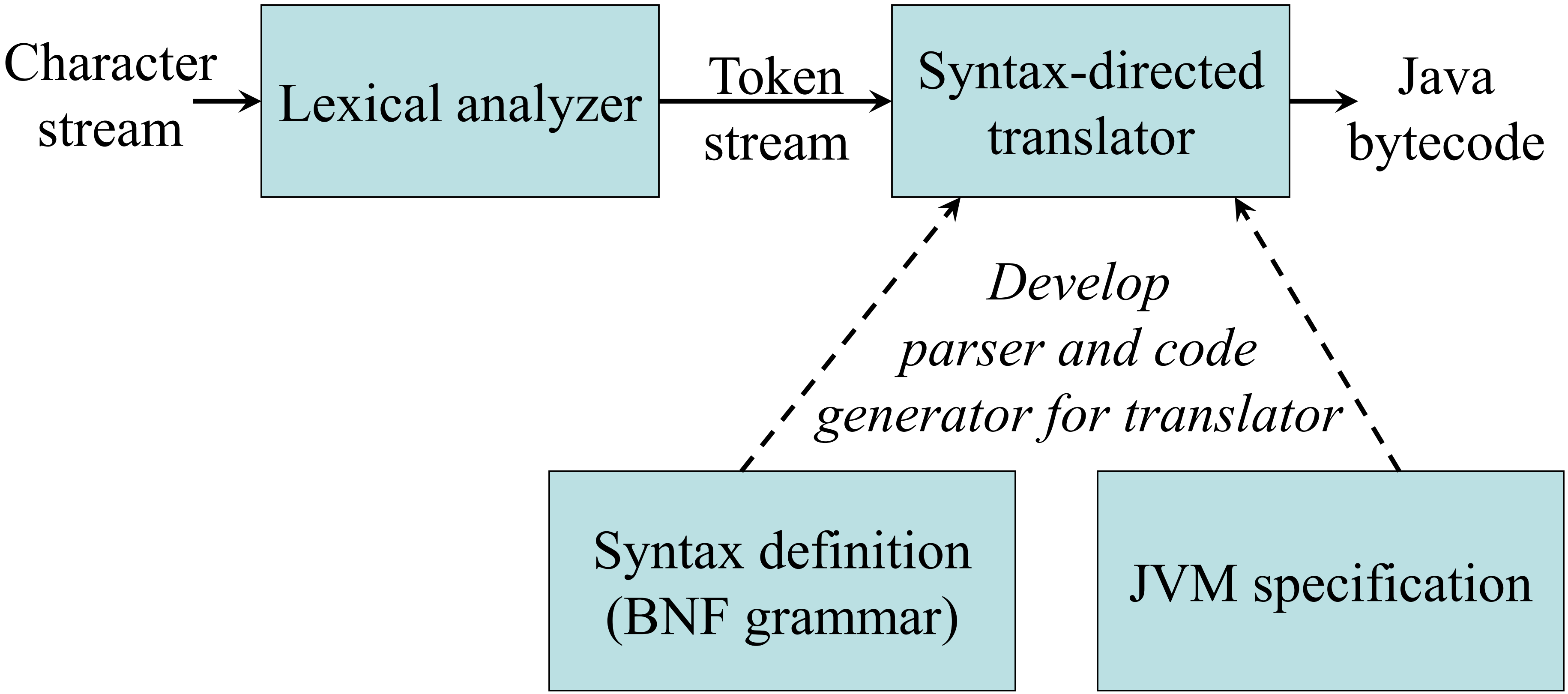
A Simple Syntax-Directed Translator

Chapter 2

Overview

- This chapter contains introductory material to Chapters 3 to 8
- Building a simple compiler
  - Syntax definition
  - Syntax directed translation
  - Predictive parsing

The Structure of our Compiler



Syntax Definition

- 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 =

$$list \rightarrow list + digit$$

$$list \rightarrow list \;\mathbf{-}\; digit$$

$$list \rightarrow digit$$

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

Derivation

- Given a CF grammar 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

<u>*list*</u>
⇒ <u>*list*</u> + *digit*
⇒ <u>*list*</u> **-** *digit* + *digit*
⇒ <u>*digit*</u> **-** *digit* + *digit*
⇒ **9 -** <u>*digit*</u> + *digit*
⇒ **9 - 5** + <u>*digit*</u>
⇒ **9 - 5** + **2**

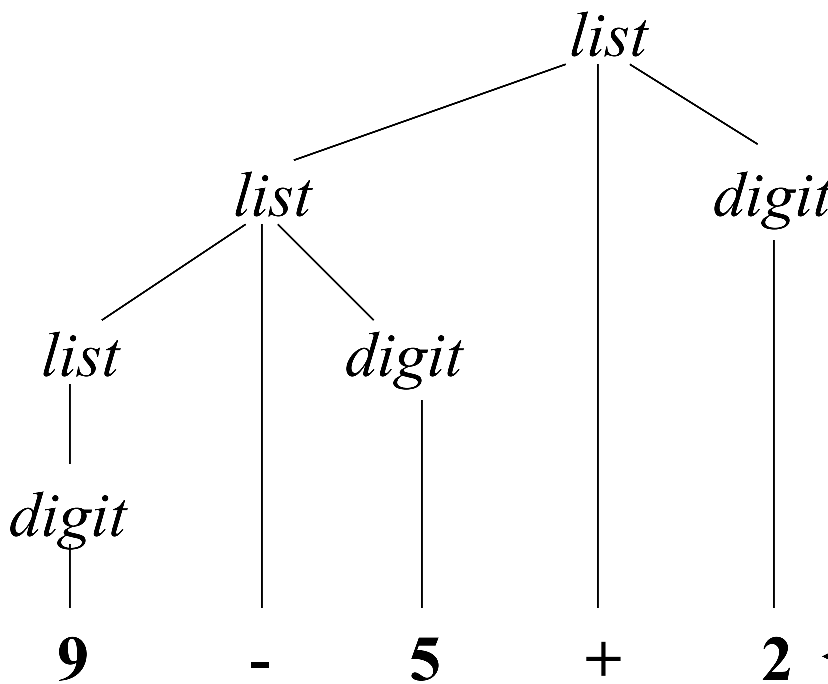
This is an example *leftmost derivation*, because we replaced the leftmost nonterminal (underlined) in each step

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 children X_1, X_2, ..., X_n where X_i is a (non)terminal or ε (ε denotes the *empty string*)

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:

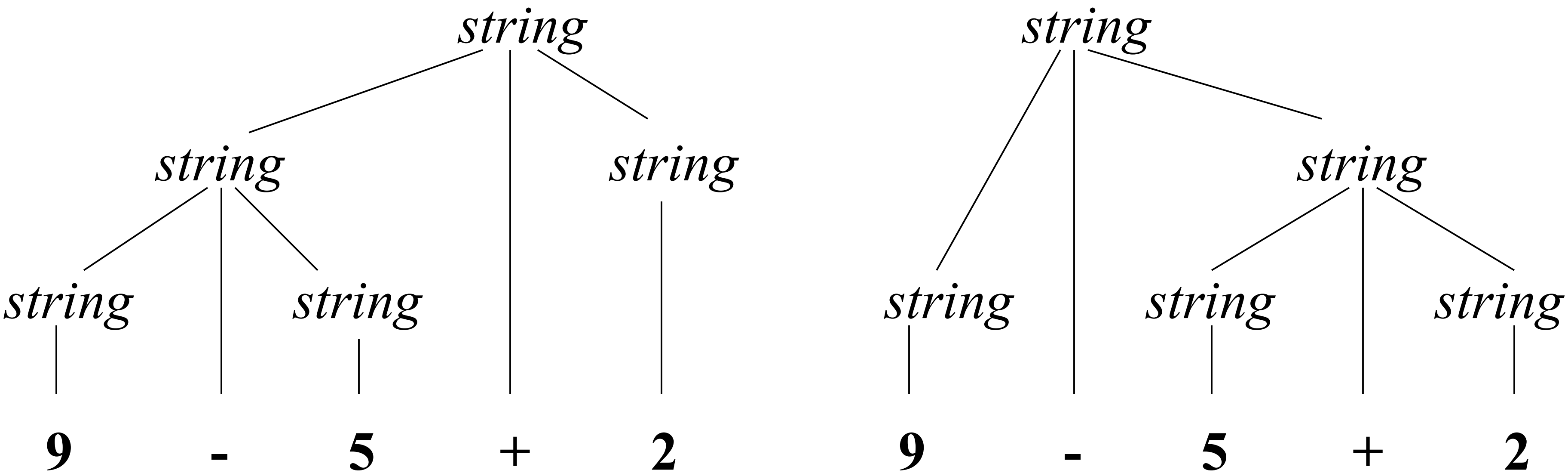
$$G = <\{string\}, \{+,-,\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5},\mathbf{6},\mathbf{7},\mathbf{8},\mathbf{9}\}, P, string>$$

with production $P =$

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

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

Ambiguity (cont’d)



Associativity of Operators

Left-associative operators have *left-recursive* productions

$$left \rightarrow left + term \mid term$$

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

Right-associative operators have *right-recursive* productions

$$right \rightarrow term = right \mid term$$

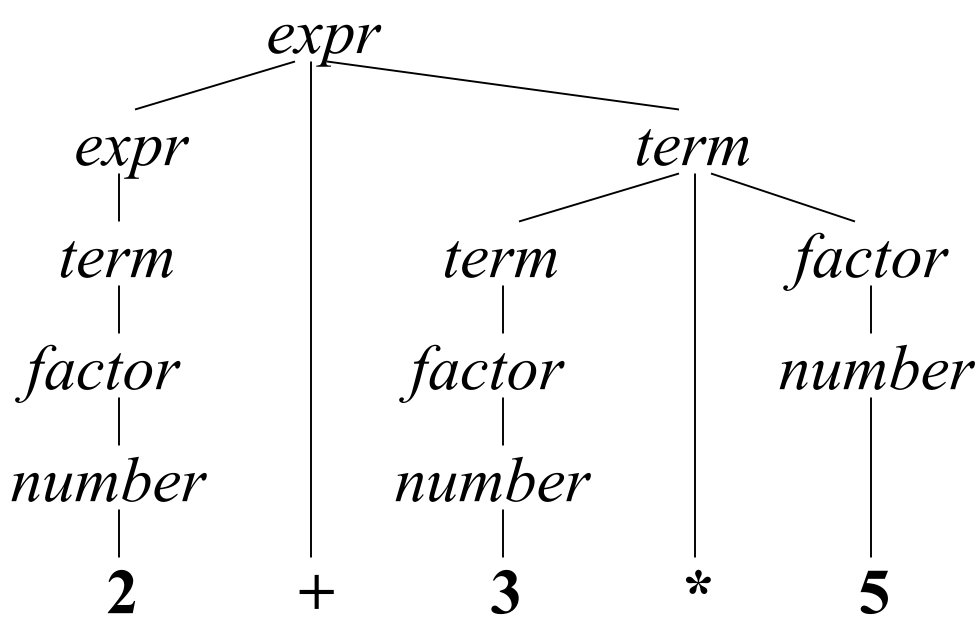
String **a=b=c** has the same meaning as **a=(b=c)**

Precedence of Operators

Operators with higher precedence “bind more tightly”

$$\textit{expr} \rightarrow \textit{expr} + \textit{term} \mid \textit{term}$$
$$\textit{term} \rightarrow \textit{term} * \textit{factor} \mid \textit{factor}$$
$$\textit{factor} \rightarrow \textit{number} \mid \textbf{(}\ \textit{expr}\ \textbf{)}$$

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



Syntax of Statements

$$
\begin{aligned}
\mathit{stmt} \rightarrow\ & \textbf{id} \mathrel{\textbf{:=}} \mathit{expr} \\
\mid\ & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\
\mid\ & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
\mid\ & \textbf{while}\ \mathit{expr}\ \textbf{do}\ \mathit{stmt} \\
\mid\ & \textbf{begin}\ \mathit{opt\_stmts}\ \textbf{end} \\
\mathit{opt\_stmts} \rightarrow\ & \mathit{stmt}\ \textbf{;}\ \mathit{opt\_stmts} \\
\mid\ & \varepsilon
\end{aligned}
$$

Syntax-Directed Translation

- Uses a CF grammar to specify the syntactic structure of the language
- AND associates a set of *attributes* with (non)terminals
- AND associates with each production a set of *semantic rules* for computing values for the attributes
- The attributes contain the translated form of the input after the computations are completed

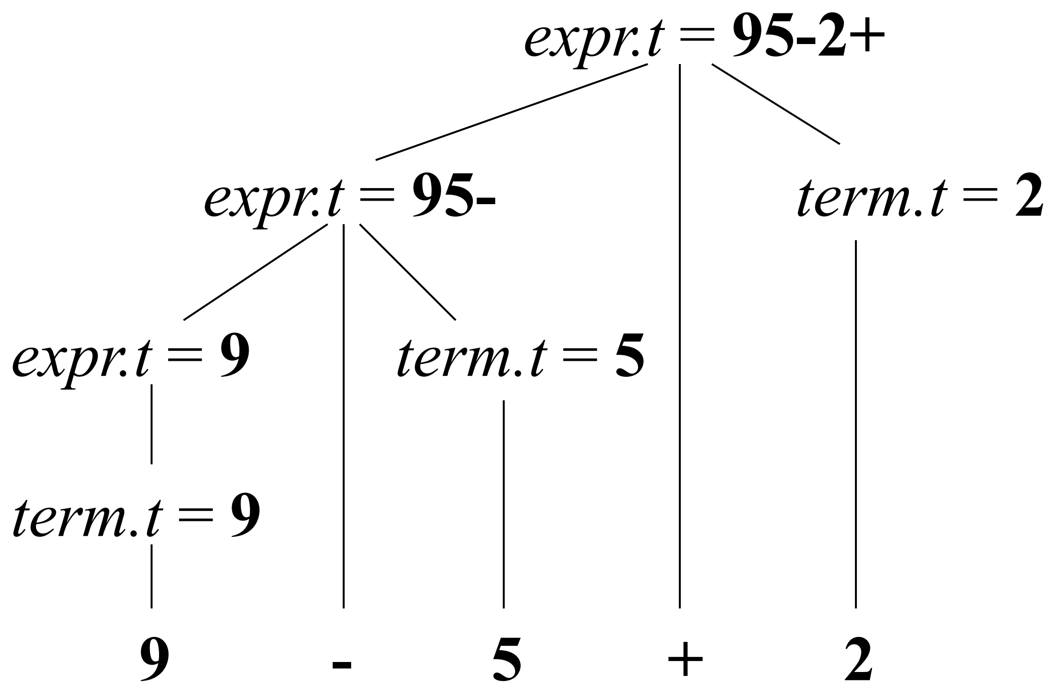
Synthesized and Inherited Attributes

- 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

Production	Semantic Rule
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \; // \; term.t \; //$ "+"
$expr \rightarrow expr_1$ **-** $term$	$expr.t := expr_1.t \; // \; term.t \; //$ "**-**"
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow$ **0**	$term.t :=$ "0"
$term \rightarrow$ **1**	$term.t :=$ "1"
…	…
$term \rightarrow$ **9**	$term.t :=$ "9"

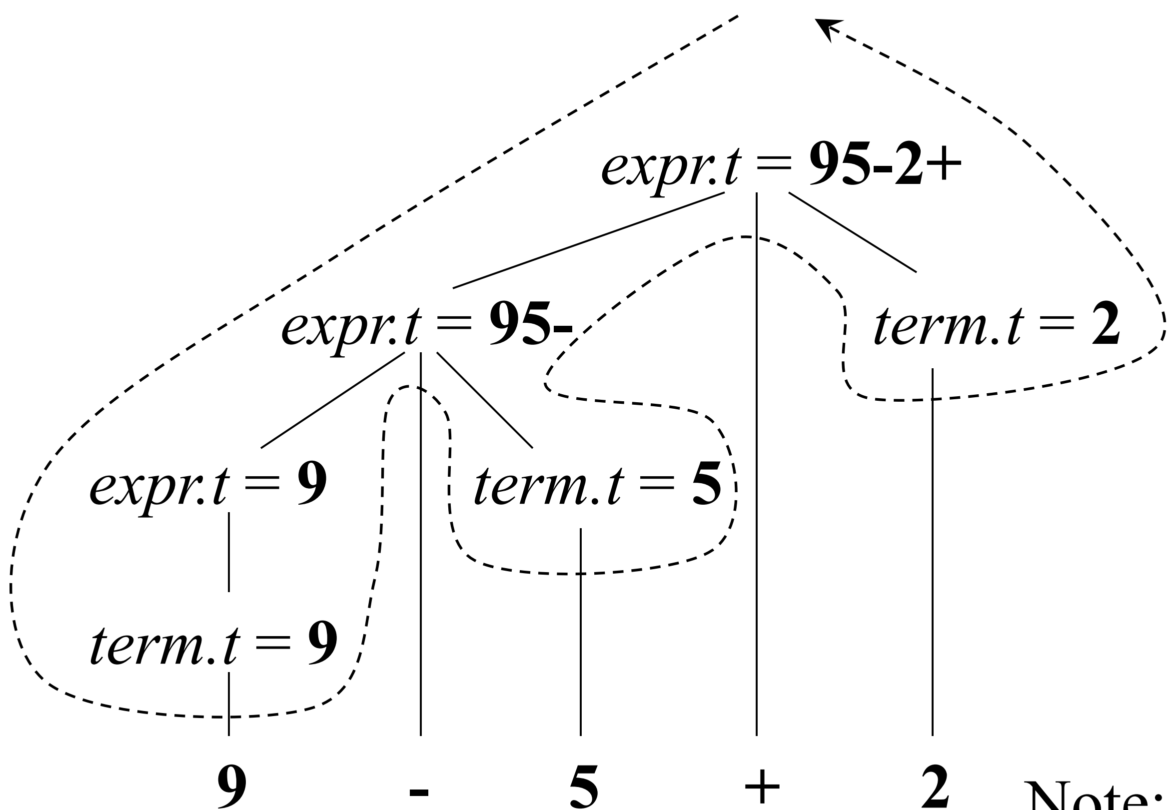
Example Annotated Parse Tree



Depth-First Traversals

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

Depth-First Traversals (Example)

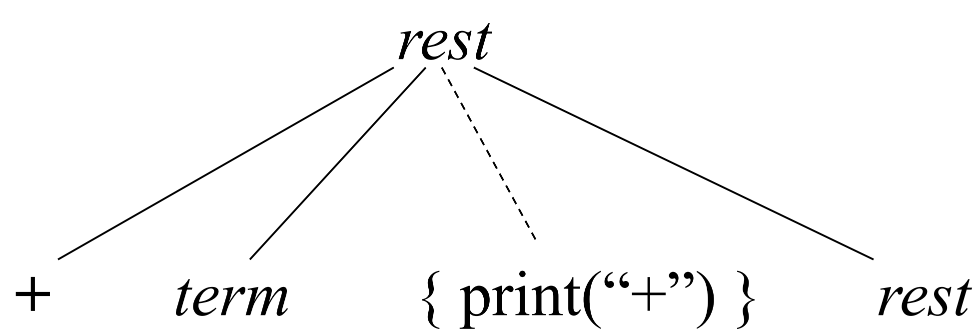


Note: all attributes are of the synthesized type

Translation Schemes

- A *translation scheme* is a CF grammar embedded with *semantic actions*

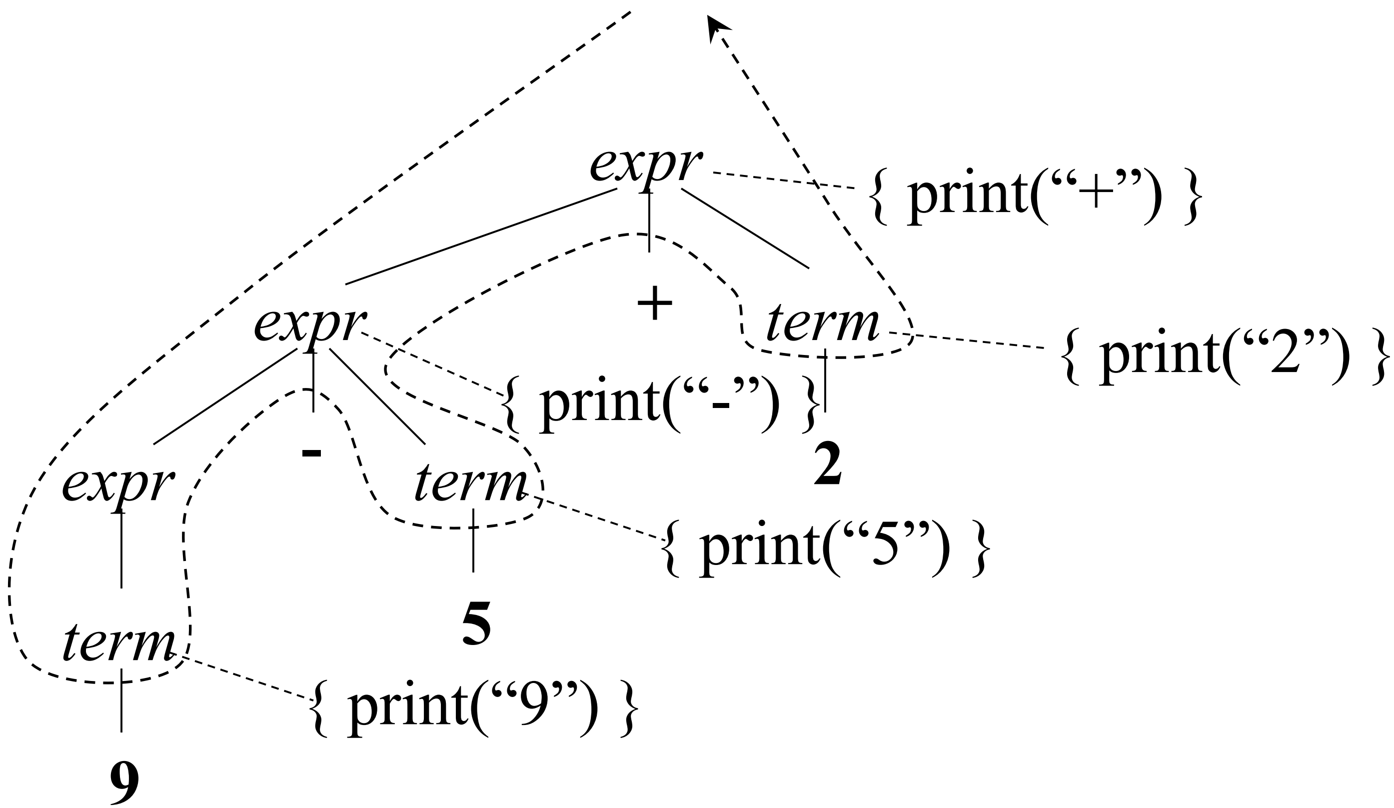
$$\mathit{rest} \rightarrow + \ \mathit{term}\ \underbrace{\{\ \text{print(“+”)}\ \}}_{\text{Embedded semantic action}}\ \mathit{rest}$$



Example Translation Scheme

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

Example Translation Scheme (cont'd)



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

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
- *Top-down parsing* "constructs" parse tree from root to leaves
- *Bottom-up parsing* "constructs" parse tree from leaves to root

Predictive Parsing

- *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
  - 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

Example Predictive Parser (Grammar)

type → *simple*
| **^ id**
| **array [** *simple* **] of** *type*
simple → **integer**
| **char**
| **num dotdot num**

Example Predictive Parser (Program Code)

```
procedure match(t : token);
begin
    if lookahead = t then
        lookahead := nexttoken()
    else error()
end;

procedure type();
begin
    if lookahead in { 'integer', 'char', 'num' } then
        simple()
    else if lookahead = '^' then
        match('^'); match(id)
    else if lookahead = 'array' then
        match('array'); match('['); simple();
        match(']'); match('of'); type()
    else error()
end;

procedure simple();
begin
    if lookahead = 'integer' then
        match('integer')
    else if lookahead = 'char' then
        match('char')
    else if lookahead = 'num' then
        match('num');
        match('dotdot');
        match('num')
    else error()
end;
```

Example Predictive Parser (Execution Step 1)

type()

Check *lookahead* and call *match*

match(‘**array**’)

Input: **array** **[** **num** **dotdot** **num** **]** **of** **integer**

↑

lookahead

Example Predictive Parser (Execution Step 2)

type()

match(‘**array**’) *match*(‘**[**’)

Input: **array** **[** **num** **dotdot** **num** **]** **of** **integer**

lookahead

Example Predictive Parser (Execution Step 3)

type()

match(‘**array**’) *match*(‘**[**’) *simple*()

match(‘**num**’)

<u>Input</u>: **array [num dotdot num] of integer**

↑ *lookahead*

Example Predictive Parser (Execution Step 4)

type()

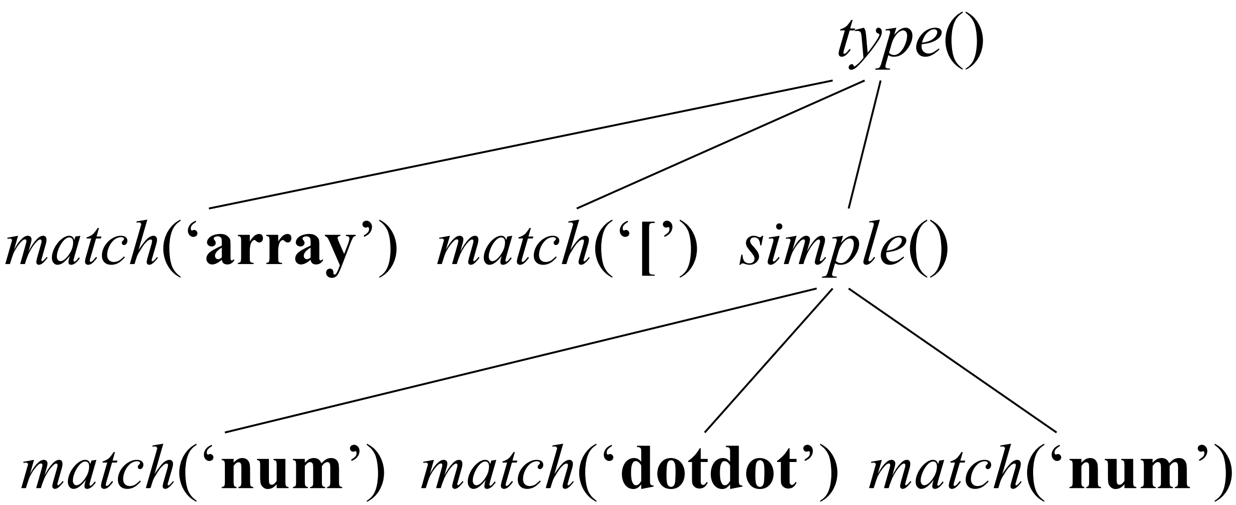
match(‘**array**’) *match*(‘**[**’) *simple*()

match(‘**num**’) *match*(‘**dotdot**’)

Input: **array [num dotdot num] of integer**

lookahead

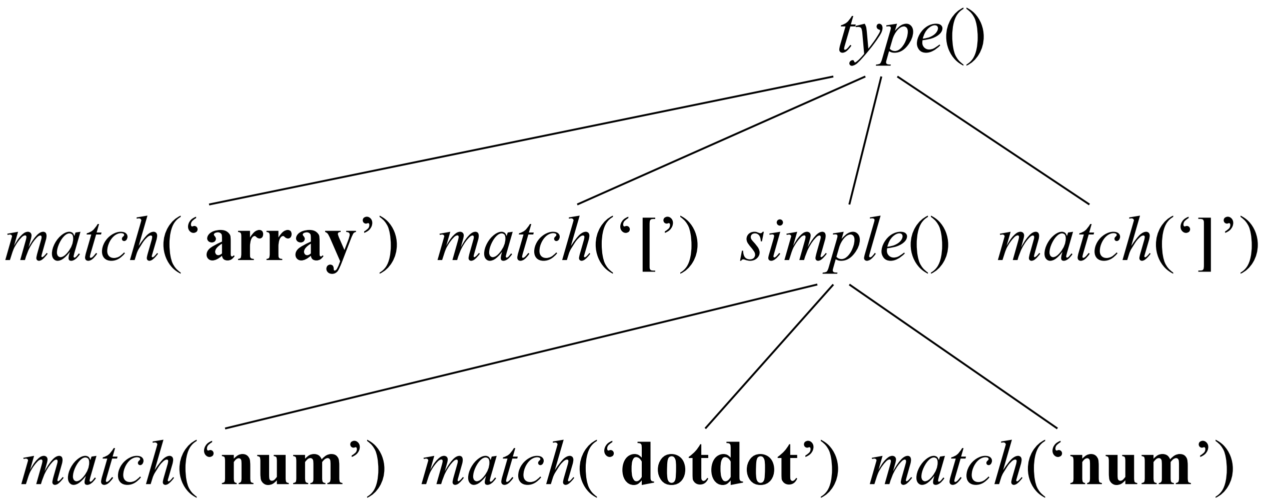
Example Predictive Parser (Execution Step 5)



Input: **array** **[** **num** **dotdot** **num** **]** **of** **integer**

lookahead

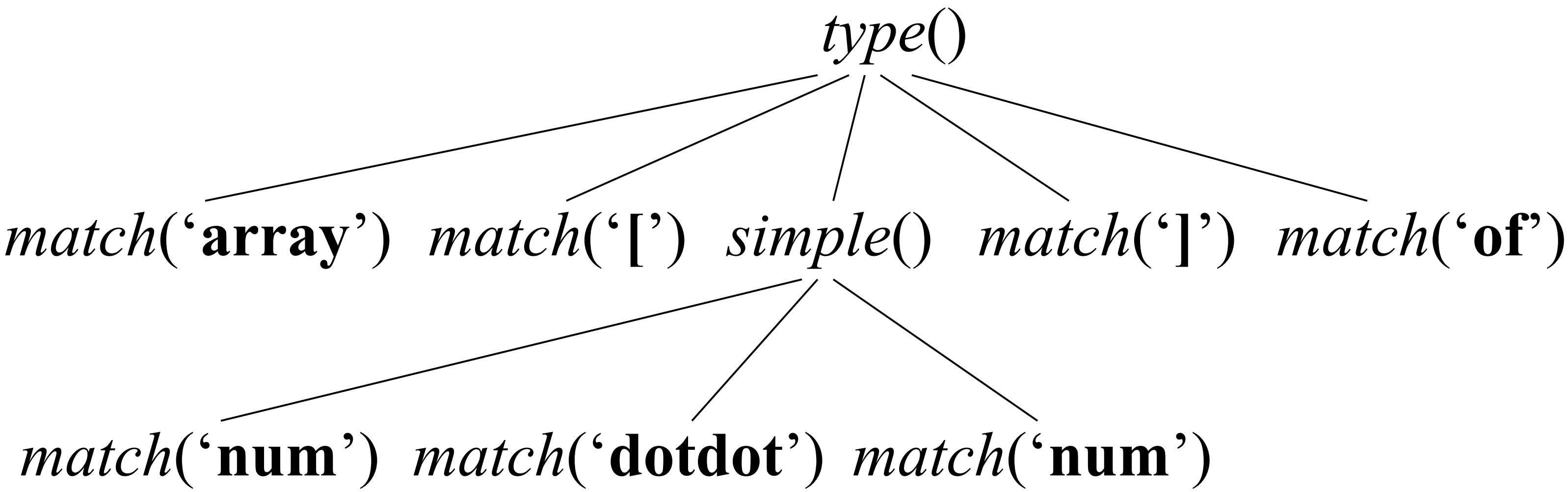
Example Predictive Parser (Execution Step 6)



Input: **array [num dotdot num] of integer**

lookahead

Example Predictive Parser (Execution Step 7)

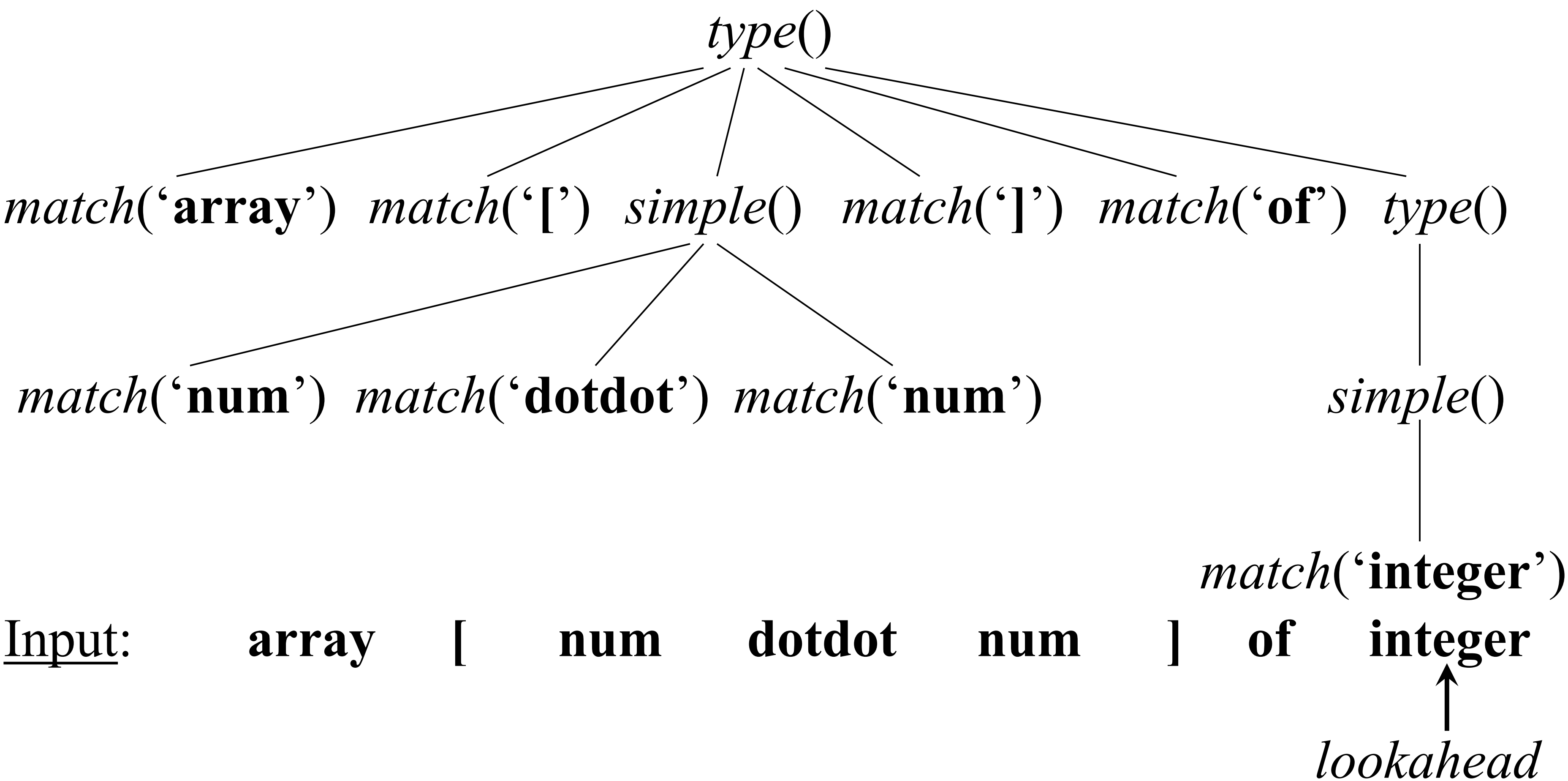


Input: **array [num dotdot num] of integer**

↑

lookahead

Example Predictive Parser (Execution Step 8)



FIRST

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

type → *simple*
  | **^ id**
  | **array [** *simple* **] of** *type*
simple → **integer**
  | **char**
  | **num dotdot num**

FIRST(*simple*) = { **integer**, **char**, **num** }
FIRST(**^ id**) = { **^** }
FIRST(*type*) = { **integer**, **char**, **num**, **^**, **array** }

Using FIRST

We use FIRST to write a predictive parser as follows

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

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

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

Then FIRST (α) and FIRST(β) 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

$$\begin{aligned} stmt \rightarrow\ & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt} \\ \mid\ & \textbf{if}\ \text{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \end{aligned}$$

We can use *left factoring* to fix the problem

$$\begin{aligned} stmt &\rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \textit{stmt opt\_else} \\ opt\_else &\rightarrow \textbf{else}\ \textit{stmt} \\ &\mid \varepsilon \end{aligned}$$

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*

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

A Translator for Simple Expressions

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

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") }

$expr \rightarrow term\ rest$

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

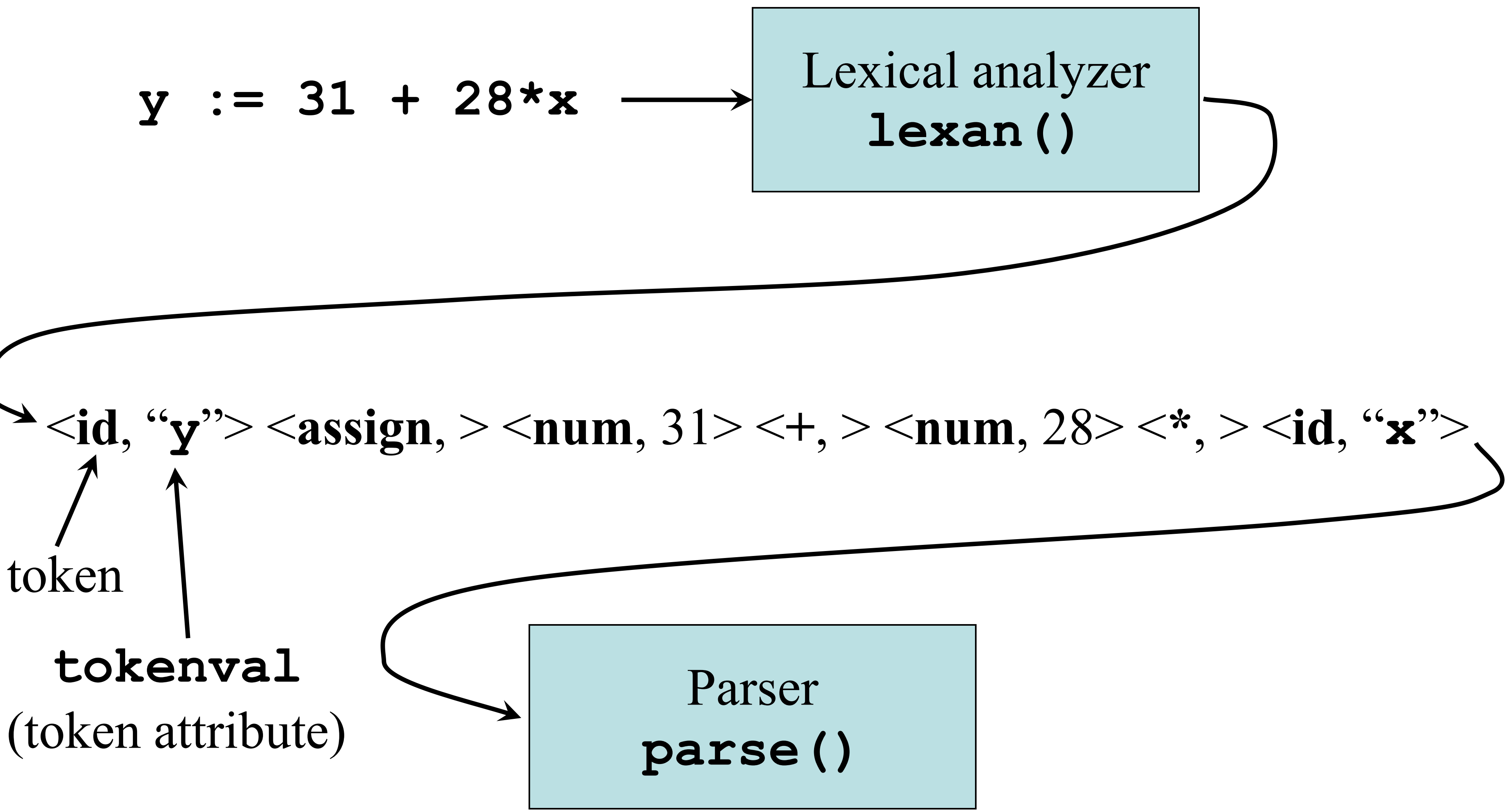
$term \rightarrow \mathbf{0}\ \{\ \mathrm{print}(\text{“0”})\ \}$
$term \rightarrow \mathbf{1}\ \{\ \mathrm{print}(\text{“1”})\ \}$
$\ldots$
$term \rightarrow \mathbf{9}\ \{\ \mathrm{print}(\text{“9”})\ \}$

```
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);
}
```

Adding a Lexical Analyzer

- Typical tasks of the lexical analyzer:
  - Remove white space and comments
  - Encode constants as tokens
  - Recognize keywords
  - Recognize identifiers

The Lexical Analyzer



Token Attributes

$factor \rightarrow$ **(** $expr$ **)**
| **num** { print(**num**.value) }

```
#define NUM 256 /* token returned by lexan */

factor()
{   if (lookahead == '(')
    {   match('('); expr(); match(')');
    }
    else if (lookahead == NUM)
    {   printf(" %d ", tokenval); match(NUM);
    }
    else error();
}
```

Symbol Table

The symbol table is globally accessible (to all phases of the compiler)

Each entry in the symbol table contains a string and a token value:

```
struct entry
{   char *lexptr; /* lexeme (string) */
    int token;
};
struct entry symtable[];
```

`insert(s, t)`: returns array index to new entry for string **`s`** token **`t`**

`lookup(s)`: returns array index to entry for string **`s`** or 0

Possible implementations:
- simple C code (see textbook)
- hashtables

Identifiers

$$\textit{factor} \rightarrow \textbf{(}\ \textit{expr}\ \textbf{)}$$
$$\quad \mid \textbf{id}\ \{\ \text{print}(\textbf{id}.\text{string})\ \}$$

```
#define ID 259 /* token returned by lexan() */

factor()
{   if (lookahead == '(')
    {   match('('); expr(); match(')');
    }
    else if (lookahead == ID)
    {   printf(" %s ", symtable[tokenval].lexptr);
        match(NUM);
    }
    else error();
}
```

Handling Reserved Keywords

We simply initialize the global symbol table with the set of keywords

```
/* global.h */
#define DIV 257 /* token */
#define MOD 258 /* token */
#define ID  259 /* token */

/* init.c */
insert("div", DIV);
insert("mod", MOD);

/* lexer.c */
int lexan()
{   …
    tokenval = lookup(lexbuf);
    if (tokenval == 0)
        tokenval = insert(lexbuf, ID);
    return symtable[p].token;
}
```

Handling Reserved Keywords (cont'd)

morefactors → **div** *factor* { print('DIV') } *morefactors*
| **mod** *factor* { print('MOD') } *morefactors*
| …

```
/* parser.c */
morefactors()
{   if (lookahead == DIV)
    {   match(DIV); factor(); printf("DIV"); morefactors();
    }
    else if (lookahead == MOD)
    {   match(MOD); factor(); printf("MOD"); morefactors();
    }
    else
        …
}
```