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 (just a little bit…)
The Structure of our Compiler

Source Program (Character stream) → Lexical analyzer → Token stream → Syntax-directed translator → Java bytecode

Syntax definition (BNF grammar) → Develop parser and code generator for translator → JVM specification
The Structure of our Compiler

Source Program (Character stream) → Lexical analyzer → Syntax-directed translator → Java bytecode

Syntax definition (BNF grammar) → Develop parser and code generator for translator

JVM specification
Syntax Definition

• Context-free grammar is a 4-tuple with
  – A set of tokens (\textit{terminal} symbols)
  – A set of \textit{nonterminals}
  – A set of \textit{productions}
  – A designated \textit{start symbol}
Example Grammar

Context-free grammar for simple expressions:

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

with productions \( P = \)

\[ \text{list} \rightarrow \text{list} + \text{digit} \]
\[ \text{list} \rightarrow \text{list} - \text{digit} \]
\[ \text{list} \rightarrow \text{digit} \]
\[ \text{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]
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

\[
\begin{align*}
\text{list} & \Rightarrow \text{list} + \text{digit} \\
& \Rightarrow \text{list} - \text{digit} + \text{digit} \\
& \Rightarrow \text{digit} - \text{digit} + \text{digit} \\
& \Rightarrow 9 - \text{digit} + \text{digit} \\
& \Rightarrow 9 - 5 + \text{digit} \\
& \Rightarrow 9 - 5 + 2
\end{align*}
\]

This is an example \textit{leftmost derivation}, because we replaced the leftmost nonterminal (underlined) in each step.

Likewise, a \textit{rightmost derivation} replaces the rightmost nonterminal in each step.
Parse Trees

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

\[ string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid \ldots \mid 9 \]

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

\[
left \rightarrow left + \text{term} \mid \text{term}
\]

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

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

\[
right \rightarrow \text{term} = right \mid \text{term}
\]

String \(a=b=c\) has the same meaning as \(a=(b=c)\)
Precedence of Operators

Operators with higher precedence “bind more tightly”

\[
\begin{align*}
expr & \rightarrow expr \ + \ term \ | \ term \\
term & \rightarrow term \ * \ factor \ | \ factor \\
factor & \rightarrow number \ | \ ( \ expr )
\end{align*}
\]

String \(2+3*5\) has the same meaning as \(2+(3*5)\)
Syntax of Statements

\[
stmt \rightarrow \text{id} := \text{expr} \\
\quad | \text{if} \ \text{expr} \ \text{then} \ \text{stmt} \\
\quad | \text{if} \ \text{expr} \ \text{then} \ \text{stmt} \ \text{else} \ \text{stmt} \\
\quad | \text{while} \ \text{expr} \ \text{do} \ \text{stmt} \\
\quad | \text{begin} \ \text{opt_stmts} \ \text{end} \\
\]

\[
\text{opt_stmts} \rightarrow \text{stmt} ; \ \text{opt_stmts} \\
\quad | \varepsilon
\]
The Structure of our Compiler

Source Program (Character stream) → Lexical analyzer → Token stream → Syntax-directed translator → Java bytecode

Syntax definition (BNF grammar) → Develop parser and code generator for translator → JVM specification
Syntax-Directed Translation

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

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

Production

\[ \text{expr} \rightarrow \text{expr}_1 + \text{term} \]
\[ \text{expr} \rightarrow \text{expr}_1 - \text{term} \]
\[ \text{expr} \rightarrow \text{term} \]
\[ \text{term} \rightarrow 0 \]
\[ \text{term} \rightarrow 1 \]
\[ \vdots \]
\[ \text{term} \rightarrow 9 \]

Semantic Rule

\[ \text{expr}.t := \text{expr}_1.t \mathbin{//} \text{term}.t \mathbin{//} "+" \]
\[ \text{expr}.t := \text{expr}_1.t \mathbin{//} \text{term}.t \mathbin{//} "-" \]
\[ \text{expr}.t := \text{term}.t \]
\[ \text{term}.t := "0" \]
\[ \text{term}.t := "1" \]
\[ \vdots \]
\[ \text{term}.t := "9" \]
Example Annotated Parse Tree

```
expr.t = "95-2+

expr.t = "95-"
  expr.t = "9"
    term.t = "9"
      9

  term.t = "5"
    5

term.t = "2"
  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*

\[
\text{rest} \rightarrow + \text{ term} \{ \text{ print(“+”) } \} \text{ rest}
\]
Example Translation Scheme

\[
\begin{align*}
expr & \rightarrow expr + term \quad \{ \text{print("+") } \} \\
expr & \rightarrow expr - term \quad \{ \text{print("-") } \} \\
expr & \rightarrow term \\
term & \rightarrow 0 \quad \{ \text{print("0") } \} \\
term & \rightarrow 1 \quad \{ \text{print("1") } \} \\
\cdots & \quad \cdots \\
term & \rightarrow 9 \quad \{ \text{print("9") } \}
\end{align*}
\]
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 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
  – Each nonterminal has one (recursive) procedure that is responsible for parsing the nonterminal’s syntactic category of input tokens
  – When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information

• **Predictive parsing** is a special form of recursive descent parsing where we use one lookahead token to unambiguously determine the parse operations
Example Predictive Parser
(Grammar)

\[
\begin{align*}
type & \rightarrow simple \\
& \mid ^\text{id} \\
& \mid \text{array } [ \text{simple} ] \text{ of type} \\
simple & \rightarrow \text{integer} \\
& \mid \text{char} \\
& \mid \text{num dotdot num}
\end{align*}
\]
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)

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

Check `lookahead` and call `match`

`match('array')`

`type()`
Example Predictive Parser
(Execution Step 2)

Input: `array [ num dotdot num ] of integer`
Example Predictive Parser
(Execution Step 3)

Input: array [ num dotdot num ] of integer

lookahead
Example Predictive Parser
(Execution Step 4)

Input: array [ num dotdot num ] of integer

lookahead
Example Predictive Parser
(Execution Step 5)

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

Diagram:
```
type()

match('array') match('[') simple()

match('num') match('dotdot') match('num')
```
Example Predictive Parser
(Execution Step 6)

Input: array [ num dotdot num ] of integer

looking ahead
Example Predictive Parser
(Execution Step 7)

Input: `array [ num dotdot num ] of integer`
Example Predictive Parser
(Execution Step 8)

Input: array [ num dotdot num ] of integer
FIRST

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

\[
type \rightarrow simple \\
\mid ^\wedge \text{id} \\
\mid \text{array [ simple ] of type}
\]

\[
simple \rightarrow \text{integer} \\
\mid \text{char} \\
\mid \text{num dotdot num}
\]

FIRST(\(simple\)) = \{ \text{integer, char, num} \}
FIRST(\(^\wedge \text{id}\)) = \{ ^\wedge \}
FIRST(\(type\)) = \{ \text{integer, char, num, ^, array} \}
How to use FIRST

We use FIRST to write a predictive parser as follows

\[
\begin{align*}
\text{expr} & \rightarrow \text{term rest} \\
\text{rest} & \rightarrow + \text{term rest} \\
& \mid - \text{term rest} \\
& \mid \varepsilon
\end{align*}
\]

procedure \(\text{rest}()\); begin
if \(\text{lookahead}\) in \(\text{FIRST}(+ \text{term rest})\) then
\quad \text{match}(‘+’); \text{term}(); \text{rest}()
else if \(\text{lookahead}\) in \(\text{FIRST}(- \text{term rest})\) then
\quad \text{match}(‘-’); \text{term}(); \text{rest}()
else return
end;

When a nonterminal \(A\) has two (or more) productions as in
\[
A \rightarrow \alpha \\
\mid \beta
\]
Then FIRST (\(\alpha\)) and FIRST(\(\beta\)) must be disjoint for predictive parsing to work
Left Factoring

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

$$stmt \rightarrow \textbf{if} expr \ \textbf{then} \ stmt \ \textbf{endif}$$

$$| \ \textbf{if} expr \ \textbf{then} \ stmt \ \textbf{else} \ stmt \ \textbf{endif}$$

We can use left factoring to fix the problem

$$stmt \rightarrow \textbf{if} \ expr \ \textbf{then} \ stmt \ \textbf{opt\_else}$$

$$opt\_else \rightarrow \textbf{else} \ stmt \ \textbf{endif}$$

$$| \ \textbf{endif}$$
Left Recursion

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

$$A \rightarrow A \alpha$$
$$\mid \beta$$
$$\mid \gamma$$

We can eliminate \textit{left recursive productions} by systematically rewriting the grammar using \textit{right recursive productions}

$$A \rightarrow \beta R$$
$$\mid \gamma R$$
$$R \rightarrow \alpha R$$
$$\mid \varepsilon$$
A Translator for Simple Expressions

\[
\begin{align*}
expr & \rightarrow expr + term \quad \{ \text{print(“+”)} \} \\
expr & \rightarrow expr - term \quad \{ \text{print(“-”)} \} \\
expr & \rightarrow term \\
term & \rightarrow 0 \quad \{ \text{print(“0”) } \} \\
term & \rightarrow 1 \quad \{ \text{print(“1”) } \} \\
& \quad \cdots \\
term & \rightarrow 9 \quad \{ \text{print(“9”) } \}
\end{align*}
\]

After left recursion elimination:

\[
\begin{align*}
expr & \rightarrow term \ rest \\
rest & \rightarrow + \ term \ \{ \text{print(“+”) } \ } rest \ | \ - \ term \ \{ \text{print(“-”) } \ } rest \ | \ \epsilon \\
term & \rightarrow 0 \ \{ \text{print(“0”) } \} \\
term & \rightarrow 1 \ \{ \text{print(“1”) } \} \\
& \quad \cdots \\
term & \rightarrow 9 \ \{ \text{print(“9”) } \}
\end{align*}
\]
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);
}
The Structure of our Compiler

Source Program (Character stream) → Lexical analyzer → Token stream → Syntax-directed translator → Java bytecode

- Syntax definition (BNF grammar)
- JVM specification

Develop parser and code generator for translator
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 “lexer”

\[ \text{y} := 31 + 28 \times x \]

Lexical analyzer

\text{lexer()}

Parser

\text{parse()}

\text{id, “y”} \leftarrow \text{assign, } \text{num, 31} \leftarrow “+” \leftarrow \text{num, 28} \leftarrow “*” \leftarrow \text{id, “x”}

\text{token}

(lookahead)

\text{tokenval}

(token attribute)
Token Attributes

\[
factor \rightarrow ( \ expr \ ) \\
| \ \textbf{num} \ \{ \ \text{print}(\text{num}.value) \} \\
\]

\#define \textbf{NUM} \ 256 /* token returned by lexan */

\begin{verbatim}
factor()
{
    if (lookahead == '(')
    {
        match('('); expr(); match(')');
    }
    else if (lookahead == NUM)
    {
        printf(" %d ", tokenval); match(NUM);
    }
    else error();
}
\end{verbatim}
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:

```c
struct entry
{
   char *lexptr; /* lexeme (string) for tokenval */
   int token;
};
```

```c
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 as in the project
- hashtables
Identifiers

\[
\text{factor} \rightarrow (\text{expr}) \\
| \text{id} \{ \text{print(id.string)} \}
\]

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

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

provided by the lexer for ID
Handling Reserved Keywords

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

```c
/* 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) /* not found */
       tokenval = insert(lexbuf, ID);
    return symtable[p].token;
}
```
Handling Reserved Keywords (cont’d)

morefactors \rightarrow \texttt{div factor} \{ \text{print( ‘DIV’ )} \} \text{morefactors}
| \texttt{mod factor} \{ \text{print( ‘MOD’ )} \} \text{morefactors}
| \ldots

/* parser.c */
morefactors()
{
    \text{if (lookahead == DIV)}
    \{ \text{match(DIV); factor(); printf(“DIV”); morefactors();} \}
    \text{else if (lookahead == MOD)}
    \{ \text{match(MOD); factor(); printf(“MOD”); morefactors();} \}
    \text{else}
    \{ \ldots \}
}
The Structure of our Compiler

Source Program (Character stream) → Lexical analyzer → Syntax-directed translator → Java bytecode

- Token stream
- Syntax definition (BNF grammar)
- JVM specification

Develop parser and code generator for translator
Abstract Stack Machines

Instructions
1. push 5
2. rvalue 2
3. +
4. rvalue 3
5. *
6. ...

Stack

Data

\[ \begin{array}{c}
16 \\
7 \\
... \\
\end{array} \]

\[ \begin{array}{c}
0 \\
11 \\
7 \\
... \\
\end{array} \]
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

\(*, /, \ldots\) ditto for other arithmetic operations

\(<, \&, \ldots\) 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
The Structure of our Compiler

Source Program (Character stream) → Lexical analyzer → Token stream → Syntax-directed translator → Java bytecode

- Syntax definition (BNF grammar)
- JVM specification

Develop parser and code generator for translator
Translation of Expressions to Abstract Machine Code

To produce code by string concatenation, we augment the left-factored and left-recursion-eliminated grammar for expressions as follows:

\[
\begin{align*}
expr & \rightarrow term \ rest \ \{ \ expr.t := term.t \ // \ rest.t \} \\
rest & \rightarrow + \ term \ rest_1 \ \{ \ rest.t := term.t \ // \ ‘+’ \ // \ rest_1.t \} \\
rest & \rightarrow - \ term \ rest_1 \ \{ \ rest.t := term.t \ // \ ‘-’ \ // \ rest_1.t \} \\
rest & \rightarrow \varepsilon \ \{ \ rest.t := ‘’ \} \\
term & \rightarrow \text{num} \ \{ \ term.t := ‘push’ \ // \ \text{num}.value \} \\
term & \rightarrow \text{id} \ \{ \ term.t := ‘rvalue’ \ // \ \text{id}.lexeme \}
\end{align*}
\]
Syntax-Directed Translation of Expressions (cont’d)

\[ expr.t = \text{‘rvalue x’} // \text{‘push 3’} // \text{‘+’} \]

\[ term.t = \text{‘rvalue x’} \quad \text{rest.t} = \text{‘push 3’} // \text{‘+’} \]

\[ x \quad + \quad 3 \quad \varepsilon \]
Translation Scheme to Generate Abstract Machine Code

As an alternative to producing code by string concatenation, we can emit code “on the fly” as follows

\[
expr \rightarrow \text{term moreterms} \\
moreterms \rightarrow + \text{term} \{ \text{print(‘+’)} \} \text{moreterms} \\
moreterms \rightarrow - \text{term} \{ \text{print(‘-’)} \} \text{moreterms} \\
moreterms \rightarrow \epsilon \\
\text{term} \rightarrow \text{factor morefactors} \\
morefactors \rightarrow \ast \text{factor} \{ \text{print(‘*’)} \} \text{morefactors} \\
morefactors \rightarrow \text{div factor} \{ \text{print(‘DIV’)} \} \text{morefactors} \\
morefactors \rightarrow \text{mod factor} \{ \text{print(‘MOD’)} \} \text{morefactors} \\
morefactors \rightarrow \epsilon \\
\text{factor} \rightarrow ( \text{expr} ) \\
\text{factor} \rightarrow \text{num} \{ \text{print(‘push’ // num.value)} \} \\
\text{factor} \rightarrow \text{id} \{ \text{print(‘rvalue’ // id.lexeme)} \}
Translation Scheme to Generate Abstract Machine Code (cont’d)

\[
stmt \rightarrow \text{id} := \{ \text{print(‘lvalue ’} \ // \ \text{id}.lexeme) \} \ \text{expr} \ \{ \text{print(‘:=’) } \}
\]

<table>
<thead>
<tr>
<th>lvalue id.lexeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>code for expr</td>
</tr>
<tr>
<td>:=</td>
</tr>
</tbody>
</table>
Translation Scheme to Generate Abstract Machine Code (cont’d)

\[ stmt \rightarrow \textbf{if} \ expr \{ \ out := \text{newlabel}(); \ \text{print(‘gofalse ’ }\ // \ out) \} \]
\[ \textbf{then} \ stmt \{ \ \text{print(‘label ’ }\ // \ out) \} \]

<table>
<thead>
<tr>
<th>code for expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>gofalse out</td>
</tr>
<tr>
<td>code for stmt</td>
</tr>
<tr>
<td>label out</td>
</tr>
</tbody>
</table>
Translation Scheme to Generate Abstract Machine Code (cont’d)

\[
\text{stmt} \rightarrow \text{while} \{ \text{test} := \text{newlabel()}; \text{print(‘label ’ // test)} \} \\
\text{expr} \{ \text{out} := \text{newlabel()}; \text{print(‘gofalse ’ // out)} \} \\
\text{do stmt} \{ \text{print(‘goto ’ // test // ‘label ’ // out)} \}
\]

<table>
<thead>
<tr>
<th>label test</th>
</tr>
</thead>
<tbody>
<tr>
<td>code for expr</td>
</tr>
<tr>
<td>gofalse out</td>
</tr>
<tr>
<td>code for stmt</td>
</tr>
<tr>
<td>goto test</td>
</tr>
<tr>
<td>label out</td>
</tr>
</tbody>
</table>
Translation Scheme to Generate Abstract Machine Code (cont’d)

\[
\begin{align*}
  \text{start} & \rightarrow \text{stmt} \{ \text{print(’halt’)} \} \\
  \text{stmt} & \rightarrow \text{begin opt_stmts end} \\
  \text{opt_stmts} & \rightarrow \text{stmt ; opt_stmts} | \epsilon
\end{align*}
\]
The Structure of our Compiler

Source Program (Character stream) → Lexical analyzer → Token stream → Syntax-directed translator → Java bytecode

Develop parser and code generator for translator

Syntax definition (BNF grammar) → JVM specification
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
  
  [http://docs.oracle.com/javase/specs/](http://docs.oracle.com/javase/specs/)
Runtime Data Areas (§ 2.5)

- method code
- constant pool
- frame
  - local vars & method args
  - operand stack
- heap

pc
Constant Pool (§ 2.5.5)

- 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 (because these 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 (§ 2.6)

• 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 (§ 2.2, § 2.3, § 2.4)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>byte</code></td>
<td>a 8-bit signed two’s complement integer</td>
</tr>
<tr>
<td><code>short</code></td>
<td>a 16-bit signed two’s complement integer</td>
</tr>
<tr>
<td><code>int</code></td>
<td>a 32-bit signed two’s complement integer</td>
</tr>
<tr>
<td><code>long</code></td>
<td>a 64-bit signed two’s complement integer</td>
</tr>
<tr>
<td><code>char</code></td>
<td>a 16-bit Unicode character</td>
</tr>
<tr>
<td><code>float</code></td>
<td>a 32-bit IEEE 754 single-precision float value</td>
</tr>
<tr>
<td><code>double</code></td>
<td>a 64-bit IEEE 754 double-precision float value</td>
</tr>
<tr>
<td><code>boolean</code></td>
<td>a virtual type only, <strong>int</strong> is used to represent true (1) false (0)</td>
</tr>
<tr>
<td><code>returnAddress</code></td>
<td>the location of the <em>pc</em> after method invocation</td>
</tr>
<tr>
<td><code>reference</code></td>
<td>a 32-bit address reference to an object of <em>class type</em>, <em>array type</em>, or <em>interface type</em> (value can be NULL)</td>
</tr>
</tbody>
</table>

Operand stack has 32-bit slots, thus `long` and `double` occupy two slots
# Instruction Set (§ 2.11, § 6)

<table>
<thead>
<tr>
<th>opcode</th>
<th>byte</th>
<th>short</th>
<th>int</th>
<th>long</th>
<th>float</th>
<th>double</th>
<th>char</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tipush</td>
<td>bipush</td>
<td>sipush</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tconst</td>
<td></td>
<td></td>
<td>ict const</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tload</td>
<td></td>
<td></td>
<td>iload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tstore</td>
<td></td>
<td></td>
<td>ilastore</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tinc</td>
<td></td>
<td></td>
<td>iinc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taload</td>
<td></td>
<td></td>
<td>ilaload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tastore</td>
<td></td>
<td></td>
<td>ilastore</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tadd</td>
<td></td>
<td></td>
<td>iadd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tsub</td>
<td></td>
<td></td>
<td>isub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tmul</td>
<td></td>
<td></td>
<td>imul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tdiv</td>
<td></td>
<td></td>
<td>idiv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trem</td>
<td></td>
<td></td>
<td>irem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tneg</td>
<td></td>
<td></td>
<td>ineg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tshl</td>
<td></td>
<td></td>
<td>ishl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tshr</td>
<td></td>
<td></td>
<td>ishr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tand</td>
<td></td>
<td></td>
<td>iand</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tor</td>
<td></td>
<td></td>
<td>ior</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Txor</td>
<td></td>
<td></td>
<td>ixor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2T</td>
<td></td>
<td></td>
<td>i2l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2T</td>
<td></td>
<td></td>
<td>i2f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2T</td>
<td></td>
<td></td>
<td>i2d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2T</td>
<td></td>
<td></td>
<td>l2i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2T</td>
<td></td>
<td></td>
<td>l2f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2T</td>
<td></td>
<td></td>
<td>l2d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2T</td>
<td></td>
<td></td>
<td>f2i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2T</td>
<td></td>
<td></td>
<td>f2f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2T</td>
<td></td>
<td></td>
<td>f2d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d2T</td>
<td></td>
<td></td>
<td>d2i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d2T</td>
<td></td>
<td></td>
<td>d2f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmp</td>
<td></td>
<td></td>
<td>lcmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmp</td>
<td></td>
<td></td>
<td>fcml</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmp</td>
<td></td>
<td></td>
<td>dcmpl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmp</td>
<td></td>
<td></td>
<td>fcmpg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmp</td>
<td></td>
<td></td>
<td>dcmpg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if_TcmpOP</td>
<td></td>
<td></td>
<td>if_icmpOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ireturn</td>
<td></td>
<td></td>
<td>ireturn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ireturn</td>
<td></td>
<td></td>
<td>lreturn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ireturn</td>
<td></td>
<td></td>
<td>freturn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ireturn</td>
<td></td>
<td></td>
<td>dreturn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ireturn</td>
<td></td>
<td></td>
<td>areturn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actual Type</th>
<th>Computational Type</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>int</td>
<td>category 1</td>
</tr>
<tr>
<td>byte</td>
<td>int</td>
<td>category 1</td>
</tr>
<tr>
<td>char</td>
<td>int</td>
<td>category 1</td>
</tr>
<tr>
<td>short</td>
<td>int</td>
<td>category 1</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>category 1</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>category 1</td>
</tr>
<tr>
<td>reference</td>
<td>reference</td>
<td>category 1</td>
</tr>
<tr>
<td>returnAddress</td>
<td>returnAddress</td>
<td>category 1</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>category 2</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>category 2</td>
</tr>
</tbody>
</table>
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” (self) and super class refs (indexed in the pool)
  - Class fields
  - Class methods
javac, javap, java

Hello.java

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

Compiler

```
javac Hello.java
```

Disassembler

```
javap -c Hello
```

JVM

```
java Hello
```
javap -c Hello

Local variable 0 = “this”

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

}
Field/Method Descriptors (§ 4.3)

Field Type:

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

MethodDescriptor: ( ParameterDescriptor* ) ReturnDescriptor

ParameterDescriptor: Field Type

ReturnDescriptor: Field Type

v
The Structure of our Compiler

Source Program (Character stream) → Lexical analyzer → Token stream → Syntax-directed translator → Java bytecode

Syntax definition (BNF grammar) → Develop parser and code generator for translator

JVM specification
Generating Code for the JVM

\[
\begin{align*}
expr & \to \text{term moreterms} \\
moreterms & \to + \text{term} \{ \text{emit(iadd)} \} \ moreterms \\
moreterms & \to - \text{term} \{ \text{emit(isub)} \} \ moreterms \\
moreterms & \to \varepsilon \\
term & \to \text{factor morefactors} \\
morefactors & \to \ast \text{factor} \{ \text{emit(imul)} \} \ morefactors \\
morefactors & \to \text{divfactor} \{ \text{emit(idiv)} \} \ morefactors \\
morefactors & \to \text{modfactor} \{ \text{emit(irem)} \} \ morefactors \\
morefactors & \to \varepsilon \\
\text{factor} & \to ( \ expr \ ) \\
\text{factor} & \to \text{int16} \{ \text{emit3(sipush, int16.value)} \} \\
\text{factor} & \to \text{id} \{ \text{emit2(iload, id.index)} \} 
\end{align*}
\]
Generating Code for the JVM (cont’d)

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

\[ stmt \rightarrow \text{if expr} \{ \text{emit(iconst\_0); loc := pc; emit3(if.icmpeq, 0)} \}
\text{then stmt} \{ \text{backpatch(loc, pc-loc)} \} \]

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