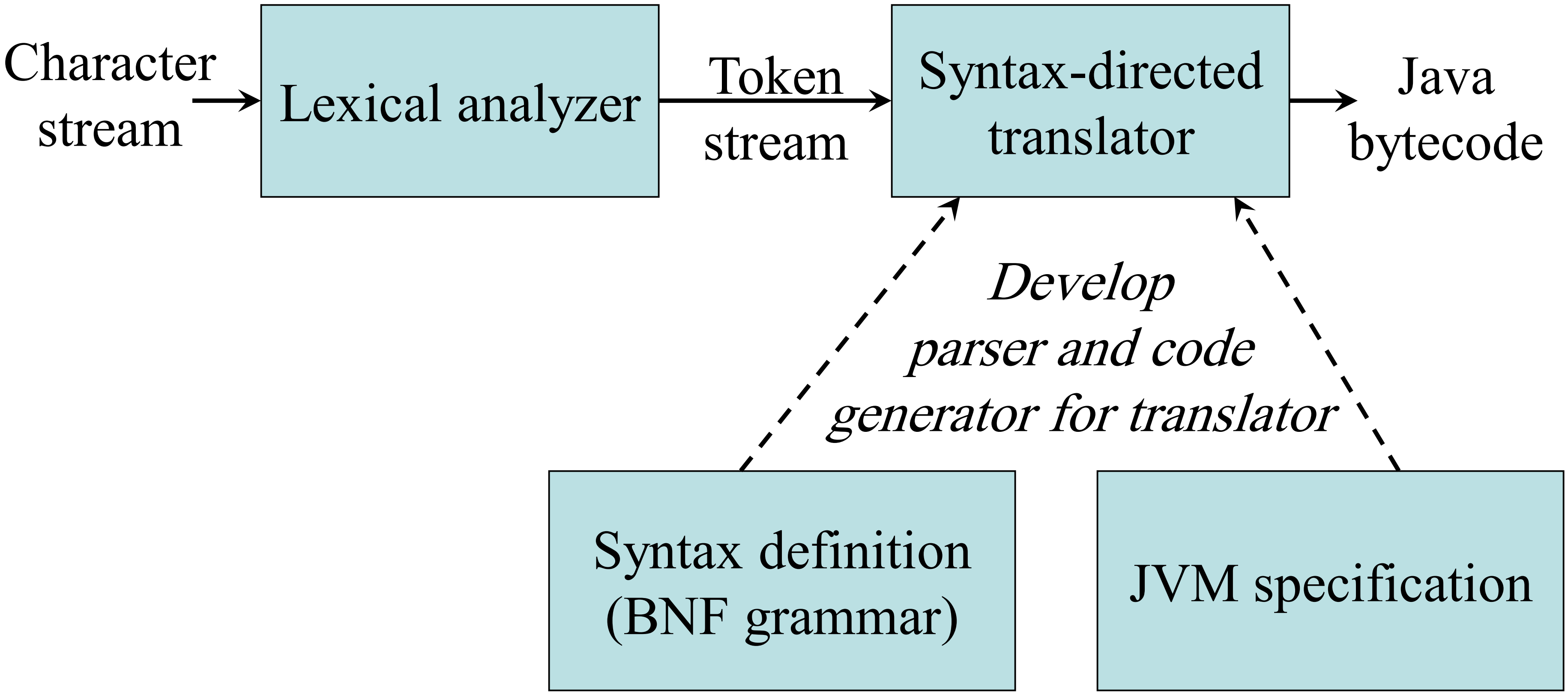
A Simple One-Pass Compiler (to Generate Code for the JVM)

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 JVM abstract stack machine
  - Generating Java bytecode for the JVM

The Structure of our Compiler



Character stream

Lexical analyzer

Token stream

Syntax-directed translator

Java bytecode

Develop parser and code generator for translator

Syntax definition (BNF grammar)

JVM specification

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:

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

with productions $P =$

$$\mathit{list} \rightarrow \mathit{list} + \mathit{digit}$$

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

$$\mathit{list} \rightarrow \mathit{digit}$$

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

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

$\underline{\mathit{list}}$

$\Rightarrow \underline{\mathit{list}} + \mathit{digit}$

$\Rightarrow \underline{\mathit{list}}\ \mathbf{-}\ \mathit{digit} + \mathit{digit}$

$\Rightarrow \underline{\mathit{digit}}\ \mathbf{-}\ \mathit{digit} + \mathit{digit}$

$\Rightarrow \mathbf{9\ -}\ \underline{\mathit{digit}} + \mathit{digit}$

$\Rightarrow \mathbf{9\ -\ 5\ +}\ \underline{\mathit{digit}}$

$\Rightarrow \mathbf{9\ -\ 5\ +\ 2}$

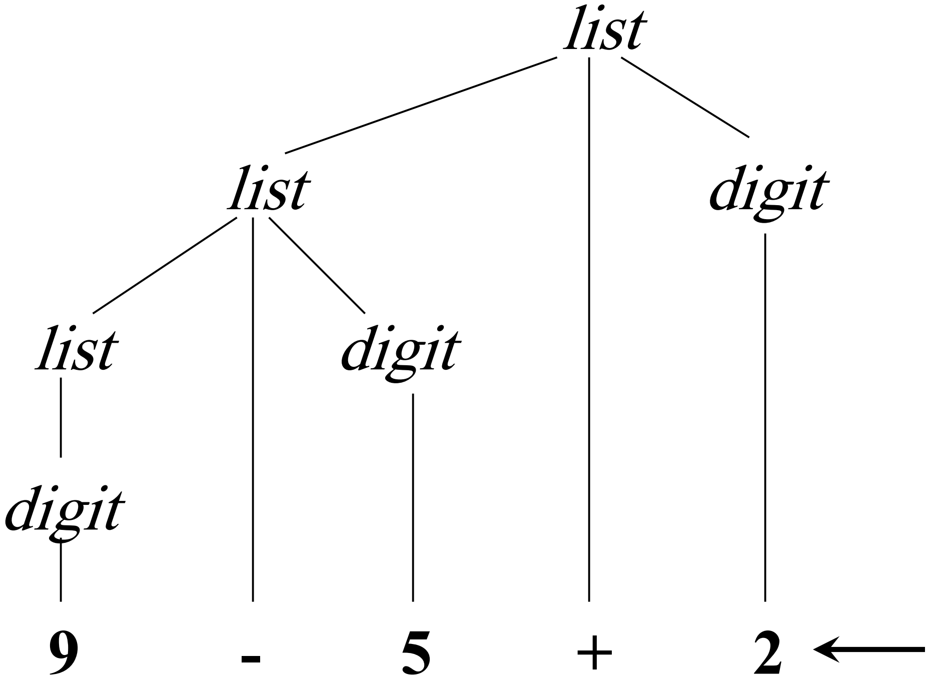
This is an example *leftmost derivation*, because we replaced the leftmost nonterminal (underlined) in each step

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 $X_1, X_2, \ldots, X_n$ 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:

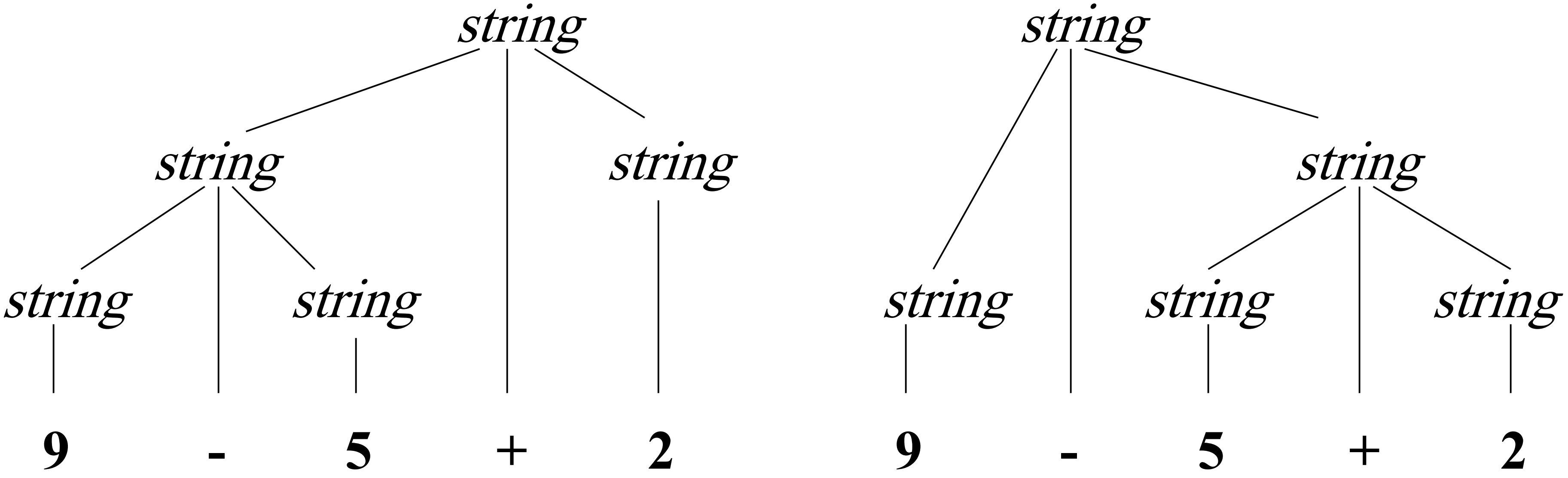
$$G = <\{\mathit{string}\}, \{\mathbf{+},\mathbf{-},\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5},\mathbf{6},\mathbf{7},\mathbf{8},\mathbf{9}\}, P, \mathit{string}>$$

with production $P =$

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

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

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

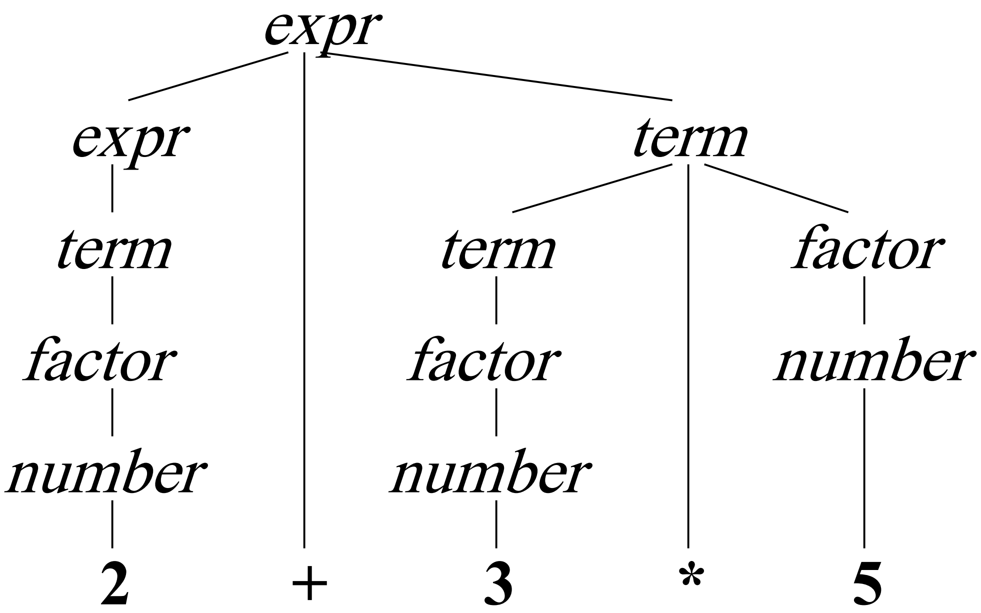
String **a=b=c** has the same meaning as **a=(b=c)**

Precedence of Operators

Operators with higher precedence “bind more tightly”

$$\begin{aligned}
&\mathit{expr} \rightarrow \mathit{expr} + \mathit{term} \mid \mathit{term} \\
&\mathit{term} \rightarrow \mathit{term} * \mathit{factor} \mid \mathit{factor} \\
&\mathit{factor} \rightarrow \mathit{number} \mid \textbf{(}\ \mathit{expr}\ \textbf{)}
\end{aligned}$$

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



expr

expr *term*

term *term* *factor*

factor *factor* *number*

number *number*

2 + **3** * **5**

Syntax of Statements

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

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

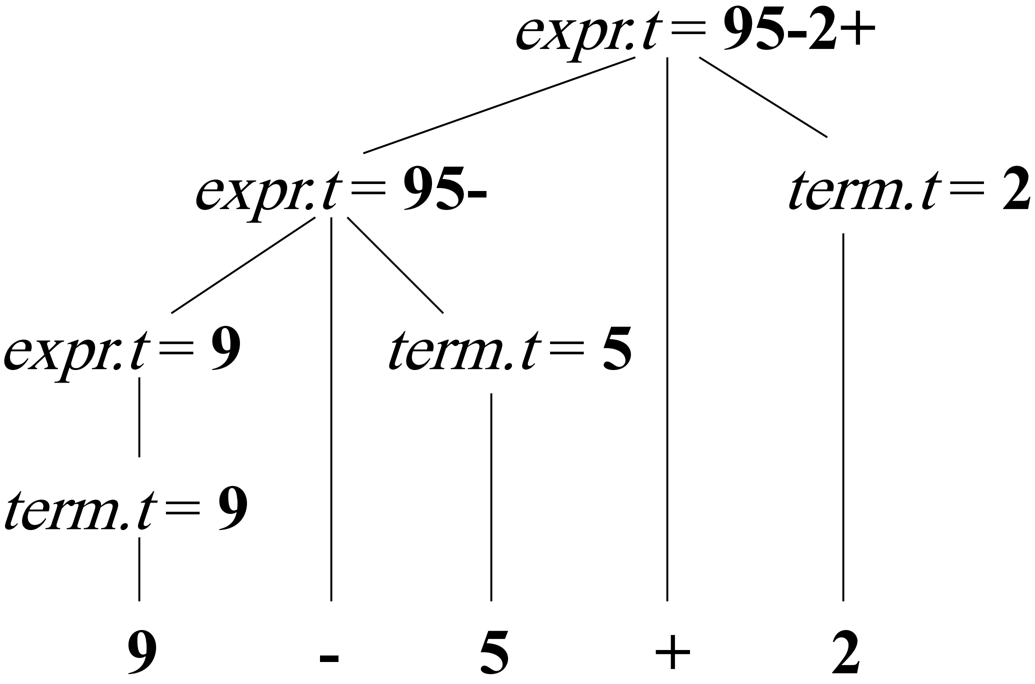
Synthesized and Inherited Attributes

- 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
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \,//\, term.t \,//$ “+”
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \,//\, term.t \,//$ “-”
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow \mathbf{0}$	$term.t :=$ “0”
$term \rightarrow \mathbf{1}$	$term.t :=$ “1”
…	…
$term \rightarrow \mathbf{9}$	$term.t :=$ “9”

Example Annotated Parse Tree



expr.t = **95-2+**

expr.t = **95-**

term.t = **2**

expr.t = **9**

term.t = **5**

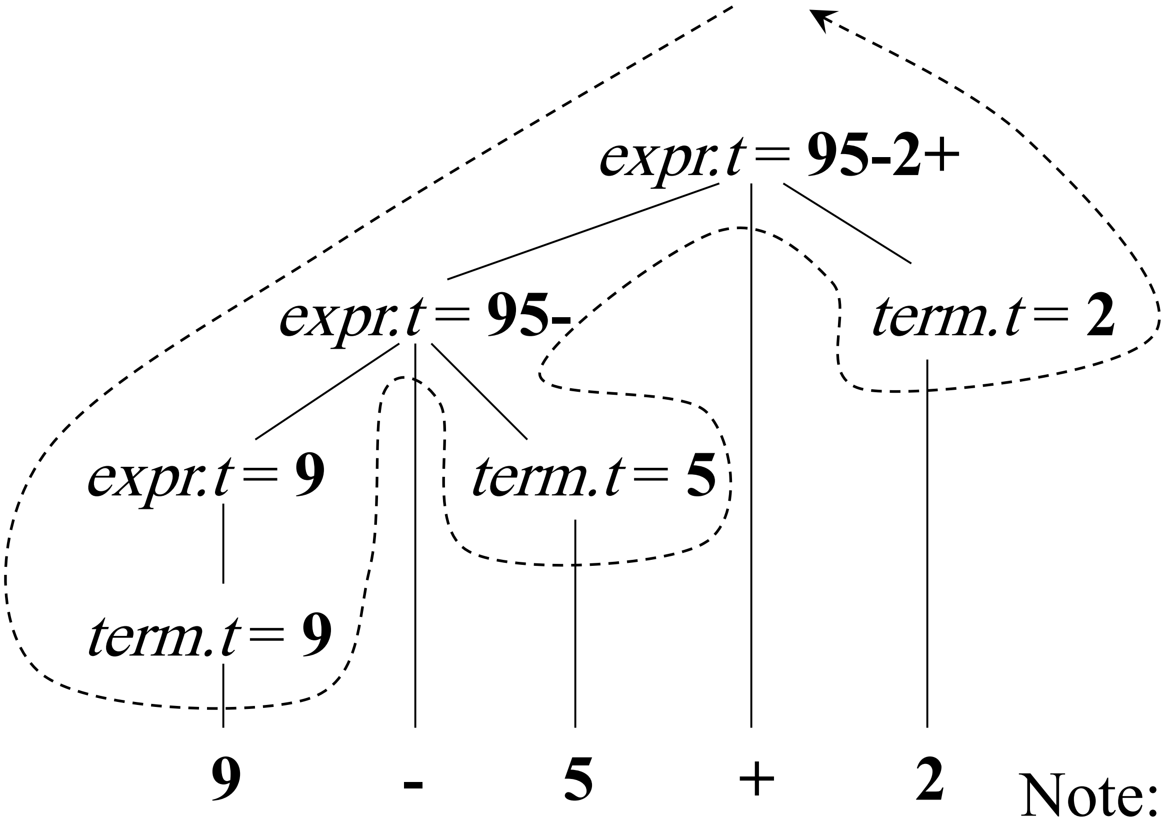
term.t = **9**

9 **-** **5** **+** **2**

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)

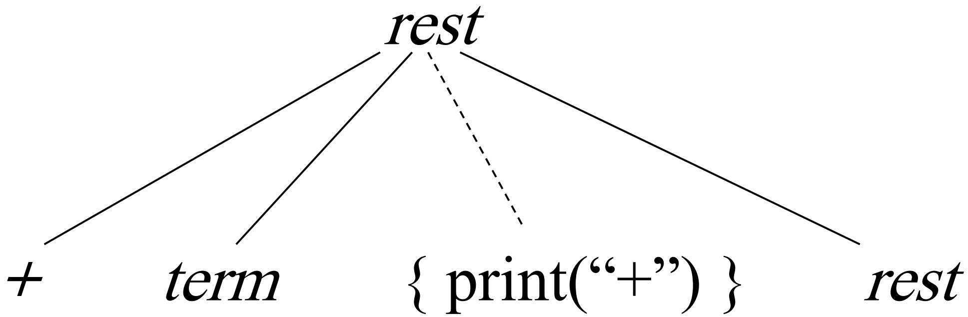


Note: all attributes are of the synthesized type

Translation Schemes

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

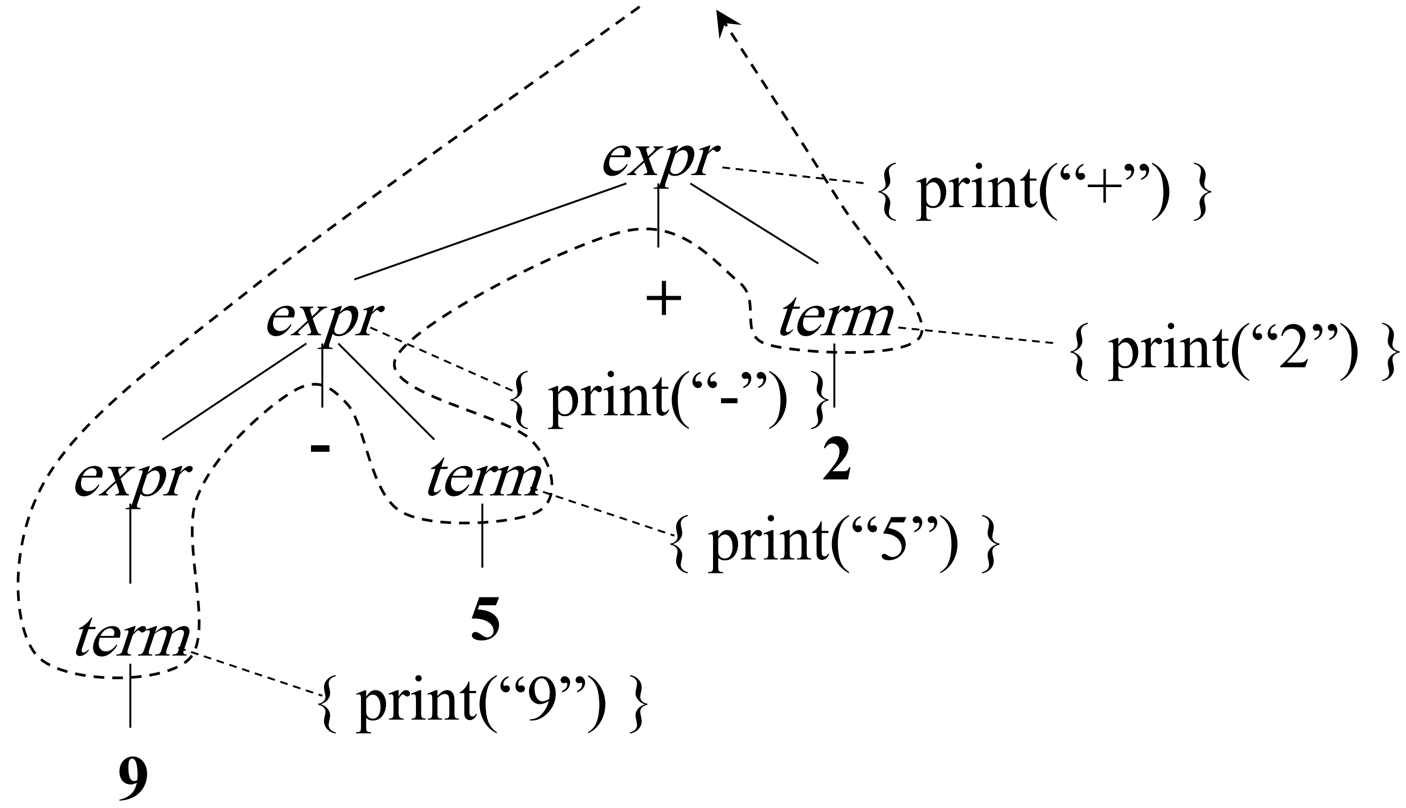
$$\mathit{rest} \to + \; \mathit{term} \; \underbrace{\{\ \text{print(“+”)}\ \}}_{\text{Embedded semantic action}} \; \mathit{rest}$$



Example Translation Scheme

expr → *expr* + *term* { print(“+”) }
expr → *expr* - *term* { print(“-”) }
expr → *term*
term → **0** { print(“0”) }
term → **1** { print(“1”) }
… …
term → **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)

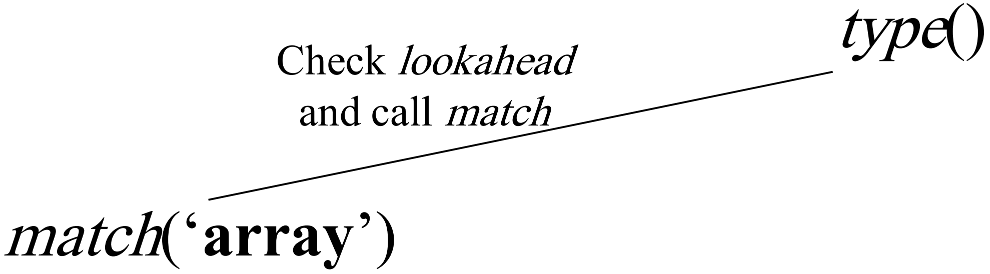
type → *simple*
/ **^ id**
/ **array [** *simple* **] of** *type*
simple → **integer**
/ **char**
/ **num dotdot num**

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*

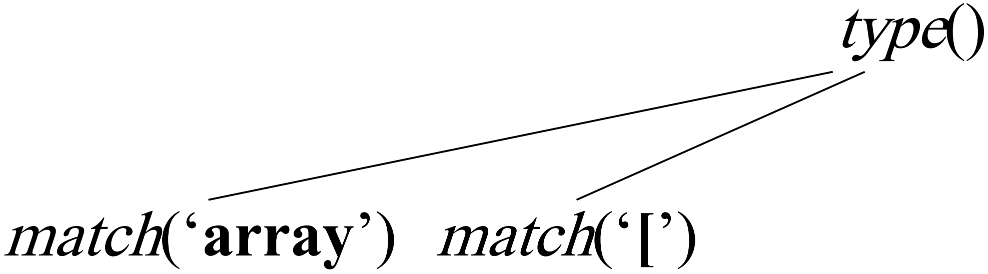
match(‘**array**’)

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

↑

lookahead

Example Predictive Parser (Execution Step 2)



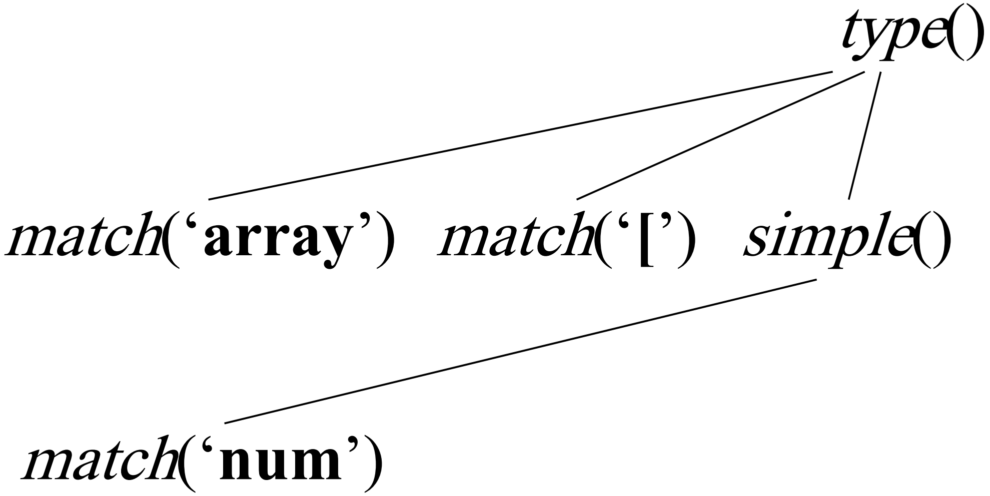
type()

match(‘**array**’) *match*(‘**[**’)

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

lookahead

Example Predictive Parser (Execution Step 3)



type()

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

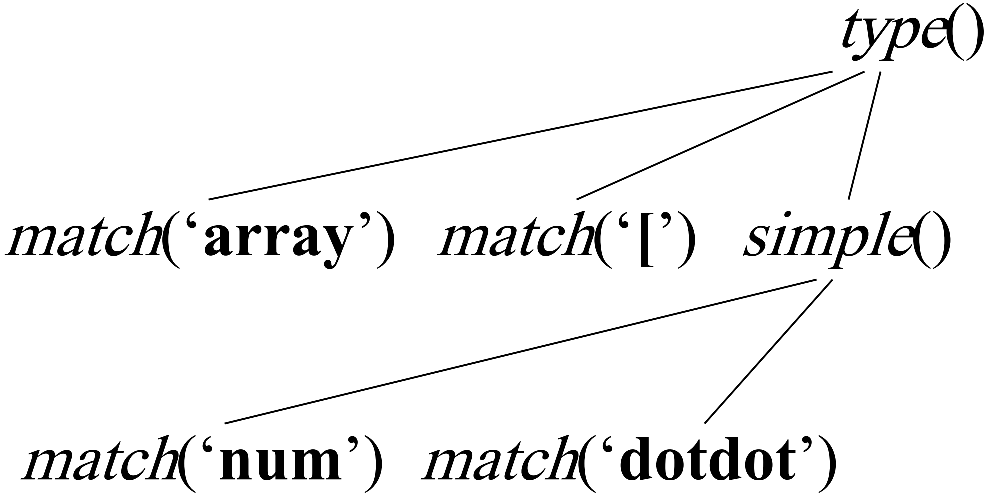
match(‘**num**’)

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

↑

lookahead

Example Predictive Parser (Execution Step 4)



type()

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

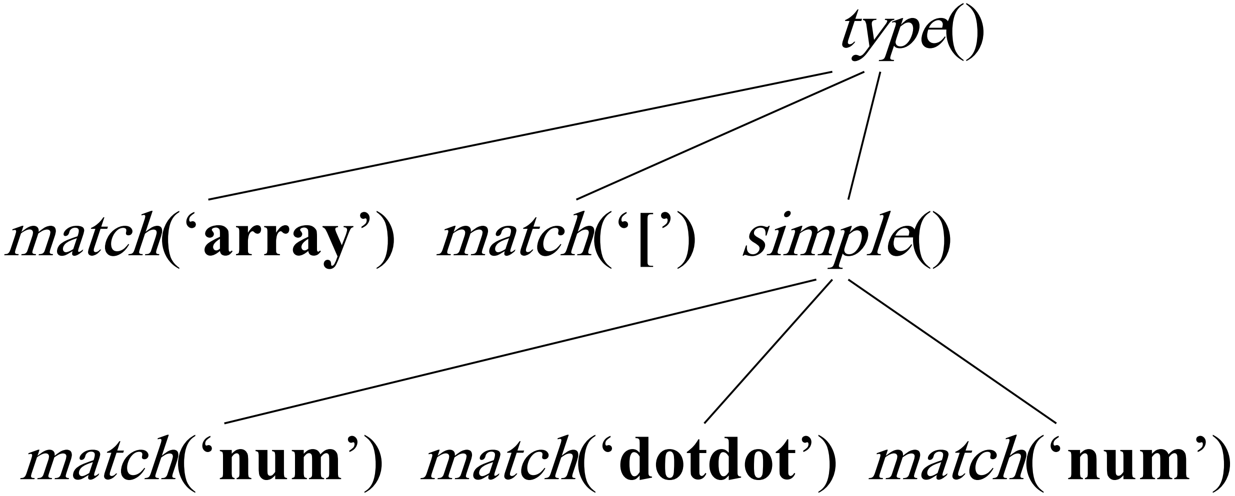
match(‘**num**’) *match*(‘**dotdot**’)

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

↑

lookahead

Example Predictive Parser (Execution Step 5)



type()

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

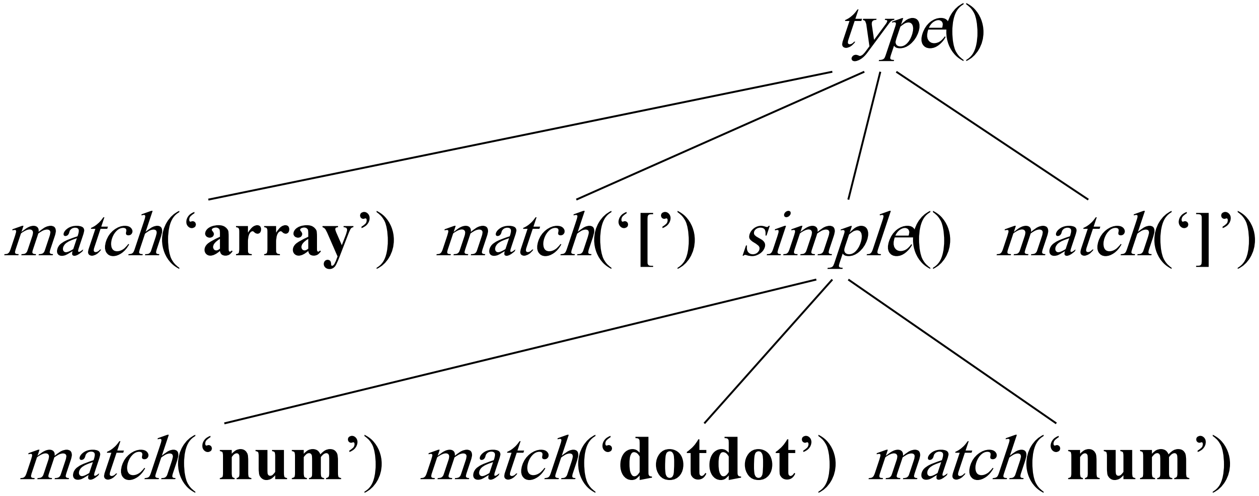
match(‘**num**’) *match*(‘**dotdot**’) *match*(‘**num**’)

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

↑

lookahead

Example Predictive Parser (Execution Step 6)



type()

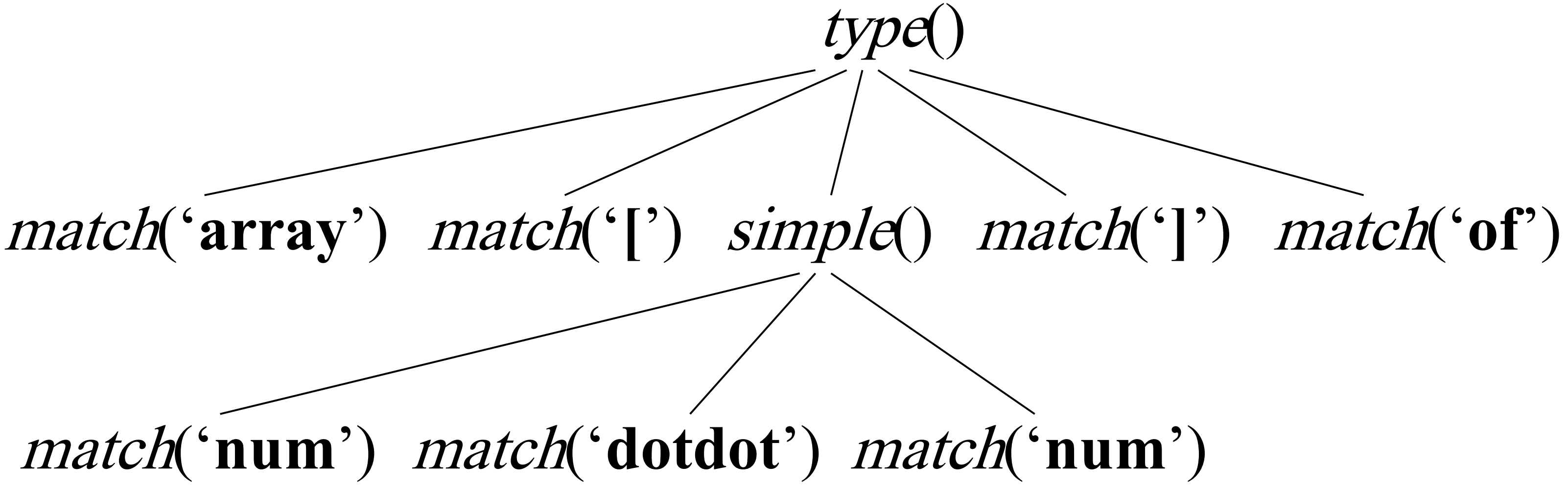
match(‘**array**’) *match*(‘**[**’) *simple*() *match*(‘**]**’)

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

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

↑ *lookahead*

Example Predictive Parser (Execution Step 7)



type()

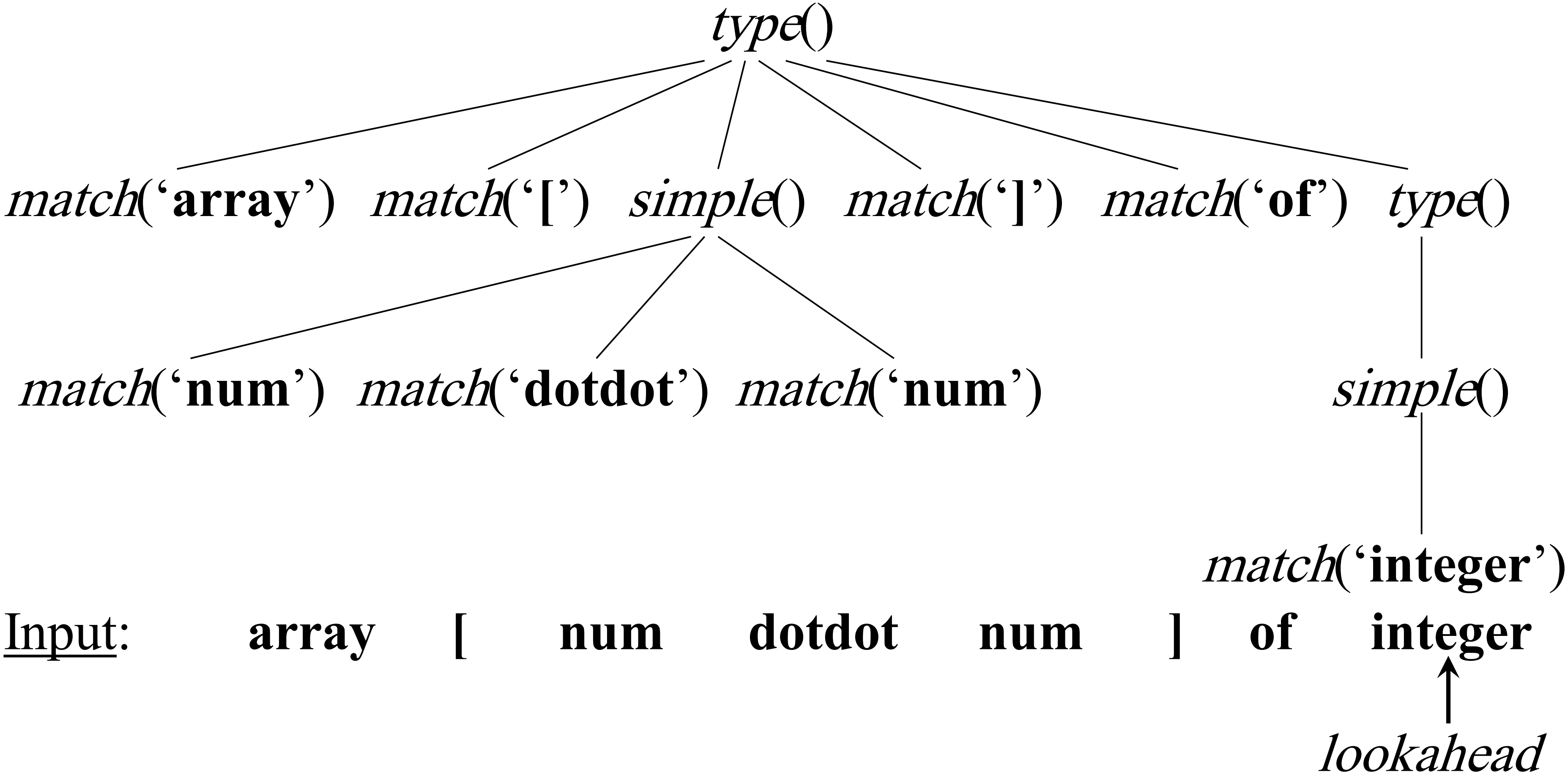
match(‘**array**’) *match*(‘**[**’) *simple*() *match*(‘**]**’) *match*(‘**of**’)

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

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

↑ *lookahead*

Example Predictive Parser (Execution Step 8)



type()

match(‘**array**’) *match*(‘**[**’) *simple*() *match*(‘**]**’) *match*(‘**of**’) *type*()

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

match(‘**integer**’)

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

↑

lookahead

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

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

```
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

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

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

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

We can use *left factoring* to fix the problem

$$\begin{aligned} stmt &\to \textbf{if}\ \text{expr}\ \textbf{then}\ stmt\ opt\_else \\ opt\_else &\to \textbf{else}\ stmt \\ &/\ \varepsilon \end{aligned}$$

Left Recursion

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

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

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

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

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 → *term rest*

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

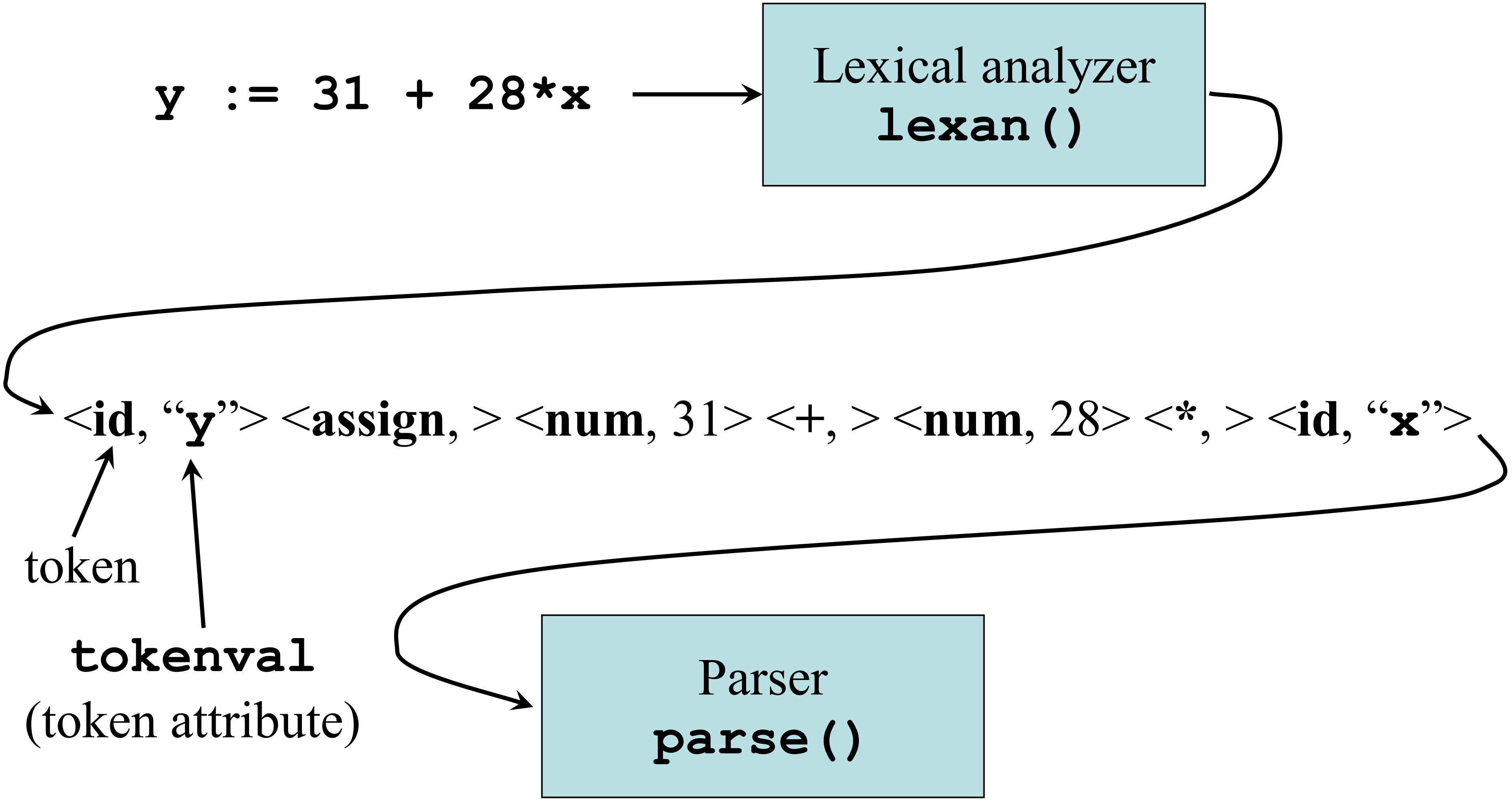
term → **0** { print(“0”) }
term → **1** { print(“1”) }
…
term → **9** { print(“9”) }

```
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);
}
```

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



`y := 31 + 28*x`

Lexical analyzer
`lexan()`

<**id**, “**`y`**”> <**assign**, > <**num**, 31> <+, > <**num**, 28> <*, > <**id**, “**`x`**”>

token

`tokenval`
(token attribute)

Parser
`parse()`

Token Attributes

factor → **(** *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

factor → **(** *expr* **)**
| **id** { print(**id**.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

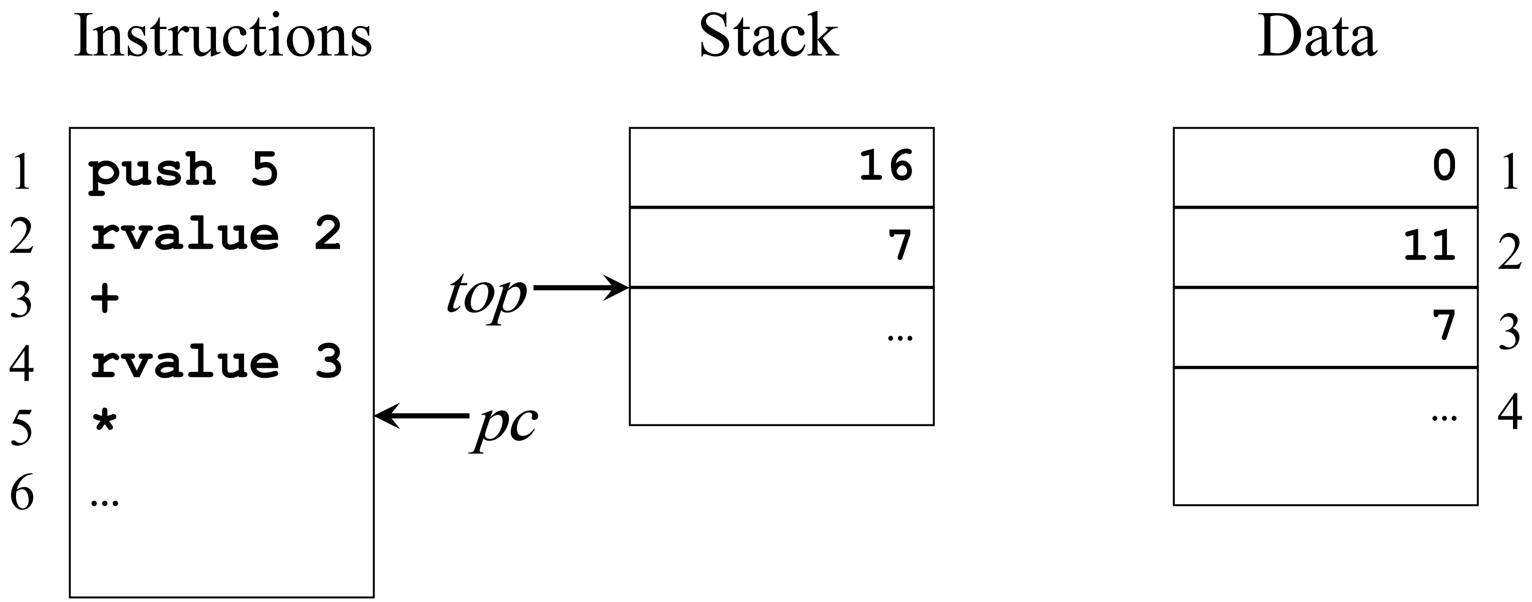
```
/* global.h */
#define DIV 257 /* token */
#define MOD 258 /* token */
#define ID  259 /* token */

/* 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
        …
}
```

Abstract Stack Machines



Instructions

```
1  push 5
2  rvalue 2
3  +
4  rvalue 3
5  *
6  …
```

pc

Stack

16
7
…

top

Data

0	1
11	2
7	3
…	4

Generic Instructions for Stack Manipulation

`push` *v*	push constant value *v* onto the stack
`rvalue` *l*	push contents of data location *l*
`lvalue` *l*	push address of data location *l*
`pop`	discard value on top of the stack
`:=`	the r-value on top is placed in the l-value below it and both are popped
`copy`	push a copy of the top value on the stack
`+`	add value on top with value below it pop both and push result
`-`	subtract value on top from value below it pop both and push result
`*, /,` …	ditto for other arithmetic operations
`<, &,` …	ditto for relational and logical operations

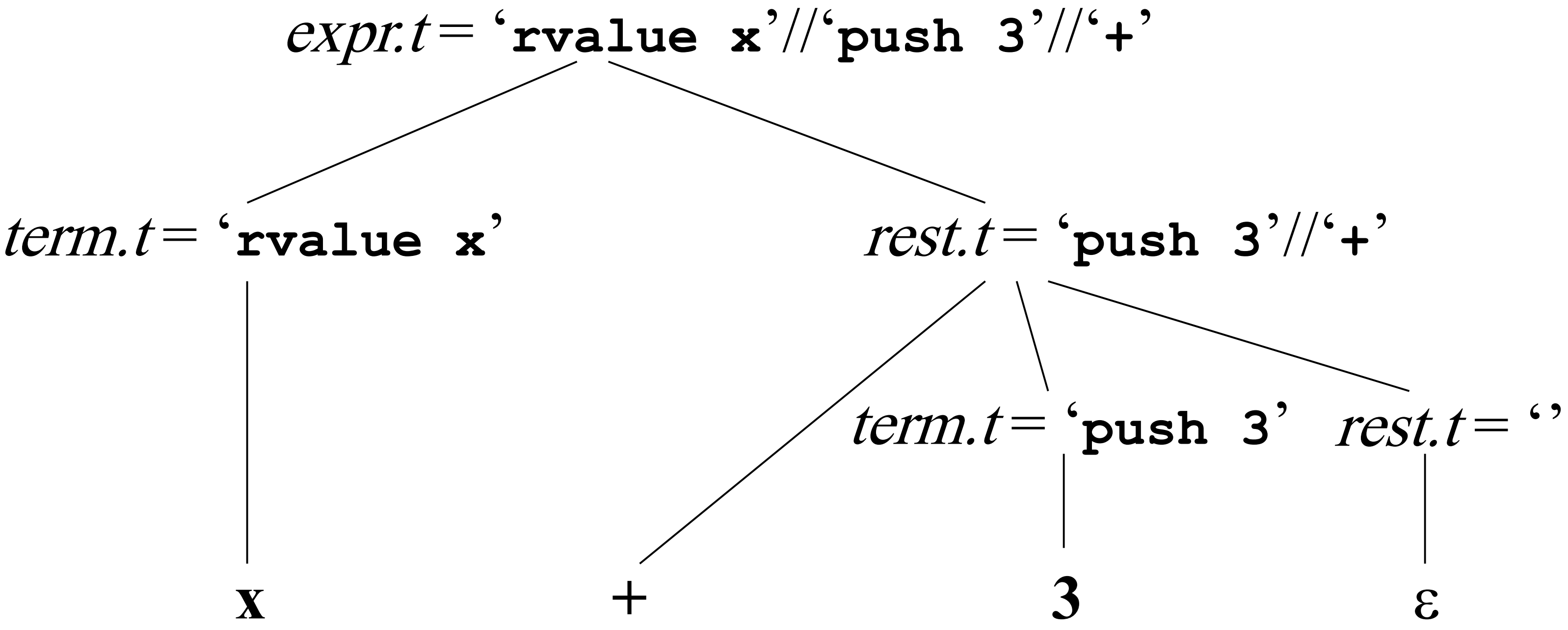
Generic Control Flow Instructions

`label` *l*	label instruction with *l*
`goto` *l*	jump to instruction labeled *l*
`gofalse` *l*	pop the top value, if zero then jump to *l*
`gotrue` *l*	pop the top value, if nonzero then jump to *l*
`halt`	stop execution
`jsr` *l*	jump to subroutine labeled *l*, push return address
`return`	pop return address and return to caller

Syntax-Directed Translation of Expressions

expr → *term rest* { *expr.t* := *term.t* // *rest.t* }
rest → **+** *term* $rest_1$ { *rest.t* := *term.t* // ‘**+**’ // $rest_1.t$ }
rest → **-** *term* $rest_1$ { *rest.t* := *term.t* // ‘**-**’ // $rest_1.t$ }
rest → ε { *rest.t* := ‘’ }
term → **num** { *term.t* := ‘`push `’ // **num**.*value* }
term → **id** { *term.t* := ‘`rvalue `’ // **id**.*lexeme* }

Syntax-Directed Translation of Expressions (cont’d)



expr.t = ‘**`rvalue x`**’//‘**`push 3`**’//‘**`+`**’

term.t = ‘**`rvalue x`**’

rest.t = ‘**`push 3`**’//‘**`+`**’

term.t = ‘**`push 3`**’

rest.t = ‘’

x

+

3

ε

Translation Scheme to Generate Abstract Machine Code

expr → *term moreterms*
moreterms → + *term* { print(‘`+`’) } *moreterms*
moreterms → - *term* { print(‘`-`’) } *moreterms*
moreterms → ε
term → *factor morefactors*
morefactors → * *factor* { print(‘`*`’) } *morefactors*
morefactors → **div** *factor* { print(‘`DIV`’) } *morefactors*
morefactors → **mod** *factor* { print(‘`MOD`’) } *morefactors*
morefactors → ε
factor → **(** *expr* **)**
factor → **num** { print(‘`push `’ // **num**.*value*) }
factor → **id** { print(‘`rvalue `’ // **id**.*lexeme*) }

Translation Scheme to Generate Abstract Machine Code (cont'd)

stmt → **id :=** { print('**`lvalue `**' // **id**.*lexeme*) } *expr* { print('**`:=`**') }

`lvalue` **id**.*lexeme*
code for *expr*
`:=`

Translation Scheme to Generate Abstract Machine Code (cont’d)

stmt → **if** *expr* { *out* := newlabel(); print(‘`gofalse `’ // *out*) }
then *stmt* { print(‘`label `’// *out*) }

code for *expr*
`gofalse` *out*
code for *stmt*
`label` *out*

Translation Scheme to Generate Abstract Machine Code (cont’d)

stmt → **while** { *test* := newlabel(); print(‘**`label `**’ // *test*) }
expr { *out* := newlabel(); print(‘**`gofalse `**’ // *out*) }
do *stmt* { print(‘**`goto `**’ // *test* // ‘**`label `**’ // *out*) }

`label` *test*
code for *expr*
`gofalse` *out*
code for *stmt*
`goto` *test*
`label` *out*

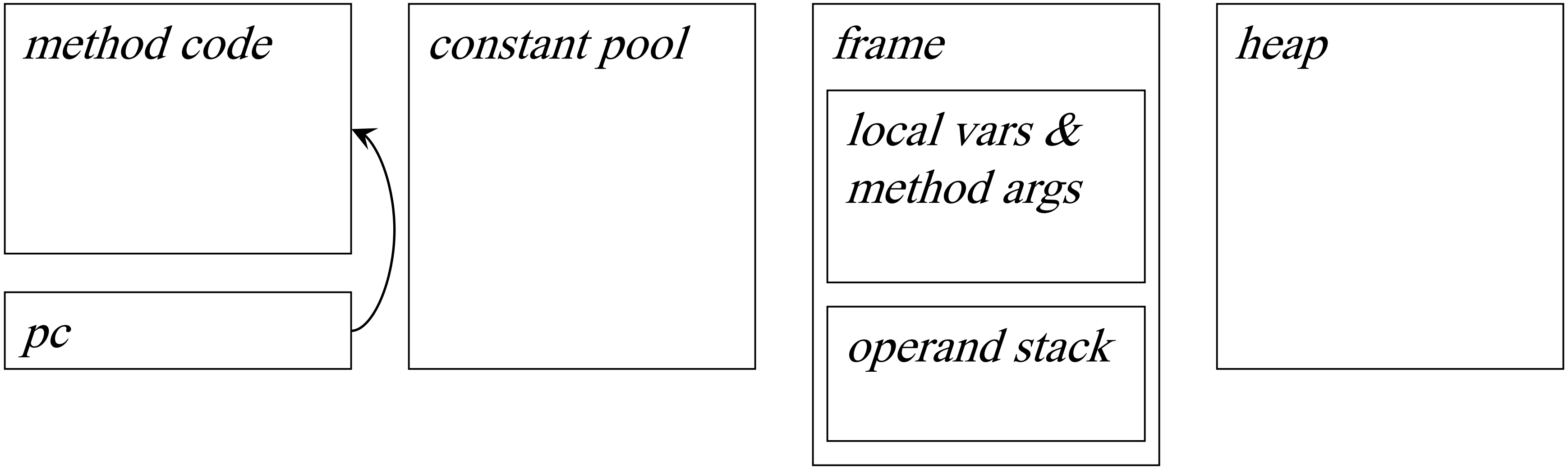
Translation Scheme to Generate Abstract Machine Code (cont’d)

start → *stmt* { print(‘`halt`’) }
stmt → **begin** *opt_stmts* **end**
opt_stmts → *stmt* **;** *opt_stmts* | ε

The JVM

- Abstract stack machine architecture
  - Emulated in software with JVM interpreter
  - *Just-In-Time* (JIT) compilers
  - Hardware implementations available
- Java *bytecode*
  - Platform independent
  - Small
  - Safe
- The Java^{TM} Virtual Machine Specification, 2nd ed.
  `http://java.sun.com/docs/books/vmspec`

Runtime Data Areas (§3.5)



method code

pc

constant pool

frame

local vars & method args

operand stack

heap

Constant Pool (§3.5.5)

- Serves a function similar to that of a symbol table
- Contains several kinds of constants
- Method and field references, strings, float constants, and integer constants larger than 16 bit cannot be used as operands of bytecode instructions and must be loaded on the operand stack from the constant pool
- Java *bytecode verification* is a pre-execution process that checks the consistency of the bytecode instructions and constant pool

Frames (§3.6)

- A new *frame* (also known as *activation record*) is created each time a method is invoked
- A frame is destroyed when its method invocation completes
- Each frame contains an array of variables known as its *local variables* indexed from 0
  - Local variable 0 is "*this*" (unless the method is static)
  - Followed by method parameters
  - Followed by the local variables of blocks
- Each frame contains an *operand stack*

Data Types (§3.2, §3.3, §3.4)

`byte`	a 8-bit signed two’s complement integer
`short`	a 16-bit signed two’s complement integer
`int`	a 32-bit signed two’s complement integer
`long`	a 64-bit signed two’s complement integer
`char`	a 16-bit Unicode character
`float`	a 32-bit IEEE 754 single-precision float value
`double`	a 64-bit IEEE 754 double-precision float value
`boolean`	a virtual type only, **`int`** is used to represent true (1) false (0)
`returnAddress`	the location of the *pc* after method invocation
`reference`	a 32-bit address reference to an object of *class type*, *array type*, or *interface type* (value can be NULL)

Operand stack has 32-bit slots, thus **`long`** and **`double`** occupy two slots

Instruction Set (§3.11, §6)

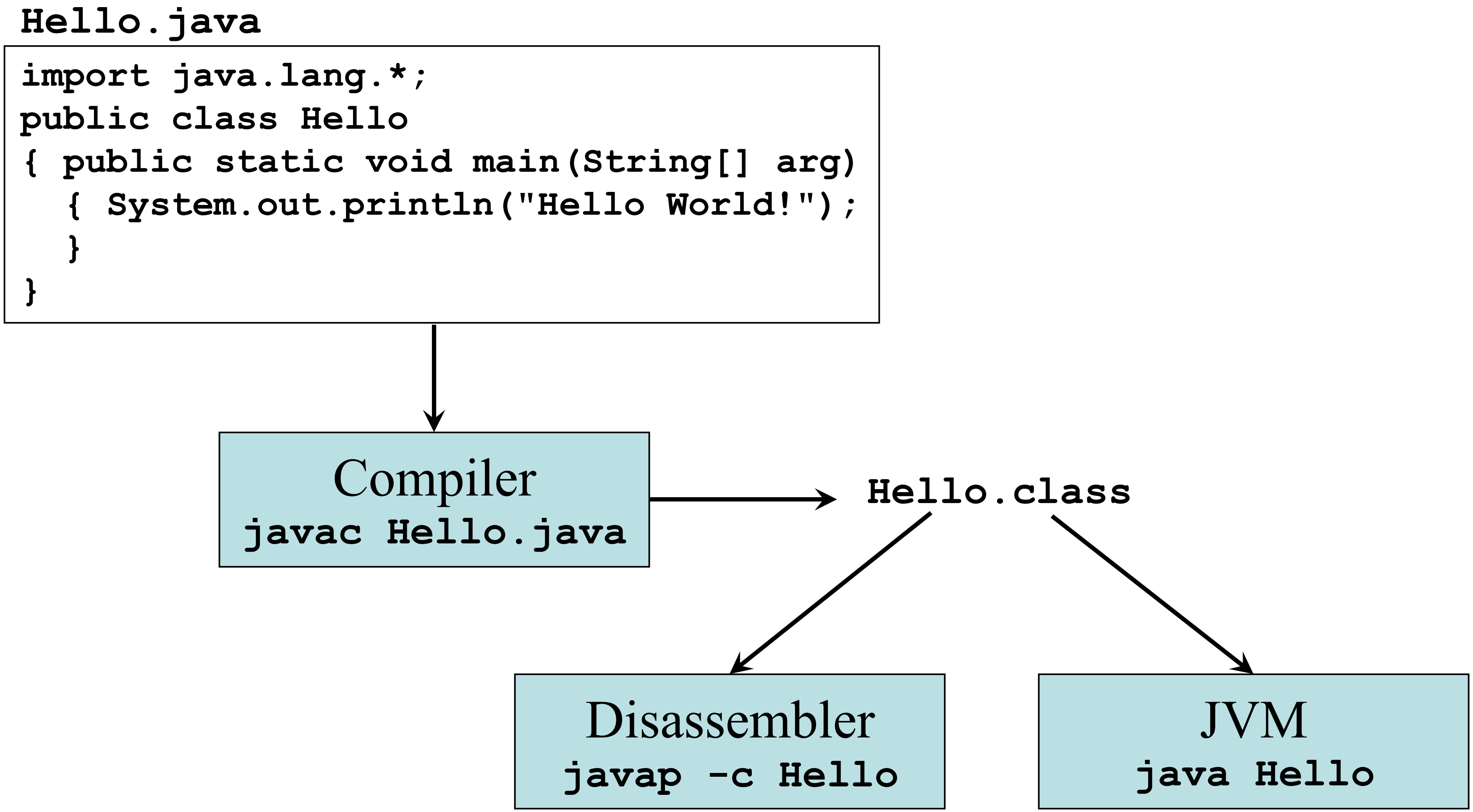
opcode	***byte***	***short***	***int***	***long***	***float***	***double***	***char***	***reference***
Tipush	*bipush*	*sipush*						
Tconst			*iconst*	*lconst*	*fconst*	*dconst*		*aconst*
Tload			*iload*	*lload*	*fload*	*dload*		*aload*
Tstore			*istore*	*lstore*	*fstore*	*dstore*		*astore*
Tinc			*iinc*					
Taload	*baload*	*saload*	*iaload*	*laload*	*faload*	*daload*	*caload*	*aaload*
Tastore	*bastore*	*sastore*	*iastore*	*lastore*	*fastore*	*dastore*	*castore*	*aastore*
Tadd			*iadd*	*ladd*	*fadd*	*dadd*		
Tsub			*isub*	*lsub*	*fsub*	*dsub*		
Tmul			*imul*	*lmul*	*fmul*	*dmul*		
Tdiv			*idiv*	*ldiv*	*fdiv*	*ddiv*		
Trem			*irem*	*lrem*	*frem*	*drem*		
Tneg			*ineg*	*lneg*	*fneg*	*dneg*		
Tshl			*ishl*	*lshl*				
Tshr			*ishr*	*lshr*				
Tushr			iushr	lushr				
Tand			*iand*	*land*				
Tor			*ior*	*lor*				
Txor			*ixor*	*lxor*				
i2T	*i2b*	*i2s*		*i2l*	*i2f*	*i2d*		
l2T			*l2i*		*l2f*	*l2d*		
f2T			*f2i*	*f2l*		*f2d*		
d2T			*d2i*	*d2l*	*d2f*			
Tcmp				*lcmp*				
Tcmpl					*fcmpl*	*dcmpl*		
Tcmpg					*fcmpg*	*dcmpg*		
if_TcmpOP			*if_icmpOP*					*if_acmpOP*
Treturn			*ireturn*	*lreturn*	*freturn*	*dreturn*		*areturn*

Actual Type	***Computational Type***	***Category***
boolean	int	category 1
byte	int	category 1
char	int	category 1
short	int	category 1
int	int	category 1
float	float	category 1
reference	reference	category 1
returnAddress	returnAddress	category 1
long	long	category 2
double	double	category 2

The Class File Format (§4)

- A *class file* consists of a stream of 8-bit bytes
- 16-, 32-, and 64-bit quantities are stored in 2, 4, and 8 consecutive bytes in *big-endian* order
- Contains several components, including:
  - Magic number **`0xCAFEBABE`**
  - Version info
  - Constant pool
  - This and super class references (index into pool)
  - Class fields
  - Class methods

javac, javap, java



Hello.java

```
import java.lang.*;
public class Hello
{ public static void main(String[] arg)
  { System.out.println("Hello World!");
  }
}
```

Compiler

`javac Hello.java`

Hello.class

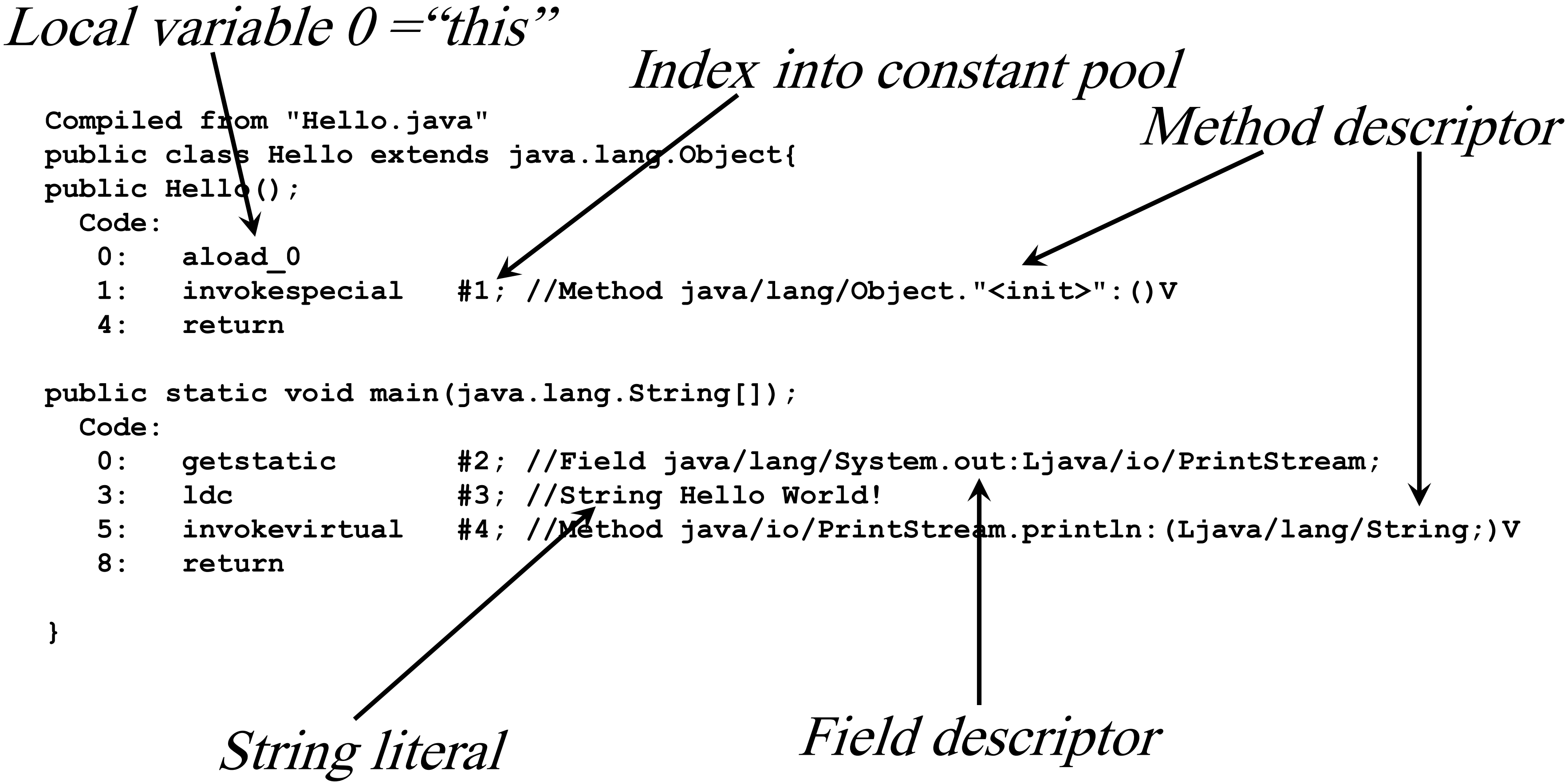
Disassembler

`javap -c Hello`

JVM

`java Hello`

javap -c Hello



Local variable 0 =“this”

Index into constant pool

Method descriptor

```
Compiled from "Hello.java"
public class Hello extends java.lang.Object{
public Hello();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   ldc             #3; //String Hello World!
   5:   invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   8:   return

}
```

String literal

Field descriptor

Field/Method Descriptors (§4.3)

FieldType:

BaseType Character	Type	Interpretation
B	byte	signed byte
C	char	Unicode character
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L<classname>;	reference	an instance of class <classname>
S	short	signed short
Z	boolean	true or false
[	reference	one array dimension

MethodDescriptor:
    (*ParameterDescriptor**) *ReturnDescriptor*

ParameterDescriptor:
    FieldType

ReturnDescriptor:
    FieldType
    `V`

Generating Code for the JVM

expr → *term moreterms*

moreterms → + *term* { emit(**`iadd`**) } *moreterms*

moreterms → **-** *term* { emit(**`isub`**) } *moreterms*

moreterms → ε

term → *factor morefactors*

morefactors → ***** *factor* { emit(**`imul`**) } *morefactors*

morefactors → **div** *factor* { emit(**`idiv`**) } *morefactors*

morefactors → **mod** *factor* { emit(**`irem`**) } *morefactors*

morefactors → ε

factor → **(** *expr* **)**

factor → **int8** { emit2(**`bipush`**, **int8**.*value*) }

factor → **int16** { emit3(**`sipush`**, **int16**.*value*) }

factor → **int32** { *idx* := newpoolint(**int32**.*value*);
emit2(**`ldc`**, *idx*) }

factor → **id** { emit2(**`iload`**, **id**.*index*) }

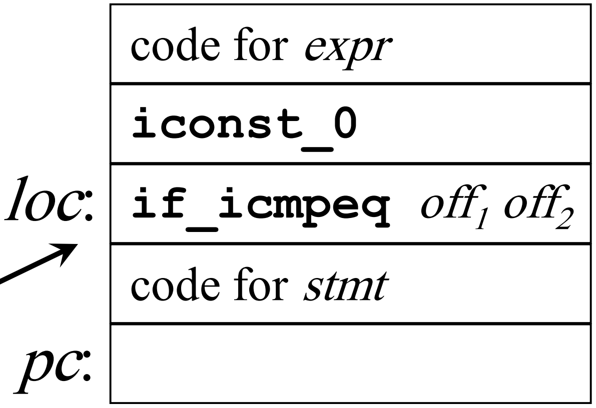
Generating Code for the JVM (cont’d)

stmt → **id :=** *expr* { emit2(`istore`, **id**.*index*) }

code for *expr*

`istore` **id**.*index*

stmt → **if** *expr* { emit(`iconst_0`); *loc* := pc;
emit3(`if_icmpeq`, 0) }
then *stmt* { backpatch(*loc*, pc-*loc*) }



backpatch() sets the offsets of the relative branch when the target *pc* value is known