# CS 4300: Compiler Theory

# Chapter 2
# A Simple Syntax- Directed Translator

*Xuejun Liang*

*2019 Fall*

## 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

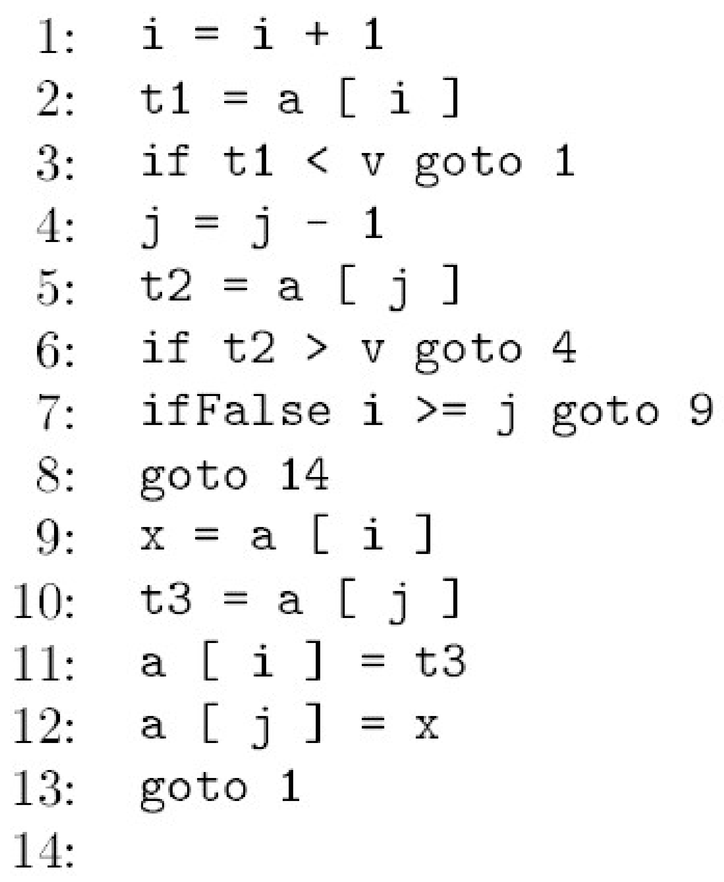
## An Example Source Code

```
{
    int i; int j; float[100] a; float v; float x;

    while ( true ) {
        do i = i+1; while ( a[i] < v );
        do j = j-1; while ( a[j] > v );
        if ( i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
}
```

Figure 2.1: A code fragment to be translated

# The Generated Intermediate Code



```
 1:  i = i + 1
 2:  t1 = a [ i ]
 3:  if t1 < v goto 1
 4:  j = j - 1
 5:  t2 = a [ j ]
 6:  if t2 > v goto 4
 7:  ifFalse i >= j goto 9
 8:  goto 14
 9:  x = a [ i ]
10:  t3 = a [ j ]
11:  a [ i ] = t3
12:  a [ j ] = x
13:  goto 1
14:
```

Figure 2.2: Simplified intermediate code for the program fragment in Fig. 2.1

# Compiler Front End

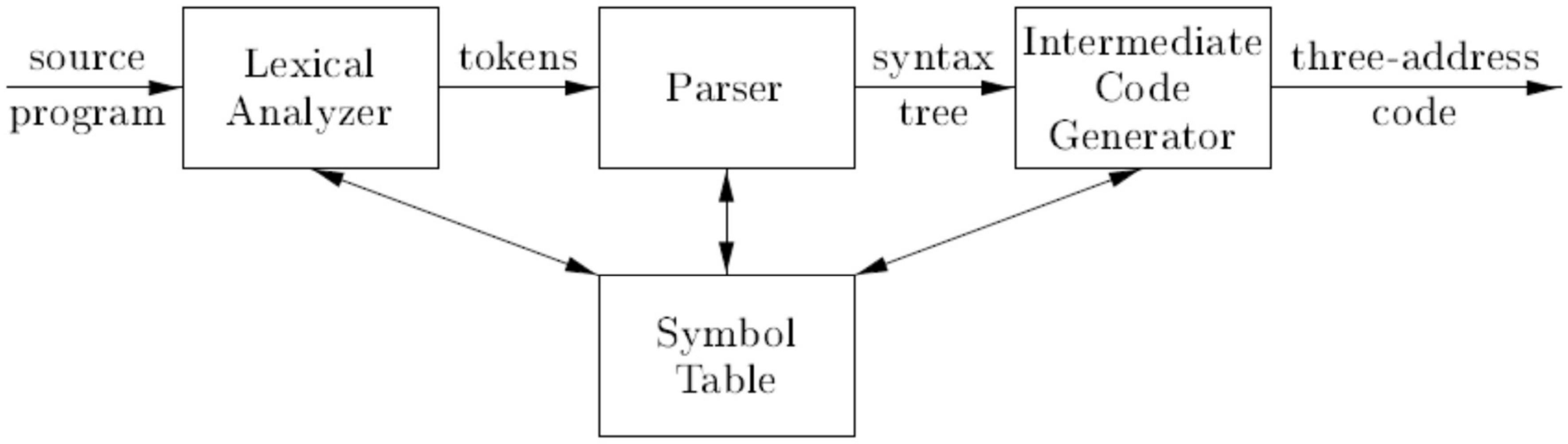


Figure 2.3: A model of a compiler front end

- For simplicity, the parser will use the syntax-directed translation of infix expressions to postfix form.
- For example, the postfix form of the expression 9-5+2 is 95-2+

## 2. Syntax Definition

An **if-else** statement in Java can have the form

if ( expression ) statement else statement

This structuring rule can be expressed as

*stmt* → **if** ( *expr* ) *stmt* **else** *stmt*

The rule called production, left side called head, and right side called body

- Context-free grammar is a 4-tuple with
  - A set of tokens (*terminal* symbols)
  - A set of *nonterminals*
  - A set of *productions*
  - A designated *start symbol*

## Example Grammar

Context-free grammar for simple expressions:

$G = <\{list, digit\}, \{+,-,\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5},\mathbf{6},\mathbf{7},\mathbf{8},\mathbf{9}\}, P, list>$

with productions $P =$

$list \to list + digit$

$list \to list - digit$

$list \to digit$

$digit \to \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$

## Derivation and Parsing

- A grammar derives strings (called derivation) by
  - beginning with the start symbol and repeatedly
  - replacing a nonterminal by the body of a production for that nonterminal.
- The terminal strings that can be derived from the start symbol form the language defined by the grammar
- Parsing is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar, and if it cannot be derived from the start symbol of the grammar, then reporting syntax errors within the string.

# Derivation Example

$\underline{\textit{list}}$
$\Rightarrow \underline{\textit{list}} + \textit{digit}$
$\Rightarrow \underline{\textit{list}}\ \textbf{-}\ \textit{digit} + \textit{digit}$
$\Rightarrow \underline{\textit{digit}}\ \textbf{-}\ \textit{digit} + \textit{digit}$
$\Rightarrow \textbf{9 - }\underline{\textit{digit}} + \textit{digit}$
$\Rightarrow \textbf{9 - 5} + \underline{\textit{digit}}$
$\Rightarrow \textbf{9 - 5 + 2}$

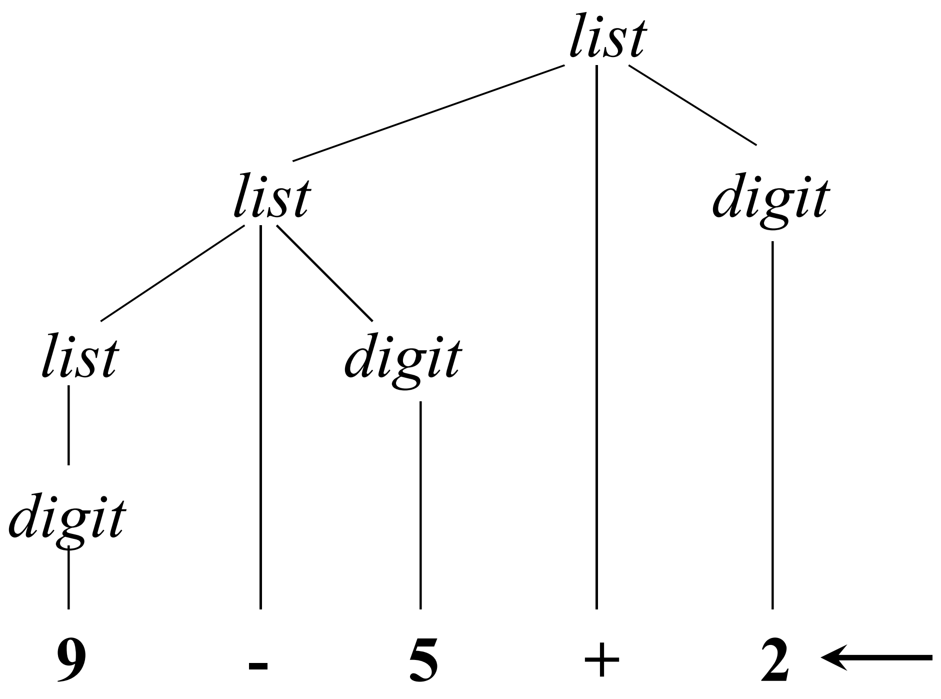
- This is an example *leftmost derivation*, because we replaced the leftmost nonterminal (underlined) in each step.
- Likewise, a *rightmost derivation* replaces the rightmost nonterminal in each step

## Parse Trees

- The *root* of the tree is labeled by the start symbol
- Each *leaf* of the tree is labeled by a terminal (=token) or ε
- Each *interior node* is labeled by a nonterminal
- If $A \rightarrow X_1\, X_2\, \ldots\, X_n$ is a production, then node $A$ has immediate *children* $X_1, X_2, \ldots, X_n$ where $X_i$ is a (non)terminal or ε (ε denotes the *empty string*)

## Parse Tree Example

Parse tree of the string **9-5+2** using grammar *G*



The sequence of leafs is called the *yield* of the parse tree

## Ambiguity

A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be ambiguous

Consider the following context-free grammar:

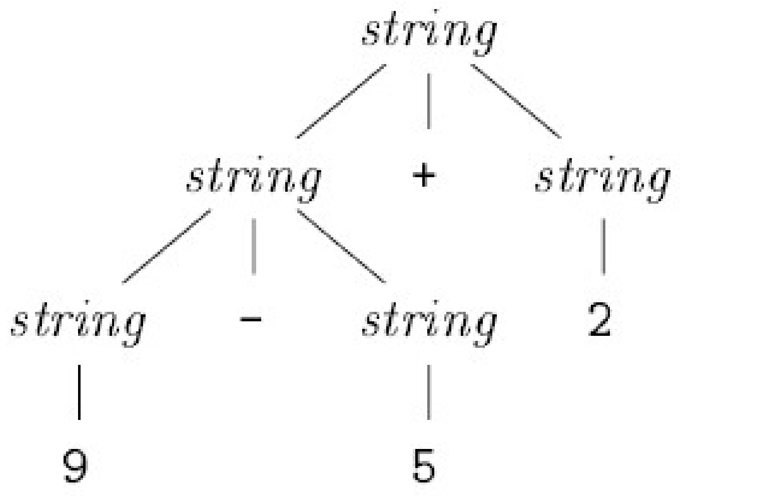
$$G = <\{string\},\ \{\mathbf{+,-,0,1,2,3,4,5,6,7,8,9}\},\ P,\ string>$$

with production $P$ =

$$string \to string + string \mid string \mathbf{-} string \mid \mathbf{0} \mid \mathbf{1} \mid \ldots \mid \mathbf{9}$$

This grammar is *ambiguous*, because more than one parse tree represents the string **9-5+2**

# Two parse trees for 9-5+2



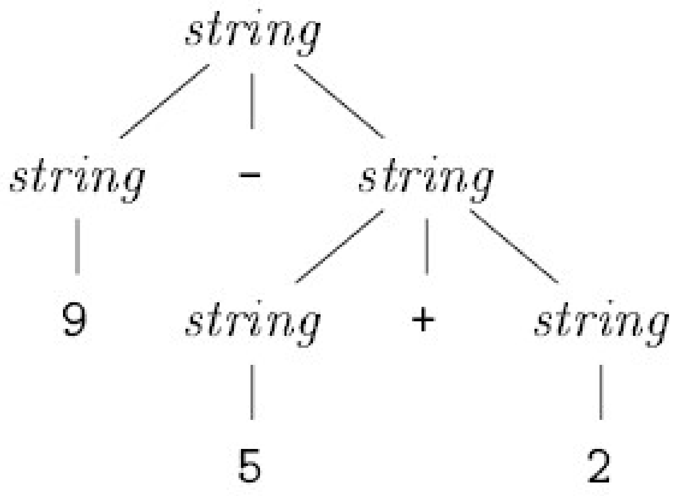


Figure 2.6: Two parse trees for 9-5+2

## Associativity of Operators

*Left-associative* operators have *left-recursive* productions

$$left \rightarrow left + digit \mid digit$$

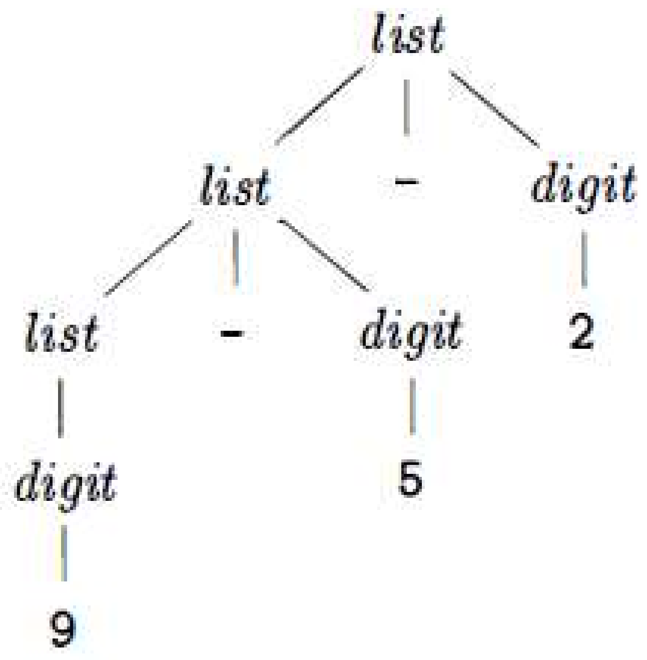
String **9+5+2** has the same meaning as **(9+5)+2**

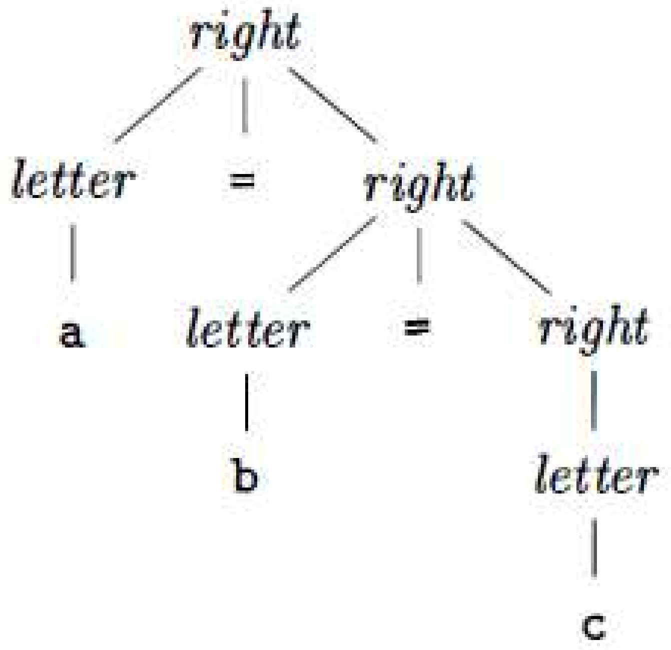
*Right-associative* operators have *right-recursive* productions

$$right \rightarrow letter = right \mid letter$$

String **a=b=c** has the same meaning as **a=(b=c)**

## Parse trees for left- and right-associative grammars



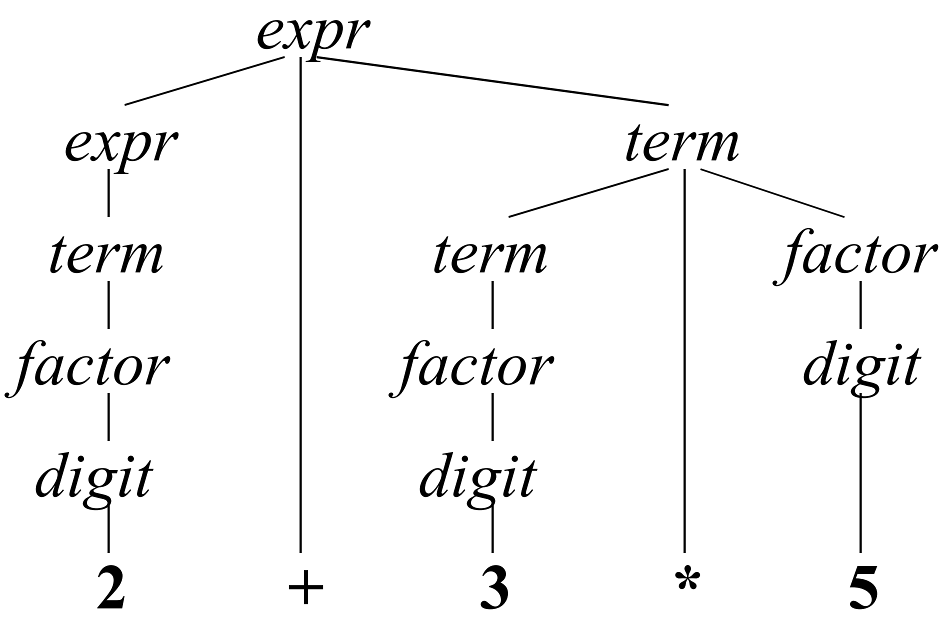


# Precedence of Operators

Operators with higher precedence "bind more tightly"

$$expr \rightarrow expr + term \mid term$$
$$term \rightarrow term * factor \mid factor$$
$$factor \rightarrow digit \mid \mathbf{(}\ expr\ \mathbf{)}$$

String **2+3*5** has the same meaning as **2+(3*5)**



# Syntax (Grammar)

Expressions

$$\begin{array}{rcl}
expr & \rightarrow & expr + term \mid expr - term \mid term \\
term & \rightarrow & term * factor \mid term \,/\, factor \mid factor \\
factor & \rightarrow & \textbf{digit} \mid (\ expr\ )
\end{array}$$

Subset of Java Statements

$$\begin{array}{rcl}
stmt & \rightarrow & \textbf{id} = expression\ ; \\
 & \mid & \textbf{if}\ (\ expression\ )\ stmt \\
 & \mid & \textbf{if}\ (\ expression\ )\ stmt\ \textbf{else}\ stmt \\
 & \mid & \textbf{while}\ (\ expression\ )\ stmt \\
 & \mid & \textbf{do}\ stmt\ \textbf{while}\ (\ expression\ )\ ; \\
 & \mid & \{\ stmts\ \}
\end{array}$$

$$\begin{array}{rcl}
stmts & \rightarrow & stmts\ stmt \\
 & \mid & \epsilon
\end{array}$$

## 3. Syntax-Directed Translation

- Uses a CF grammar to specify the syntactic structure of the language
- AND associates a set of *attributes* with the terminals and nonterminals of the grammar
- AND associates with each production a set of *semantic rules* to compute values of attributes
- A parse tree is traversed and semantic rules applied: after the tree traversal(s) are completed, the attribute values on the nonterminals contain the translated form of the input

## Synthesized and Inherited Attributes

- An attribute is said to be …
  - *synthesized* if its value at a parse-tree node is determined from the attribute values at the children of the node
  - *inherited* if its value at a parse-tree node is determined by the parent (by enforcing the parent’s semantic rules)

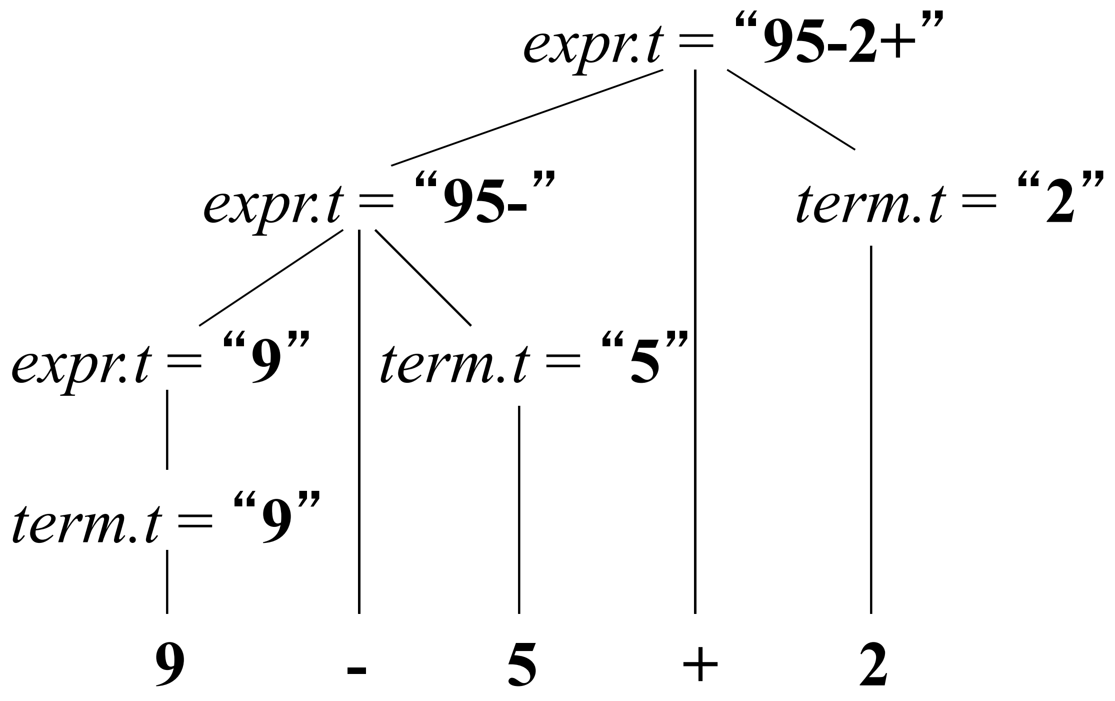
# Example Attribute Grammar

String concat operator

| Production | Semantic Rule |
|---|---|
| $expr \rightarrow expr_1 + term$ | $expr.t := expr_1.t \;\|\|\; term.t \;\|\|\;$ "+" |
| $expr \rightarrow expr_1 - term$ | $expr.t := expr_1.t \;\|\|\; term.t \;\|\|\;$ "-" |
| $expr \rightarrow term$ | $expr.t := term.t$ |
| $term \rightarrow \mathbf{0}$ | $term.t :=$ "0" |
| $term \rightarrow \mathbf{1}$ | $term.t :=$ "1" |
| … | … |
| $term \rightarrow \mathbf{9}$ | $term.t :=$ "9" |

Syntax-directed definition for infix to postfix translation

# Example Annotated Parse Tree



Attribute values at nodes in a parse tree

# Depth-First Traversals

```
**procedure** *visit*(*n* : *node*);
**begin**
    **for** each child *c* of *n*, from left to right **do**
        *visit*(*c*);
    evaluate semantic rules at node *n*
**end**
```

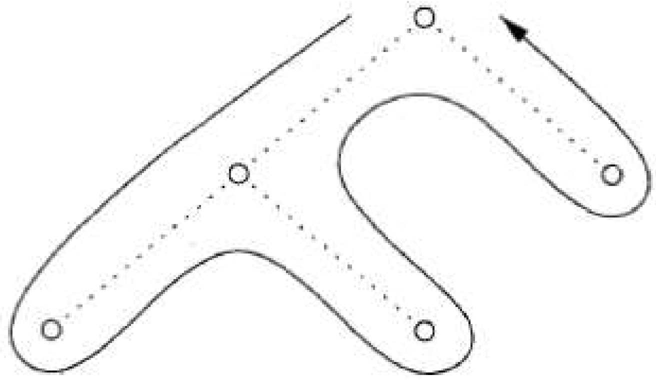
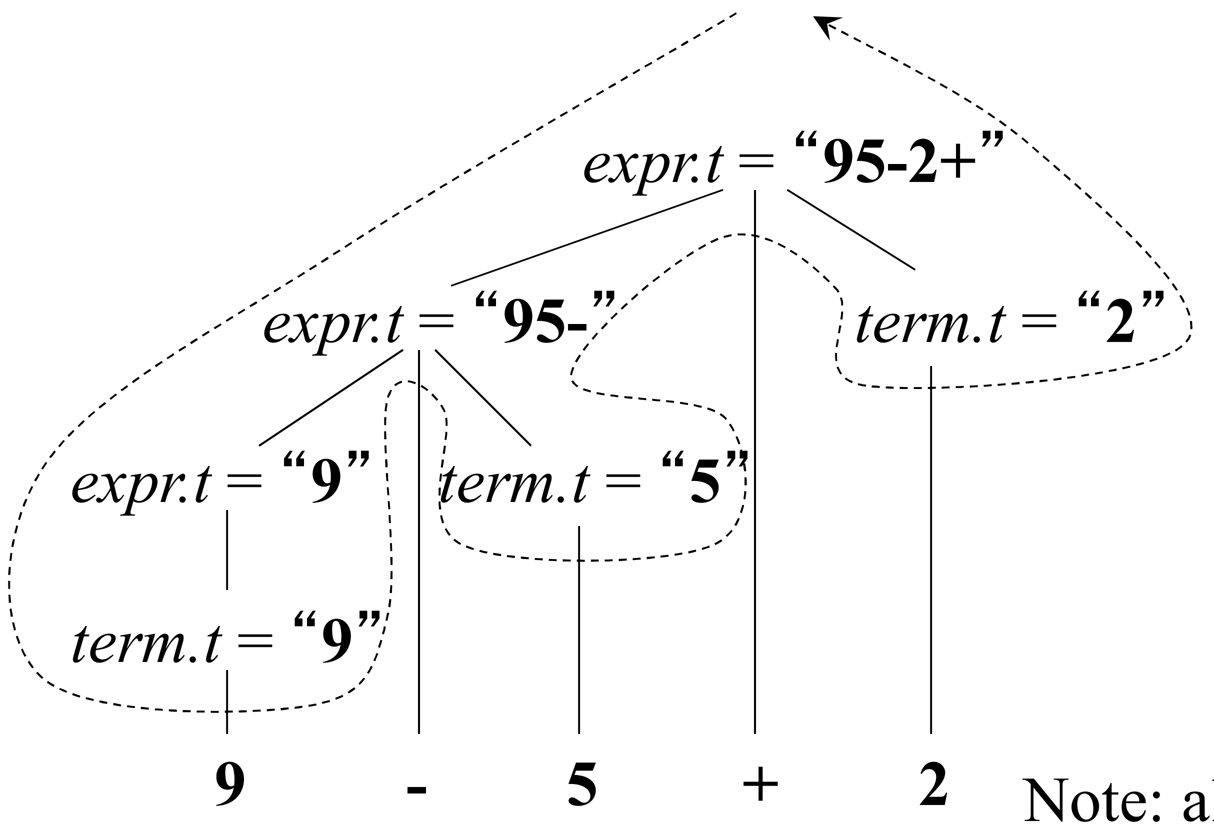


Figure 2.12: Example of a depth-first traversal of a tree

# Depth-First Traversals (Example)

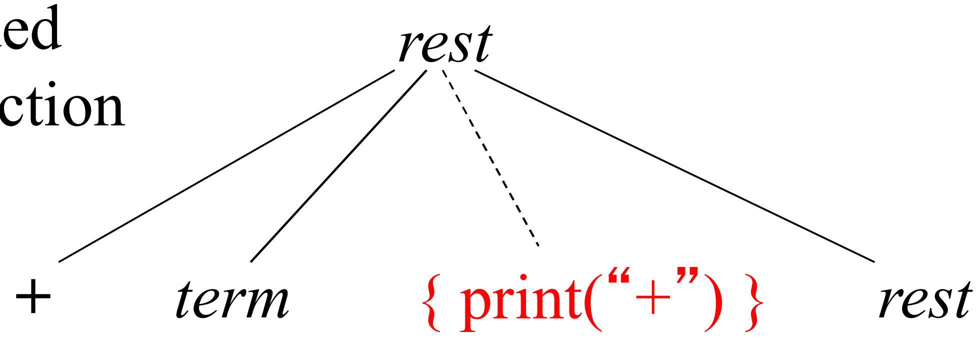


Note: all attributes are of the synthesized type

## Translation Schemes

- A translation scheme is a CF grammar embedded with semantic actions by attaching program fragments to productions in the grammar

$$rest \rightarrow + \ term\ \underbrace{\{\ \text{print}(\text{“+”})\ \}}_{\text{Embedded semantic action}}\ rest$$



An extra leaf is constructed for a semantic action

# Example Translation Scheme

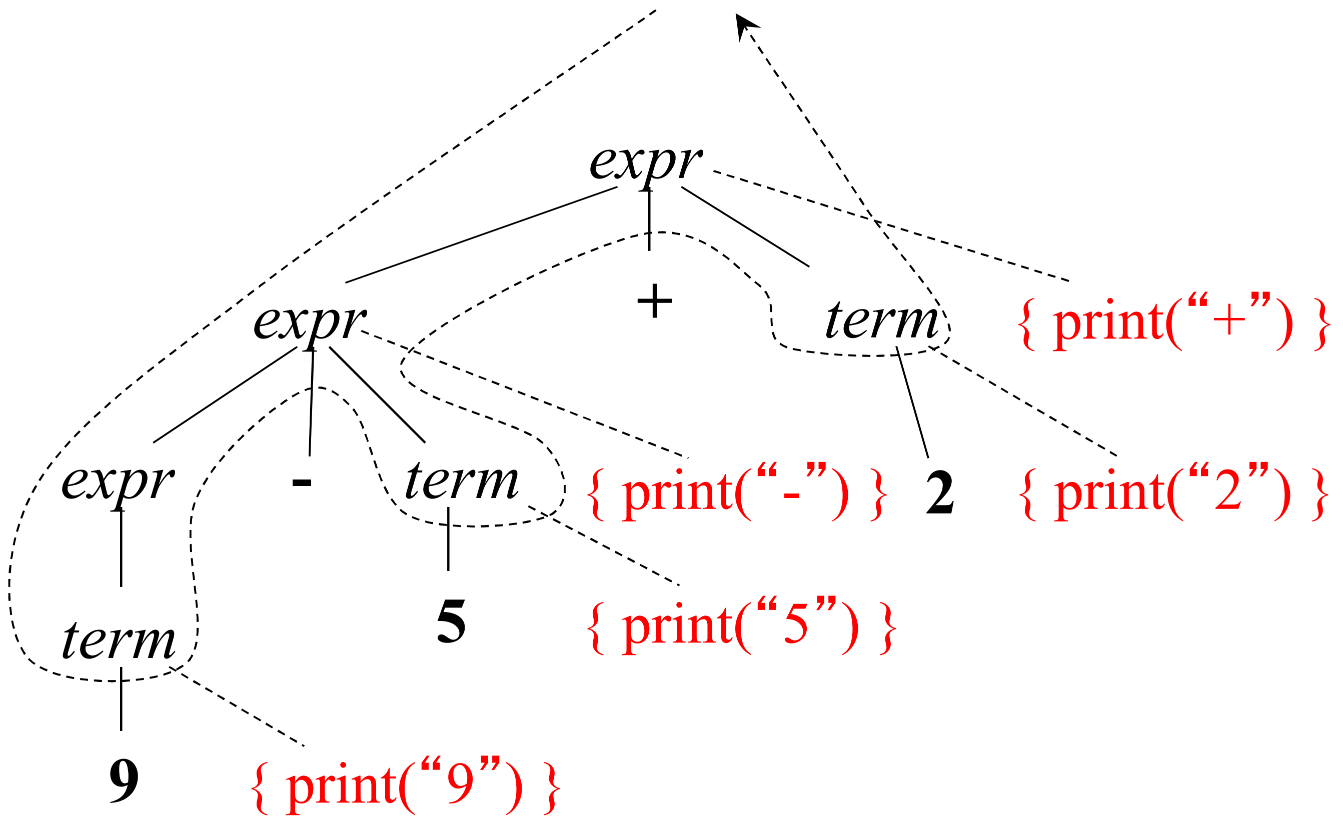
## Grammar

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

Actions for translating into postfix notation

# Example Translation Scheme

## Parse Tree



Translates **9-5+2** into postfix **95-2+**

## 4. Parsing

- Parsing = *process of determining if a string of tokens can be generated by a grammar*
- For any CF grammar there is a parser that takes at most $O(n^3)$ time to parse a string of $n$ tokens
- Linear algorithms suffice for parsing programming language source code
- *Top-down parsing* "constructs" a parse tree from root to leaves
- *Bottom-up parsing* "constructs" a parse tree from leaves to root

## Top-Down Parsing

- The top-down construction of a parse tree is done by starting with the root, labeled with the starting nonterminal , and repeatedly performing the following two steps.
  1. At node N, labeled with nonterminal A, select one of the productions for A and construct children at N for the symbols in the production body.
  2. Find the next node at which a subtree is to be constructed, typically the leftmost unexpanded nonterminal of the tree.

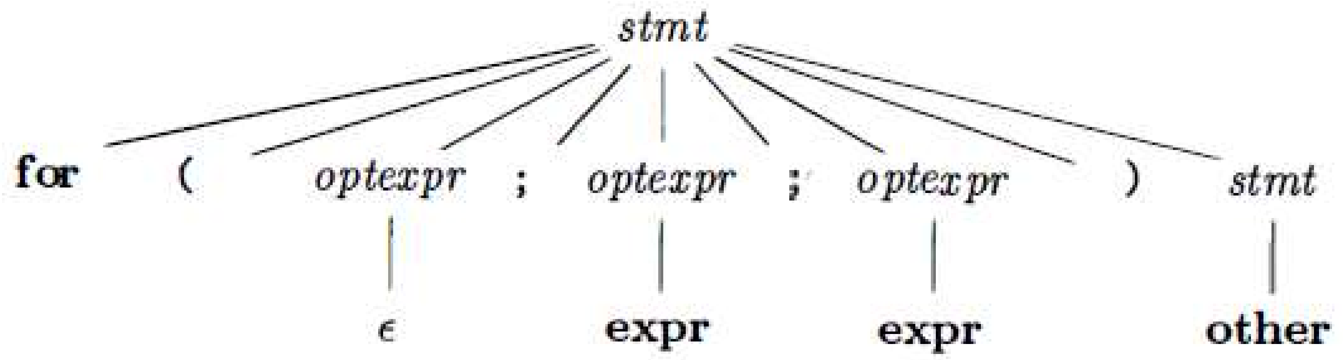
# Top-Down Parsing Example

Grammar

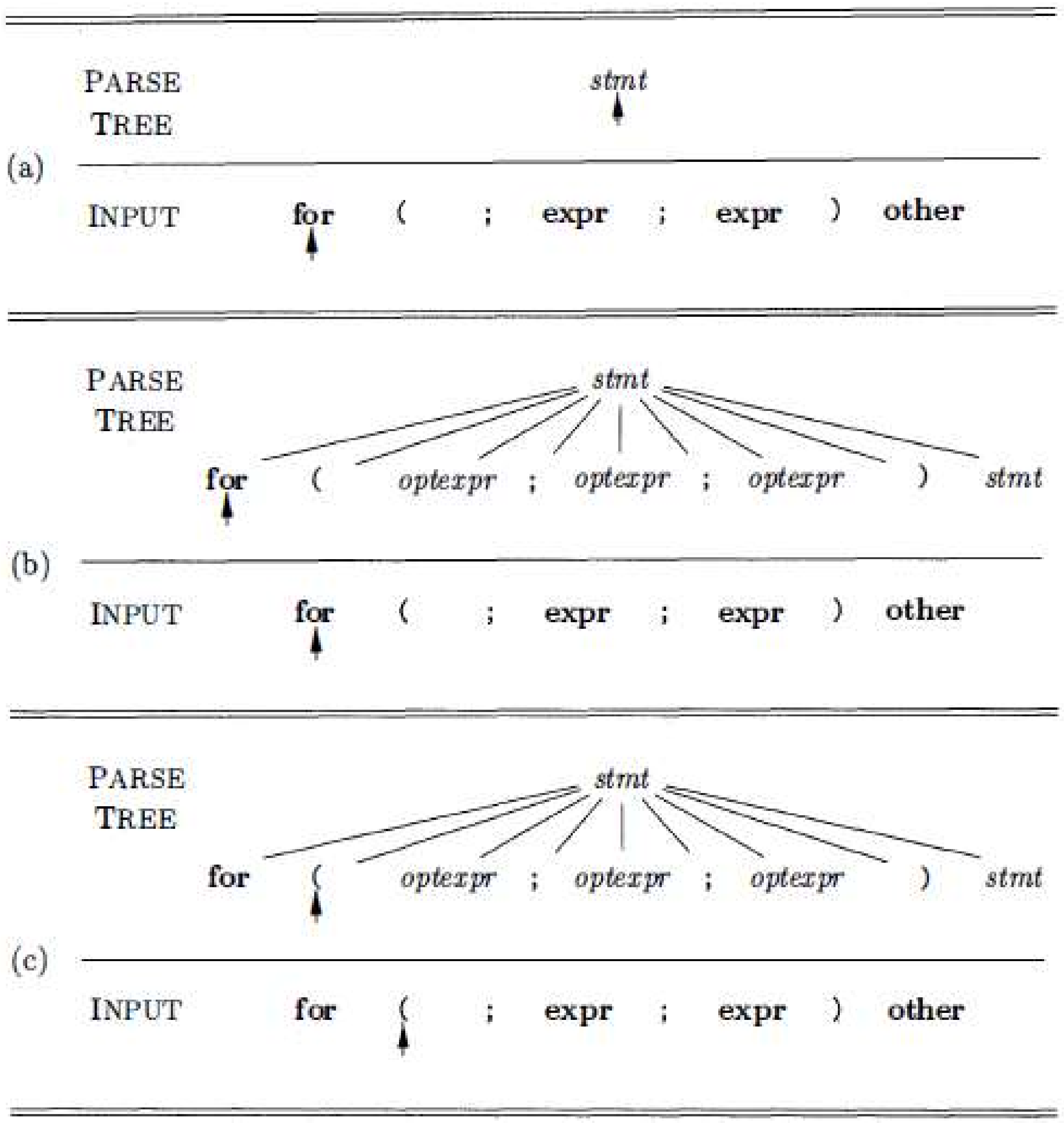
$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{expr} \ ; \\
 & | & \textbf{if} \ ( \ \textbf{expr} \ ) \ \mathit{stmt} \\
 & | & \textbf{for} \ ( \ \mathit{optexpr} \ ; \ \mathit{optexpr} \ ; \ \mathit{optexpr} \ ) \ \mathit{stmt} \\
 & | & \textbf{other} \\
 & & \\
\mathit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

Input string

for ( ; expr ; expr ) other



A parse tree according to the grammar


(a)
PARSE TREE
stmt
INPUT
for ( ; expr ; expr ) other
(b)
PARSE TREE
stmt
for ( optexpr ; optexpr ; optexpr ) stmt
INPUT
for ( ; expr ; expr ) other
(c)
PARSE TREE
stmt
for ( optexpr ; optexpr ; optexpr ) stmt
INPUT
for ( ; expr ; expr ) other

## Predictive Parsing

- *Recursive descent parsing* is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input.
  - Each nonterminal has one (recursive) procedure that is responsible for parsing the nonterminal's syntactic category of input tokens
  - When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information
- *Predictive parsing* is a special form of recursive descent parsing where we use one lookahead token to unambiguously determine the parse operations

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr(); match(';'); optexpr(); match(';'); optexpr();
                match(')'); stmt(); break;
        case other;
                match(other); break;
        default:
                report("syntax error");
        }
}

void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
        if ( lookahead == t ) lookahead = nextTerminal;
        else report("syntax error");
}
```

## FIRST

FIRST(α) is the set of terminals that appear as the first symbols of one or more strings generated from α

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{expr}\ ; \\
& | & \textbf{if}\ (\ \textbf{expr}\ )\ \mathit{stmt} \\
& | & \textbf{for}\ (\ \mathit{optexpr}\ ;\ \mathit{optexpr}\ ;\ \mathit{optexpr}\ )\ \mathit{stmt} \\
& | & \textbf{other} \\
\\
\mathit{optexpr} & \rightarrow & \epsilon \\
& | & \textbf{expr}
\end{array}
$$

FIRST(*stmt*) = { **expr**, **if**, **for**, **other** }
FIRST(**expr**) = {**expr**}
FIRST( **for** (*optexpr* ; *optexpr ; optexpr )* *stmt*) = {**for**}

# How to use FIRST

We use FIRST to write a predictive parser as follows

$$
\begin{aligned}
expr &\rightarrow term\ rest \\
rest &\rightarrow +\ term\ rest \\
&\mid -\ term\ rest \\
&\mid \varepsilon
\end{aligned}
$$

```
procedure rest();
begin
    if lookahead in FIRST(+ term rest) then
        match( '+' ); term(); rest()
    else if lookahead in FIRST(- term rest) then
        match( '-' ); term(); rest()
    else return
end;
```

When a nonterminal *A* has two (or more) productions as in

$$
\begin{aligned}
A &\rightarrow \alpha \\
&\mid \beta
\end{aligned}
$$

Then FIRST ($\alpha$) and FIRST($\beta$) must be disjoint for predictive parsing to work

## Left Factoring

When more than one production for nonterminal *A* starts with the same symbols, the FIRST sets are not disjoint

*stmt* → **if** *expr* **then** *stmt* **endif**
| **if** expr **then** *stmt* **else** *stmt* **endif**

We can use *left factoring* to fix the problem

*stmt* → **if** expr **then** *stmt opt_else*
*opt_else* → **else** *stmt* **endif**
| **endif**

## Left Recursion

When a production for nonterminal *A* starts with a self reference then a predictive parser loops forever

$$
\begin{aligned}
A &\rightarrow A\ \alpha \\
&\mid \beta \\
&\mid \gamma
\end{aligned}
$$

We can eliminate *left recursive productions* by systematically rewriting the grammar using *right recursive productions*

$$
\begin{aligned}
A &\rightarrow \beta\ R \\
&\mid \gamma\ R \\
R &\rightarrow \alpha\ R \\
&\mid \varepsilon
\end{aligned}
$$

# 5. A Translator for Simple Expressions

## Actions for translating into postfix notation

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

## Translation scheme after left recursion elimination

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

## Example Parse Tree

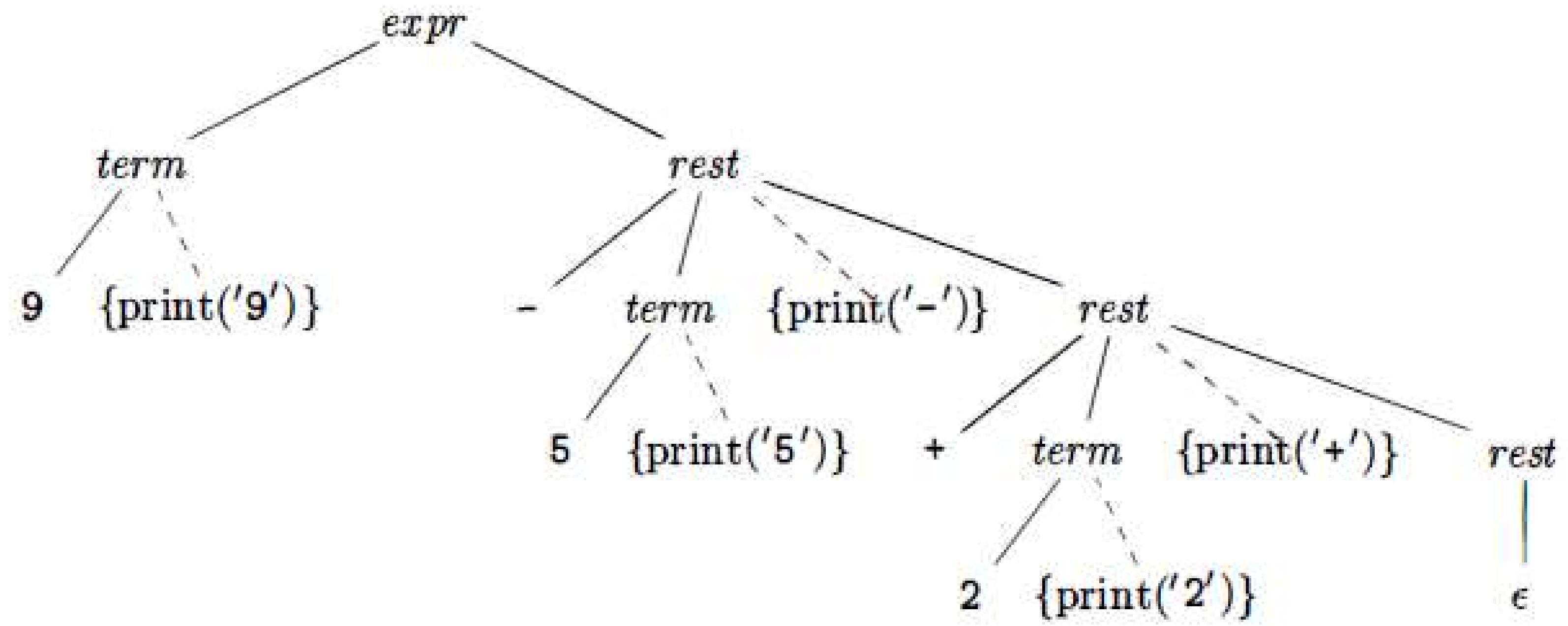


Figure 2.24: Translation of 9-5+2 to 95-2+

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

```
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;
        }
    }
```

## … 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

$expr \rightarrow term\ rest$

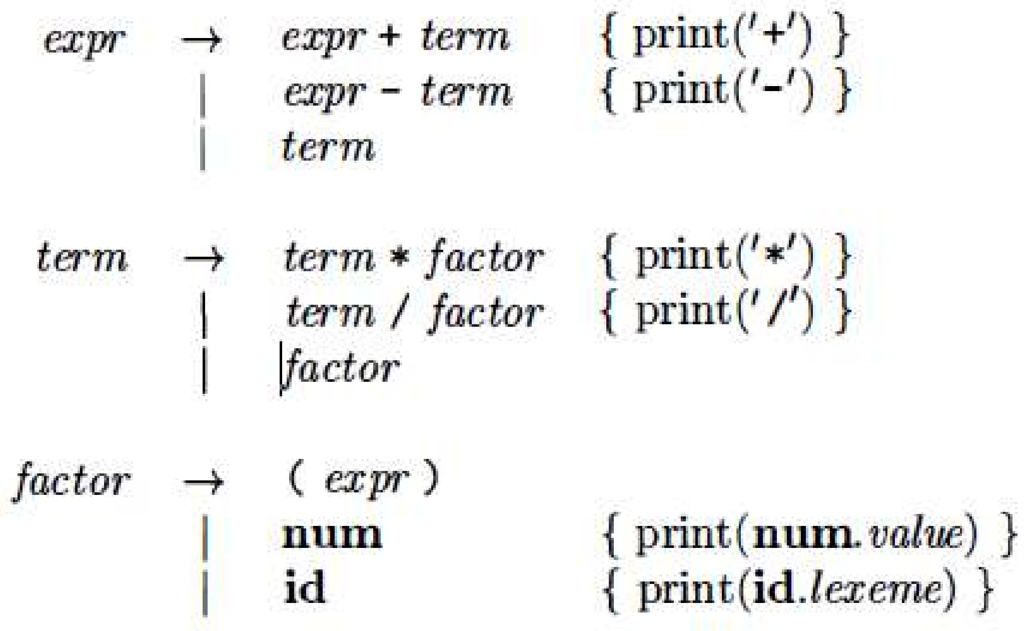
$rest \rightarrow +\ term\ \{\ \text{print(“+”)}\ \}\ rest$
$\quad |\ -\ term\ \{\ \text{print(“-”)}\ \}\ rest$
$\quad |\ \varepsilon$

$term \rightarrow \mathbf{0}\ \{\ \text{print(“0”)}\ \}$
$term \rightarrow \mathbf{1}\ \{\ \text{print(“1”)}\ \}$
…
$term \rightarrow \mathbf{9}\ \{\ \text{print(“9”)}\ \}$

```
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



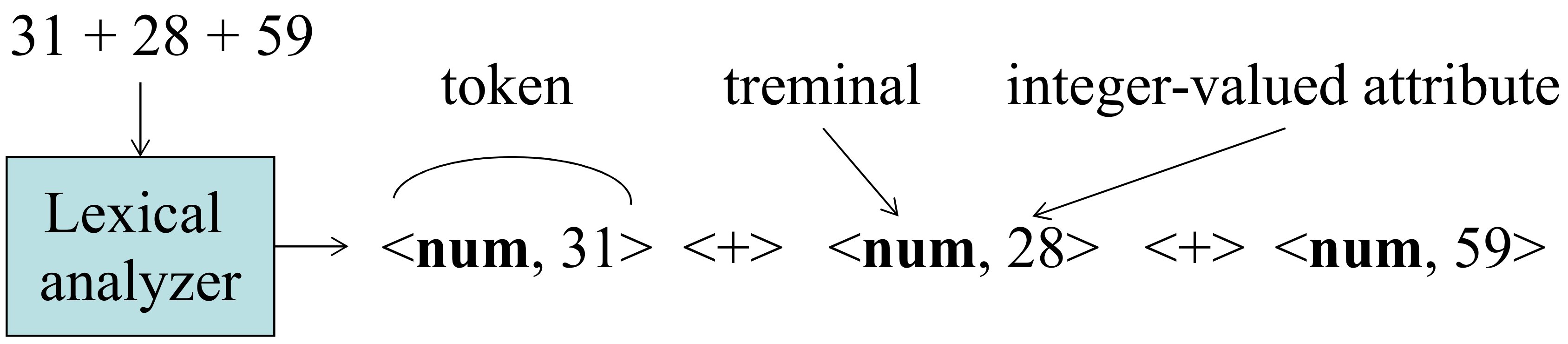
$$
\begin{array}{lcll}
expr & \rightarrow & expr + term & \{\ \text{print}('+')\ \} \\
 & | & expr - term & \{\ \text{print}('-')\ \} \\
 & | & term & \\
 & & & \\
term & \rightarrow & term * factor & \{\ \text{print}('*')\ \} \\
 & | & term\ /\ factor & \{\ \text{print}('/')\ \} \\
 & | & factor & \\
 & & & \\
factor & \rightarrow & (\ expr\ ) & \\
 & | & \textbf{num} & \{\ \text{print}(\textbf{num}.value)\ \} \\
 & | & \textbf{id} & \{\ \text{print}(\textbf{id}.lexeme)\ \}
\end{array}
$$

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

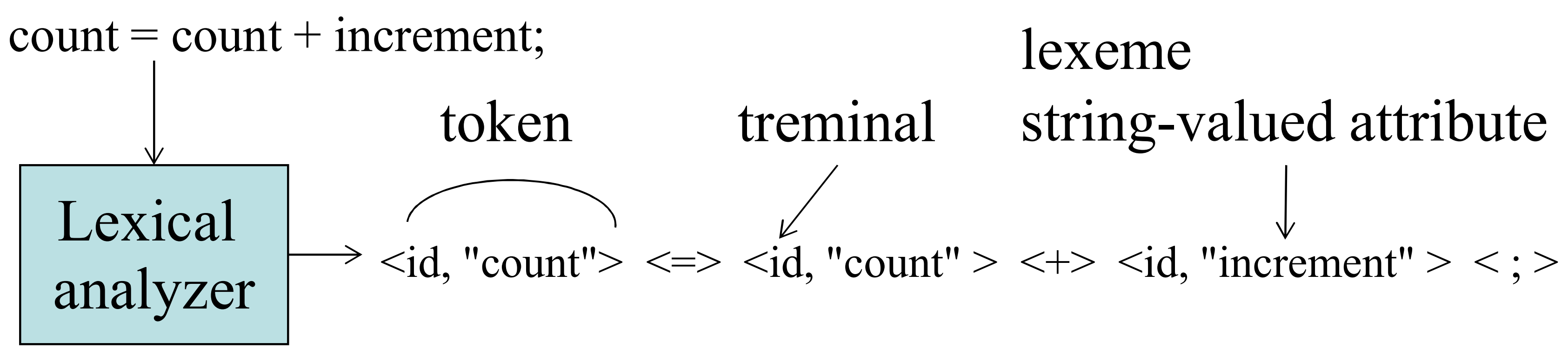
## Constants (Number)



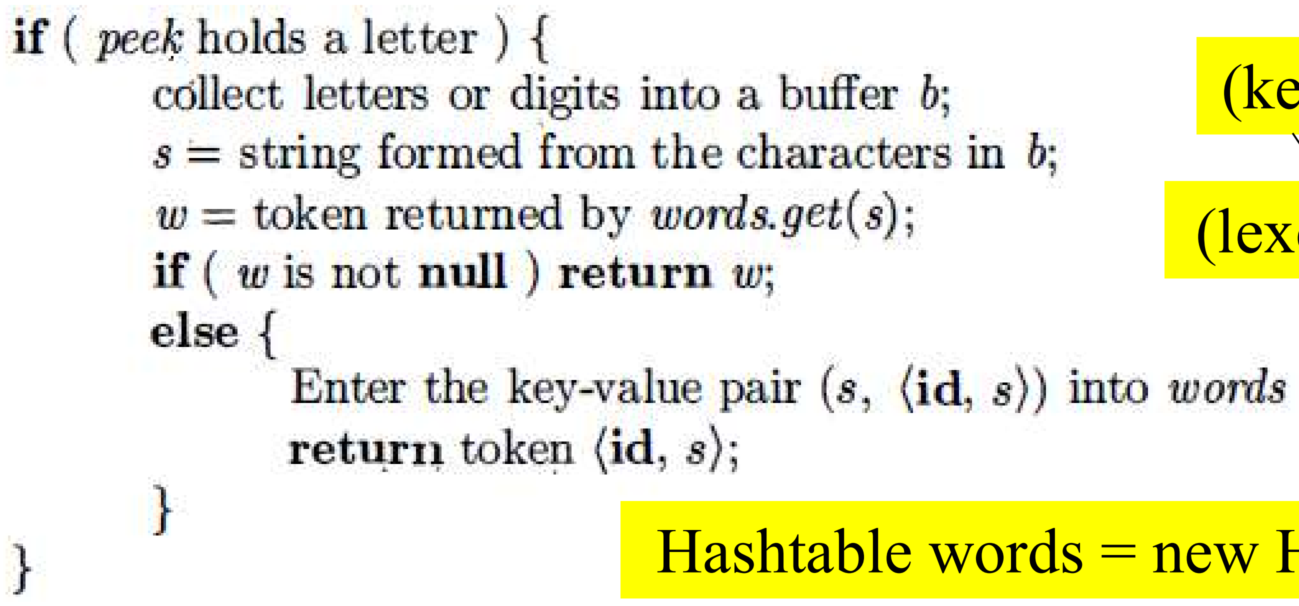
```
if ( peek holds a digit ) {
        v = 0;
        do {
                v = v * 10 + integer value of digit peek;
                peek = next input character;
        } while ( peek holds a digit );
        return token ⟨num, v⟩;
}
```

Grouping digits into integers

# Keywords and Identifiers


count = count + increment;
Lexical analyzer
token
treminal
lexeme
string-valued attribute
<id, "count"> <=> <id, "count" > <+> <id, "increment" > < ; >

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



```
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⟩;
        }
}
```

(key, value)

(lexeme, token)

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;
}
```

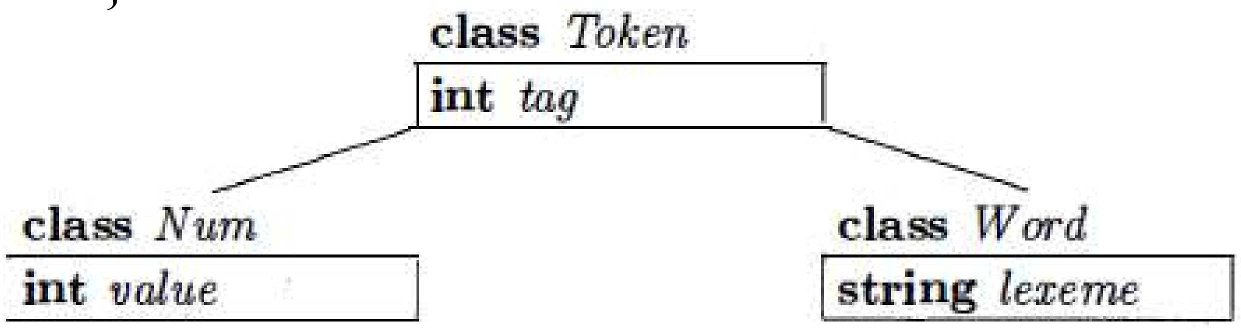


Figure 2.32: Class *Token* and subclasses *Num* and *Word*

## 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;
}
```

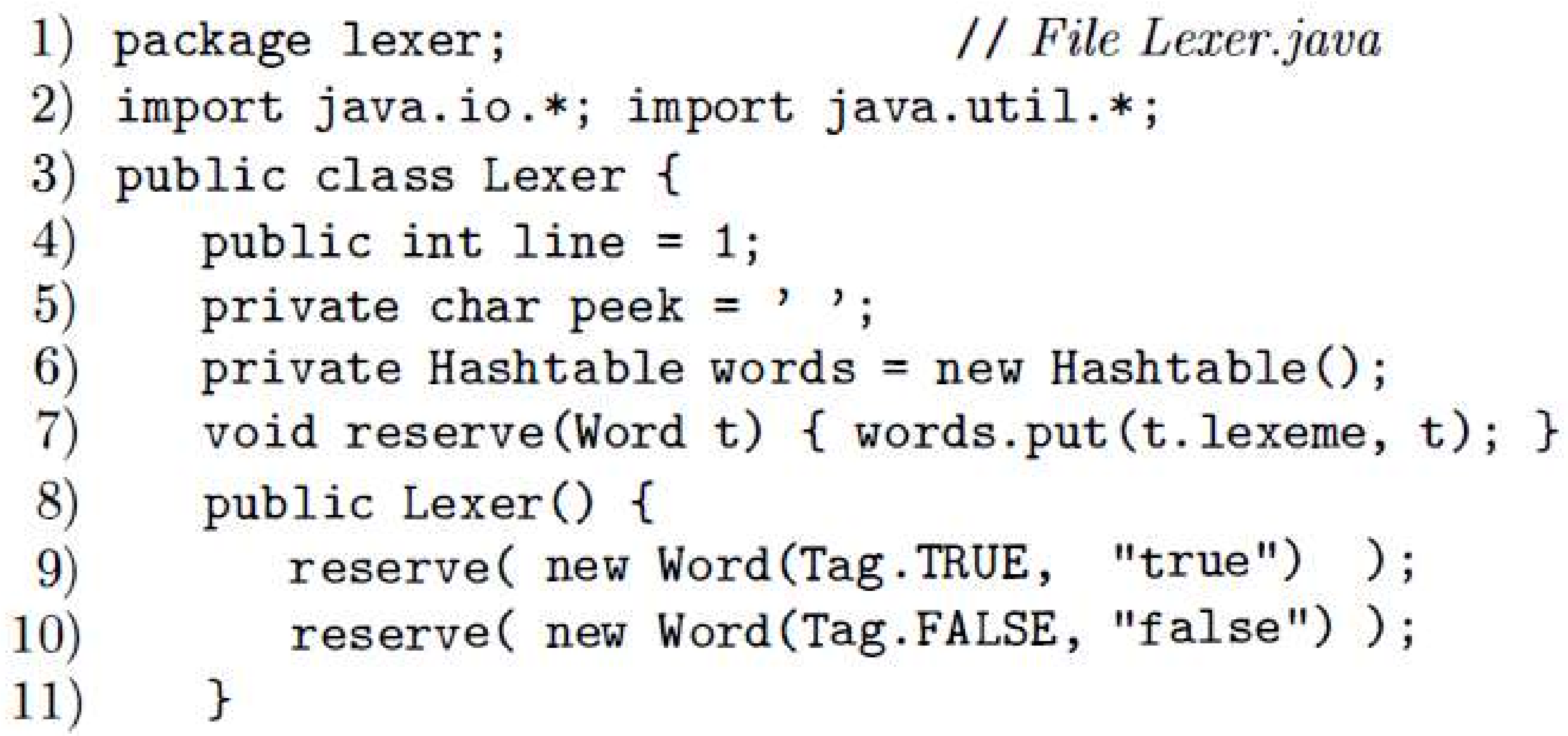
In C++, constant is defined as below
**#define NUM 256**

## 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;
   }
}
```

# 7. Symbol Tables

Given input :

```
{ int x ; char y ; { bool y ; x ; y ; } x ; y ; }
```

The goal is to produce output:

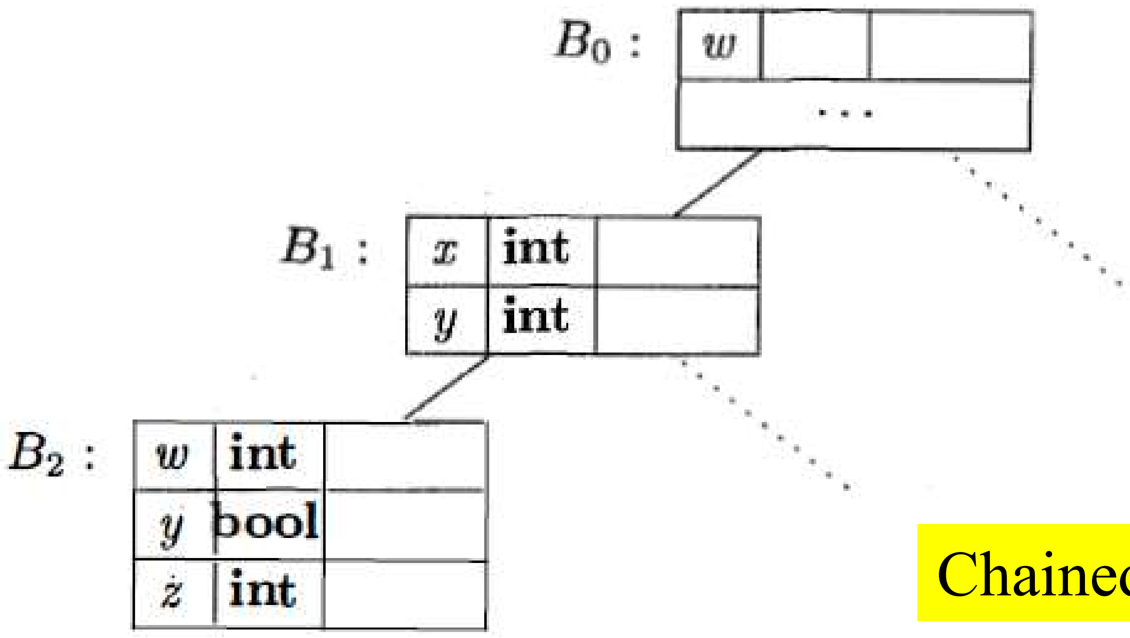
```
{ { x : int ; y : bool ; } x : int ; y : char ; }
```

The most- closely nested rule for blocks

an identifier x is in the scope of the most-closely nested declaration of x; that is, the declaration of x found by examining blocks inside-out, starting with the block in which x appears

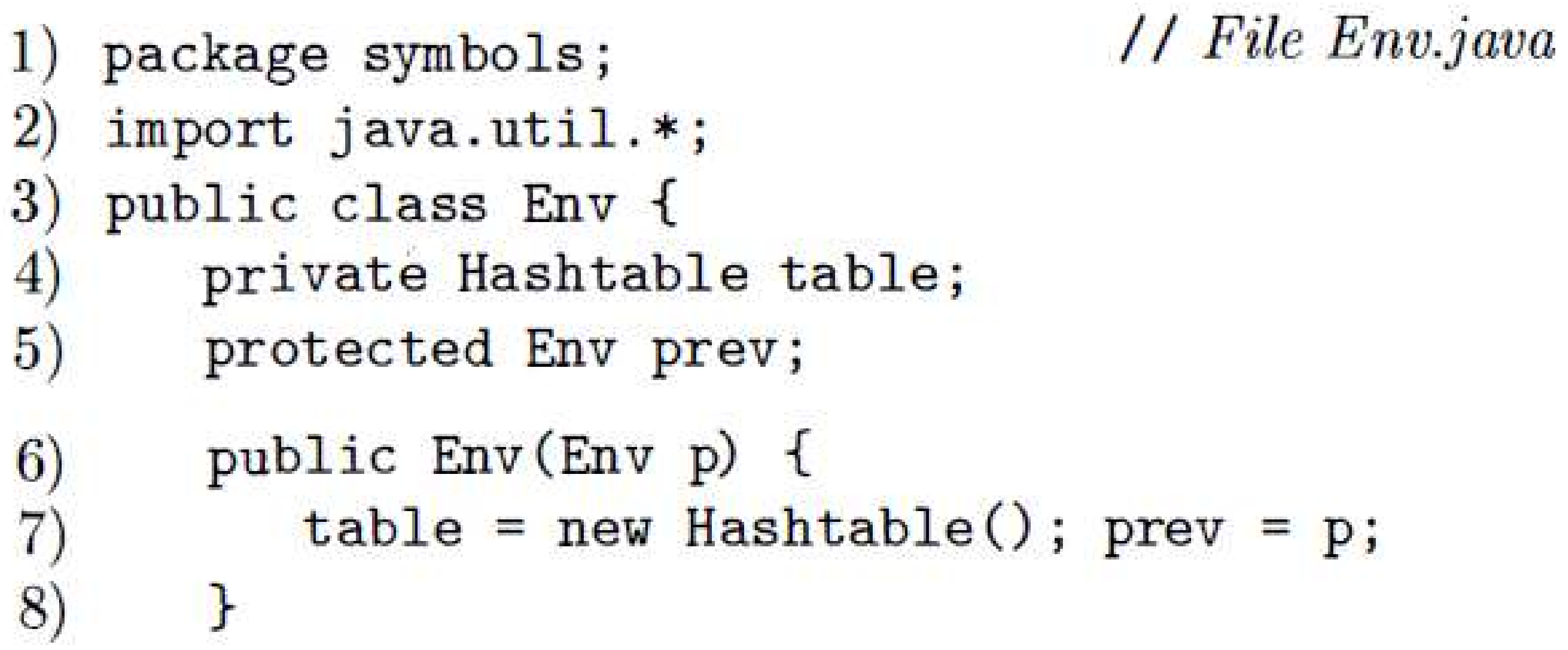
## Symbol Table Per Scope

```
{   int $x_1$; int $y_1$;
    {   int $w_2$; bool $y_2$; int $z_2$;
            ⋯ $w_2$ ⋯; ⋯ $x_1$ ⋯; ⋯ $y_2$ ⋯; ⋯ $z_2$ ⋯;
    }
    ⋯ $w_0$ ⋯; ⋯ $x_1$ ⋯; ⋯ $y_1$ ⋯;
}
```



Chained symbol tables

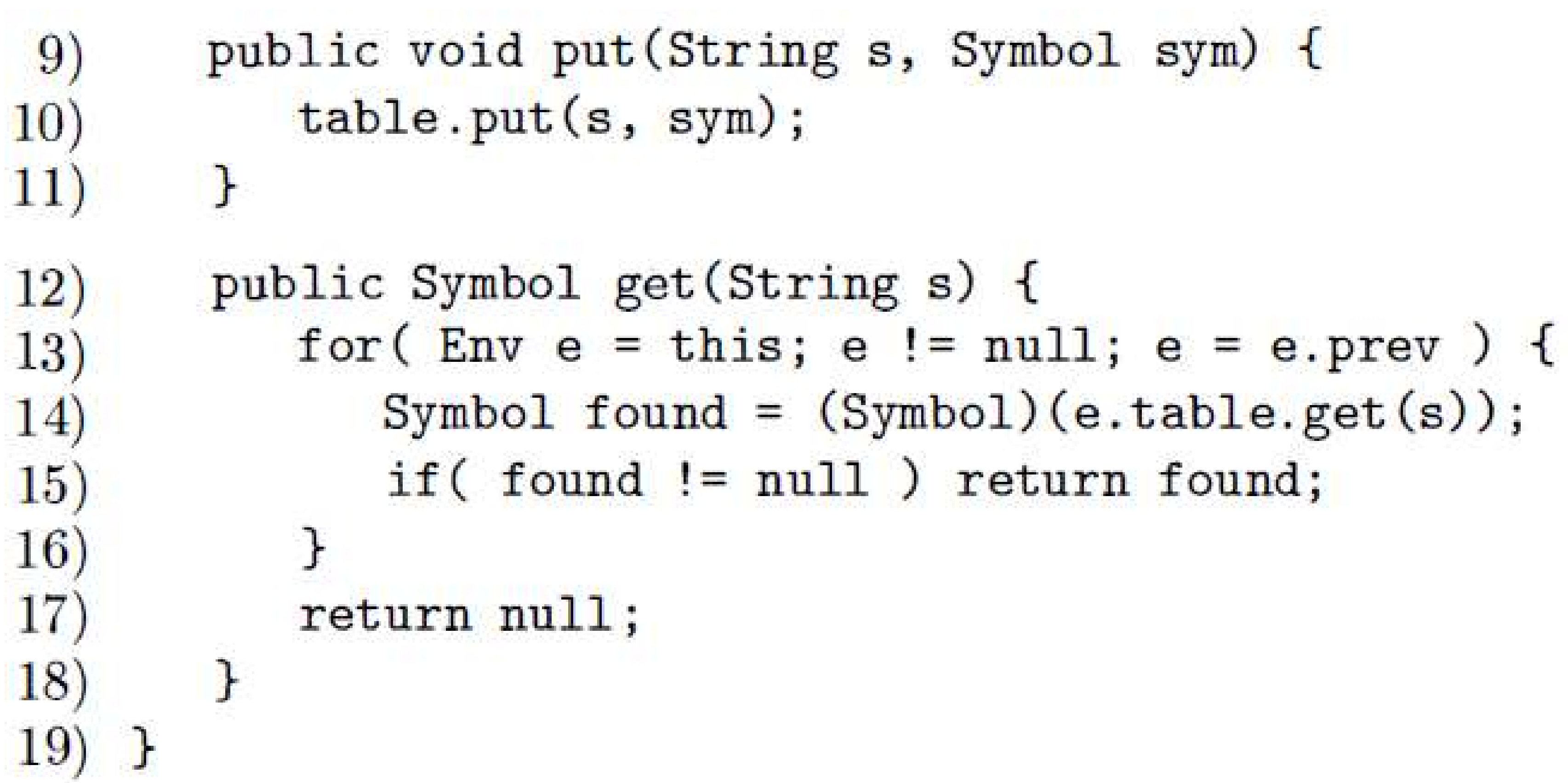
## Class Env implements chained symbol tables 1/2



```
package symbols;                              // File Env.java
import java.util.*;
public class Env {
    private Hashtable table;
    protected Env prev;

    public Env(Env p) {
        table = new Hashtable(); prev = p;
    }
```

## Class Env implements chained symbol tables 2/2



```
    public void put(String s, Symbol sym) {
       table.put(s, sym);
    }
    public Symbol get(String s) {
       for( Env e = this; e != null; e = e.prev ) {
          Symbol found = (Symbol)(e.table.get(s));
          if( found != null ) return found;
       }
       return null;
    }
}
```

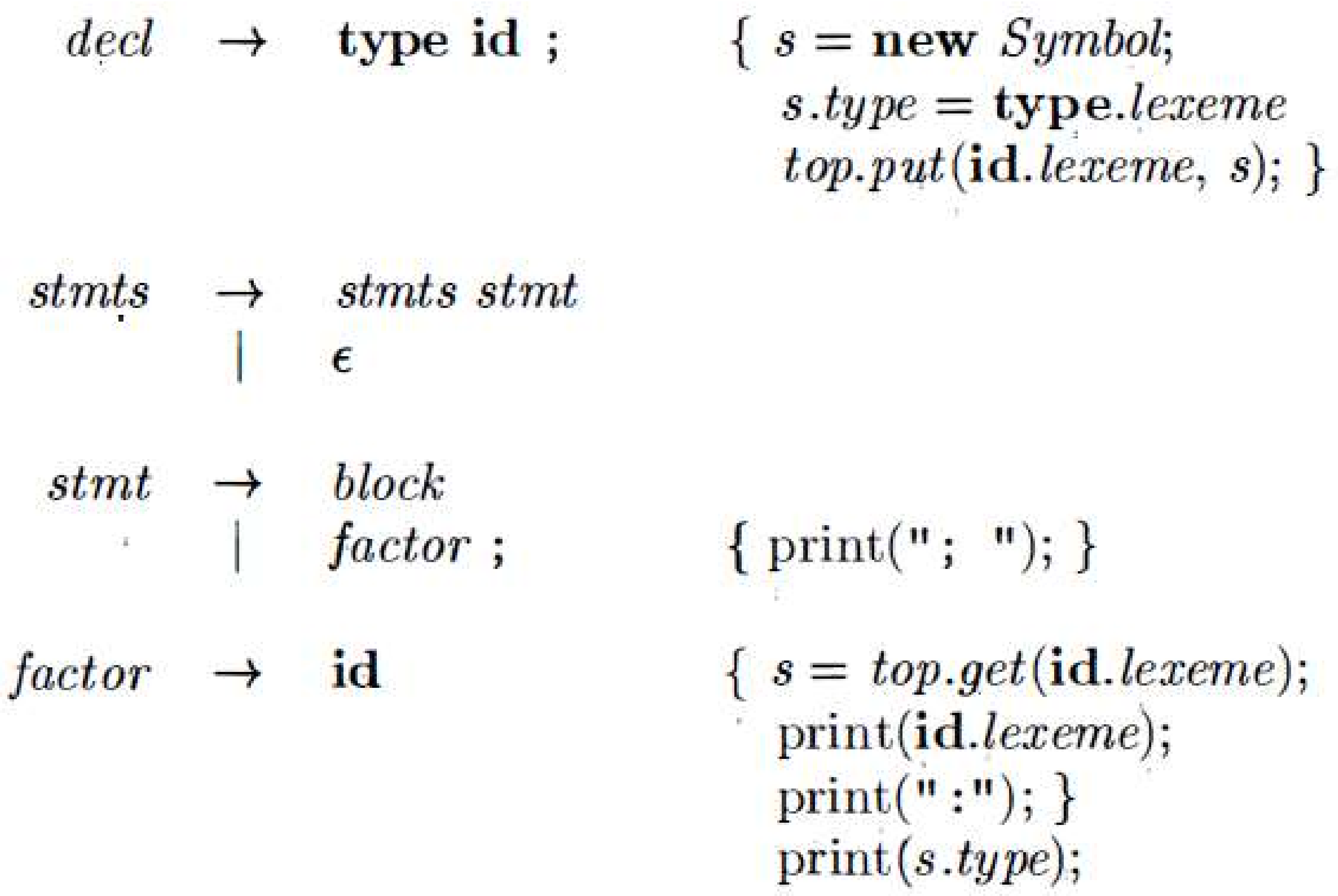
# The Use of Symbol Tables 1/2

The use of symbol tables for translating a language with blocks

$$
\begin{array}{lcll}
\mathit{program} & \rightarrow & & \{\ \mathit{top} = \textbf{null};\ \} \\
 & & \mathit{block} & \\
 & & & \\
\mathit{block} & \rightarrow & \text{'\{'} & \{\ \mathit{saved} = \mathit{top}; \\
 & & & \quad \mathit{top} = \textbf{new}\ \mathit{Env}(\mathit{top}); \\
 & & & \quad \texttt{print("\{ ");}\ \} \\
 & & \mathit{decls}\ \mathit{stmts}\ \text{'\}'} & \{\ \mathit{top} = \mathit{saved}; \\
 & & & \quad \texttt{print("\} ");}\ \} \\
 & & & \\
\mathit{decls} & \rightarrow & \mathit{decls}\ \mathit{decl} & \\
 & \mid & \epsilon &
\end{array}
$$

# The Use of Symbol Tables 2/2

The use of symbol tables for translating a language with blocks

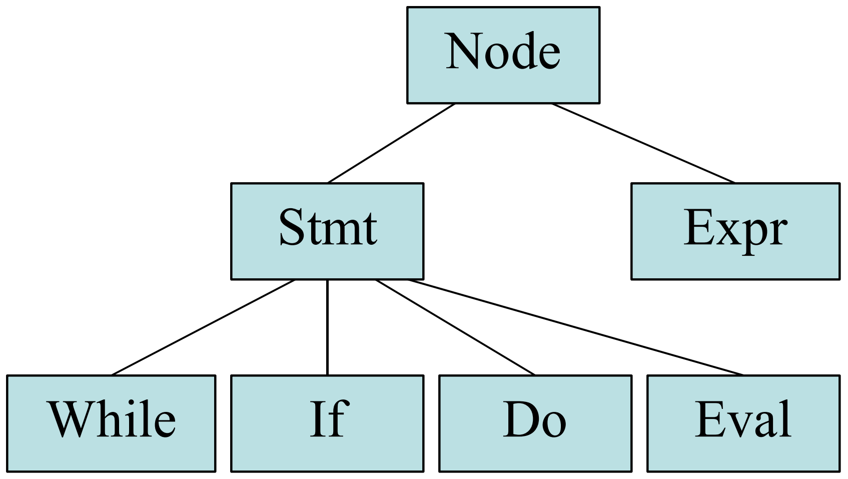


$$
\begin{array}{lcll}
\mathit{decl} & \rightarrow & \textbf{type id ;} & \{\ s = \textbf{new } \mathit{Symbol}; \\
 & & & \quad s.\mathit{type} = \textbf{type}.\mathit{lexeme} \\
 & & & \quad \mathit{top.put}(\textbf{id}.\mathit{lexeme},\ s);\ \} \\
 & & & \\
\mathit{stmts} & \rightarrow & \mathit{stmts}\ \mathit{stmt} & \\
 & | & \epsilon & \\
 & & & \\
\mathit{stmt} & \rightarrow & \mathit{block} & \\
 & | & \mathit{factor}\ ; & \{\ \texttt{print("; ");}\ \} \\
 & & & \\
\mathit{factor} & \rightarrow & \textbf{id} & \{\ s = \mathit{top.get}(\textbf{id}.\mathit{lexeme}); \\
 & & & \quad \texttt{print}(\textbf{id}.\mathit{lexeme}); \\
 & & & \quad \texttt{print(":");}\ \} \\
 & & & \quad \texttt{print}(s.\mathit{type});
\end{array}
$$

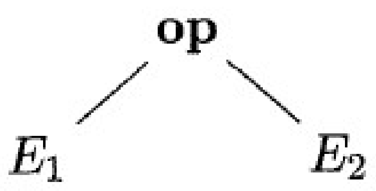
## 8. Intermediate Code Generation

- Consider intermediate representations for expressions and statements (No declarations)
- Two most important intermediate representations are
  - Trees, including parse trees and (abstract) syntax trees
  - Linear representations, especially "three-address code.“
- Construction of Syntax Trees
  - Syntax Trees for Statements
  - Representing Blocks in Syntax Trees
  - Syntax Trees for Expressions
- Static Checking
- Emit three-address code along with the syntax tree

# Syntax Trees for Statements



One class per statement



AST

**while** ( *expr* ) *stmt*

while statement

**new** *While* $(x, y)$

Create a While node

$stmt \rightarrow$ **if** ( $expr$ ) $stmt_1$ $\{\ stmt.n = \textbf{new}\ \mathit{If}(expr.n, stmt_1.n);\ \}$

Production

Semantic action

# Syntax Trees for Statements (Cont.)

$$\left.\begin{array}{llll}
\textit{program} & \rightarrow & \textit{block} & \{\ \text{return } \textit{block.n};\ \} \\
\textit{block} & \rightarrow & \text{'\{' } \textit{stmts} \text{ '\}'} & \{\ \textit{block.n} = \textit{stmts.n};\ \}
\end{array}\right\}\text{Block}$$

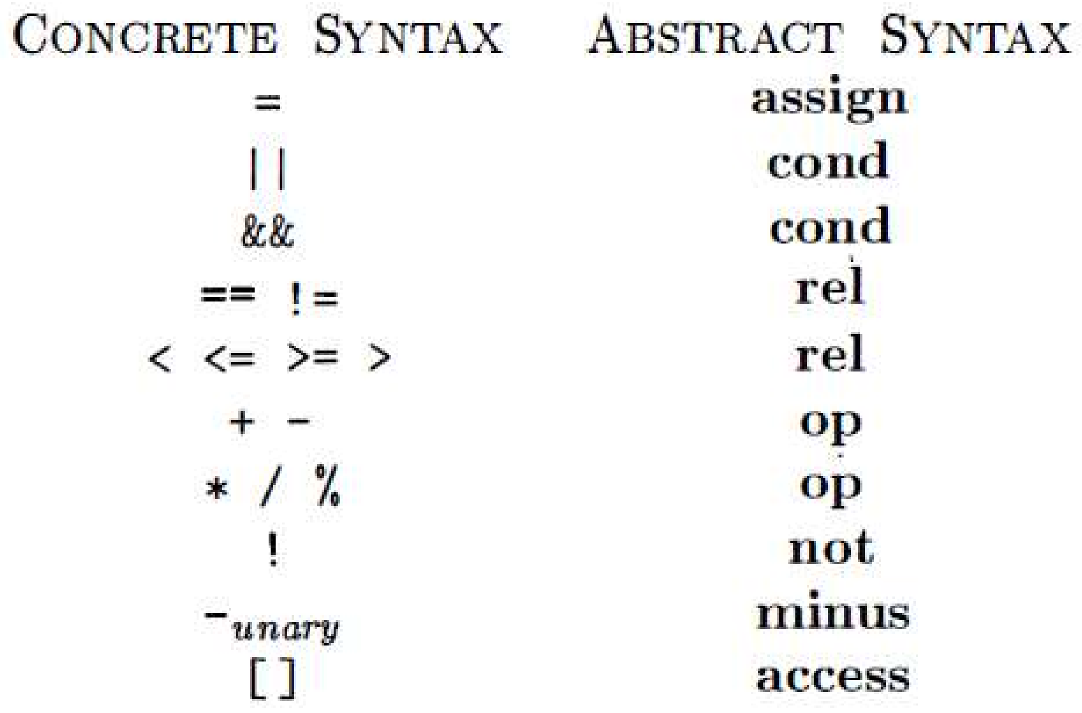
$$\begin{array}{llll}
\textit{stmts} & \rightarrow & \textit{stmts}_1\ \textit{stmt} & \{\ \textit{stmts.n} = \textbf{new}\ \textit{Seq}(\textit{stmts}_1.n, \textit{stmt.n});\ \} \\
 & | & \epsilon & \{\ \textit{stmts.n} = \textbf{null};\ \}
\end{array}$$

sequence

$$\begin{array}{llll}
\textit{stmt} & \rightarrow & \textit{expr}\ ; & \{\ \textit{stmt.n} = \textbf{new}\ \textit{Eval}(\textit{expr.n});\ \} \\
 & | & \textbf{if}\ (\ \textit{expr}\ )\ \textit{stmt}_1 & \\
 & & & \{\ \textit{stmt.n} = \textbf{new}\ \textit{If}(\textit{expr.n}, \textit{stmt}_1.n);\ \} \\
 & | & \textbf{while}\ (\ \textit{expr}\ )\ \textit{stmt}_1 & \\
 & & & \{\ \textit{stmt.n} = \textbf{new}\ \textit{While}(\textit{expr.n}, \textit{stmt}_1.n);\ \} \\
 & | & \textbf{do}\ \textit{stmt}_1\ \textbf{while}\ (\ \textit{expr}\ )\ ; & \\
 & & & \{\ \textit{stmt.n} = \textbf{new}\ \textit{Do}(\textit{stmt}_1.n, \textit{expr.n});\ \} \\
 & | & \textit{block} & \{\ \textit{stmt.n} = \textit{block.n};\ \}
\end{array}$$

## Syntax Trees for Expressions

- Group "similar" operators to reduce the number of classes of nodes in an implementation of expressions.
- "similar" to mean that the type-checking and code-generation rules for the operators are similar



| CONCRETE SYNTAX | ABSTRACT SYNTAX |
|---|---|
| = | assign |
| \|\| | cond |
| && | cond |
| == != | rel |
| < <= >= > | rel |
| + - | op |
| * / % | op |
| ! | not |
| $-_{unary}$ | minus |
| [] | access |

## Syntax Trees for Expressions

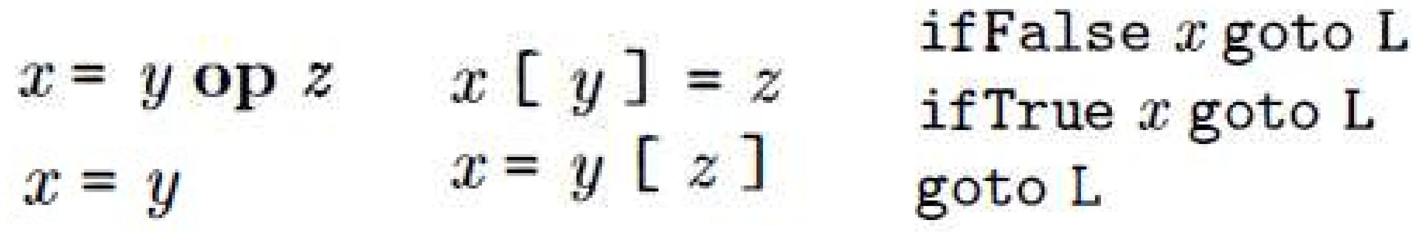
$$
\begin{array}{rcll}
expr & \rightarrow & rel = expr_1 & \{\ expr.n = \textbf{new}\ Assign('=', rel.n, expr_1.n);\ \} \\
 & | & rel & \{\ expr.n = rel.n;\ \} \\
 & & & \\
rel & \rightarrow & rel_1 < add & \{\ rel.n = \textbf{new}\ Rel('<', rel_1.n, add.n);\ \} \\
 & | & rel_1 <= add & \{\ rel.n = \textbf{new}\ Rel('\leq', rel_1.n, add.n);\ \} \\
 & | & add & \{\ rel.n = add.n;\ \} \\
 & & & \\
add & \rightarrow & add_1 + term & \{\ add.n = \textbf{new}\ Op('+', add_1.n, term.n);\ \} \\
 & | & term & \{\ add.n = term.n;\ \} \\
 & & & \\
term & \rightarrow & term_1 * factor & \{\ term.n = \textbf{new}\ Op('*', term_1.n, factor.n);\ \} \\
 & | & factor & \{\ term.n = factor.n;\ \} \\
 & & & \\
factor & \rightarrow & (\ expr\ ) & \{\ factor.n = expr.n;\ \} \\
 & | & \textbf{num} & \{\ factor.n = \textbf{new}\ Num(\textbf{num}.value);\ \}
\end{array}
$$

## Static Checking

- Static checks are consistency checks that are done during compilation
  - Syntactic Checking.
    - There is more to syntax than grammars
  - Type Checking
    - Assure that an operator or function is applied to the right number and type of operands
- L-values and R-values
  - *r*-values are what we usually think of as "values," while *l*-values are locations.
- Coercion
  - A coercion occurs if the type of an operand is automatically converted to the type expected by the operator

## Three-Address Code

- Show how to write functions that process the syntax tree and, as a side-effect, emit the necessary three-address code
- Three-Address Instructions

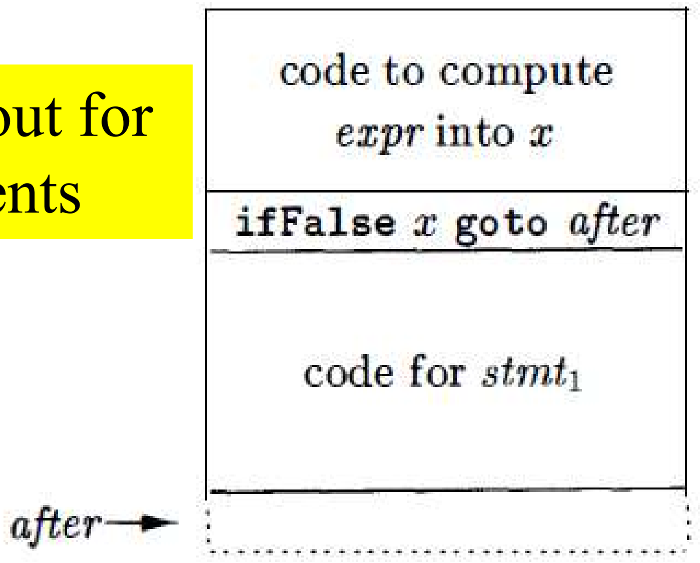


```
x = y op z      x [ y ] = z      ifFalse x goto L
x = y           x = y [ z ]      ifTrue x goto L
                                 goto L
```

- Translation of Statements
  - Example: **if** *expr* **then** $stmt_1$

**if** *expr* **then** $stmt_1$

Code layout for if-statements



.

Function gen in class If generates three-address code

```
class If extends Stmt {
        Expr E; Stmt S;
        public If(Expr x, Stmt y) { E = x; S = y; after = newlabel(); }
        public void gen() {
                Expr n = E.rvalue();
                emit( "ifFalse " + n.toString() + " goto " + after);
                S.gen();
                emit(after + ":");
        }
}
```

# Using Translation Scheme

**if** *expr* **then** $stmt_1$

$stmt \rightarrow$ **if** *expr*

```
        { after = newlabel();
          print("ifFalse goto after:"); }
    then stmt1
        { print("after:    "); }
```

| |
|---|
| code for *expr* |
| **`ifFalse goto`** *after* |
| code for $stmt_1$ |
| *after*: |

# Translation of Expressions

i - j +k

$\downarrow$

tl = i - j

t2 = tl + k

2 *a [i]

$\downarrow$

tl = a [ i ]

t2 = 2 * tl

a [2*k]

$\downarrow$

t = 2*k

a [t]

a [ i ] = 2 * a[j – k]

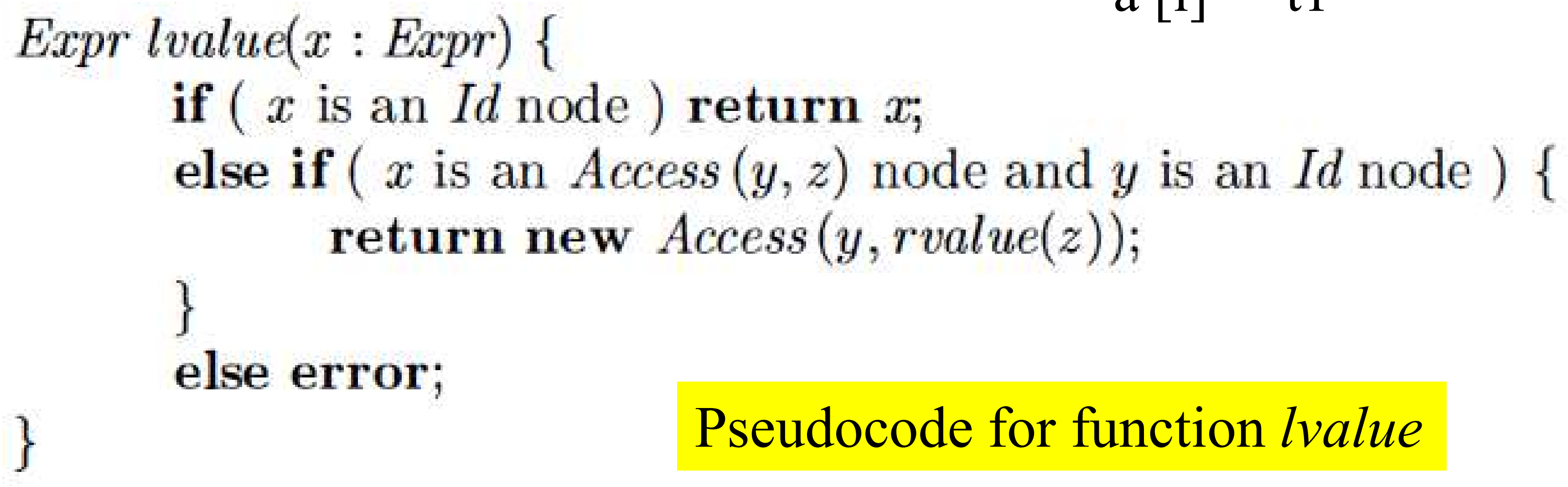
$\downarrow$

t3 = j - k

t2 = a [t3]

t1 = 2 * t2

a [i] = t1

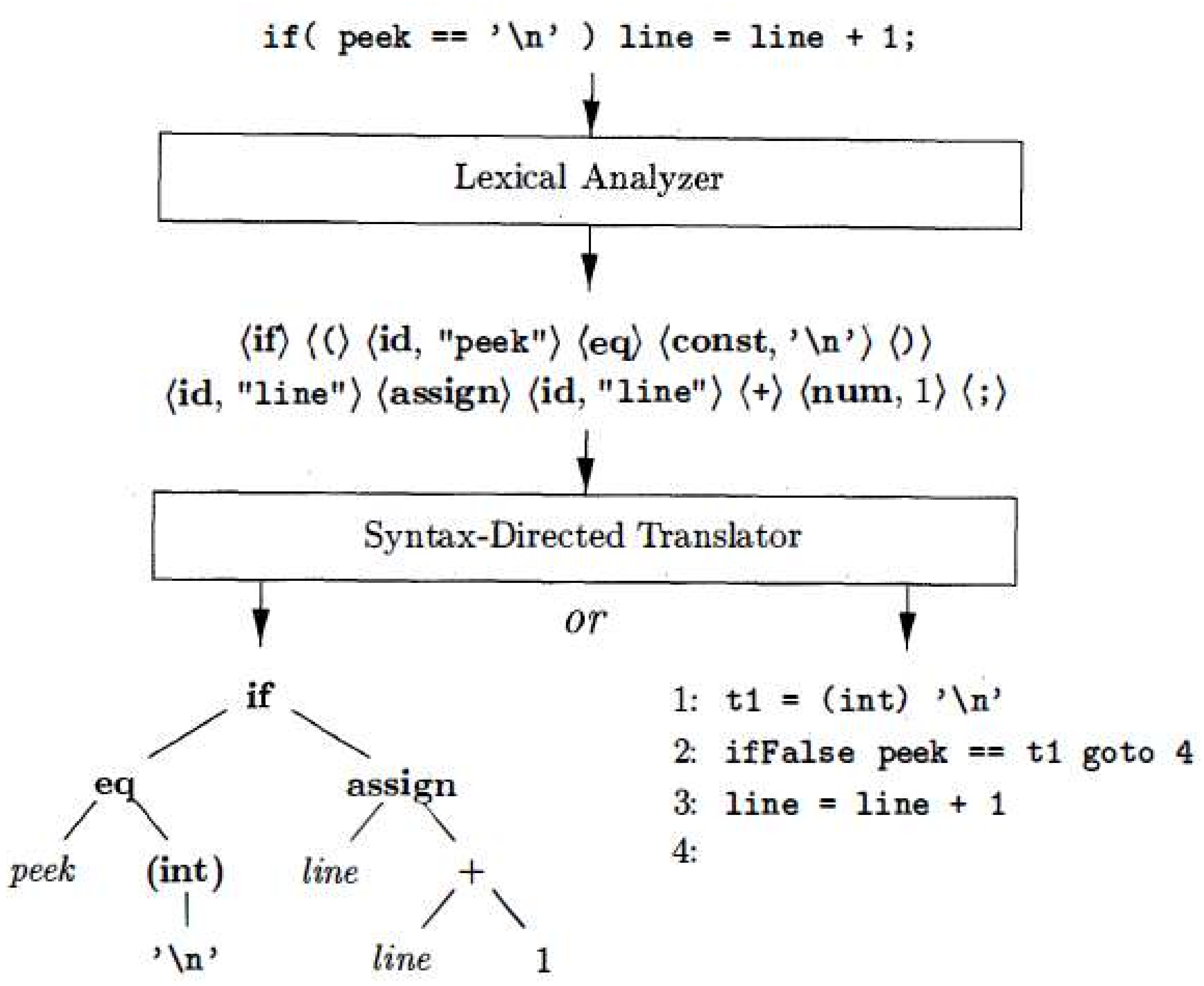


```
Expr lvalue(x : Expr) {
        if ( x is an Id node ) return x;
        else if ( x is an Access(y, z) node and y is an Id node ) {
                return new Access(y, rvalue(z));
        }
        else error;
}
```

Pseudocode for function *lvalue*

```
$Expr\ rvalue(x : Expr)$ {
        **if** ( $x$ is an $Id$ or a $Constant$ node ) **return** $x$;
        **else if** ( $x$ is an $Op$ ($\mathbf{op}, y, z$) or a $Rel$ ($\mathbf{op}, y, z$) node ) {
                $t$ = new temporary;
                emit string for $t$ = $rvalue(y)$ **op** $rvalue(z)$;
                **return** a new node for $t$;
        }
        **else if** ( $x$ is an $Access$ ($y, z$) node ) {
                $t$ = new temporary;
                call $lvalue(x)$, which returns $Access$ ($y, z'$);
                emit string for $t$ = $Access$ ($y, z'$);
                **return** a new node for $t$;
        }
        **else if** ( $x$ is an $Assign$ ($y, z$) node ) {
                $z'$ = $rvalue(z)$;
                emit string for $lvalue(y)$ = $z'$;
                **return** $z'$;
        }
}
```

Pseudocode for function *rvalue*



```
if( peek == '\n' ) line = line + 1;
```

⟨if⟩ ⟨(⟩ ⟨id, "peek"⟩ ⟨eq⟩ ⟨const, '\n'⟩ ⟨)⟩
⟨id, "line"⟩ ⟨assign⟩ ⟨id, "line"⟩ ⟨+⟩ ⟨num, 1⟩ ⟨;⟩

```
1: t1 = (int) '\n'
2: ifFalse peek == t1 goto 4
3: line = line + 1
4:
```