

A Simple One-Pass Compiler to Generate Bytecode for the JVM

Chapter 2

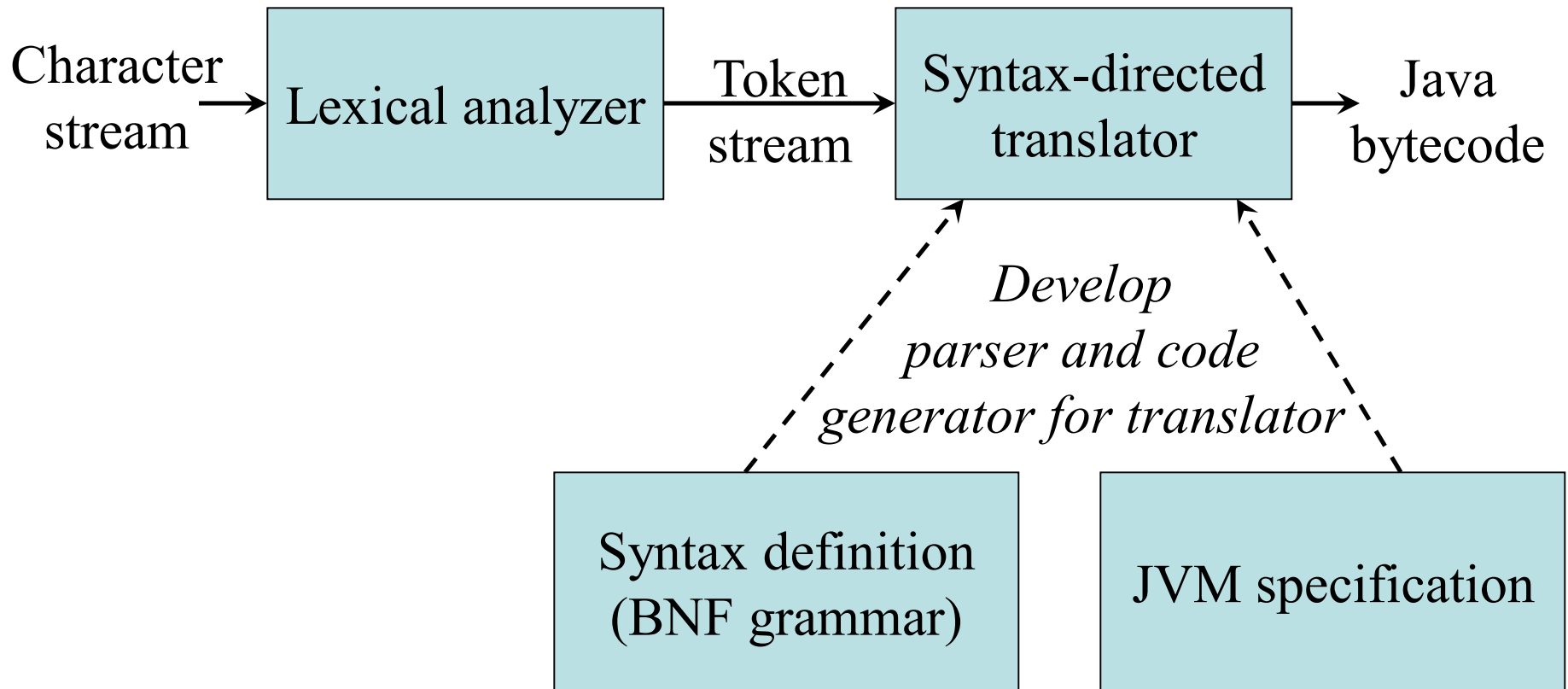
Overview

- This chapter contains introductory material to Chapters 3 to 8 of the Dragon book
- Combined with material on the JVM to prepare for the laboratory assignments

Building a Simple Compiler

- Building our compiler involves:
 - Defining the *syntax* of a programming language
 - Develop a source code parser: for our compiler we will use *predictive parsing*
 - Implementing *syntax directed translation* to generate intermediate code: our target is the JVM *abstract stack machine*
 - Generating Java *bytecode* for the JVM
 - Optimize the Java bytecode (optional)

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:

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

with productions $P =$

$$list \rightarrow list + digit$$

$$list \rightarrow list - digit$$

$$list \rightarrow digit$$

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

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

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

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

Likewise, a *rightmost derivation* replaces the rightmost

AVAILABLE AT:

Onebyzero Edu. Organized Learning, Smoother Career

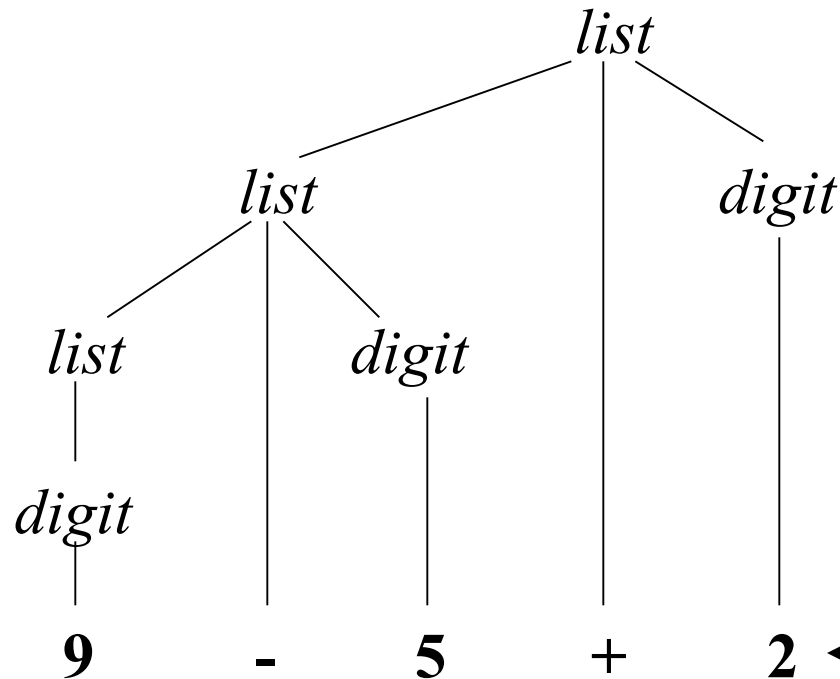
The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

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 $A \rightarrow X_1 X_2 \dots X_n$ is a production, then node A has immediate *children* $X_1, X_2, \dots, X_n$ 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:

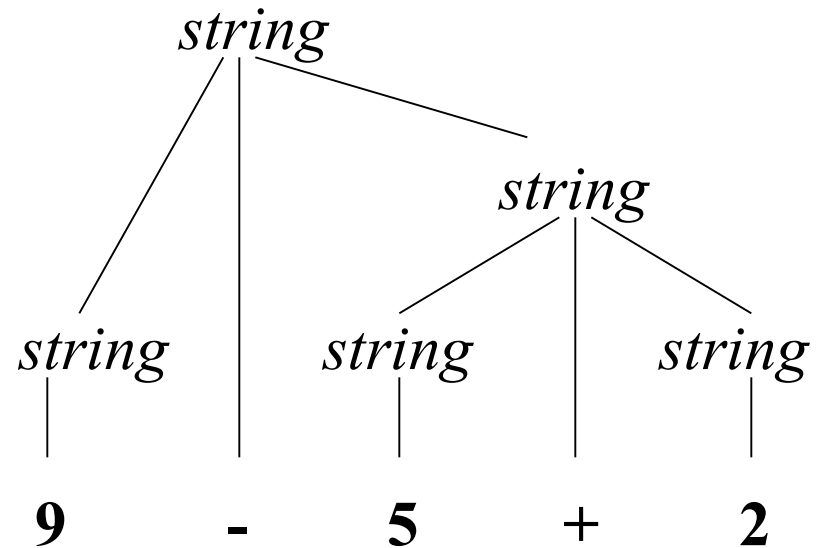
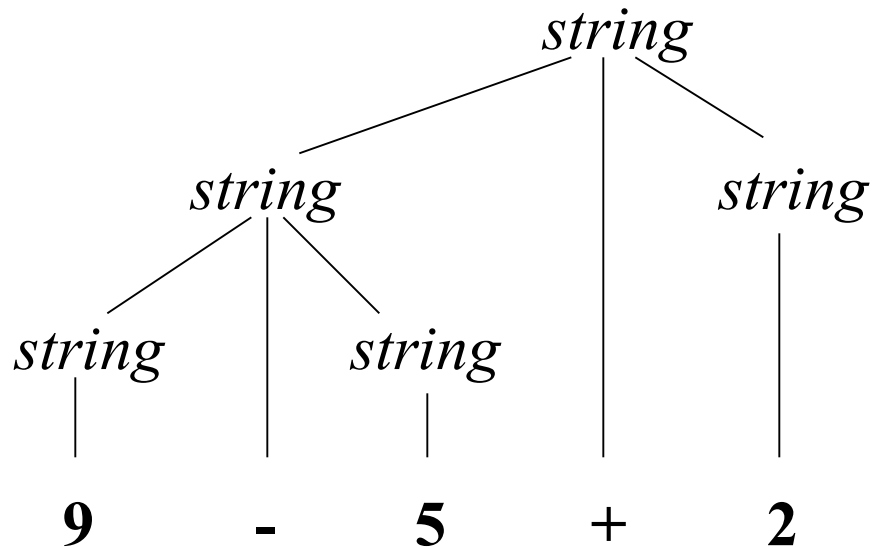
$$G = \langle \{string\}, \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, P, string \rangle$$

with production $P =$

$$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid \dots \mid 9$$

This grammar is *ambiguous*, because more than one parse tree represents 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

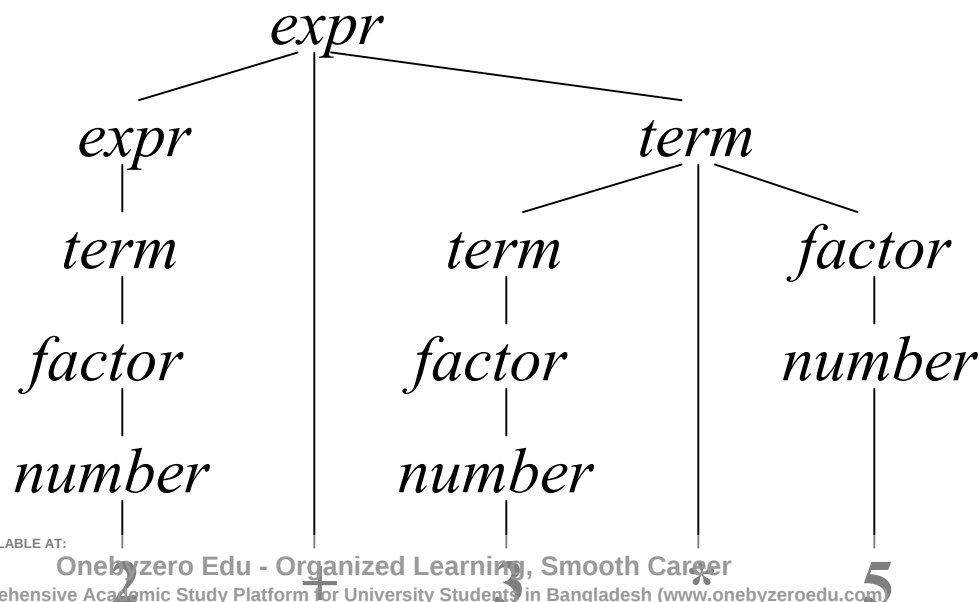
Operators with higher precedence “bind more tightly”

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

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

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

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



Syntax of Statements

$$\begin{aligned} stmt &\rightarrow \mathbf{id} := expr \\ &\quad | \mathbf{if} \, expr \, \mathbf{then} \, stmt \\ &\quad | \mathbf{if} \, expr \, \mathbf{then} \, stmt \, \mathbf{else} \, stmt \\ &\quad | \mathbf{while} \, expr \, \mathbf{do} \, stmt \\ &\quad | \mathbf{begin} \, opt_stmts \, \mathbf{end} \\ opt_stmts &\rightarrow stmt \, ; \, opt_stmts \\ &\quad | \varepsilon \end{aligned}$$

Syntax-Directed Translation

- Uses a CF grammar to specify the syntactic structure of the language
- AND associates a set of *attributes* with 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

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

$expr \rightarrow expr_1 + term$

$expr \rightarrow expr_1 - term$

$expr \rightarrow term$

$term \rightarrow 0$

$term \rightarrow 1$

...

$term \rightarrow 9$

Semantic Rule

$expr.t := expr_1.t \parallel term.t \parallel "+"$

$expr.t := expr_1.t \parallel term.t \parallel "-"$

$expr.t := term.t$

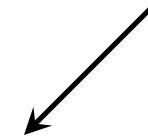
$term.t := "0"$

$term.t := "1"$

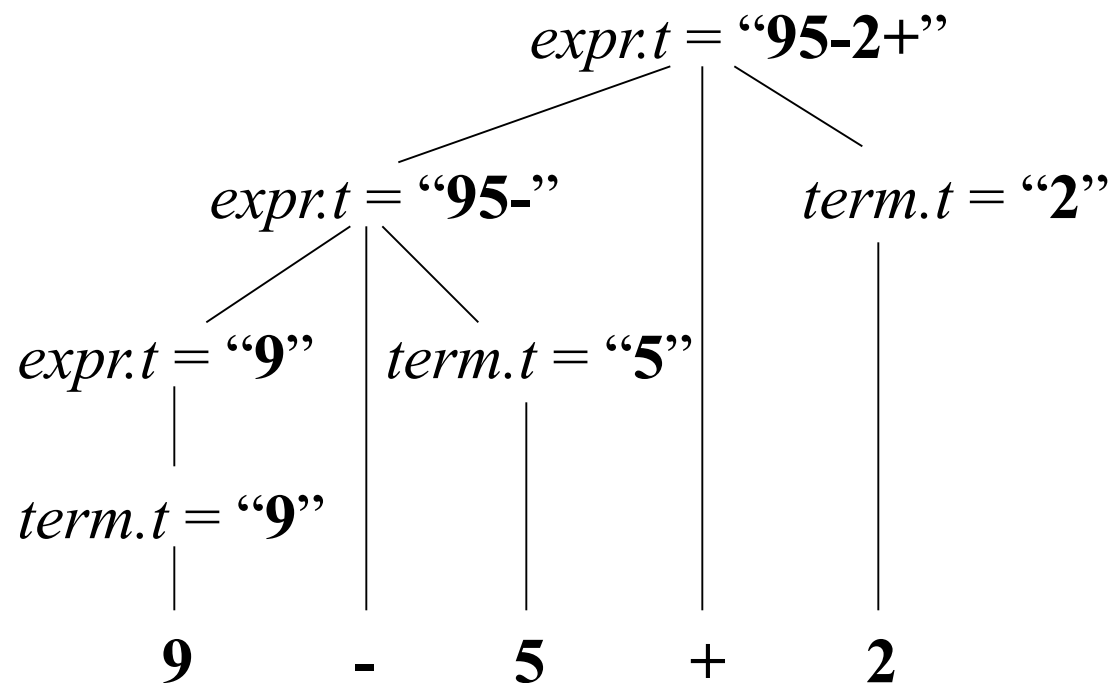
...

$term.t := "9"$

String concat operator



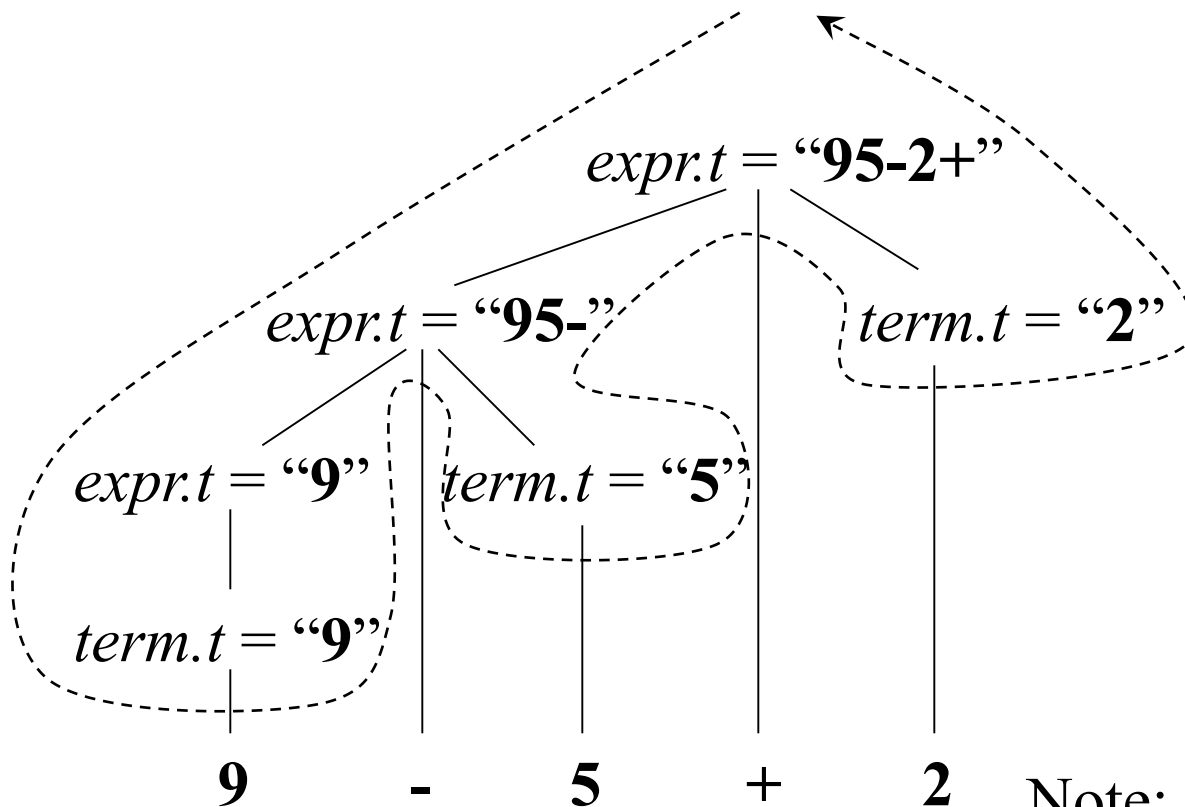
Example Annotated Parse Tree



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)



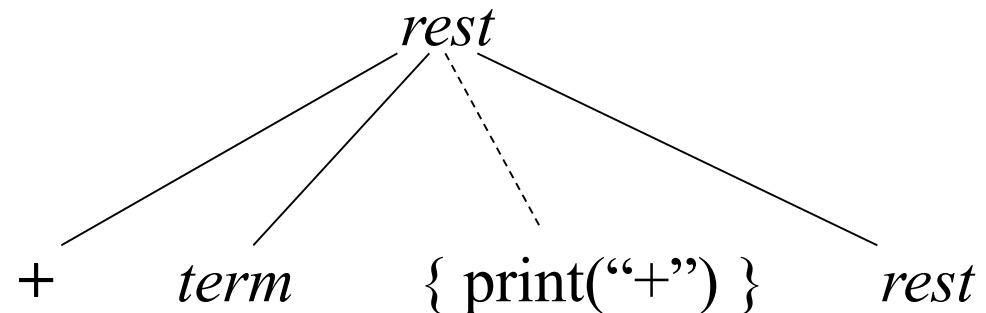
Note: all attributes are of the synthesized type

Translation Schemes

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

$$rest \rightarrow + term \{ \text{print}("+") \} rest$$

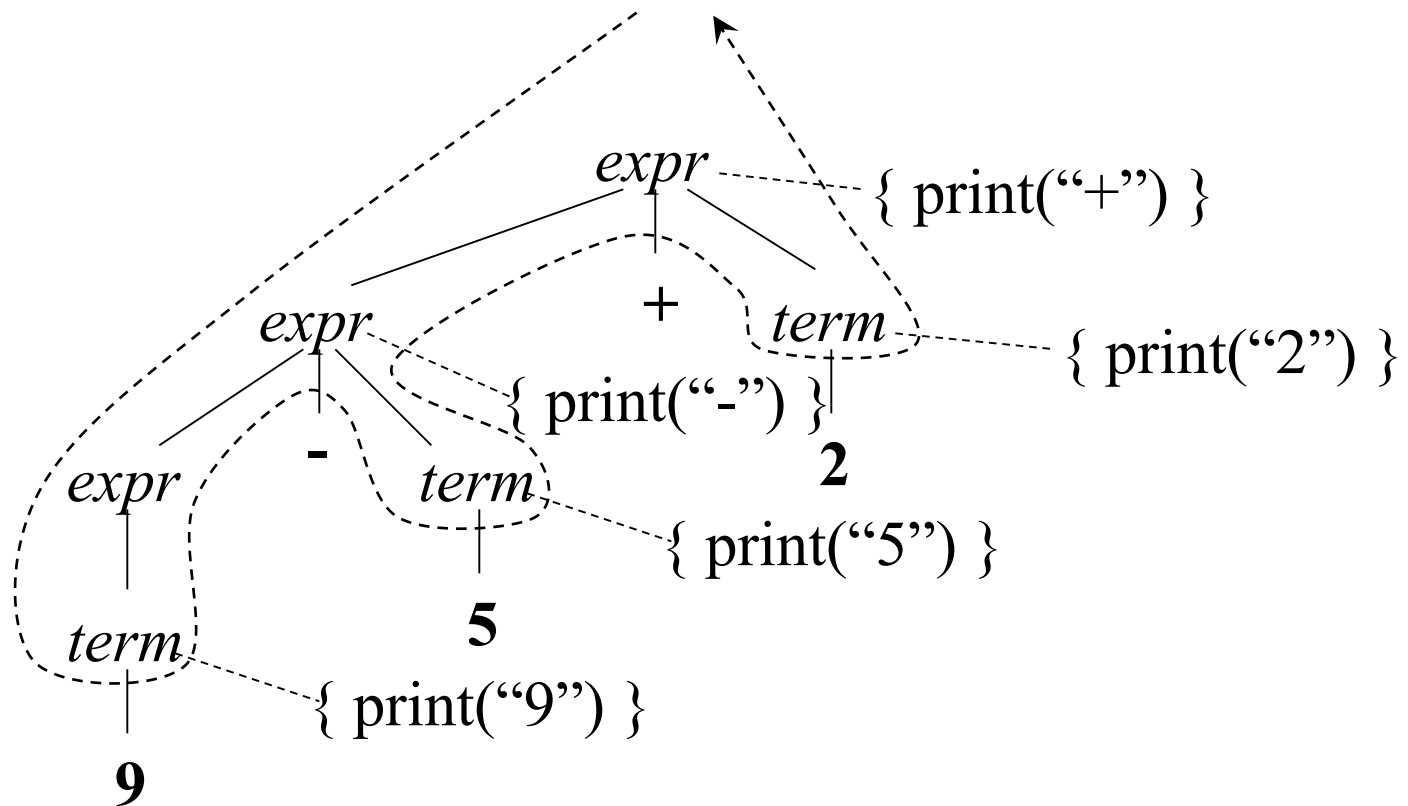

 Embedded
semantic action



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+**

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

Parsing

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

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)

$$\begin{aligned} type &\rightarrow simple \\ &| \textbf{^ id} \\ &| \textbf{array [simple] of type} \\ simple &\rightarrow \textbf{integer} \\ &| \textbf{char} \\ &| \textbf{num dotdot num} \end{aligned}$$

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)

match('array')

*Check lookahead
and call match*

type()

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

↑
lookahead

Example Predictive Parser (Execution Step 2)

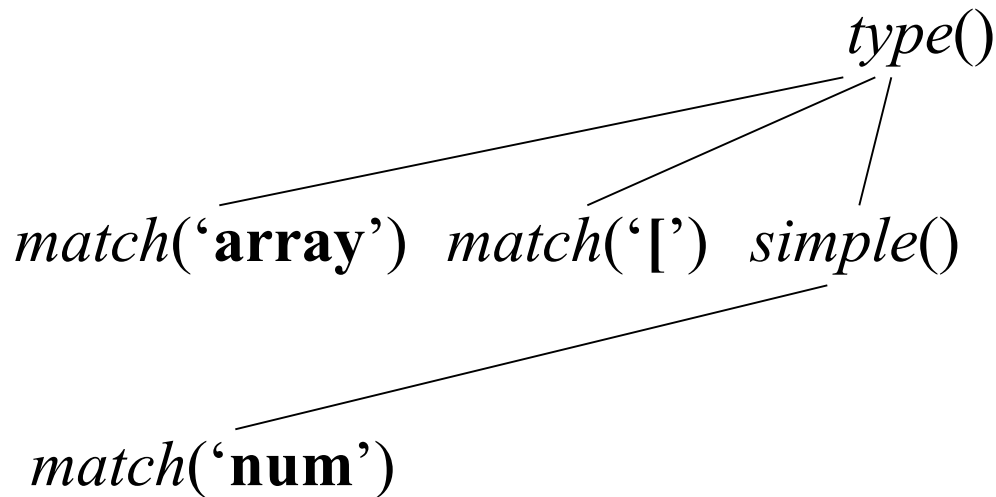
match('array') *match('[')* *type()*

Input: **array** **|** **num** **dotdot** **num** **|** **of** **integer**

AVAILABLE AT

lookahead

Example Predictive Parser (Execution Step 3)



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



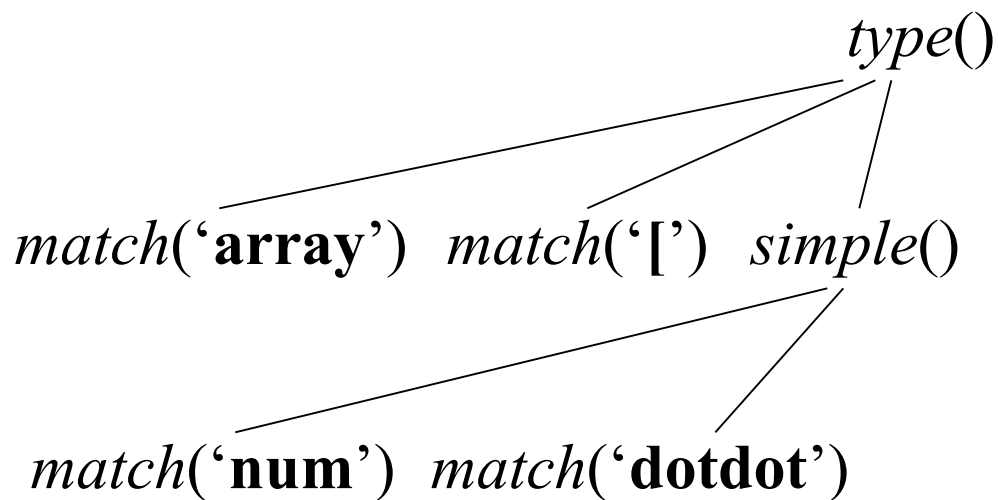
lookahead

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

Example Predictive Parser (Execution Step 4)



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

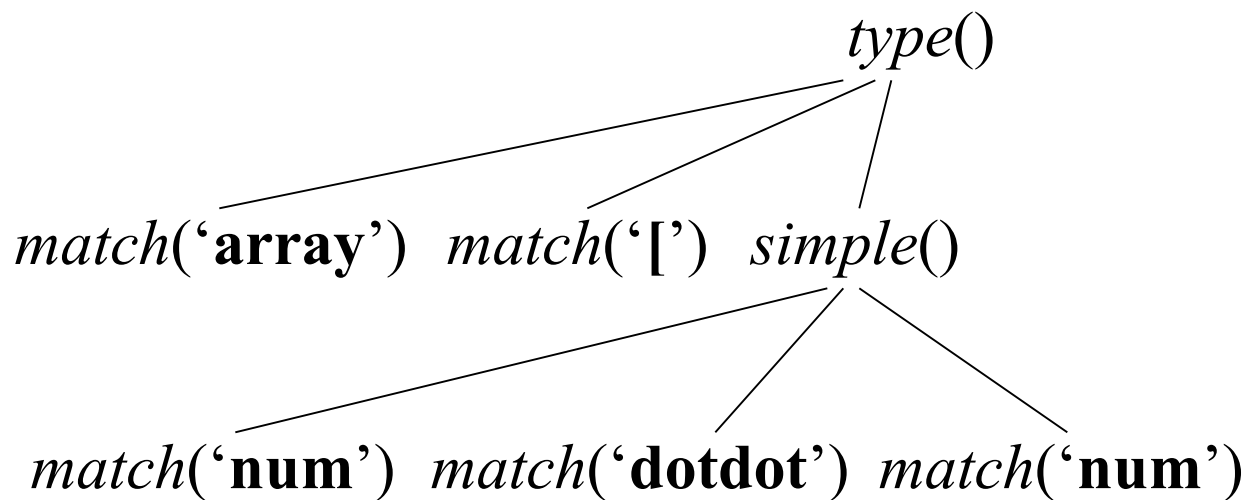


lookahead

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smoother Career
The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

Example Predictive Parser (Execution Step 5)



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

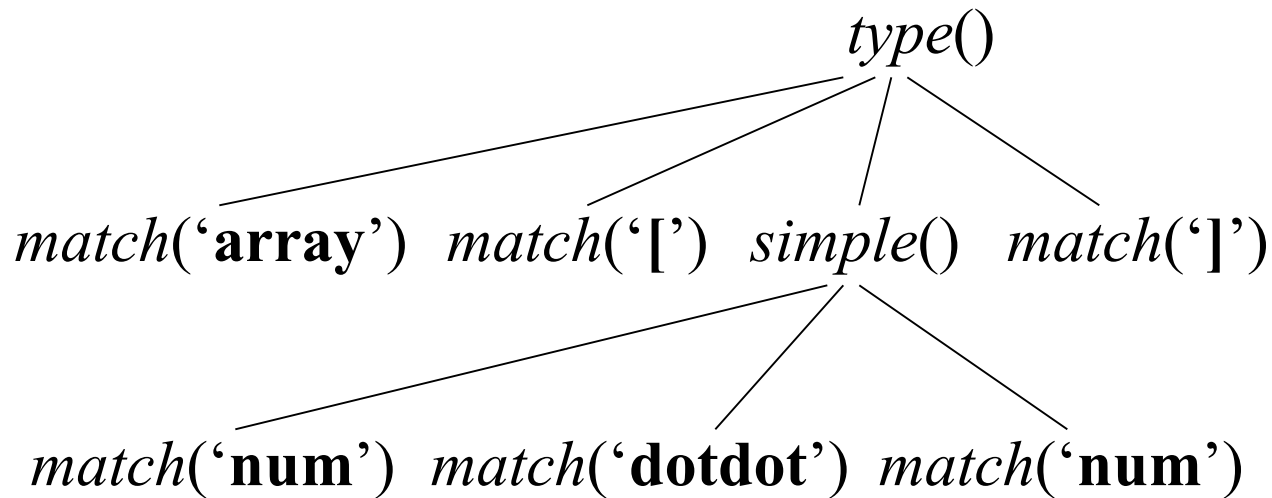


lookahead

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career
The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

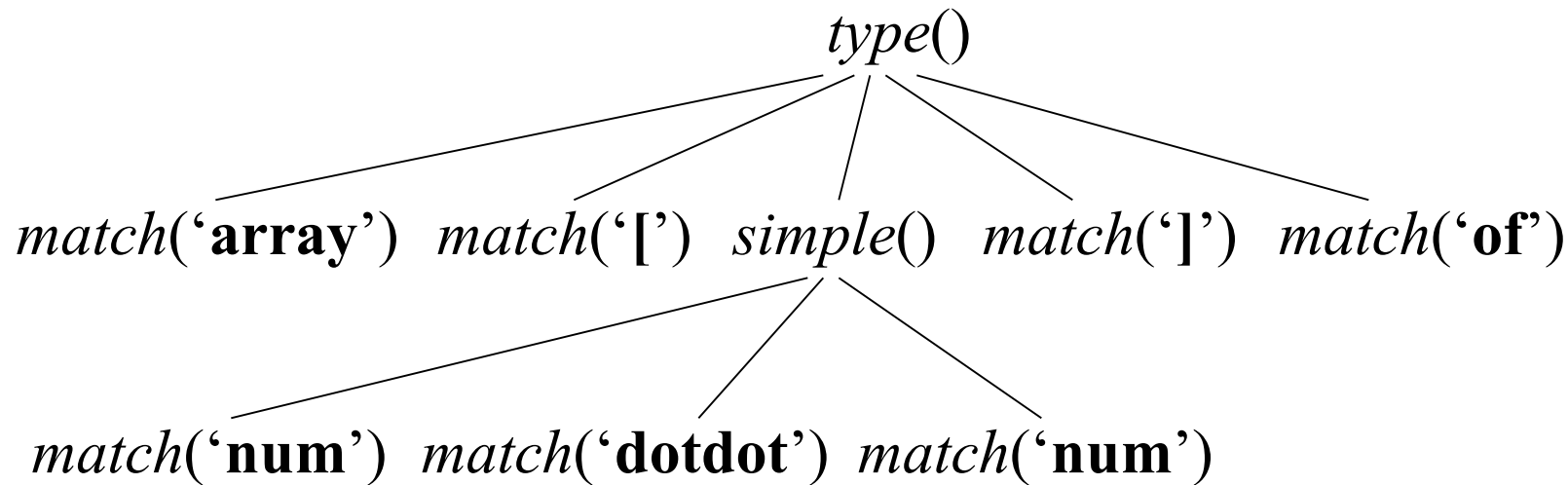
Example Predictive Parser (Execution Step 6)



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

↑
lookahead

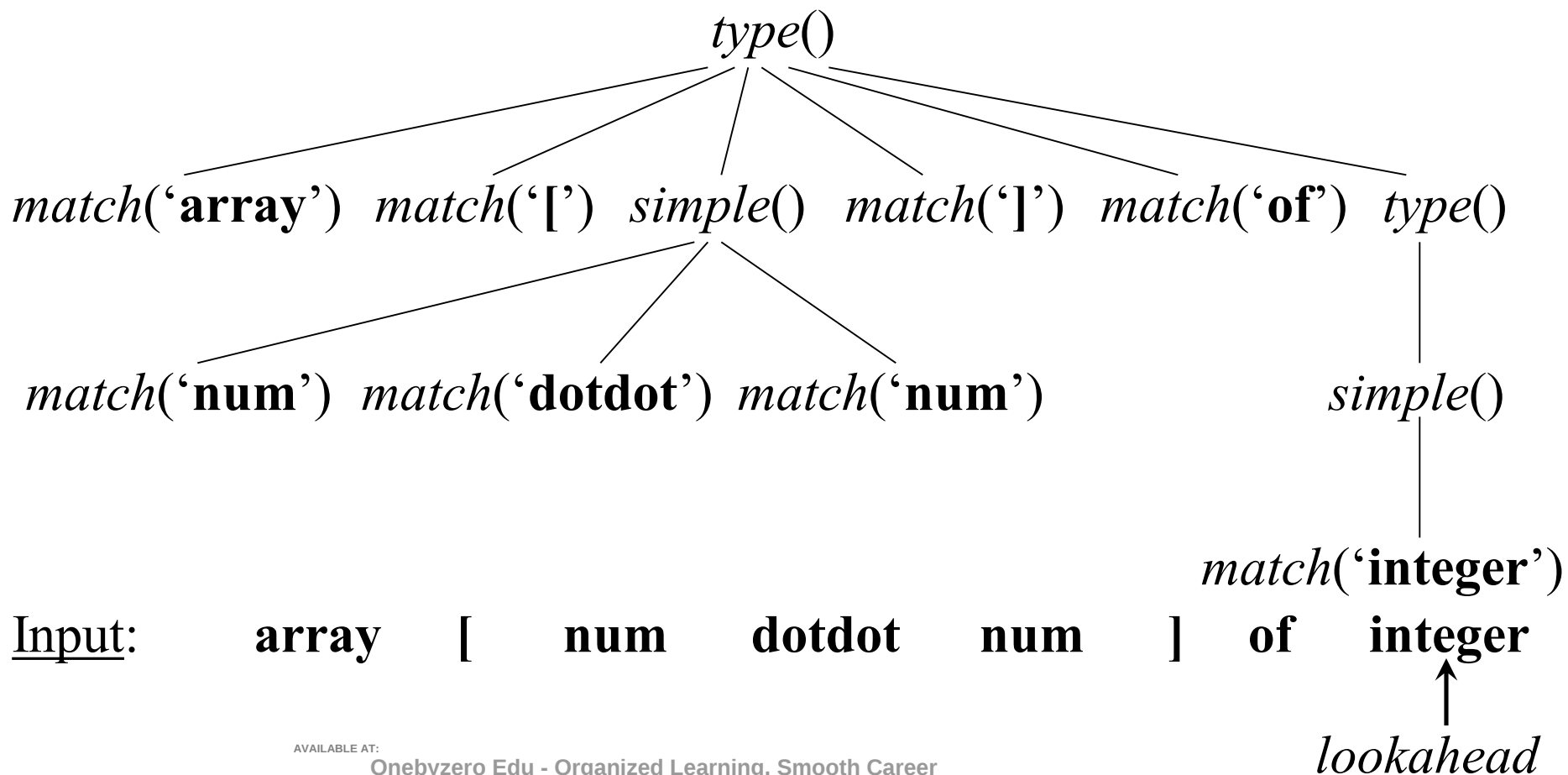
Example Predictive Parser (Execution Step 7)



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

↑
lookahead

Example Predictive Parser (Execution Step 8)



FIRST

$\text{FIRST}(\alpha)$ is the set of terminals that appear as the first symbols of one or more strings generated from α

$$\begin{aligned}
 &type \rightarrow simple \\
 &\quad | \text{ } ^ \mathbf{id} \\
 &\quad | \mathbf{array} [simple] \mathbf{of} type \\
 &simple \rightarrow \mathbf{integer} \\
 &\quad | \mathbf{char} \\
 &\quad | \mathbf{num} \mathbf{dotdot} \mathbf{num}
 \end{aligned}$$

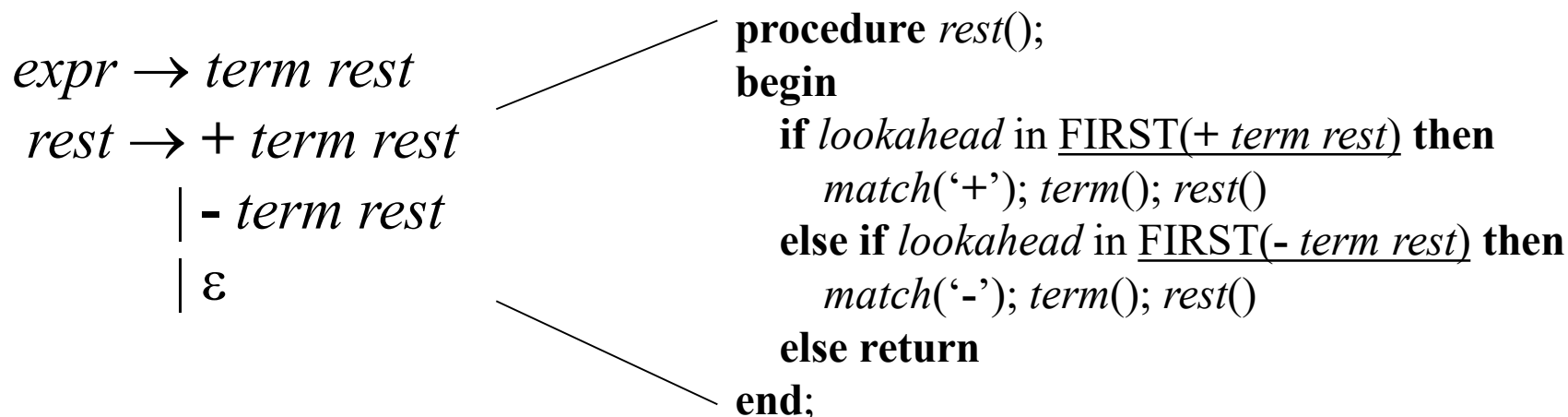
$\text{FIRST}(simple) = \{ \mathbf{integer}, \mathbf{char}, \mathbf{num} \}$

$\text{FIRST}(^ \mathbf{id}) = \{ ^ \mathbf{id} \}$

$\text{FIRST}(type) = \{ \mathbf{integer}, \mathbf{char}, \mathbf{num}, ^, \mathbf{array} \}$

How to use FIRST

We use FIRST to write a predictive parser as follows



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

$$\begin{array}{c}
 A \rightarrow \alpha \\
 | \beta
 \end{array}$$

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

$$\begin{aligned} stmt &\rightarrow \text{if } expr \text{ then } stmt \\ &\quad | \text{ if } expr \text{ then } stmt \text{ else } stmt \end{aligned}$$

We can use *left factoring* to fix the problem

$$\begin{aligned} stmt &\rightarrow \text{if } expr \text{ then } stmt \text{ opt_else} \\ opt_else &\rightarrow \text{else } stmt \\ &\quad | \epsilon \end{aligned}$$

Left Recursion

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

$$\begin{array}{c}
 A \rightarrow A \alpha \\
 | \beta \\
 | \gamma
 \end{array}$$

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

$$\begin{array}{c}
 A \rightarrow \beta R \\
 | \gamma R \\
 R \rightarrow \alpha R
 \end{array}$$

A Translator for Simple Expressions

$expr \rightarrow expr + term \quad \{ \text{print}("+") \}$

$expr \rightarrow expr - term \quad \{ \text{print}("-") \}$

$expr \rightarrow term$

$term \rightarrow 0 \quad \{ \text{print}("0") \}$

$term \rightarrow 1 \quad \{ \text{print}("1") \}$

...

$term \rightarrow 9 \quad \{ \text{print}("9") \}$

After left recursion elimination:

$expr \rightarrow term \text{ rest}$

$\text{rest} \rightarrow + term \{ \text{print}("+") \} \text{rest} \mid - term \{ \text{print}("-") \} \text{rest} \mid \varepsilon$

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

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

...

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

$expr \rightarrow term\ rest$

$rest \rightarrow +\ term\ \{ \text{print}("+") \}\ rest$
 $\quad | -\ term\ \{ \text{print}("-") \}\ rest$
 $\quad | \epsilon$

$term \rightarrow 0\ \{ \text{print}("0") \}$
 $term \rightarrow 1\ \{ \text{print}("1") \}$
 $\dots$
 $term \rightarrow 9\ \{ \text{print}("9") \}$

```

main()
{   lookahead = getchar();
    expr();
}

expr()
{   term();
    while (1) /* optimized by inlining rest()
               and removing recursive calls */
    {   if (lookahead == '+')
        {   match('+'); term(); putchar('+');
        }
        else if (lookahead == '-')
        {   match('-'); term(); putchar('-');
        }
        else break;
    }
}

term()
{   if (isdigit(lookahead))
    {   putchar(lookahead); match(lookahead);
    }
    else error();
}

match(int t)
{   if (lookahead == t)
        lookahead = getchar();
    else error();
}

error()
{   printf("Syntax error\n");
    exit(1);
}

```

Adding a Lexical Analyzer

- Typical tasks of the lexical analyzer:
 - Remove white space and comments
 - Encode constants as tokens
 - Recognize keywords
 - Recognize identifiers and store identifier names in a global symbol table

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()

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

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();
}
```

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

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)
 - hash tables

Identifiers

factor $\rightarrow$ (*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();
}
```

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

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;
}

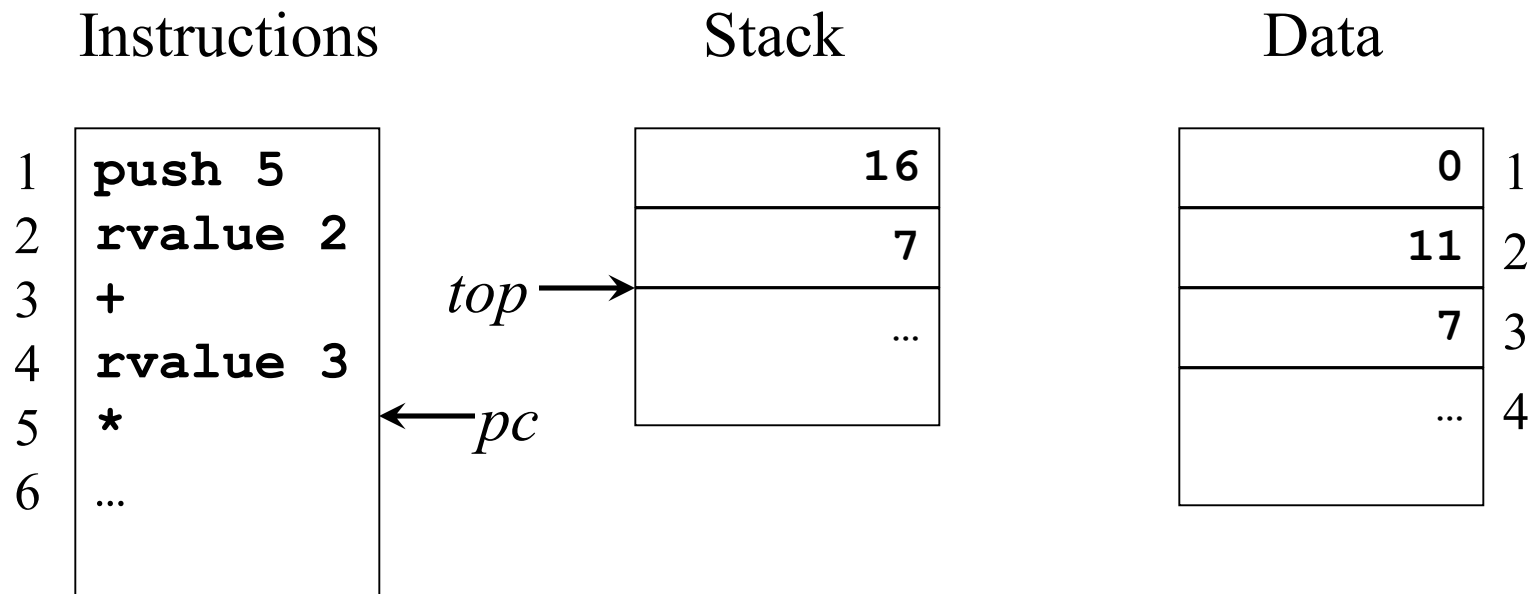
```

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



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

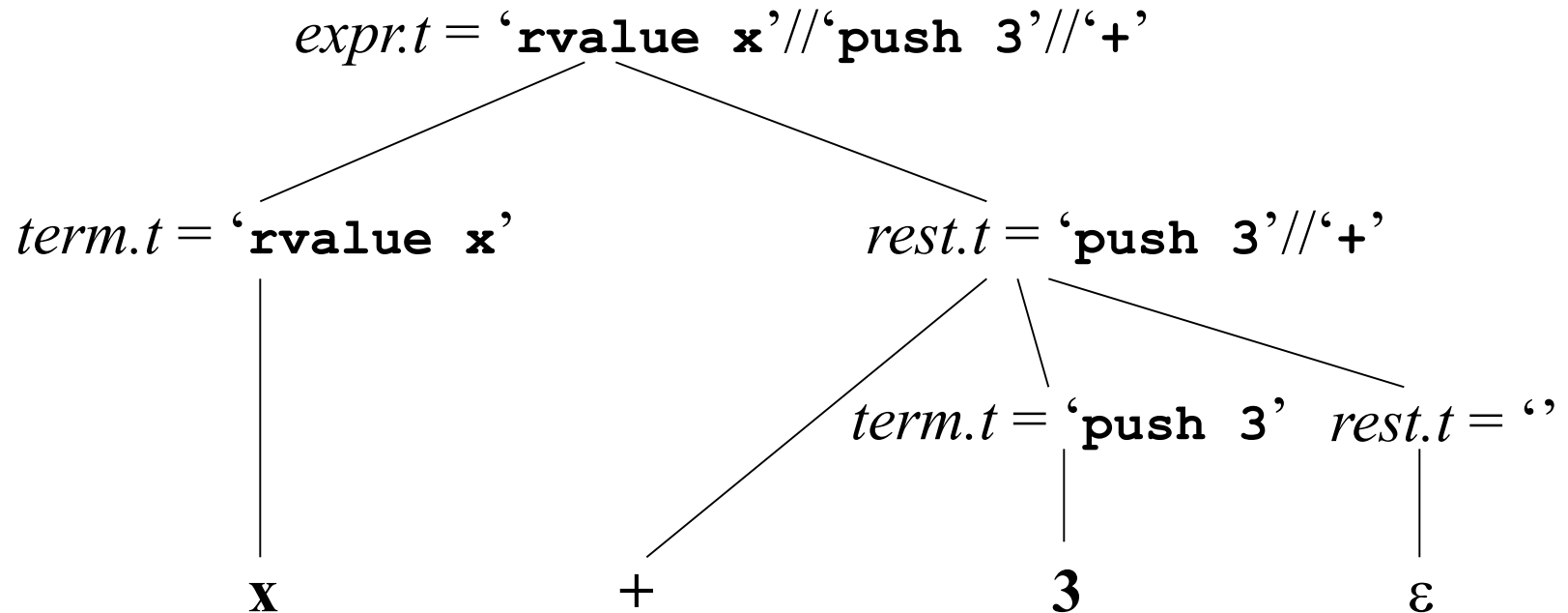
Generic Control Flow Instructions

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

Syntax-Directed Translation of Expressions

$$\begin{aligned}
 \text{expr} &\rightarrow \text{term rest} \{ \text{expr.t} := \text{term.t} // \text{rest.t} \} \\
 \text{rest} &\rightarrow + \text{term rest}_1 \{ \text{rest.t} := \text{term.t} // '+' // \text{rest}_1.t \} \\
 \text{rest} &\rightarrow - \text{term rest}_1 \{ \text{rest.t} := \text{term.t} // '-' // \text{rest}_1.t \} \\
 \text{rest} &\rightarrow \varepsilon \{ \text{rest.t} := '' \} \\
 \text{term} &\rightarrow \mathbf{num} \{ \text{term.t} := \mathbf{'push'} // \mathbf{num.value} \} \\
 \text{term} &\rightarrow \mathbf{id} \{ \text{term.t} := \mathbf{'rvalue'} // \mathbf{id.lexeme} \}
 \end{aligned}$$

Syntax-Directed Translation of Expressions (cont'd)



Translation Scheme to Generate Abstract Machine Code

expr $\rightarrow$ *term moreterms*

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

moreterms $\rightarrow$ $-$ *term* { print('-') } *moreterms*

moreterms $\rightarrow \epsilon$

term $\rightarrow$ *factor morefactors*

morefactors $\rightarrow$ $*$ *factor* { print('*') } *morefactors*

morefactors $\rightarrow$ **div** *factor* { print('DIV') } *morefactors*

morefactors $\rightarrow$ **mod** *factor* { print('MOD') } *morefactors*

morefactors $\rightarrow \epsilon$

factor $\rightarrow$ (*expr*)

factor $\rightarrow$ **num** { print('push ' // **num.value**) }

factor $\rightarrow$ **id** { print('rvalue ' // **id.lexeme**) }

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

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

$stmt \rightarrow id := \{ \text{print}('lvalue' // id.lexeme) \} expr \{ \text{print}(':=') \}$

lvalue <i>id.lexeme</i>
code for <i>expr</i>
:=

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

$stmt \rightarrow \text{if } expr \{ out := \text{newlabel}(); \text{print}(\text{'gofalse' } // out) \}$
 then $stmt \{ \text{print}(\text{'label' } // out) \}$

code for $expr$
gofalse out
code for $stmt$
label out

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

$stmt \rightarrow$ **while** { $test := \text{newlabel}()$; print('label ' // $test$) }
 $expr$ { $out := \text{newlabel}()$; print('gofalse ' // out) }
 do $stmt$ { print('goto ' // $test$ // 'label ' // out) }

label $test$
code for $expr$
gofalse out
code for $stmt$
goto $test$
label out

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

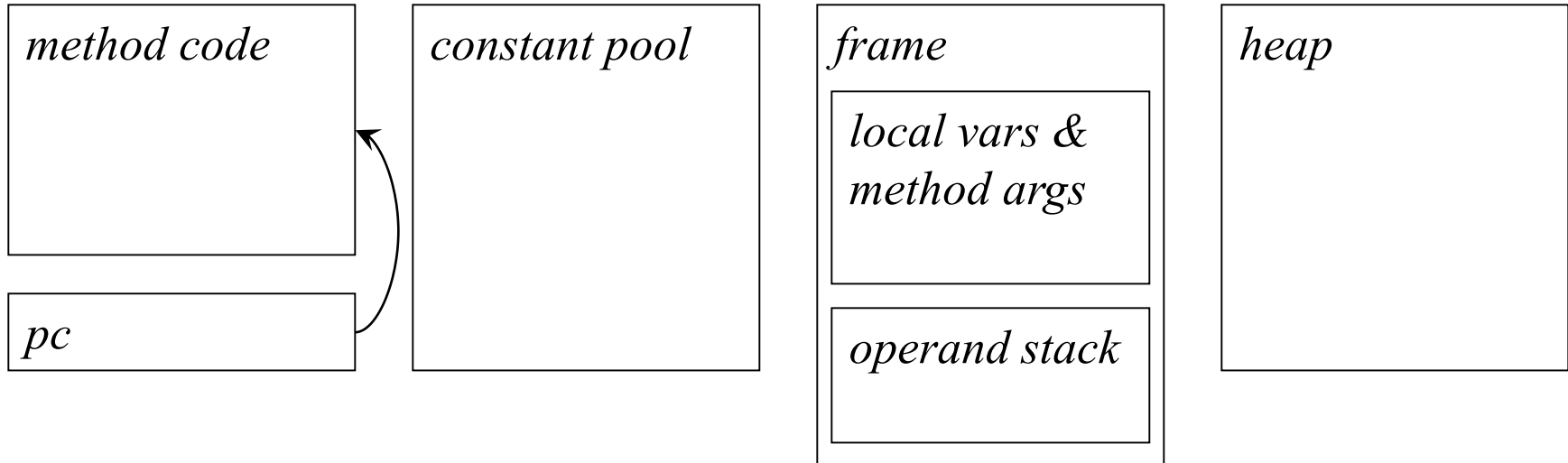
Translation Scheme to Generate Abstract Machine Code (cont'd)

$$\begin{aligned} start &\rightarrow stmt \{ \text{print}(\text{'halt'}) \} \\ stmt &\rightarrow \mathbf{begin} \textit{opt_stmts} \mathbf{end} \\ \textit{opt_stmts} &\rightarrow stmt ; \textit{opt_stmts} \mid \varepsilon \end{aligned}$$

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™ Virtual Machine Specification, 2nd ed.
<http://java.sun.com/docs/books/vmspec>

Runtime Data Areas (§3.5)



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 <i>pc</i> after method invocation
reference	a 32-bit address reference to an object of <i>class type</i> , <i>array type</i> , or <i>interface type</i> (value can be NULL)

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

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

Instruction Set (§3.11, §6)

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

<i>Actual Type</i>	<i>Computational Type</i>	<i>Category</i>
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

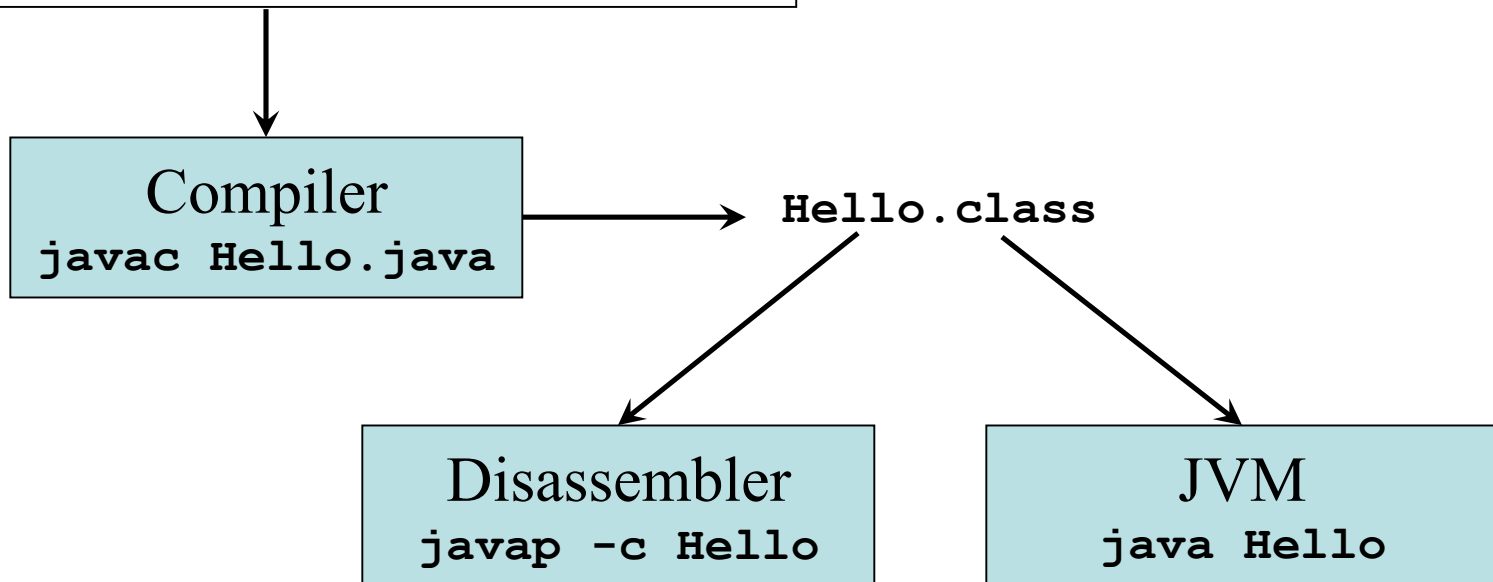
The Class File Format (§4)

- 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 **0xCAFEBAFE**
 - 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!");  
  }  
}
```



AVAILABLE AT:

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

AVAILABLE AT:

Onebyzero Edu - Organized Learning, Smooth Career

The Comprehensive Academic Study Platform for University Students in Bangladesh (www.onebyzeroedu.com)

Field/Method Descriptors (§4.3)

FieldType:

<i>BaseType</i> 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

Generating Code for the JVM

expr $\rightarrow$ *term moreterms*

moreterms $\rightarrow$ **+** *term* { emit(**iadd**) } *moreterms*

moreterms $\rightarrow$ **-** *term* { emit(**isub**) } *moreterms*

moreterms $\rightarrow$ ϵ

term $\rightarrow$ *factor morefactors*

morefactors $\rightarrow$ ***** *factor* { emit(**imul**) } *morefactors*

morefactors $\rightarrow$ **div** *factor* { emit(**idiv**) } *morefactors*

morefactors $\rightarrow$ **mod** *factor* { emit(**irem**) } *morefactors*

morefactors $\rightarrow$ ϵ

factor $\rightarrow$ (*expr*)

factor $\rightarrow$ **int8** { emit2(**bipush**, **int8.value**) }

factor $\rightarrow$ **int16** { emit3(**sipush**, **int16.value**) }

factor $\rightarrow$ **int32** { *idx* := newpoolint(**int32.value**);
emit2(**ldc**, *idx*) }

factor $\rightarrow$ **id** { emit2(**iload**, **id.index**) }

Generating Code for the JVM (cont'd)

$stmt \rightarrow id := expr \{ \text{emit2}(\text{istore}, id.index) \}$

code for <i>expr</i>
istore <i>id.index</i>

$stmt \rightarrow \text{if } expr \{ \text{emit}(\text{iconst_0}); loc := pc;$
 $\text{emit3}(\text{if_icmpeq}, 0) \}$
 then *stmt* $\{ \text{backpatch}(loc, pc - loc) \}$

code for <i>expr</i>
iconst_0
if_icmpeq <i>off₁</i> <i>off₂</i>
code for <i>stmt</i>

loc:

pc:

$\text{backpatch}()$ sets the offsets of the relative branch
when the target *pc* value is known