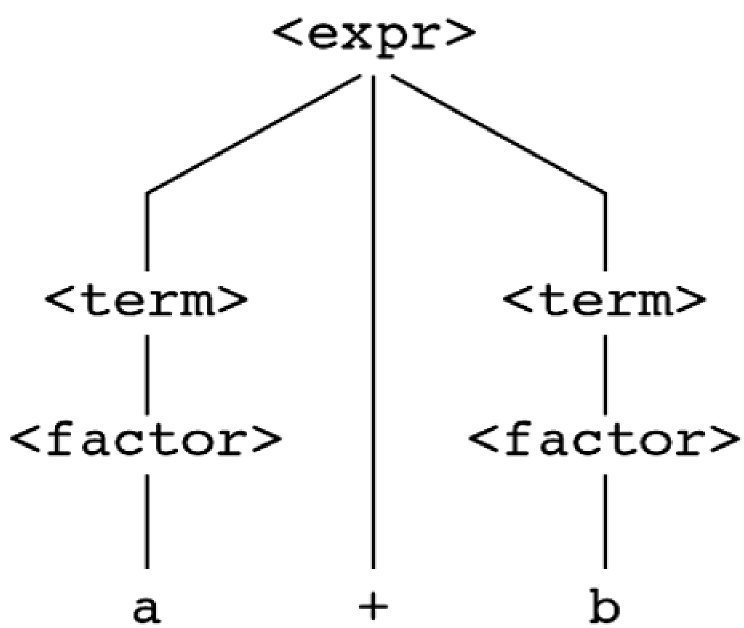
<factor> $\rightarrow$ id | '('<expr>')'

Example/4

```
void factor() {
    printf("Enter <factor>\n");
    /* Determine which RHS */
   if (nextToken == ID_CODE)
               lex();
   else if (nextToken == LEFT_PAREN_CODE){
        lex();
        expr();
        if (nextToken == RIGHT_PAREN_CODE)
               lex();
        else
               error();
   }
   else {
        error(); /* Neither RHS matches */
   }
   printf("Exit <factor>\n");
 }
```

Figure 4.2

Parse tree for a + b



Calling sequences:

Call lex //a
Enter <expr>
Enter <term>
Enter <factor>
Call lex //+
Exit <factor>
Exit <term>
Call lex //b
Enter <term>
Enter <factor>
Call lex // end of input
Exit <factor>
Exit <term>
Exit <expr>

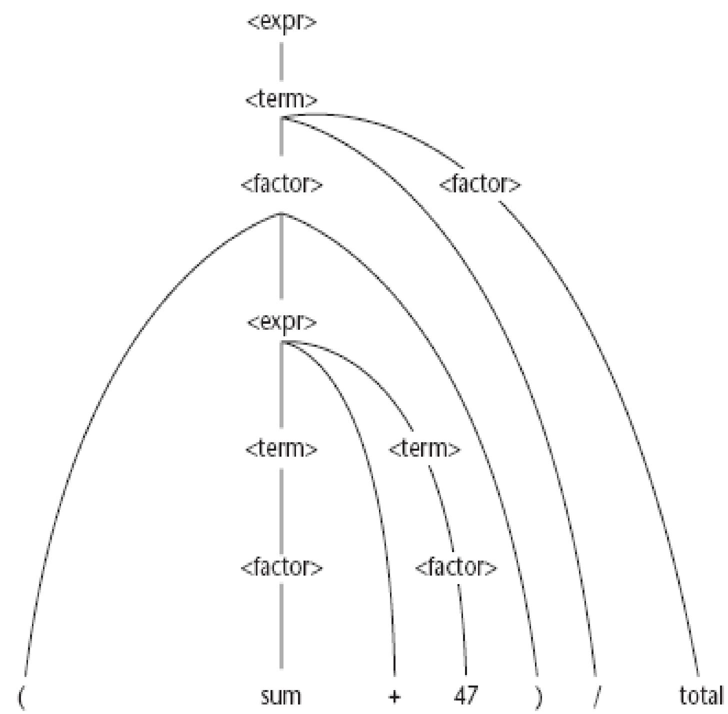
<expr> → <term> {(‘+’ | ’-”) <term>}
<term> → <factor> {(‘*’ | ’/’)<factor>}
<factor> → id | ‘(‘<expr>‘)’

Calling sequences:

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>

Figure 4.2

Parse tree for (sum + 47) / total



<expr> → <term> {(‘+’ | ’-”) <term>}
<term> → <factor> {(‘*’ | ’/’)<factor>}
<factor> → id | ‘(‘<expr>‘)’

Recursive Descent Parser: drawback

- Problem with left recursion
  - <A> $\rightarrow$ <A> + <B> (direct left recursion)
  - <A> $\rightarrow$ <B> c <D> (indirect left recursion)
    <B> $\rightarrow$ <A> b
  - A grammar can be modified to remove left recursion (p181)

- Inability to determine the correct RHS on the basis of one token of lookahead
  - Yes Example: $A \rightarrow aB \mid bAb \mid Bb$
    $B \rightarrow cB \mid d$
  - No Example: $A \rightarrow aC \mid Bd$
    $B \rightarrow ac$
    $C \rightarrow c$
  - Pairwise disjoint test and left factoring
  - EBNF does not have this problem

Bottom-up parsers

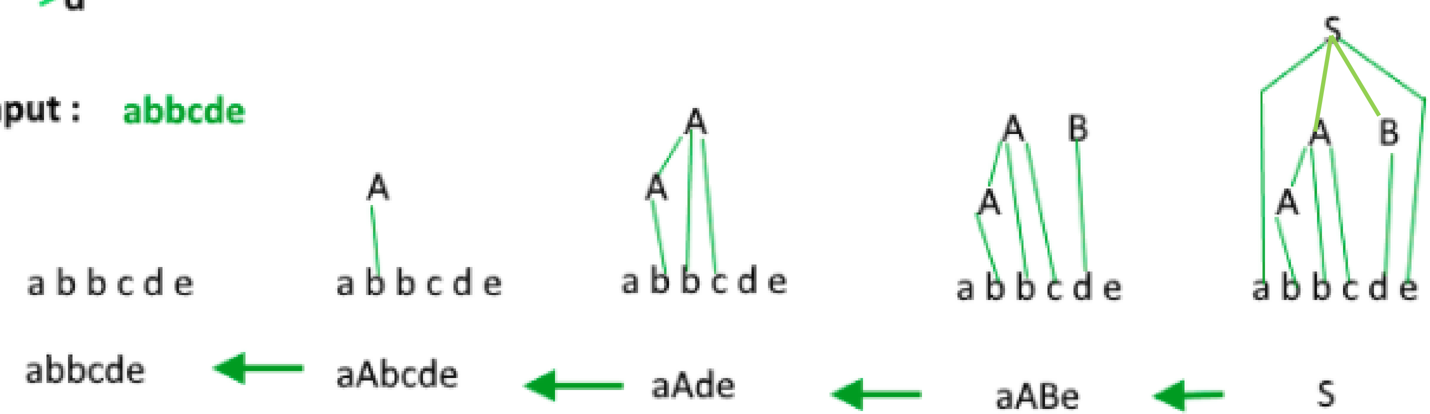
- Begin at the leaves and progressing toward the root.
  - Corresponds to the reverse of rightmost derivation
  - are more powerful than top-down methods
  - LR

S → aABe

A → Abc | b

B → d

Input : abbcde



Bottom-up parsers Again

from the derivation aspect

- Bottom-up parsers
  - Given a right sentential form, α, determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the rightmost derivation
  - The most common bottom-up parsing algorithms are in the LR family

Every string of symbols (terminal and nonterminal) in a derivation is a *sentential form*
A right sentential form is a sentential form appears in a rightmost derivation.

Shift-Reduce Algorithms

– Implementation of bottom-up parsing
  - Uses stack (parse stack)
– Reduce is the action of replacing substring (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

An shift-reduce example

from the book

$$E \to E + T \mid T$$

$$T \to T * F \mid F$$

$$F \to (E) \mid \mathrm{id}$$

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow E + \underline{T * F} \\
&\Rightarrow E + T * \underline{\mathrm{id}} \\
&\Rightarrow E + \underline{F} * \mathrm{id} \\
&\Rightarrow E + \underline{\mathrm{id}} * \mathrm{id} \\
&\Rightarrow \underline{T} + \mathrm{id} * \mathrm{id} \\
&\Rightarrow \underline{F} + \mathrm{id} * \mathrm{id} \\
&\Rightarrow \underline{\mathrm{id}} + \mathrm{id} * \mathrm{id}
\end{aligned}
$$

Corresponds to the reverse of rightmost derivation

An shift-reduce example
from the book

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

. id + id * id (shift)
⇒ Id . + id * id (reduce)
⇒ F. + id * id (reduce)
⇒ T. + id * id (reduce)
⇒ E. + id * id (shift)
⇒ E + . Id * id (shift)
⇒ E + id. * id (reduce)
⇒ E + F. * id (reduce)
⇒ E + T. * id(pick shift over reduce)
⇒ E + T *. Id(shift)
⇒ E + T * F (reduce)
⇒ E + T
⇒ E

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 recursive-descent parser is an LL parser
- Parsing problem for bottom-up parsers: find the substring of current sentential form that can be reduced to some left-hand side.