CS 4300: Compiler Theory

Chapter 2
A Simple Syntax-Directed Translator

Dr. Xuejun Liang

Outline

- This chapter is an introduction to the compiling techniques in Chapters 3 to 6 of the Dragon book
- It illustrates the techniques by developing a working Java program that translates representative programming language statements into three-address code
- The major topics are
    2. Syntax Definition
    3. Syntax-Directed Translation
    4. Parsing
    5. A Translator for Simple Expressions
    6. Lexical Analysis
    7. Symbol Tables
    8. Intermediate Code Generation

5. A Translator for Simple Expressions

Actions for translating into postfix notation

$expr \rightarrow expr + term$	{ print("+") }
$expr \rightarrow expr - term$	{ print("-") }
$expr \rightarrow term$	
$term \rightarrow \mathbf{0}$	{ print("0") }
$term \rightarrow \mathbf{1}$	{ print("1") }
…	…
$term \rightarrow \mathbf{9}$	{ print("9") }

Translation scheme after left recursion elimination

$expr \rightarrow term\ rest$

$rest \rightarrow + term$ { print("+") } $rest$ | $- term$ { print("-") } $rest$ | ε

$term \rightarrow \mathbf{0}$ { print("0") }

$term \rightarrow \mathbf{1}$ { print("1") }

…

$term \rightarrow \mathbf{9}$ { print("9") }

Example Parse Tree

expr → *term rest*
rest → + *term* { print("+") } *rest*
| **-** *term* { print("-") } *rest*
| ε
term → **0** { print("0") }
term → **1** { print("1") }
…
term → **9** { print("9") }

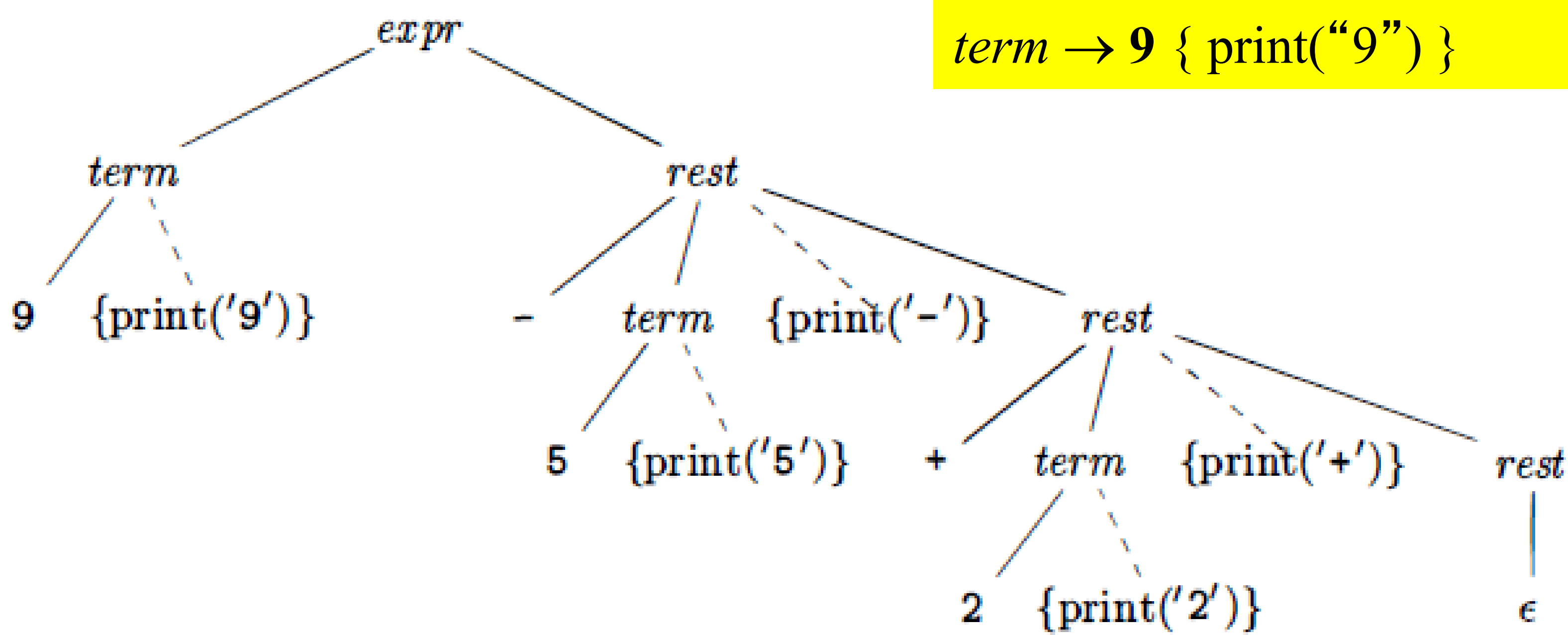


Figure 2.24: Translation of 9-5+2 to 95-2+

Pseudocode for nonterminals *expr*, *rest*, and *term*.

expr → *term rest*
rest → **+** *term* { print("+") } *rest*
  | **-** *term* { print("-") } *rest*
  | ε
term → **0** { print("0") }
term → **1** { print("1") }
…
term → **9** { print("9") }

```
void expr() {
        term(); rest();
}

void rest() {
        if ( lookahead == '+' ) {
                match('+'); term(); print('+'); rest();
        }
        else if ( lookahead == '-' ) {
                match('-'); term(); print('-'); rest();
        }
        else { } /* do nothing with the input */ ;
}

void term() {
        if ( lookahead is a digit ) {
                t = lookahead; match(lookahead); print(t);
        }
        else report("syntax error");
}
```

Java program to translate …

```
import java.io.*;
class Parser {
    static int lookahead;

    public Parser() throws IOException {
        lookahead = System.in.read();
    }

    void expr() throws IOException {
        term();
        while(true) {
            if( lookahead == '+' ) {
                match('+'); term(); System.out.write('+');
            }
            else if( lookahead == '-' ) {
                match('-'); term(); System.out.write('-');
            }
            else return;
        }
    }
```

expr → *term rest*

rest → + *term* { print("+") } *rest*
| - *term* { print("-") } *rest*
| ε

… infix expressions into postfix form

```
    void term() throws IOException {
        if( Character.isDigit((char)lookahead) ) {
            System.out.write((char)lookahead); match(lookahead);
        }
        else throw new Error("syntax error");
    }

    void match(int t) throws IOException {
        if( lookahead == t ) lookahead = System.in.read();
        else throw new Error("syntax error");
    }
}

public class Postfix {
    public static void main(String[] args) throws IOException {
        Parser parse = new Parser();
        parse.expr(); System.out.write('\n');
    }
}
```

C++ program

$$expr \rightarrow term\ rest$$

$$\begin{aligned} rest &\rightarrow +\ term\ \{\ \text{print}(\text{“+”})\ \}\ rest \\ &\mid -\ term\ \{\ \text{print}(\text{“-”})\ \}\ rest \\ &\mid \varepsilon \end{aligned}$$

$$term \rightarrow \mathbf{0}\ \{\ \text{print}(\text{“0”})\ \}$$

$$term \rightarrow \mathbf{1}\ \{\ \text{print}(\text{“1”})\ \}$$

…

$$term \rightarrow \mathbf{9}\ \{\ \text{print}(\text{“9”})\ \}$$

```
main()
{   lookahead = getchar();
    expr();
}
expr()
{   term();
    while (1) /* optimized by inlining rest()
                 and removing recursive calls */
    {   if (lookahead == ‘+’)
        {   match(‘+’); term(); putchar(‘+’);
        }
        else if (lookahead == ‘-’)
        {   match(‘-’); term(); putchar(‘-’);
        }
        else break;
    }
}
term()
{   if (isdigit(lookahead))
    {   putchar(lookahead); match(lookahead);
    }
    else error();
}
match(int t)
{   if (lookahead == t)
        lookahead = getchar();
    else error();
}
error()
{   printf(“Syntax error\n”);
    exit(1);
}
```

6. Lexical Analysis

- The expression only deals with single digit integer and no white space is allowed. So, no lexical analysis is needed.
- Expend to multiple digit integer and to include identifiers

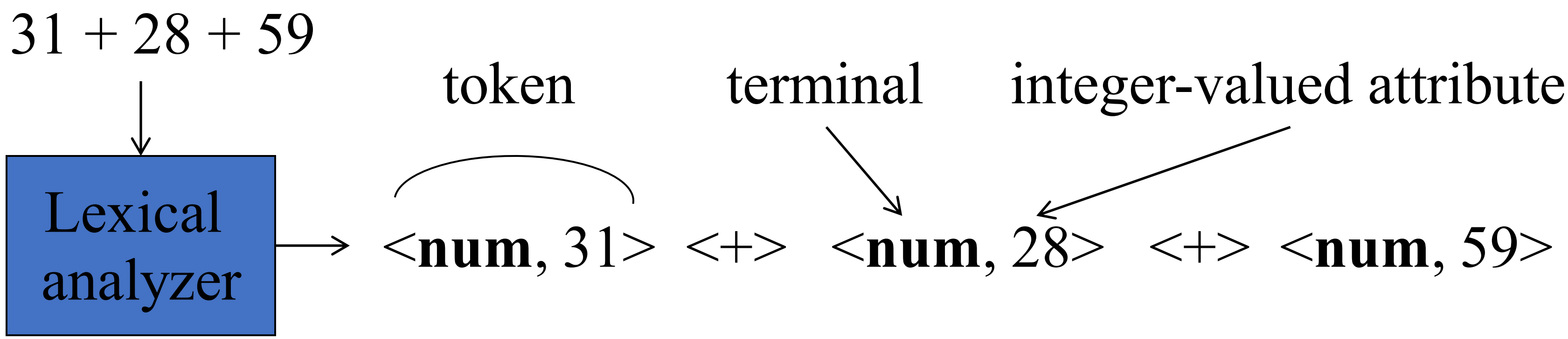
expr	$\rightarrow$	*expr* + *term*	{ print('+') }
	|	*expr* - *term*	{ print('-') }
	|	*term*	
term	$\rightarrow$	*term* * *factor*	{ print('*') }
	|	*term* / *factor*	{ print('/') }
	|	*factor*	
factor	$\rightarrow$	(*expr*)	
	|	**num**	{ print(**num**.*value*) }
	|	**id**	{ print(**id**.*lexeme*) }

Figure 2.28: Actions for translating into postfix notation

Lexical Analyzer

- To expend to multiple digit integer and to include identifiers, a lexical analyzer is needed.
- Typical tasks of the lexical analyzer:
  - Remove white space and comments
  - Encode constants as tokens
  - Recognize keywords
  - Recognize identifiers and store identifier names in a global symbol table

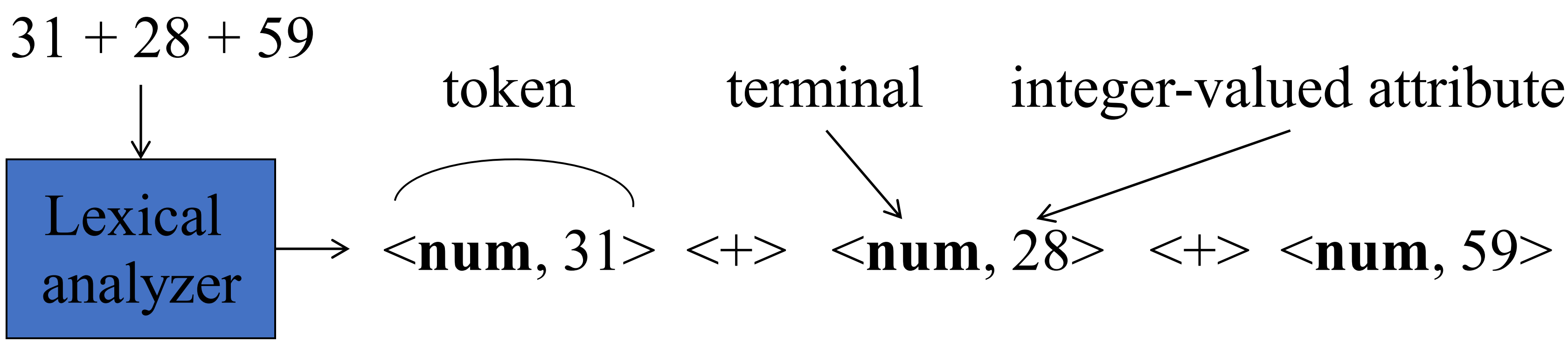
Constants (Number)




```
if ( peek holds a digit ) {
        v = 0;
        do {
                v = v * 10 + integer value of digit peek;
                peek = next input character;
        } while ( peek holds a digit );
        return token ⟨num, v⟩;
}
```

Grouping digits into integers

Constants (Number)




```
if ( peek holds a digit ) {
        v = 0;
        do {
                v = v * 10 + integer value of digit peek;
                peek = next input character;
        } while ( peek holds a digit );
        return token ⟨num, v⟩;
}
```

324

$0 \times 10 + 3 = 3$

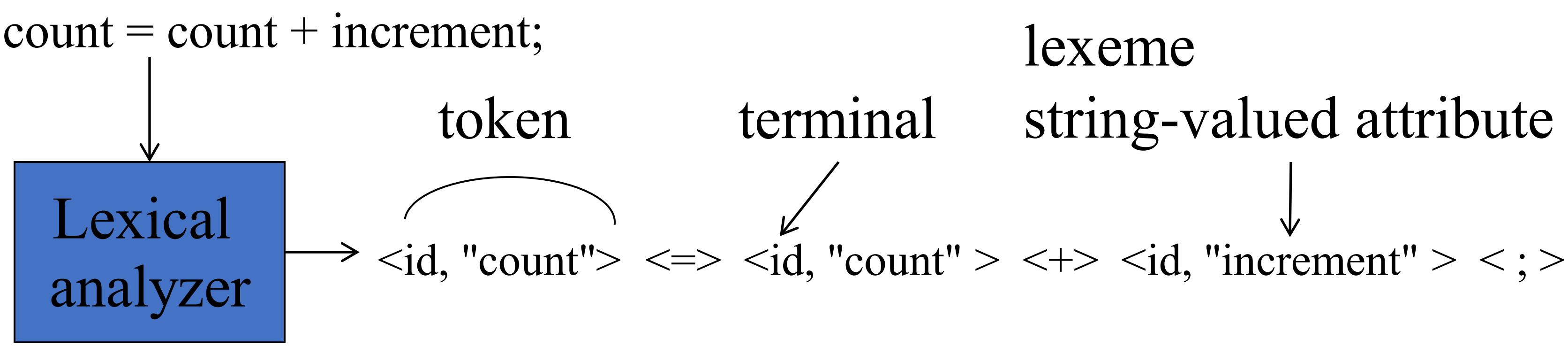
$3 \times 10 + 2 = 32$

$32 \times 10 + 4$

$= 324$

Grouping digits into integers

Keywords and Identifiers



To distinguish keywords from identifiers, use a **string table**.

(key, value)

(lexeme, token)

```
if ( peek holds a letter ) {
    collect letters or digits into a buffer b;
    s = string formed from the characters in b;
    w = token returned by words.get(s);
    if ( w is not null ) return w;
    else {
        Enter the key-value pair (s, ⟨id, s⟩) into words
        return token ⟨id, s⟩;
    }
}
```

Hashtable words = new Hashtable();

A Lexical Analyzer

pseudocode

```
Token scan () {
    skip white space;
    handle numbers;
    handle reserved words and identifiers;
    / * treat read-ahead character peek as a token * /
    Token t = new Token (peek) ;
    peek = blank /* initialization*/ ;
    return t;
}
```

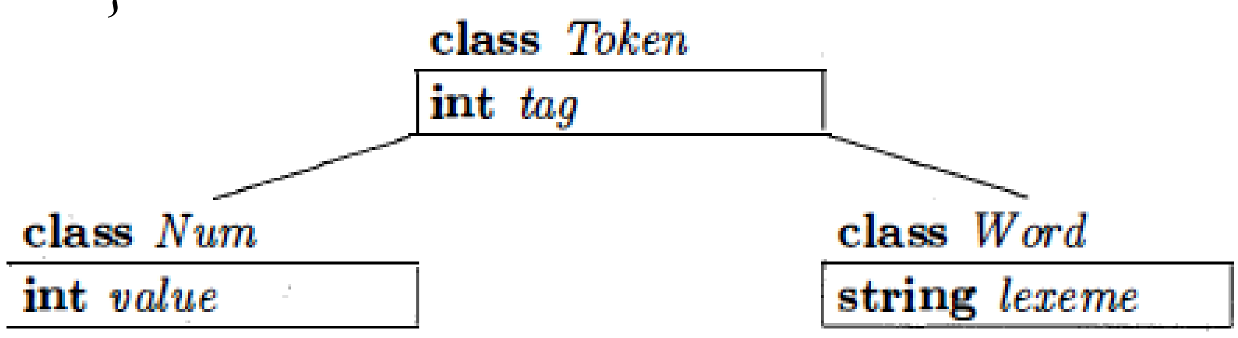


Figure 2.32: Class *Token* and subclasses *Num* and *Word*

Classes Token and Tag

```
package lexer;                      // File Token.java
public class Token {
    public final int tag;
    public Token(int t) { tag = t; }
}
```

```
package lexer;                      // File Tag.java
public class Tag {
    public final static int
        NUM = 256, ID = 257, TRUE = 258, FALSE = 259;
}
```

In C++, constant is defined as below

#define NUM 256

Subclasses Num and Word

```
package lexer;                         // File Num.java
public class Num extends Token {
    public final int value;
    public Num(int v) { super(Tag.NUM); value = v; }
}
```

```
package lexer;                         // File Word.java
public class Word extends Token {
    public final String lexeme;
    public Word(int t, String s) {
        super(t); lexeme = new String(s);
    }
}
```

Code for a lexical analyzer: Part 1 / 3

```
package lexer;                          // File Lexer.java
import java.io.*; import java.util.*;
public class Lexer {
    public int line = 1;
    private char peek = ' ';
    private Hashtable words = new Hashtable();
    void reserve(Word t) { words.put(t.lexeme, t); }
    public Lexer() {
        reserve( new Word(Tag.TRUE,  "true")  );
        reserve( new Word(Tag.FALSE, "false") );
    }
```

Code for a lexical analyzer: Part 2 / 3

```
    public Token scan() throws IOException {
        for( ; ; peek = (char)System.in.read() ) {
            if( peek == ' ' || peek == '\t' ) continue;
            else if( peek == '\n' ) line = line + 1;
            else break;
        }
        if( Character.isDigit(peek) ) {
            int v = 0;
            do {
                v = 10*v + Character.digit(peek, 10);
                peek = (char)System.in.read();
            } while( Character.isDigit(peek) );
            return new Num(v);
        }
```

Code for a lexical analyzer: Part 3 / 3

```
        if( Character.isLetter(peek) ) {
            StringBuffer b = new StringBuffer();
            do {
                b.append(peek);
                peek = (char)System.in.read();
            } while( Character.isLetterOrDigit(peek) );
            String s = b.toString();
            Word w = (Word)words.get(s);
            if( w != null ) return w;
            w = new Word(Tag.ID, s);
            words.put(s, w);
            return w;
        }
        Token t = new Token(peek);
        peek = ' ';
        return t;
    }
}
```