Chapter 2

A Simple Syntax-Directed Translator

Overview

- This chapter contains introductory material to chapters 3 to 6 of the Dragon book
- Developing a working Java program that translates representative programming language statements into three-address code, an intermediate representation
- Emphasis is on the front end of a compiler (lexical analysis, parsing, intermediate code generation)

Building a Simple Compiler

- Creating a syntax-directed translator that maps infix arithmetic expressions into postfix expressions

- Then, extending this translator to map code fragments into three-address code

- The working Java translator appears in Appendix A

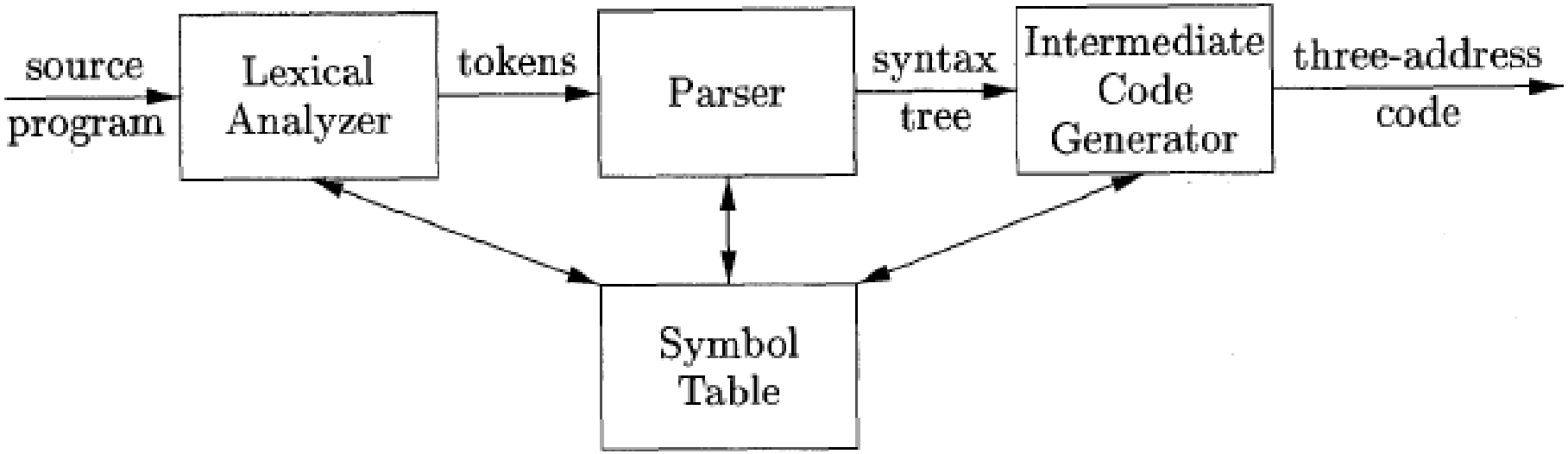
Output of a Simple Compiler

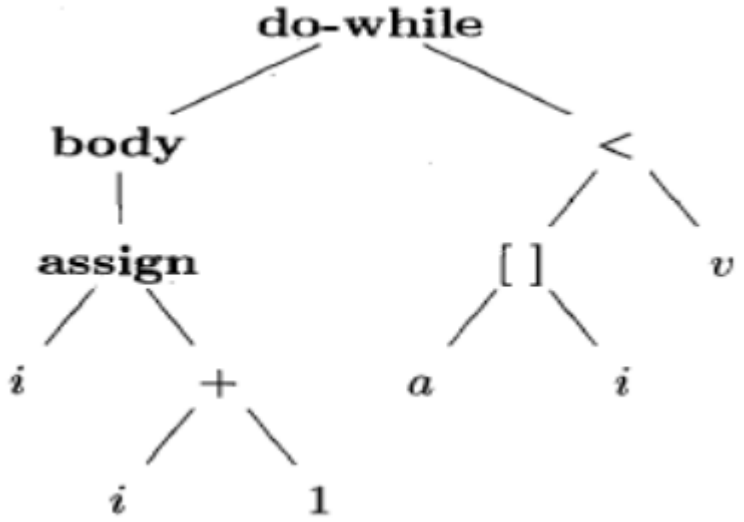

```
{
  int i; int j; float[100] a; float v; float x;

  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;
  }
}
```

```
 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:
```

A Model of a Compiler Front-end

mansoori@shirazu.ac.ir






```
do i = i+1; while ( a[i] < v );
```

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

Definition of Grammars

- Context-free grammar (CFG) 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

CFG 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:

$$list \rightarrow list + digit$$

$$list \rightarrow list - digit$$

$$list \rightarrow digit$$

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

$$list \rightarrow list + digit \mid list - digit \mid digit$$

Syntax Definition using CFG

- If statement in C:

  if (expression) statement **else** statement

  $stmt \rightarrow$ **if** ($expr$) $stmt$ **else** $stmt$

- List of parameters in a function call

  $call \rightarrow$ **id** ($optparams$)
  $optparams \rightarrow params \mid \varepsilon$
  $params \rightarrow params$, $param \mid param$

Derivation

- Given a CFG, we can determine the set of all *strings* (sequences of tokens) generated by the grammar using *derivation*
  - We begin with the start symbol
  - In each step, we replace one nonterminal in the current *sentential form* with one of the right-hand sides of a production for that nonterminal

Derivation for the Example Grammar

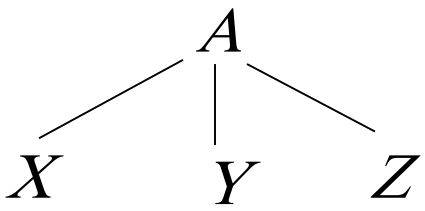
$$
\begin{aligned}
&\underline{list} \\
&\Rightarrow \underline{list} + digit \\
&\Rightarrow \underline{list} - digit + digit \\
&\Rightarrow \underline{digit} - digit + digit \\
&\Rightarrow \mathbf{9} - \underline{digit} + digit \\
&\Rightarrow \mathbf{9} - \mathbf{5} + \underline{digit} \\
&\Rightarrow \mathbf{9} - \mathbf{5} + \mathbf{2}
\end{aligned}
$$

- 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

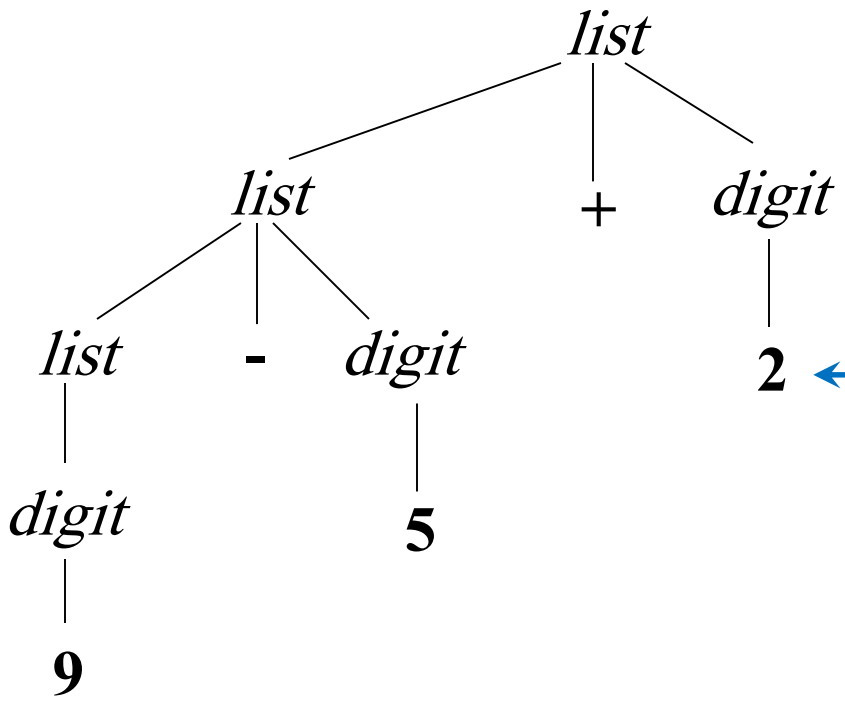
- The root of the tree is labeled by the start symbol
- Each leaf of the tree is labeled by a terminal (token) or ε (empty string)
- Each interior node is labeled by a nonterminal
- If $A \rightarrow X_1 X_2 \ldots X_n$ is a production, then node A has immediate children $X_1, X_2, \ldots, X_n$ where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

$A \rightarrow X\ Y\ Z$



Parse Tree for the Example Grammar

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



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

Ambiguity

Consider the following context-free grammar:

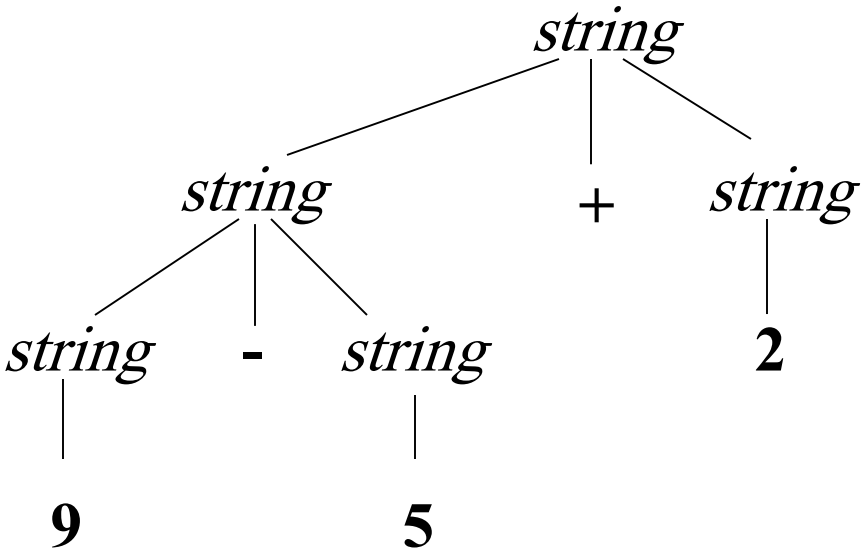
$$G = <\{string\}, \{+,-,\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5},\mathbf{6},\mathbf{7},\mathbf{8},\mathbf{9}\}, P, string>$$

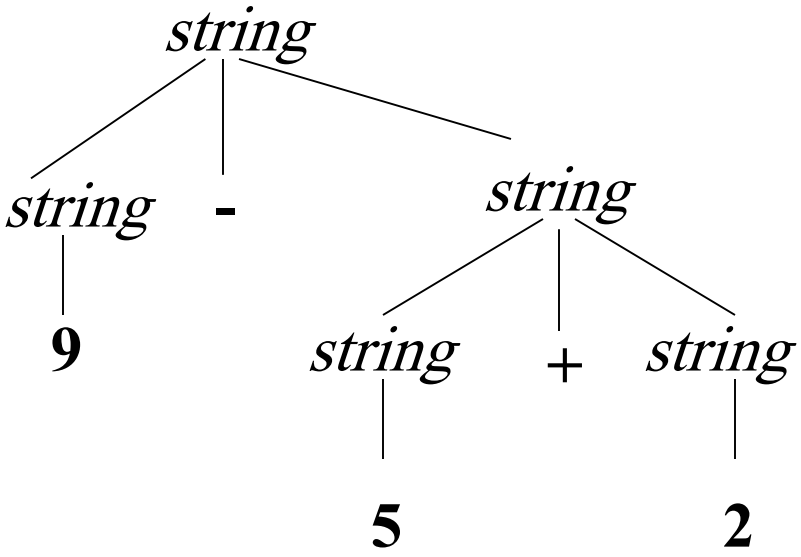
with production $P =$

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

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

Ambiguity



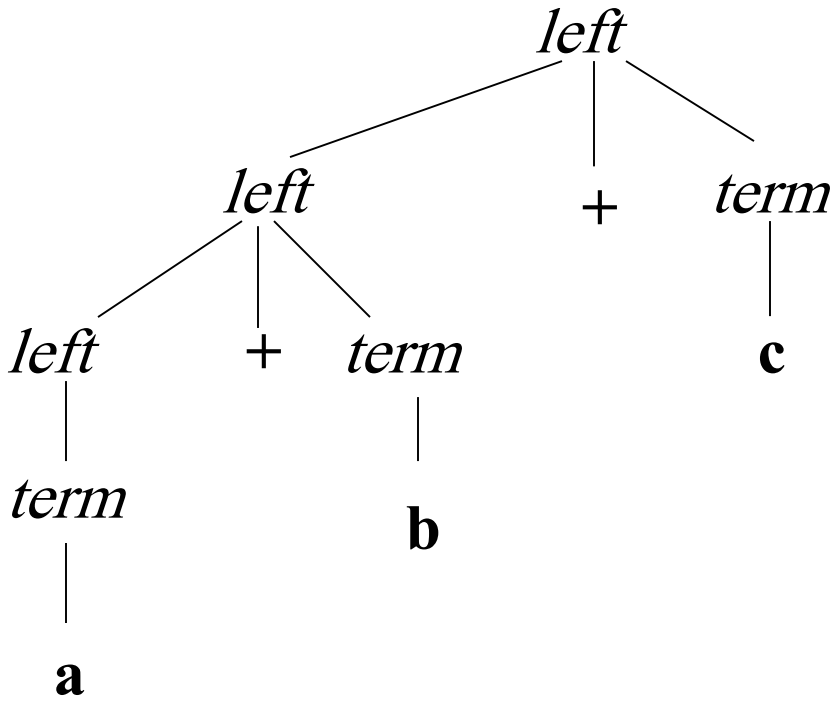


Associativity of Operators

Left-associative operators have left-recursive productions

$left \rightarrow left + term \mid term$

$term \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$



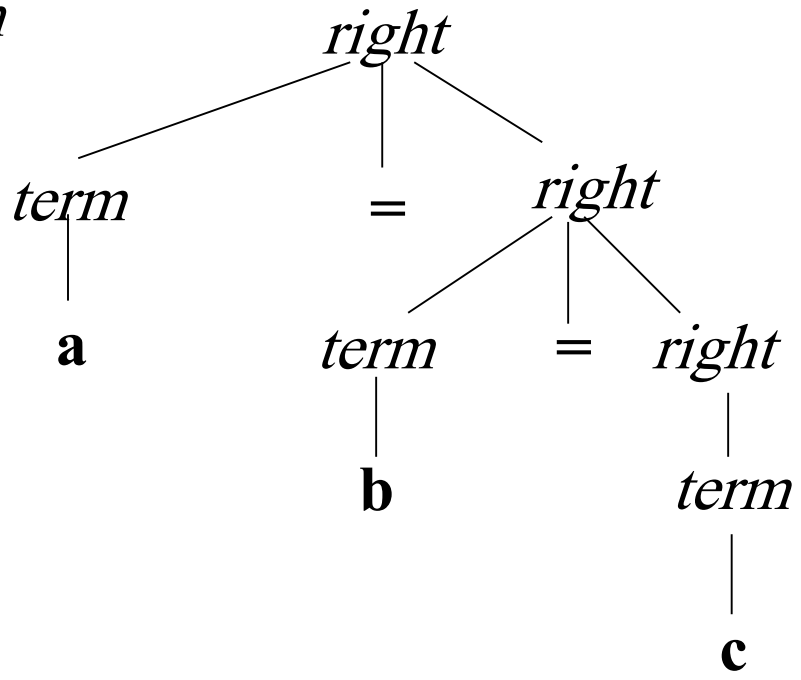
String **a+b+c** has the same meaning as **(a+b)+c**

Associativity of Operators

Right-associative operators have right-recursive productions

$right \rightarrow term = right \mid term$

$term \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$



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

Precedence of Operators

Table of precedence and associativity of operators:

operators (left-associative): + , - :: *expr*
Operators (left-associative): * , / :: *term*
basic units (digits and parenthesized expressions) :: *factor*

$$factor \rightarrow digit \mid (\ expr\ )$$

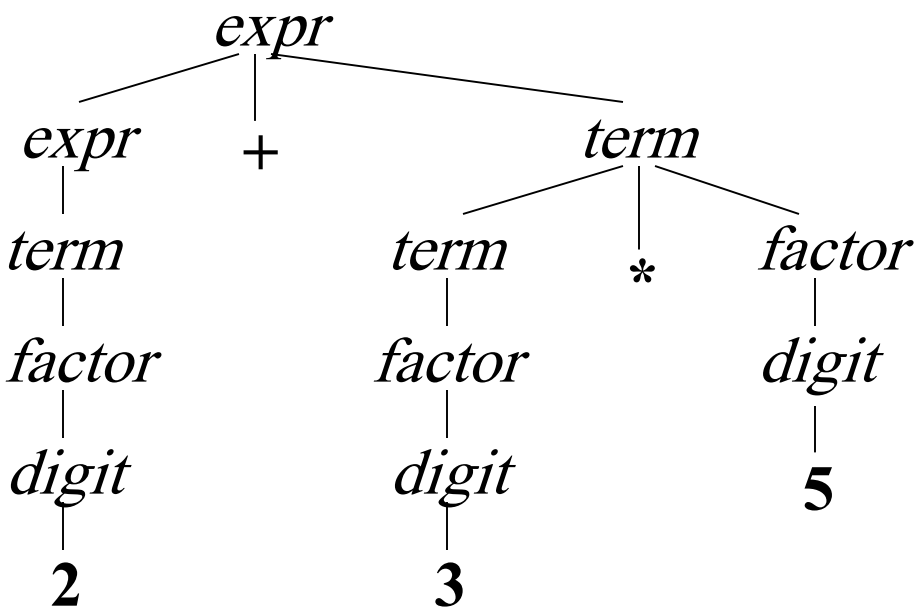
$$term \rightarrow term * factor \mid term / factor \mid factor$$

$$expr \rightarrow expr + term \mid expr - term \mid term$$

Operators with higher precedence "bind more tightly"

Precedence of Operators

expr → *expr* + *term* | *expr* - *term* | *term*
term → *term* * *factor* | *term* / *factor* | *factor*
factor → *digit* | (*expr*)
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**



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

Syntax of Statements

Keywords help to recognize statements
(except assignments and procedure calls):

$$
\begin{aligned}
\textit{stmt} \rightarrow\ & \textbf{id} = \textit{expr} \\
\mid\ & \textbf{if (}\ \textit{expr}\ \textbf{)}\ \textit{stmt} \\
\mid\ & \textbf{if (}\ \textit{expr}\ \textbf{)}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
\mid\ & \textbf{while (}\ \textit{expr}\ \textbf{)}\ \textit{stmt} \\
\mid\ & \textbf{do}\ \textit{stmt}\ \textbf{while (}\ \textit{expr}\ \textbf{) ;} \\
\mid\ & \{\ \textit{stmts}\ \}
\end{aligned}
$$

$$
\begin{aligned}
\textit{stmts} \rightarrow\ & \textit{stmts}\ \textit{stmt} \\
\mid\ & \varepsilon
\end{aligned}
$$

Parsing

- Parsing: *process of determining if a string of tokens can be generated by a grammar*
- For any CFG, 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 (efficient parsers can be constructed more easily by hand)
- *Bottom-up parsing* constructs a parse tree from leaves to root (can handle a larger class of grammars)

Top-down Construction of Parse Tree

- 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
   - Selecting the appropriate productions for *A* depends on the current terminal being scanned in the input, the *lookahead* symbol

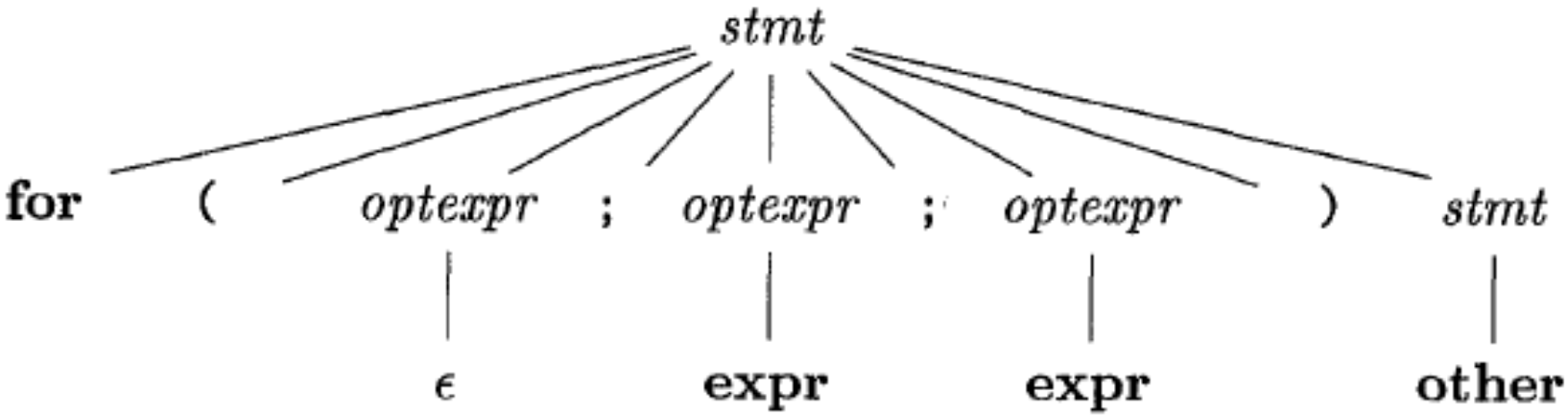
2. Find the next node at which a subtree is to be constructed, typically the leftmost unexpanded nonterminal of the tree

Top-down Construction of Parse Tree

stmt → **expr ;**
/ **if (expr)** *stmt*
/ **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*
/ **other**

optexpr → **expr**
/ **ε**

Parse tree of: **for (; expr ; expr) other**



Top-down Parsing

- For some grammars, the construction steps of *parse tree* can be implemented during a single left-to-right scan of the input string

- In general, the selection of a production for a nonterminal may involve trial and error (try a production and backtrack to try another production if the first is found to be unsuitable)
  - A production is unsuitable if, after using the production, the tree cannot be completed to match the input

Recursive Descent Parsing

- Is a top-down parsing method
  - Every nonterminal has one (*recursive*) procedure responsible for parsing the nonterminal's syntactic category of input *tokens*

$A \rightarrow X\ \mathbf{b}\ Z$

```
void A()      void X()      void Z()
{ … }         { … }         { … }
```

  - When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input *lookahead* information

$A \rightarrow \mathbf{b}\ X\ \mathbf{y}$
$\quad\quad | \quad \mathbf{c}\ Z$

```
void A()
{ if (lookahead==‘b’)
    …
  elseif (lookahead==‘c’)
    … }
```

Predictive Parsing

- Is a special form of recursive descent parsing where one lookahead token is sufficient to unambiguously determine the parse operations
- The sequence of procedure calls, during the analysis of an input string, implicitly defines a parse tree for the input

stmt → **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*

```
match(for); match('(');
optexpr(); match(';'); optexpr(); match(';'); optexpr();
match(')'); stmt();
```

Example Predictive Parsing (CFG)

stmt → **expr ;**
/ **if (expr)** *stmt*
/ **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*
/ **other**

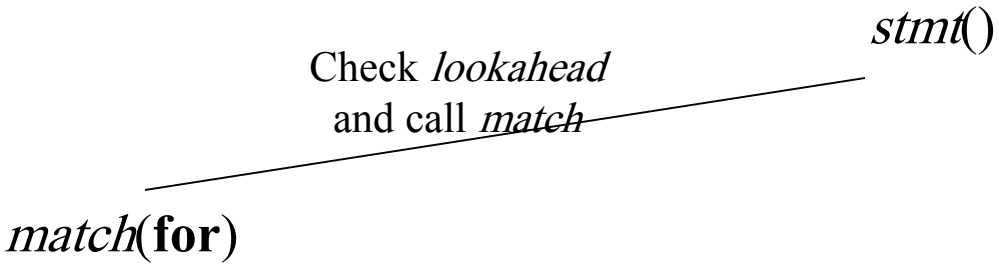
optexpr → **expr**
/ **ε**

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

Example Predictive Parsing (Code)


```
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);
}
```

Example Predictive Parser (Execution Step 1)

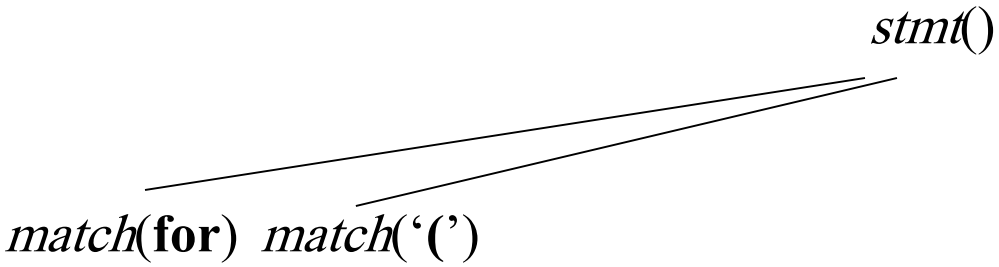


Input: **for (; expr ; expr) other**

lookahead

Example Predictive Parser (Execution Step 2)

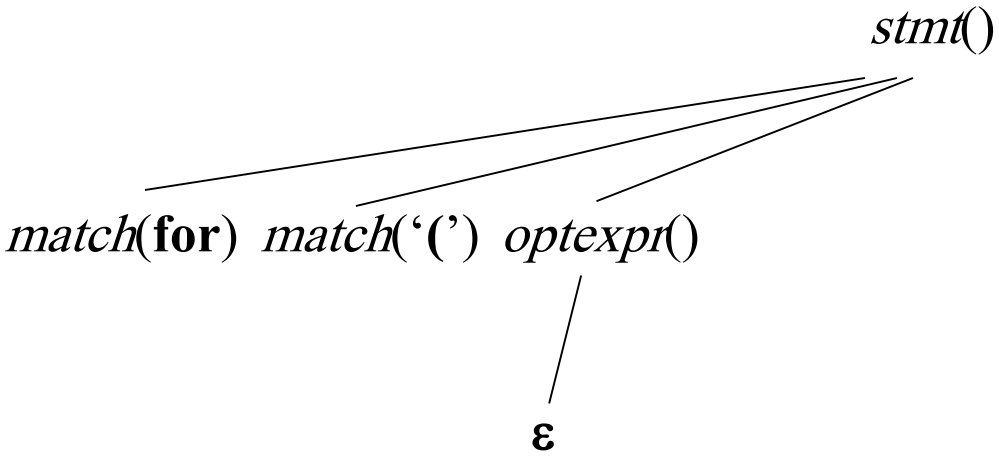
mansoori@shirazu.ac.ir



Input: **for (; expr ; expr) other**

lookahead

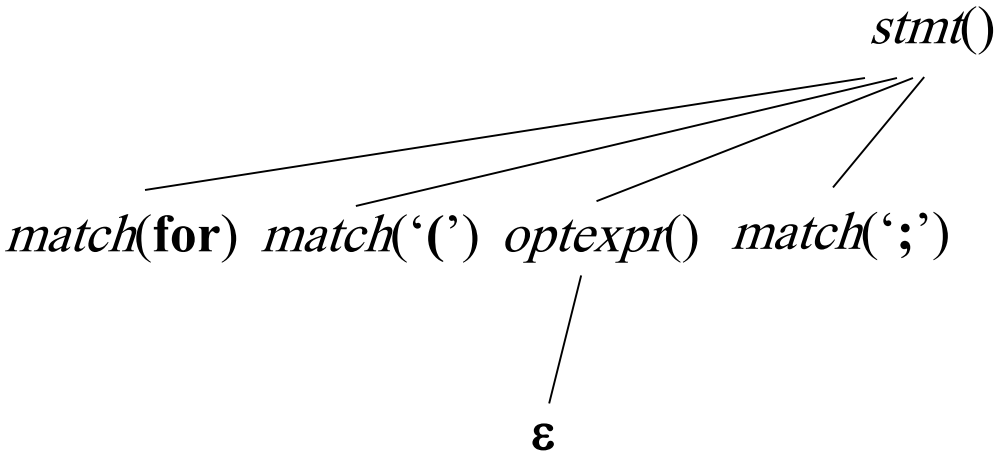
Example Predictive Parser (Execution Step 3)



Input: **for (; expr ; expr) other**

lookahead

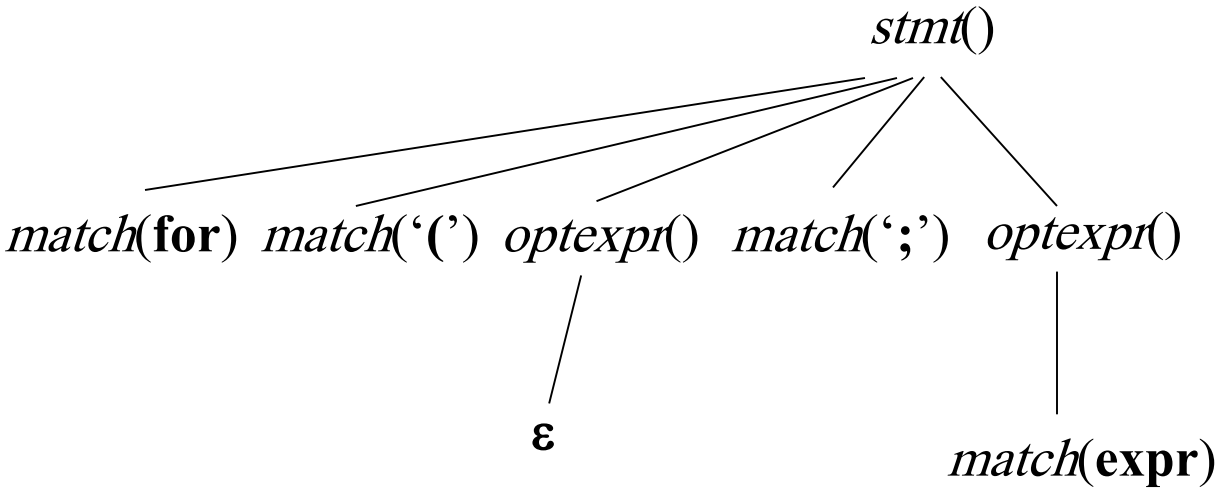
Example Predictive Parser (Execution Step 4)



Input: **for (; expr ; expr) other**

lookahead

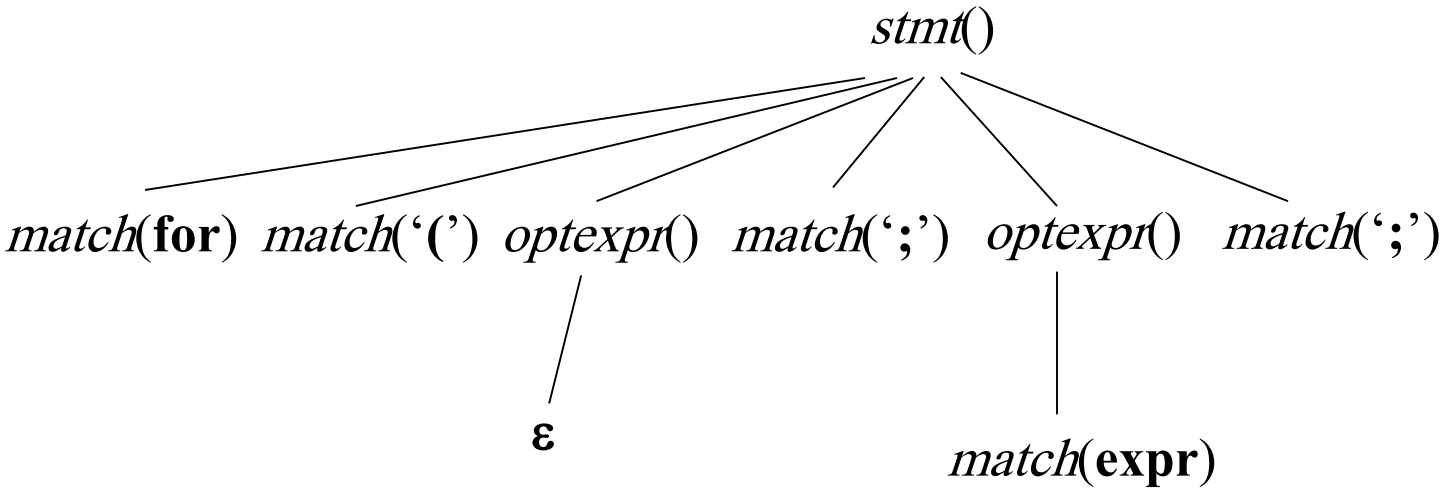
Example Predictive Parser (Execution Step 5)



Input: **for (; expr ; expr) other**

lookahead

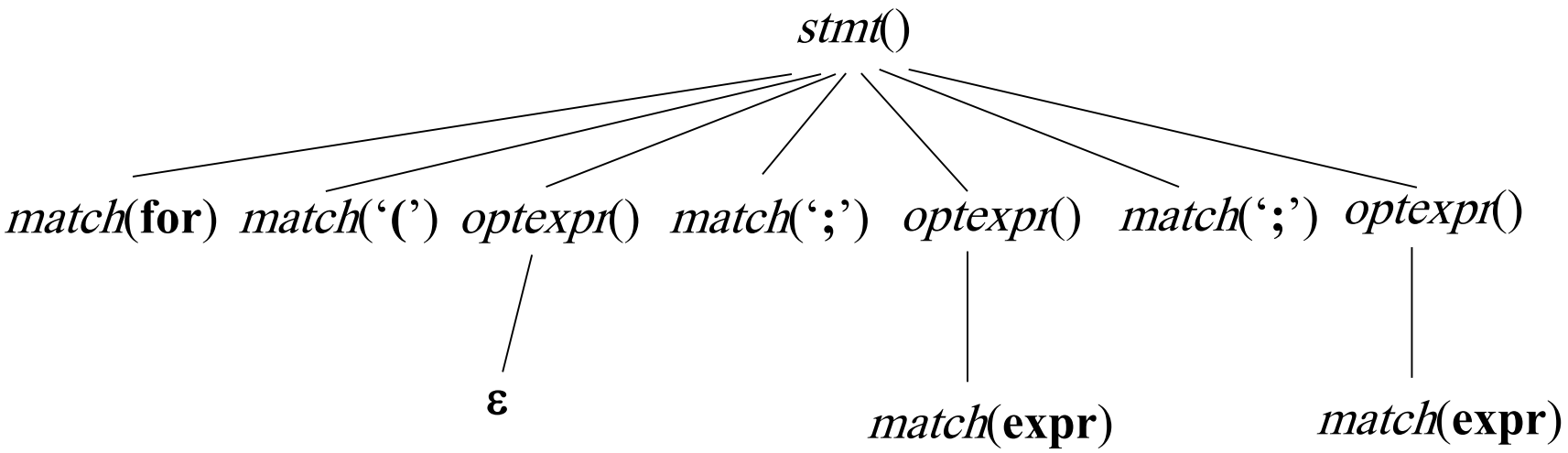
Example Predictive Parser (Execution Step 6)



Input: **for (; expr ; expr) other**

lookahead

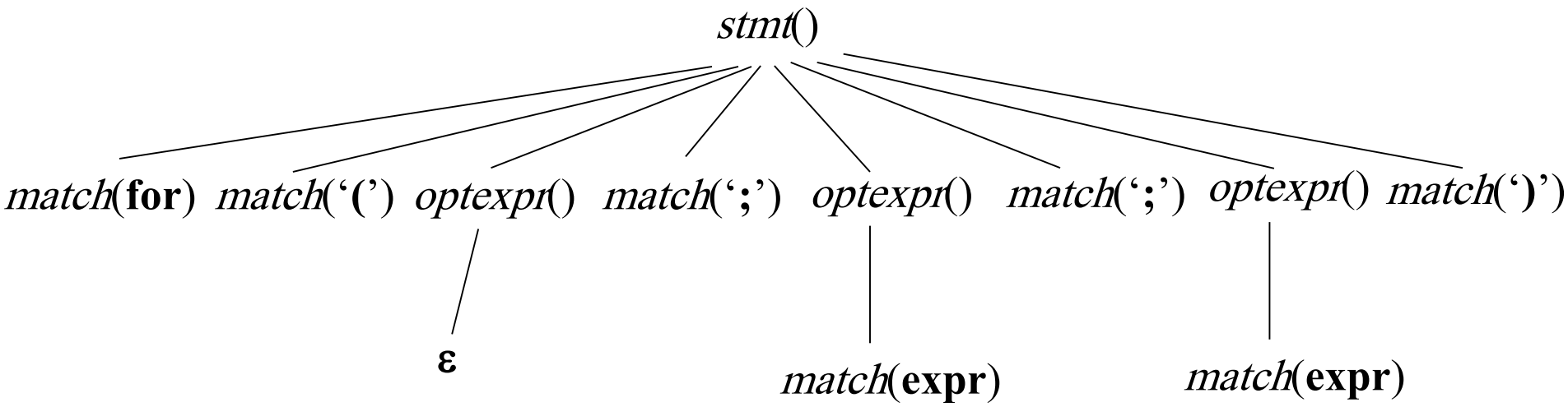
Example Predictive Parser (Execution Step 7)



Input: **for (; expr ; expr) other**

lookahead

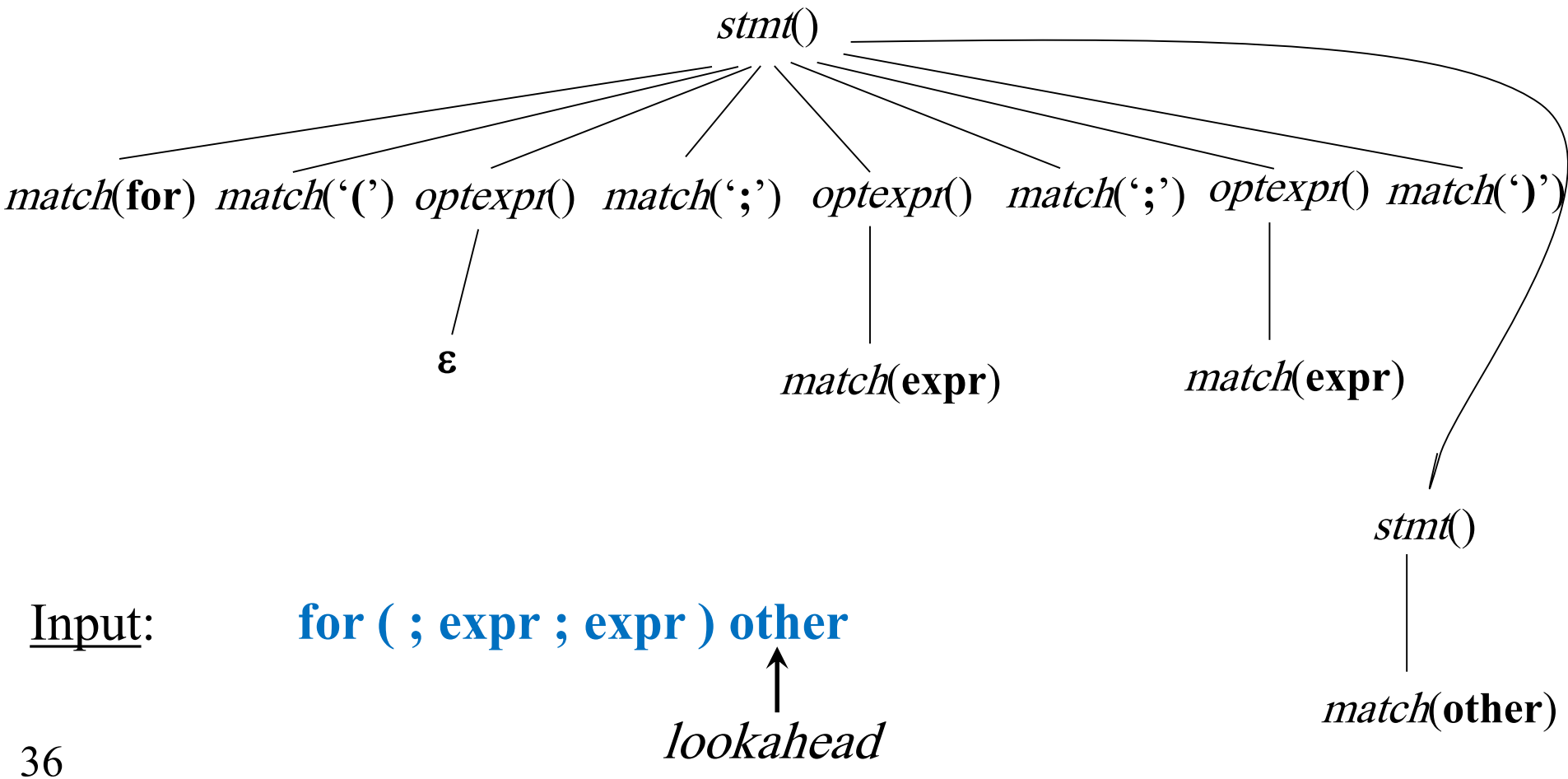
Example Predictive Parser (Execution Step 8)



Input: **for (; expr ; expr) other**

lookahead

Example Predictive Parser (Execution Step 9)



Input: **for (; expr ; expr) other**

lookahead

FIRST

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

stmt → **expr ;**
/**if (expr)** *stmt*
/**for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*
/**other**

optexpr → **expr**
/**ε**

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

Use FIRST to Design Predictive Parser

How to use FIRST set in order to design a predictive parser:

expr → *term rest*

rest → + *term rest*
| - *term rest*
| ε

```
void rest() {
  if (lookahead in FIRST(+ term rest) ) {
     match('+'); term(); rest() }
  else if (lookahead in FIRST(- term rest) ) {
     match('-'); term(); rest() }
  else return
}
```

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

$$A \rightarrow \alpha \,/\, \beta$$

Then, FIRST(α) and FIRST(β) must be disjoint for predictive parsing to work

Left Recursion in CFG

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

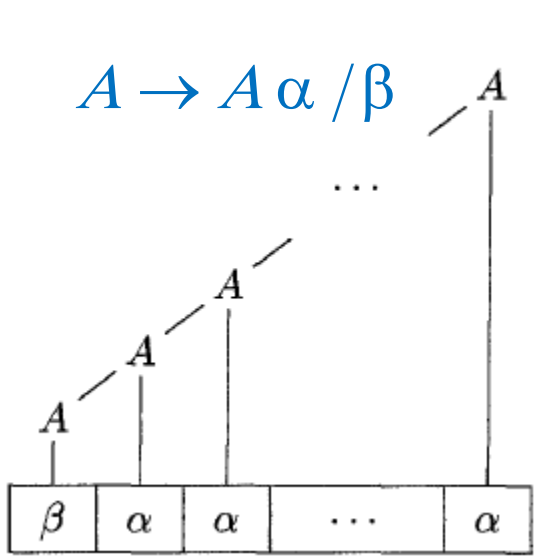
$$A \rightarrow A\,\alpha \,/\, \beta$$

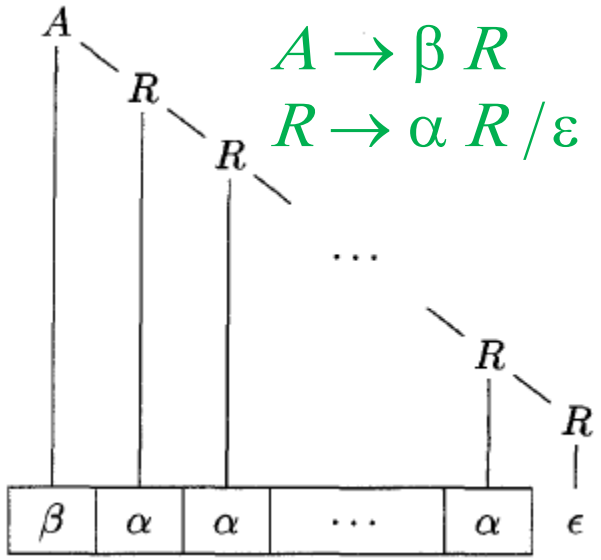
Each left-recursive production can be eliminated by rewriting the grammar using right-recursive productions

$$A \rightarrow \beta\, R$$
$$R \rightarrow \alpha\, R \,/\, \varepsilon$$

expr → *expr* + *term*
| *expr* - *term*
| *term*

expr → *term rest*
rest → + *term rest*
| - *term rest*
| ε





Syntax-Directed Translation

- Is done by attaching rules or program fragments to productions in a grammar

$expr \rightarrow expr_1 + term$

- In translation of infix expressions into postfix notation:

To translate *expr*:

translate $expr_1$;

translate *term;*

handle +;

Syntax-Directed Definition

- Uses a CFG 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 computations are completed, the attributes contain the translated form of the input

Attribute

- Is any quantity associated with a programming construct
  - Values
  - Data types of expressions
  - The number of instructions in the generated code
  - The location of the first instruction in the generated code for a construct

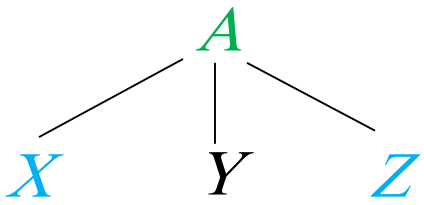
- Since nonterminals and terminals represent programming constructs, attributes are associated with nonterminals and terminals: *X.x* , *A.a* , *term.value*

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

$$A \rightarrow X\,Y\,Z$$

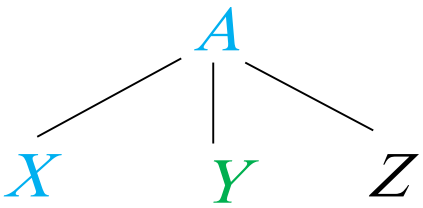
$$A.a = f(X.x,\ Z.z)$$



  - Inherited*:* if its value at a parse-tree node is determined by the parent (by enforcing the parent's semantic rules)

$$A \rightarrow X\,Y\,Z$$

$$Y.y = g(A.a,\ X.x)$$



Semantic Rules

- Rules attached to the productions of a grammar
- These rules describe how the attributes are computed at the nodes of parse tree
- For a given input string *x*, construct a parse tree for *x*
- Then, apply the semantic rules to evaluate attributes at each node in the parse tree

Syntax-Directed Definition

A syntax-directed definition associates

1. With each grammar symbol, a set of attributes, and
2. With each production, a set of semantic rules for computing the values of the attributes associated with the symbols appearing in the production

Syntax-Directed Definition Example

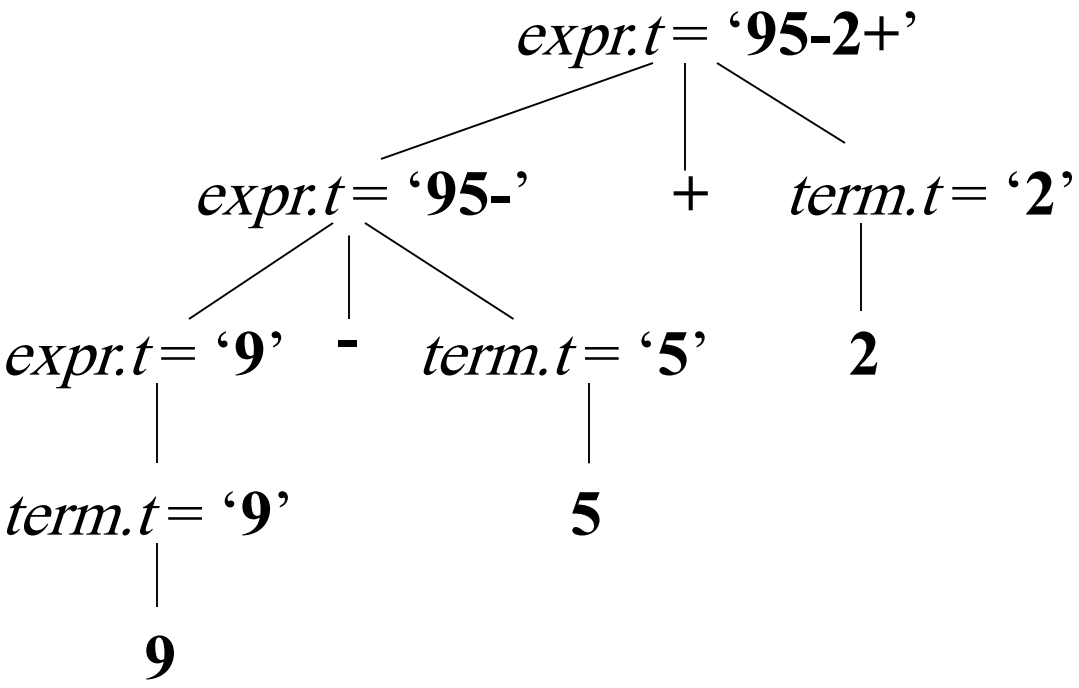
Infix to postfix translation:

String concat operator

Production	Semantic rules
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \,\|\| \, term.t \,\|\| \, \text{`+'}$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \,\|\| \, term.t \,\|\| \, \text{`-'}$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow \mathbf{0}$	$term.t = \text{`0'}$
$term \rightarrow \mathbf{1}$	$term.t = \text{`1'}$
…	…
$term \rightarrow \mathbf{9}$	$term.t = \text{`9'}$

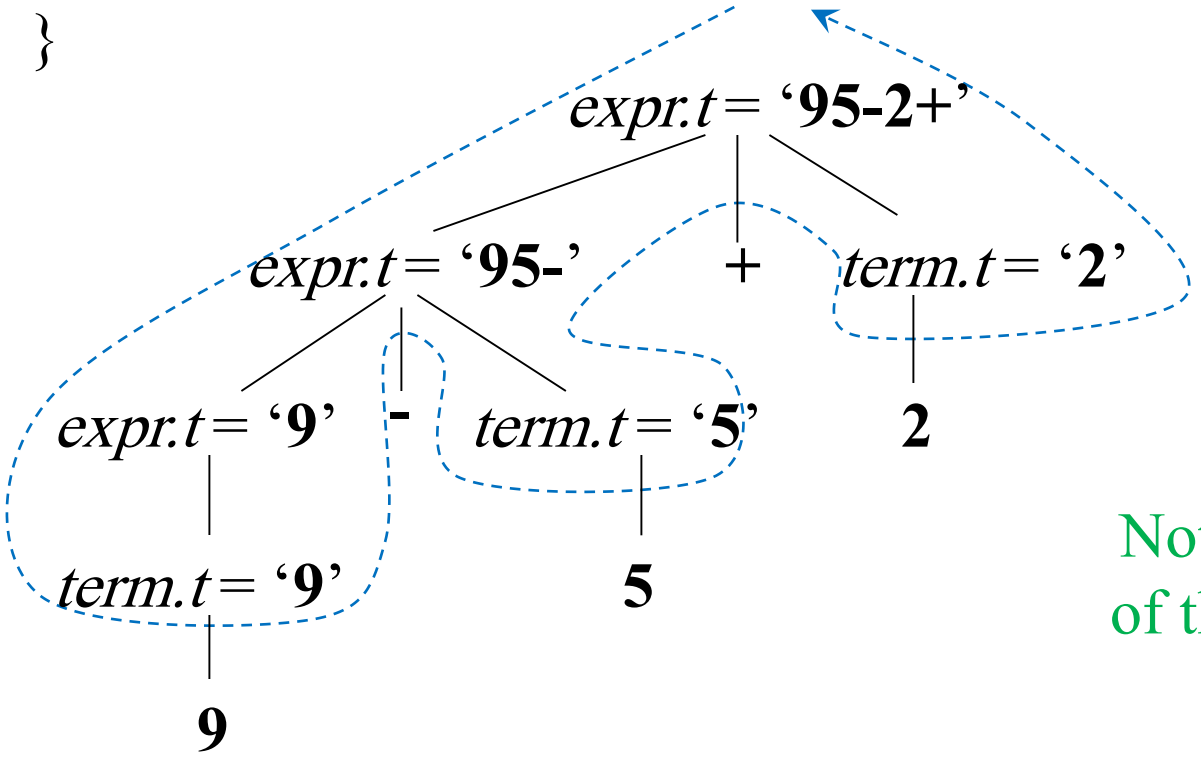
Annotated Parse Tree

A parse tree showing the attribute values at each node is called an annotated parse tree



Depth-first Postorder Traversals


```
procedure visit(node N)
{
    for ( each child C of N, from left to right)
        visit(C);
    evaluate semantic rules at node N;
}
```



Bottom-up traversal

Note: all attributes are of the synthesized type

Translation Schemes

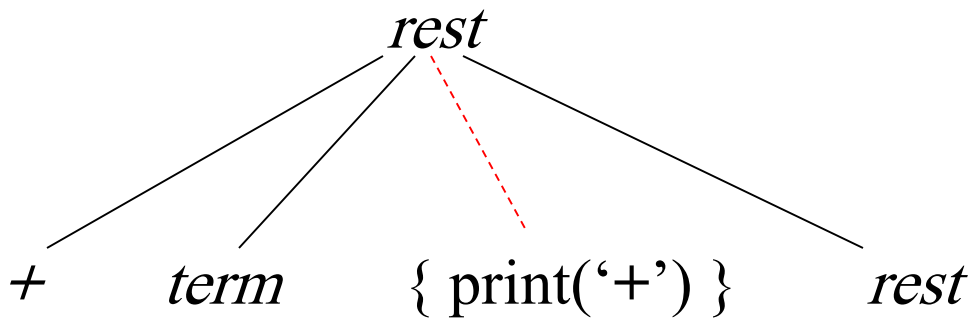
- In syntax-directed definition,
  - a translation is built up by evaluating the attributes of the nodes in the parse tree
- In translation scheme,
  - the same translation is produced incrementally, by executing program fragments (semantic actions)
  - is like a syntax-directed definition, except that the order of evaluation of the semantic rules is explicitly specified

Translation Schemes

- A translation scheme is a CFG embedded with semantic actions

$$rest \rightarrow + \; term \; \{ \text{print}(\text{`+'}) \} \; rest_1$$

Embedded
semantic action

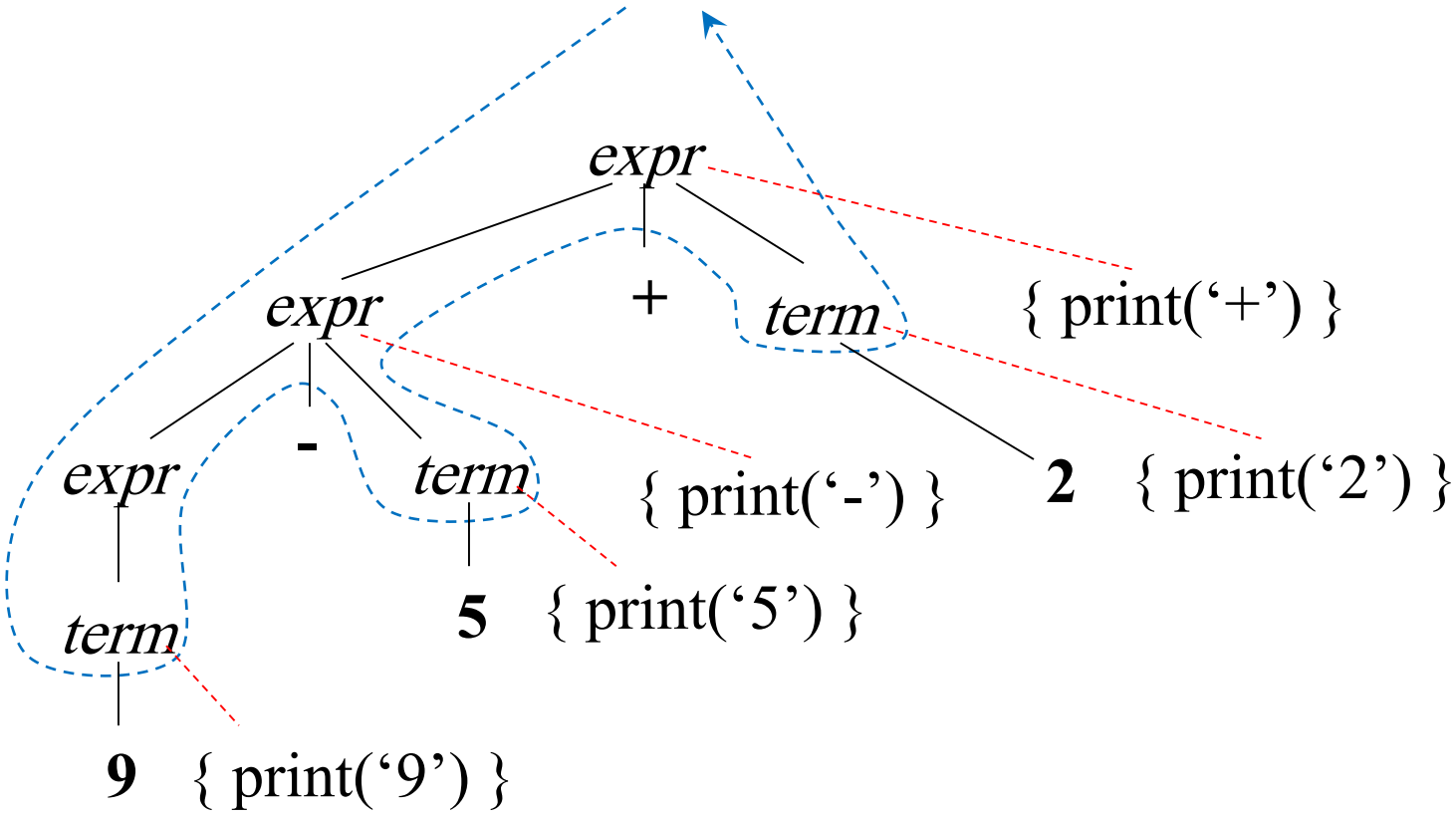


Example Translation Scheme

$expr \rightarrow expr_1 + term$	{ print('+') }
$expr \rightarrow expr_1 - term$	{ print('-') }
$expr \rightarrow term$	
$term \rightarrow \mathbf{0}$	{ print('0') }
$term \rightarrow \mathbf{1}$	{ print('1') }
…	…
$term \rightarrow \mathbf{9}$	{ print('9') }

Example Translation Scheme

mansoori@shirazu.ac.ir



Traversing annotated parse tree translates **9-5+2** into postfix **95-2+**

From Predictive Parser to Translation Scheme

1. Construct a predictive parser, ignoring the actions in productions
2. Copy the actions from the translation scheme into the parser
   - If an action appears after grammar symbol X in production p, then it is copied after the implementation of X in the code for p
   - Otherwise, if it appears at the beginning of the production, then it is copied just before the code for the production body

Left Recursion in Translation Scheme

When eliminating left-recursive from translation scheme, the semantic actions are handled as terminals

$A \rightarrow A\,\alpha\ \{a\}\ /\beta$

$A \rightarrow \beta\ R$
$R \rightarrow \alpha\ \{a\}\ R\ /\varepsilon$

expr $\rightarrow$ *expr* + *term* { print('+') }
/ *term*

expr $\rightarrow$ *term rest*
rest $\rightarrow$ + *term* { print('+') } *rest*
| ε

A Translator for Simple Expressions

$expr \rightarrow expr + term$ { 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') }

After left recursion elimination:

$expr \rightarrow term\ rest$
$rest \rightarrow + term$ { print('+') } $rest \mid - term$ { print('-') } $rest \mid \varepsilon$
$term \rightarrow \mathbf{0}$ { print('0') }
$term \rightarrow \mathbf{1}$ { print('1') }
…
$term \rightarrow \mathbf{9}$ { print('9') }

Code of Translator

expr → *term rest*

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

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

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

term → **0** { print('0') }
term → **1** { print('1') }
…
term → **9** { print('9') }

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

Code of Translator

expr → *term rest*

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

term → **0** { print('0') }
term → **1** { print('1') }
...
term → **9** { 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);
}
```

Complete Code in Java


```
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;
        }
    }

    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');
    }
}
```

Tokens **num** and **id** in Expressions

expr	→	*expr* + *term*	{ print('+') }
	|	*expr* − *term*	{ print('−') }
	|	*term*	
term	→	*term* * *factor*	{ print('*') }
	|	*term* / *factor*	{ print('/') }
	|	*factor*	
factor	→	(*expr*)	
	|	**num**	{ print(**num**.*value*) }
	|	**id**	{ print(**id**.*lexeme*) }

Adding a Lexical Analyzer

- Typical tasks of the lexical analyzer:
  - Read characters from input and group into tokens
  - Remove white space and comments
  - Encode numbers and constants as tokens
  - Recognize keywords
  - Recognize identifiers and store identifier names in a global symbol table
    - A sequence of input characters that comprises a single token is called a lexeme

Removal of White Space and Comments

1. Lexical analyzer eliminates
   - White space: blank, tab, newline
   - Comments

```
for ( ; ; peek = next input character ) {
        if ( peek is a blank or a tab ) do nothing;
        else if ( peek is a newline ) line = line+1;
        else break;
}
```

2. Modifying the grammar to incorporate white space

Reading Ahead

- Lexical analyzer may need to read ahead some characters before it can decide on the token
  - In C or Java, read ahead after '>'
    - If next character = , token: 'greater than or equal to'
    - Otherwise, token: 'greater than'
- Lexical analyzer needs an input buffer from which it can read and push back characters

Recognizing Constants

- Lexical analyzer
  - Collects the characters into integers and computes their numerical value
  - Passes to the parser a token consisting of the terminal **num** along with an integer-valued attribute computed from the digits: <**num**,25>

```
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⟩;
}
```

Recognizing Keywords and Identifiers

- Keywords: fixed character strings used to identify constructs: **for**, **do**, **if**, …
- Identifiers (**id**): character strings used to name variables, arrays, functions, …
- Grammars routinely treat identifiers as terminals to simplify the parser

count = count + increment; **id** = **id** + **id** ;

- The token for **id** has an attribute that holds the lexeme

<**id**, "count"> <=> <**id**, "count"> <+> <**id**, "increment"> <;>

Recognizing Keywords and Identifiers

- Keywords generally satisfy the rules for forming identifiers, so lexical analyzer should decide when
  - A lexeme forms a keyword
  - A lexeme forms an identifier
- If keywords are reserved
  - They cannot be used as identifiers
  - Then, a character string forms an identifier only if it is not a keyword

Using Symbol Table

- Single representation
  - Symbol table can insulate the compiler from representation of strings, so compiler phases can work with references to the table
  - References can be manipulated more efficiently than the strings themselves
- Reserved words
  - Reserved words are implemented by initializing the symbol table with them
  - When the lexical analyzer reads a lexeme of identifier, it first checks whether the lexeme is in the table
    - If so, it returns the token from the table
    - Otherwise, it returns a token of **id**

Using Symbol Table

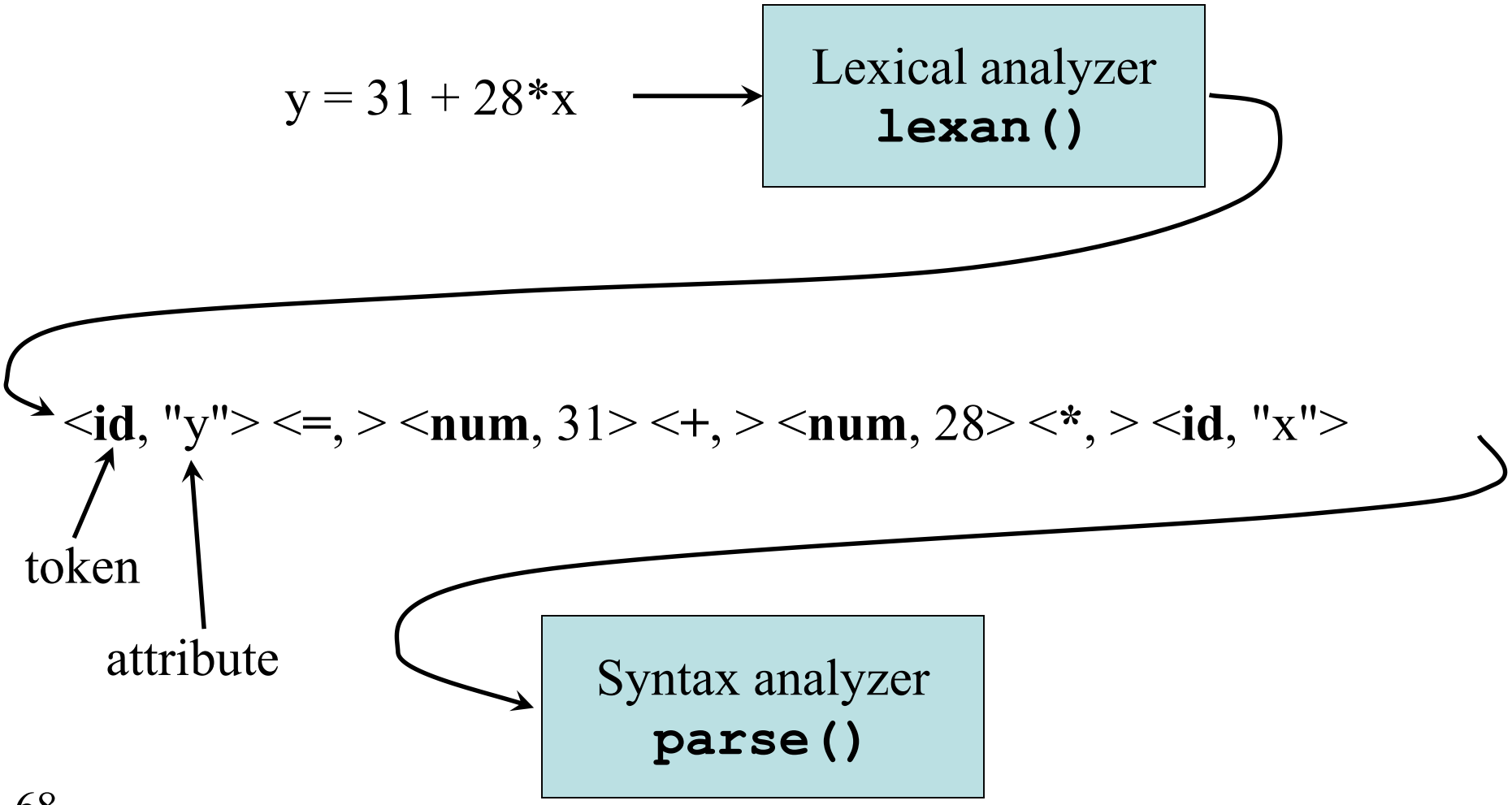
- Java: a symbol table is implemented as a hash table:

  Hashtable words = **new** *Hashtable*();

- Sets up words as a default hash table that maps keys (lexemes) to values (tokens)

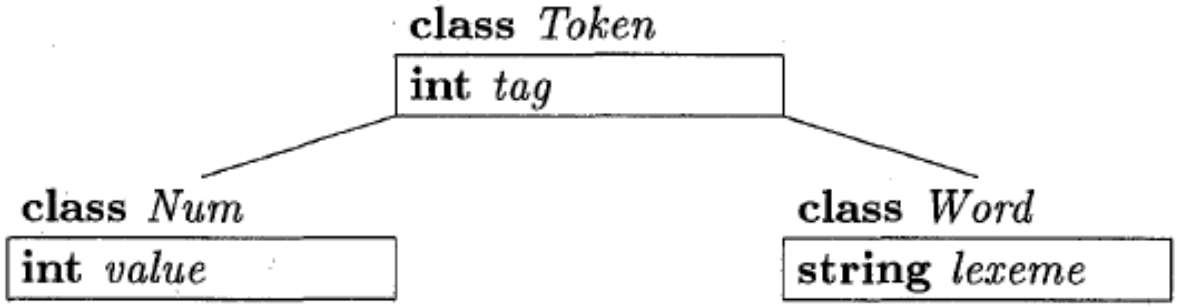
```
if ( peek holds a letter ) {
        collect letters or digits into a buffer b;
        s = string formed from the characters in b;
        w = token returned by words.get(s);
        if ( w is not null ) return w;
        else {
                Enter the key-value pair (s, ⟨id, s⟩) into words
 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

The Lexical Analyzer



Java Package for Lexical Analysis

- The package `lexer` has
  - classes for tokens
  - a class `lexer` containing function `scan`



- Class `Token` has a field `tag` that is used for parsing decisions
  - Subclass `Num` adds a field `value` for an integer value
  - Subclass `Word` adds a field `lexeme` for reserved words and identifiers

Java Package for Lexical Analysis


```
1) package lexer;                          // File Token.java
2) public class Token {
3)     public final int tag;
4)     public Token(int t) { tag = t; }
5) }
```

- The constructor `Token` is used to create token objects

  new `Token('+')`

- Where the pseudocode has terminals like **num** and **id**, the Java code uses integer constants

Java Package for Lexical Analysis


```
1) package lexer;                          // File Tag.java
2) public class Tag {
3)     public final static int
4)         NUM = 256, ID = 257, TRUE = 258, FALSE = 259;
5) }
```

```
1) package lexer;                          // File Num.java
2) public class Num extends Token {
3)     public final int value;
4)     public Num(int v) { super(Tag.NUM); value = v; }
5) }
```

```
1) package lexer;                          // File Word.java
2) public class Word extends Token {
3)     public final String lexeme;
4)     public Word(int t, String s) {
5)         super(t); lexeme = new String(s);
6)     }
7) }
```

Java Package for Lexical Analysis


```
1)  package lexer;                              // File Lexer.java
2)  import java.io.*; import java.util.*;
3)  public class Lexer {
4)     public int line = 1;
5)     private char peek = ' ';
6)     private Hashtable words = new Hashtable();
7)     void reserve(Word t) { words.put(t.lexeme, t); }
8)     public Lexer() {
9)        reserve( new Word(Tag.TRUE,  "true")  );
10)       reserve( new Word(Tag.FALSE, "false") );
11)    }
12)    public Token scan() throws IOException {
13)       for( ; ; peek = (char)System.in.read() ) {
14)          if( peek == ' ' || peek == '\t' ) continue;
15)          else if( peek == '\n' ) line = line + 1;
16)          else break;
17)       }
```

Java Package for Lexical Analysis


```
18)         if( Character.isDigit(peek) ) {
19)             int v = 0;
20)             do {
21)                 v = 10*v + Character.digit(peek, 10);
22)                 peek = (char)System.in.read();
23)             } while( Character.isDigit(peek) );
24)             return new Num(v);
25)         }
26)         if( Character.isLetter(peek) ) {
27)             StringBuffer b = new StringBuffer();
28)             do {
29)                 b.append(peek);
30)                 peek = (char)System.in.read();
31)             } while( Character.isLetterOrDigit(peek) );
32)             String s = b.toString();
33)             Word w = (Word)words.get(s);
34)             if( w != null ) return w;
35)             w = new Word(Tag.ID, s);
36)             words.put(s, w);
37)             return w;
38)         }
39)         Token t = new Token(peek);
40)         peek = ' ';
41)         return t;
42)     }
43) }
```

Symbol Tables

- The information in the symbol table
  - is collected by the analysis phases
  - and used by the synthesis phases to generate the target code
- Entries in symbol table contain information about an identifier
  - its lexeme
  - its type
  - its position in storage
  - …
- Symbol tables need to support multiple declarations for an identifier

Symbol Tables

- To implement scopes, a separate symbol table for each scope is set up
  - a program block with declarations
  - a class with an entry for each field and method
- Example of just key constructs that touch symbol tables: *blocks*, *declarations*, *factors*
- A program: *blocks* with optional *declarations* and *statements* of single identifiers (represent use of identifiers)

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

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

Symbol Table per Scope

- If blocks can be nested, several declarations of the same identifier can appear within a single block

$$block \rightarrow \text{'}\{\text{'}\ decls\ stmts\ \text{'}\}\text{'}$$

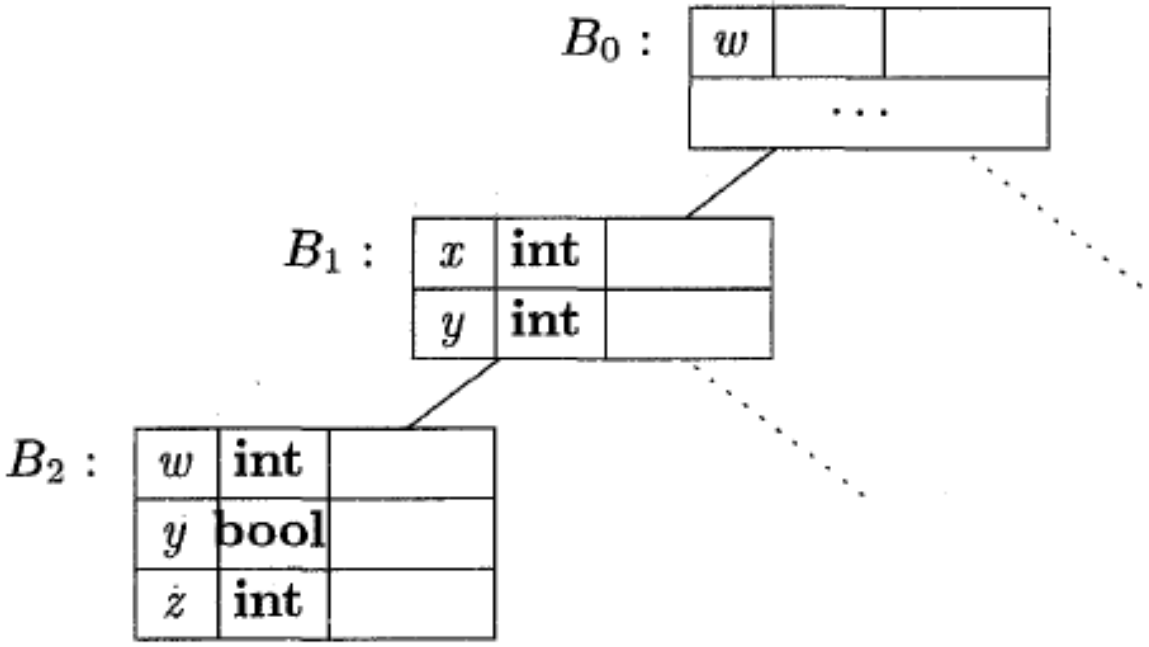
stmts can generate a block

- Most-closely nested rule for blocks: identifier x is in the scope of the most-closely nested declaration of x

```
1)  {    int x1; int y1;
2)       {    int w2; bool y2; int z2;
3)            ··· w2 ···; ··· x1 ···; ··· y2 ···; ··· z2 ···;
4)       }
5)         ··· w0 ···; ··· x1 ···; ··· y1 ···;
6)  }
```

Chained Symbol Tables

- The most-closely nested rule for blocks can be implemented by chaining symbol tables
  - Table for a nested block points to the table for its enclosing block



Java Imp. of Chained Symbol Tables


```
1)  package symbols;                          // File Env.java
2)  import java.util.*;
3)  public class Env {
4)     private Hashtable table;
5)     protected Env prev;

6)     public Env(Env p) {
7)        table = new Hashtable(); prev = p;
8)     }

9)     public void put(String s, Symbol sym) {
10)       table.put(s, sym);
11)    }

12)    public Symbol get(String s) {
13)       for( Env e = this; e != null; e = e.prev ) {
14)          Symbol found = (Symbol)(e.table.get(s));
15)          if( found != null ) return found;
16)       }
17)       return null;
18)    }
19) }
```

The Use of Symbol Tables

program	→	*block*
block	→	'{'
		decls stmts '}'
decls	→	*decls decl* | ϵ
decl	→	**type id ;**
stmts	→	*stmts stmt* | ϵ
stmt	→	*block*
	|	*factor* ;
factor	→	**id**

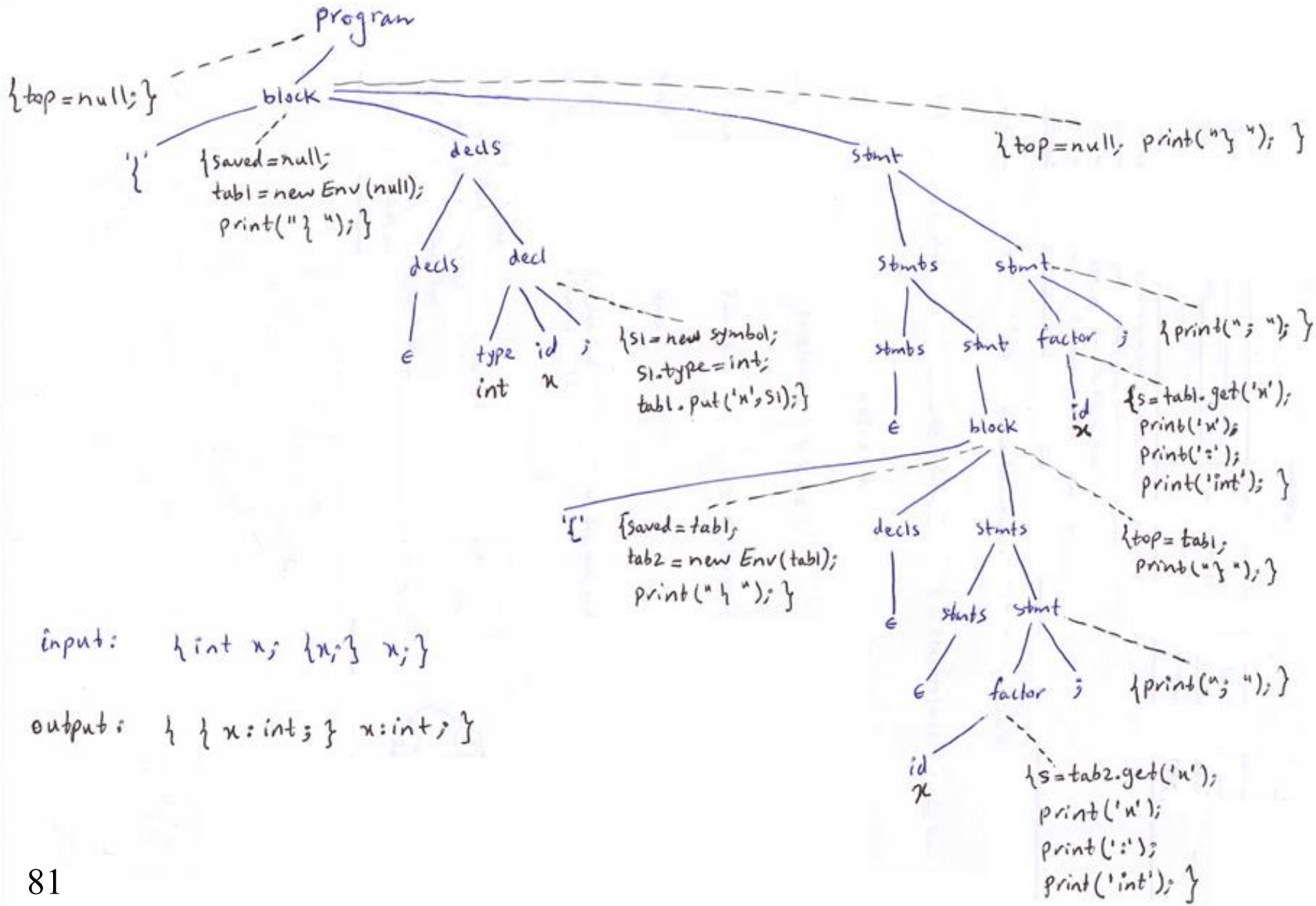

```
{ int x; char y; { bool y; x; y; } x; y; }   { { x:int; y:bool; } x:int; y:char; }
```

The Use of Symbol Tables

$$
\begin{array}{rcl}
\textit{program} & \rightarrow & \{\ \textit{top} = \textbf{null};\ \}\ \textit{block} \\
\textit{block} & \rightarrow & \text{'\{'}\ \{\ \textit{saved} = \textit{top};\ \textit{top} = \textbf{new}\ \textit{Env}(\textit{top});\ \text{print("\{ ");}\ \} \\
 & & \textit{decls}\ \textit{stmts}\ \text{'\}'}\quad \{\ \textit{top} = \textit{saved};\ \text{print("\} ");}\ \} \\
\textit{decls} & \rightarrow & \textit{decls}\ \textit{decl}\ \mid\ \epsilon \\
\textit{decl} & \rightarrow & \textbf{type id}\ ;\ \{\ \textit{s} = \textbf{new}\ \textit{Symbol};\ \textit{s.type} = \textbf{type}.\textit{lexeme} \\
 & & \qquad\qquad \textit{top.put}(\textbf{id}.\textit{lexeme},\ \textit{s});\ \} \\
\textit{stmts} & \rightarrow & \textit{stmts}\ \textit{stmt}\ \mid\ \epsilon \\
\textit{stmt} & \rightarrow & \textit{block} \\
 & \mid & \textit{factor}\ ; \qquad\qquad \{\ \text{print("; ");}\ \} \\
\textit{factor} & \rightarrow & \textbf{id} \qquad\qquad \{\ \textit{s} = \textit{top.get}(\textbf{id}.\textit{lexeme});\ \text{print}(\textbf{id}.\textit{lexeme}); \\
 & & \qquad\qquad \text{print(":");}\ \text{print}(\textit{s.type});\ \}
\end{array}
$$


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

The Use of Symbol Tables



Intermediate Code Generation

- The front-end of a compiler constructs an intermediate representation of the source program from which the back-end generates the target program

- Intermediate representations for:
  - Expressions
  - Statements

Two Intermediate Representations

- Tree representations: syntax tree
  - Syntax-tree nodes represent significant programming constructs
  - Information is added to the nodes in the form of attributes associated with the nodes

- Linear representations: three-address code
  - A sequence of elementary program steps
  - Unlike the tree, there is no hierarchical structure
  - Three-address code can be optimized

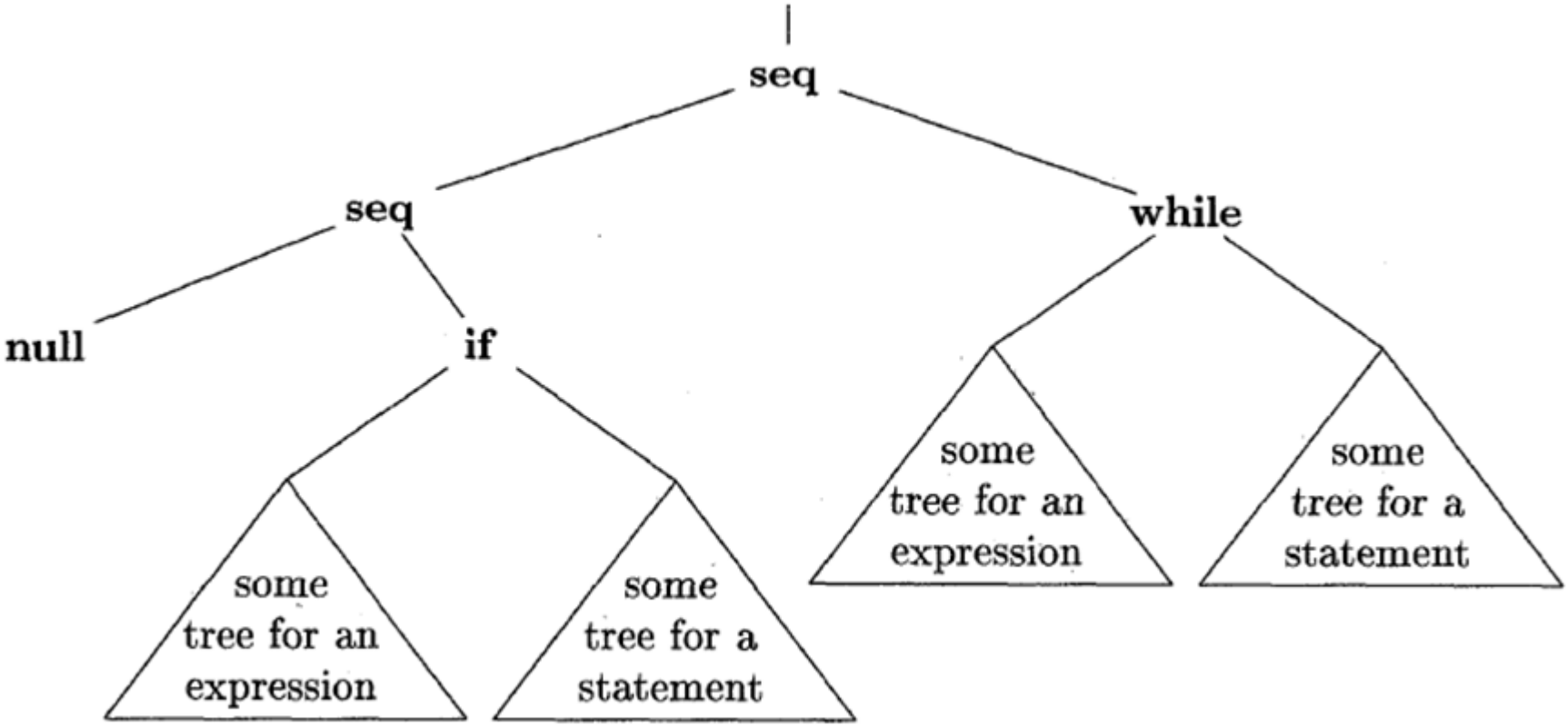
Construction of Syntax Trees for Statements

$$
\begin{aligned}
\textit{program} &\rightarrow \textit{block} \\
\textit{block} &\rightarrow \text{'\{'}\ \textit{stmts}\ \text{'\}'} \\
\textit{stmts} &\rightarrow \textit{stmts}_1\ \textit{stmt} \\
&\mid \epsilon \\
\textit{stmt} &\rightarrow \textit{expr}\ ; \\
&\mid \textbf{if}\ (\ \textit{expr}\ )\ \textit{stmt}_1 \\
&\mid \textbf{while}\ (\ \textit{expr}\ )\ \textit{stmt}_1 \\
&\mid \textbf{do}\ \textit{stmt}_1\ \textbf{while}\ (\ \textit{expr}\ ); \\
&\mid \textit{block}
\end{aligned}
$$

Construction of Syntax Trees for Statements

$$
\begin{array}{lcll}
\textit{program} & \rightarrow & \textit{block} & \{\ \text{return } \textit{block.n};\ \} \\
\textit{block} & \rightarrow & \text{'\{'}\ \textit{stmts}\ \text{'\}'} & \{\ \textit{block.n} = \textit{stmts.n};\ \} \\
\textit{stmts} & \rightarrow & \textit{stmts}_1\ \textit{stmt} & \{\ \textit{stmts.n} = \textbf{new}\ \textit{Seq}(\textit{stmts}_1.n, \textit{stmt.n});\ \} \\
 & | & \epsilon & \{\ \textit{stmts.n} = \textbf{null};\ \} \\
\textit{stmt} & \rightarrow & \textit{expr}\ ; & \{\ \textit{stmt.n} = \textbf{new}\ \textit{Eval}(\textit{expr.n});\ \} \\
 & | & \textbf{if}\ (\ \textit{expr}\ )\ \textit{stmt}_1 & \{\ \textit{stmt.n} = \textbf{new}\ \textit{If}(\textit{expr.n}, \textit{stmt}_1.n);\ \} \\
 & | & \textbf{while}\ (\ \textit{expr}\ )\ \textit{stmt}_1 & \{\ \textit{stmt.n} = \textbf{new}\ \textit{While}(\textit{expr.n}, \textit{stmt}_1.n);\ \} \\
 & | & \textbf{do}\ \textit{stmt}_1\ \textbf{while}\ (\ \textit{expr}\ )\ ; & \{\ \textit{stmt.n} = \textbf{new}\ \textit{Do}(\textit{stmt}_1.n, \textit{expr.n});\ \} \\
 & | & \textit{block} & \{\ \textit{stmt.n} = \textit{block.n};\ \}
\end{array}
$$

Construction of Syntax Trees for Statements



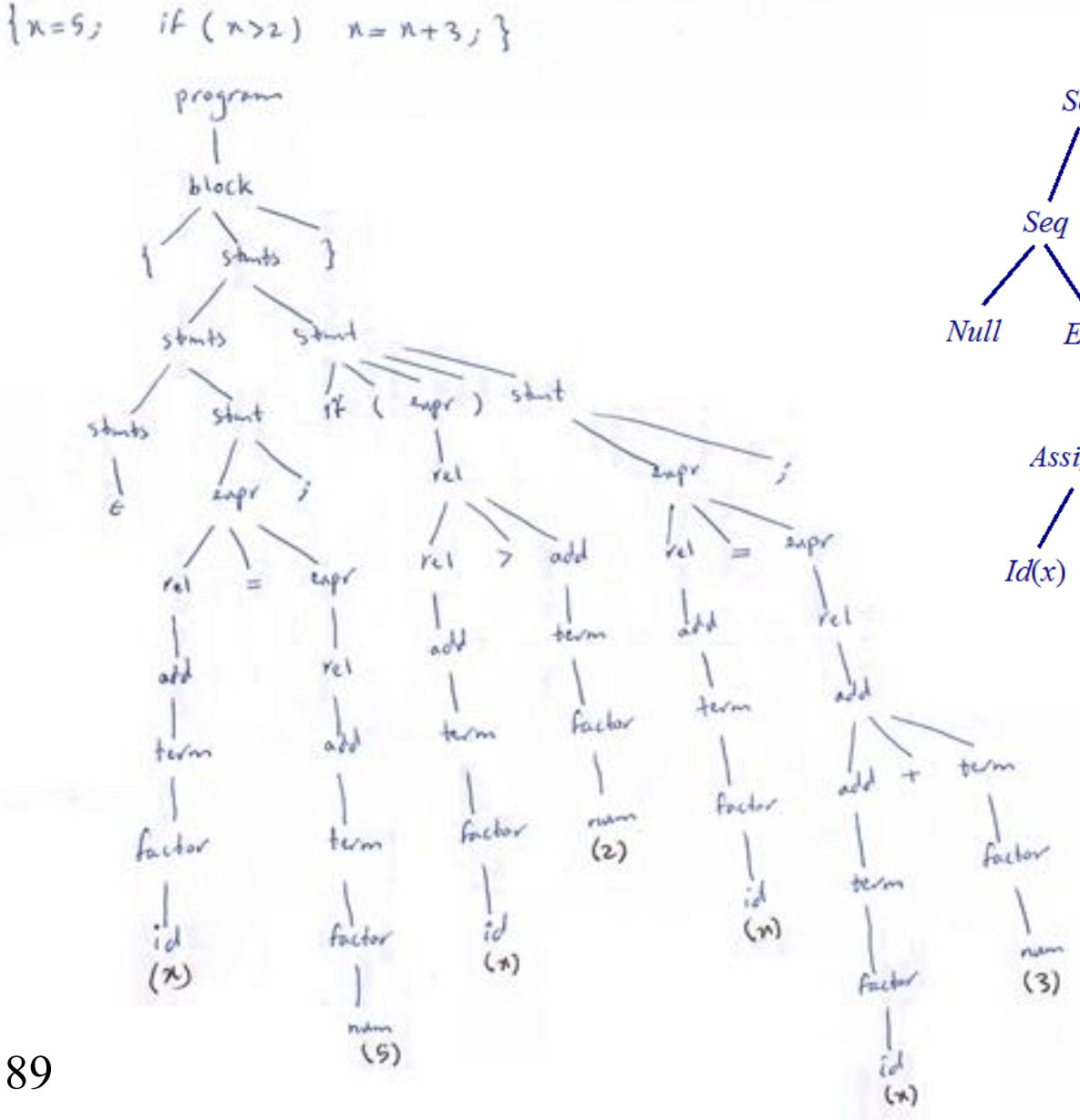
Construction of Syntax Trees for Expressions

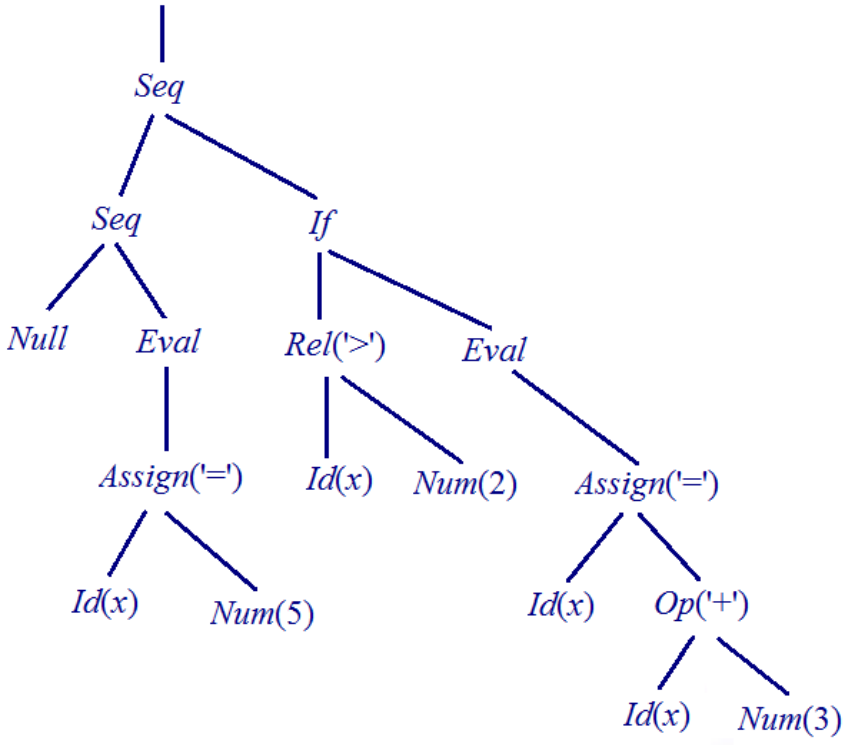
$$
\begin{aligned}
expr &\rightarrow rel = expr_1 \\
&\mid rel \\
\\
rel &\rightarrow rel_1 < add \\
&\mid rel_1 <= add \\
&\mid add \\
\\
add &\rightarrow add_1 + term \\
&\mid term \\
\\
term &\rightarrow term_1 * factor \\
&\mid factor \\
\\
factor &\rightarrow (expr) \\
&\mid \mathbf{num} \\
&\mid \mathbf{id}
\end{aligned}
$$

Construction of Syntax Trees for Expressions

$expr \rightarrow$	$rel = expr_1$	$\{\ expr.n = \textbf{new}\ Assign('=', rel.n, expr_1.n);\ \}$
$\mid$	rel	$\{\ expr.n = rel.n;\ \}$
$rel \rightarrow$	$rel_1 < add$	$\{\ rel.n = \textbf{new}\ Rel('<', rel_1.n, add.n);\ \}$
$\mid$	$rel_1 <= add$	$\{\ rel.n = \textbf{new}\ Rel('\leq', rel_1.n, add.n);\ \}$
$\mid$	add	$\{\ rel.n = add.n;\ \}$
$add \rightarrow$	$add_1 + term$	$\{\ add.n = \textbf{new}\ Op('+', add_1.n, term.n);\ \}$
$\mid$	$term$	$\{\ add.n = term.n;\ \}$
$term \rightarrow$	$term_1 * factor$	$\{\ term.n = \textbf{new}\ Op('*', term_1.n, factor.n);\ \}$
$\mid$	$factor$	$\{\ term.n = factor.n;\ \}$
$factor \rightarrow$	$(\ expr\ )$	$\{\ factor.n = expr.n;\ \}$
$\mid$	**num**	$\{\ factor.n = \textbf{new}\ Num(\textbf{num}.value);\ \}$
$\mid$	**id**	$\{\ factor.n = \textbf{new}\ Id(\textbf{id}.lexeme);\ \}$

Construction of Syntax Trees





Static Checking

- Consistency checks during compilation
  - To assure that a program can be compiled successfully
  - To catch programming errors before run

1. Syntactic checking: (not encoded in grammar)
   - An identifier being declared at most once in a scope
   - A break must have an enclosing loop or switch statement

2. L-values (locations) and R-values (values)
   - Left side of an assignment must be an l-value (identifier *i* , array access *a* [2]
   - A constant has r-value, so not on left side of an assignment

Static Checking

3. Type checking:
   - Right number and type of operands in an operator or function

     stmt → **if** (*expr*) *stmt*

     expr → $expr_1$ **relop** $expr_2$ { if ($expr_1.type == expr_2.type$)
     expr.type = Boolean;
     else
     type_error(); }
   - Coercion: type of an operand is automatically converted to the type expected by the operator
   - Overloading: meaning of an operator is determined by considering the known types of its operands and results

Three-Address Code

- Walking syntax trees to generate three-address code
- By evaluating attributes and executing code fragments at nodes in the tree
- Via writing functions that process the syntax tree and emit the necessary three-address code

Three-Address Instructions

- Binary operation: $x = y \ \mathbf{op} \ z$

  x, y, z: names, constants, compiler-generated temporaries

- Copy instruction: $x = y$

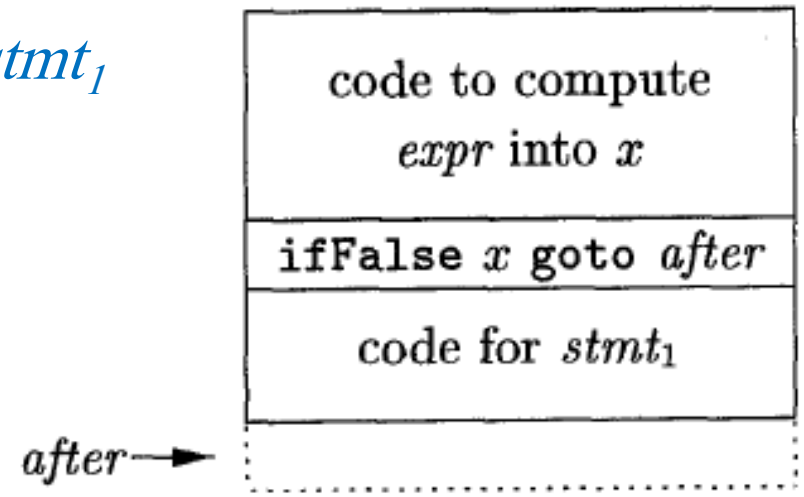
- Array access: $x\,[\,y\,] = z$
  $x = y\,[\,z\,]$

- Conditional/Unconditional jump:

`ifFalse` x `goto L`	if x is false, next execute the instruction labeled `L`
`ifTrue` x `goto L`	if x is true, next execute the instruction labeled `L`
`goto L`	next execute the instruction labeled `L`

Three-Address Code for Statements

stmt → **if** (*expr*) *stmt*$_1$




```
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 = rvalue(E);
                emit( "ifFalse " + n.toString() + " goto " + after);
                S.gen();
                emit(after + ":");
        }
}
```

Three-Address Code for Expressions

- Expressions containing binary operators **op**, array accesses, assignments, constants, identifiers

- Generating one three-address instruction for each operator node *x* of class *Expr*
  - The computed value is stored into a compiler generated temporary name, say *t*

b + **c** * **d**

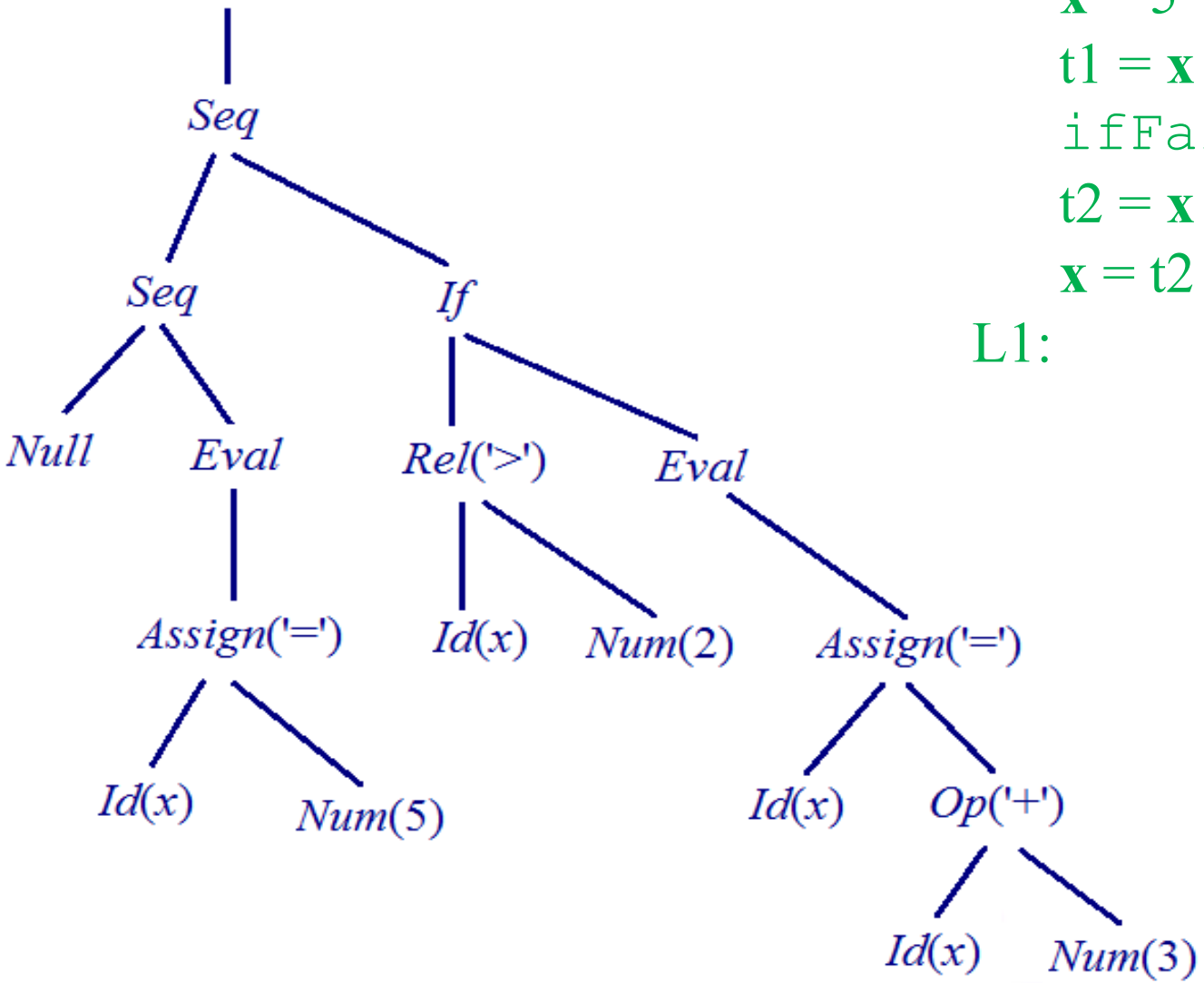
t1 = **c** * **d**
t2 = **b** + t1

Three-Address Code for Expressions


```
Expr lvalue(x : Expr) {
        if ( x is an Id node ) return x;
        else error;
}

Expr rvalue(x : Expr) {
        if ( x is an Id or a Constant node ) return x;
        else if ( x is an Op(op, y, z) or a Rel(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 Assign(y, z) node ) {
                z' = rvalue(z);
                emit string for lvalue(y) = z';
                return z';
        }
}
```

Example Three-Address Code

{ **x** = 5; if (**x** > 2) **x** = **x** + 3; }



x = 5
t1 = **x** > 2
`ifFalse` t1 `goto` L1
t2 = **x** + 3
x = t2
L1:

Three-Address Code for Expressions

- If node *x* represents the array access:

a [**b** + **c**]

t1 = **b** + **c**
a [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;
}
```

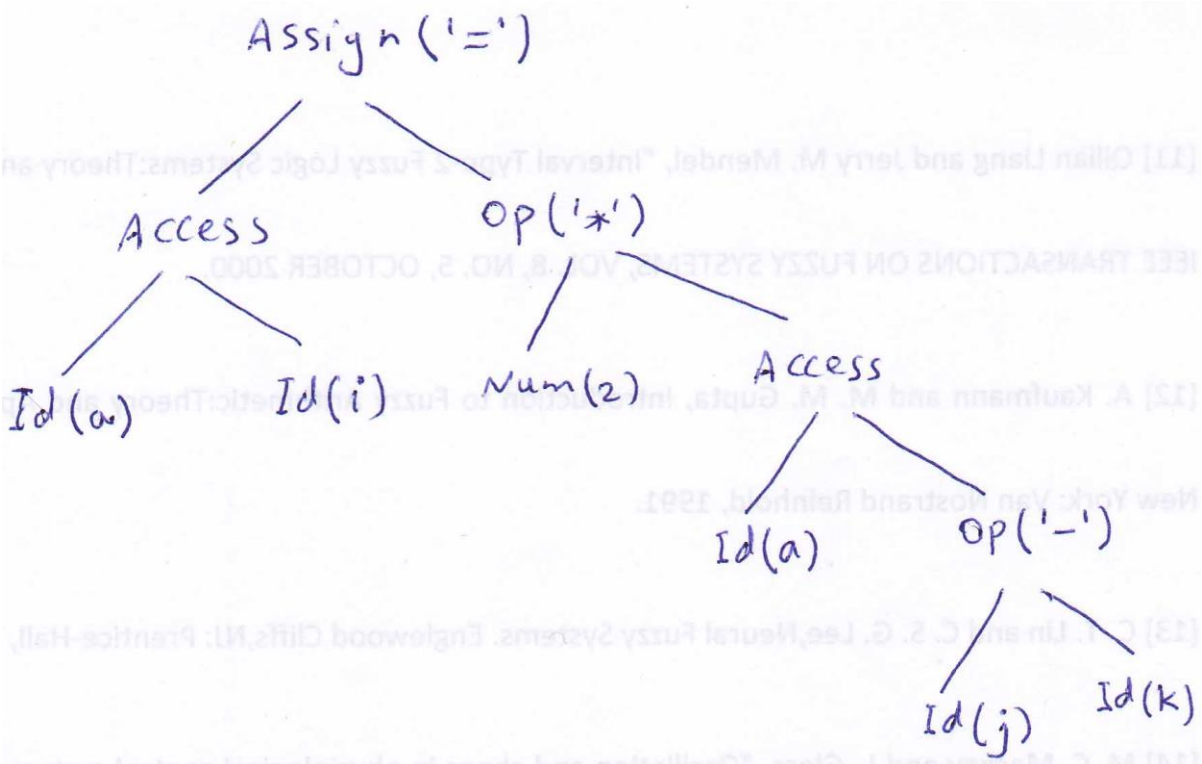
Three-Address Code for Expressions


```
Expr rvalue(x : Expr) {
    if ( x is an Id or a Constant node ) return x;
    else if ( x is an Op(op, y, z) or a Rel(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';
    }
}
```

Three-Address Code for Expressions

mansoori@shirazu.ac.ir

```
a[i] = 2*a[j-k]
```




```
t3 = j - k
t2 = a [ t3 ]
t1 = 2 * t2
a [ i ] = t1
```