



CSCI 742 - Compiler Construction

Lecture 26

Introduction to Code Generation

Instructor: Hossein Hojjat

March 30, 2018

Compiler Phases

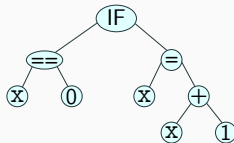
Source Code
(concrete syntax)

```
if (x==0) x=x+1;
```

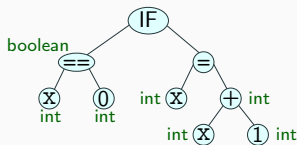
Token Stream

```
if ( x == 0 ) x = x + 1 ;
```

Abstract Syntax Tree
(AST)

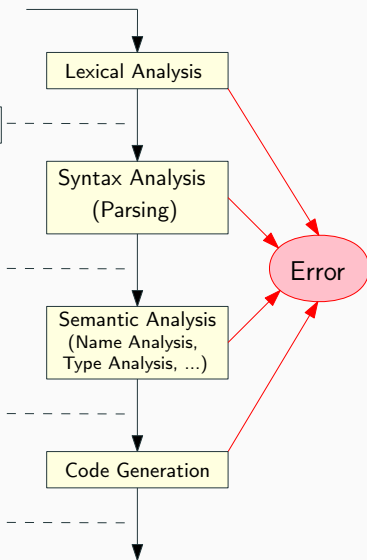


Attributed AST



Machine Code

```
16: iload_2  
17: ifne_24  
20: iload_2  
21: iconst_1  
22: iadd  
23: istore_2  
24: ...
```



Code Generation Example

- Phase after type checking emits such bytecode instructions

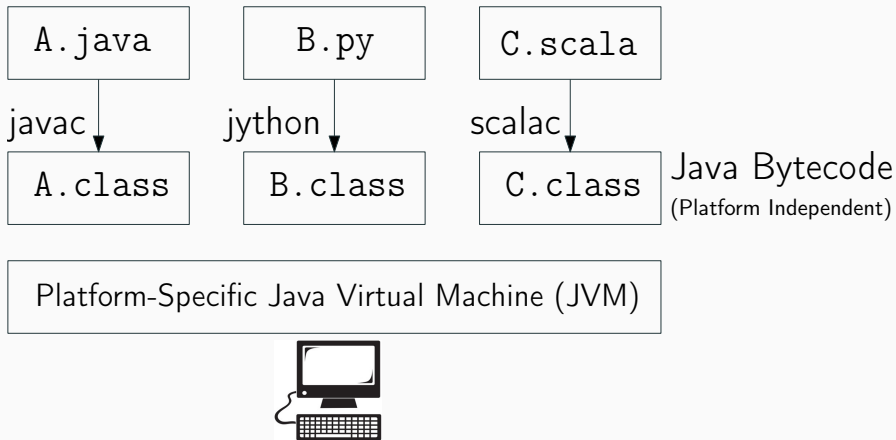
```
while (i > 0) {
  i += 2 * j + 1;
  j = j - 5;
  System.out.println(j);
}
```

```
javac Test.java
javap -c Test
```

```
5:  iload_1
6:  ifle           31
9:  iload_1
10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: iload_2
18: iconst_5
19: isub
20: istore_2
21: getstatic      #2 // System.out
24: iload_2
25: invokevirtual  #3 // println
28: goto           5
31: // ...
```

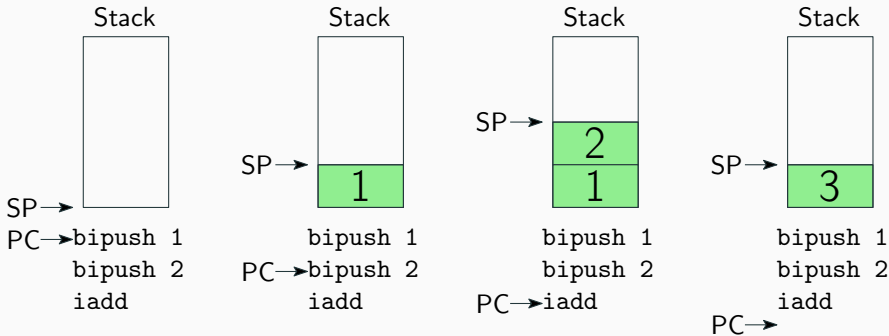
Java Virtual Machine (JVM)

- Programs are written in Java or other languages
- Compiler translates them to Java Bytecode
- Platform-specific Java Virtual Machine executes Java Bytecode



Java Virtual Machine (JVM)

- JVM is a stack machine: evaluation of expressions uses a stack (operand stack)
- Instructions fetch their arguments from the top of the operand stack
- Instructions store their results at the top of the operand stack



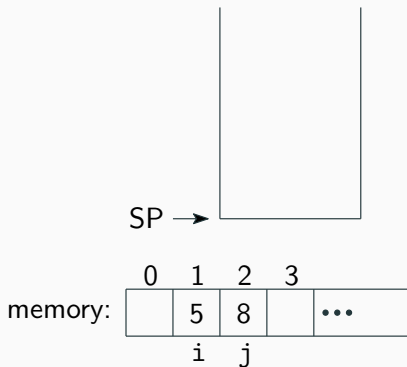
Why a Stack Machine

- A simple evaluation model: no variables or registers
- Each operation:
 - takes operands from top of stack
 - puts results back at top of stack
- Instruction “add” as opposed to “add r1, r2”
- Simpler compiler, more compact programs

Local Variables

- In memory space of a function there is an array V to store local variables and arguments
- `iload`: push the value of a local variable onto the stack
- `istore`: pop the value from the stack and store it in a local variable
- Initially the arguments x_1, \dots, x_n are stored in local variables array $V[1], \dots, V[n]$
- $V[0]$ holds the reference to `this`
 - object on which the method is invoked

Stack Machine Execution Example



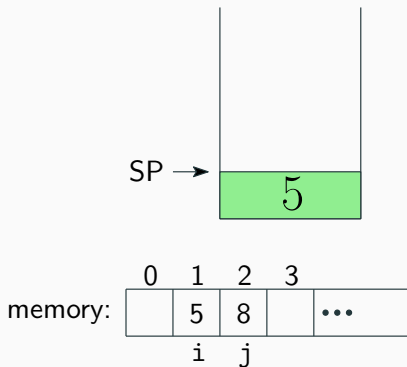
Java Bytecode

PC →

```
6: // ...
9: iload_1
10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...
```

Java Statement: `i += 2 * j + 1;`

Stack Machine Execution Example

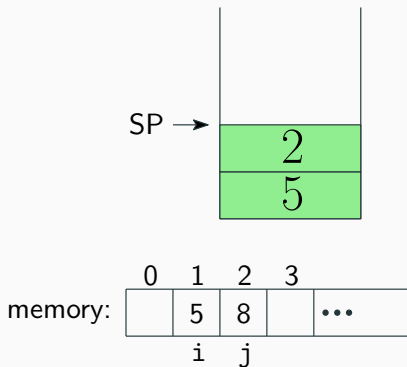


Java Bytecode

```
6: // ...
9: iload_1
PC → 10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...
```

Java Statement: `i += 2 * j + 1;`

Stack Machine Execution Example



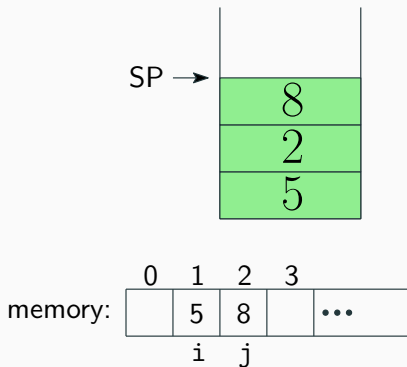
Java Bytecode

PC →

```
6: // ...
9: iload_1
10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...
```

Java Statement: `i += 2 * j + 1;`

Stack Machine Execution Example

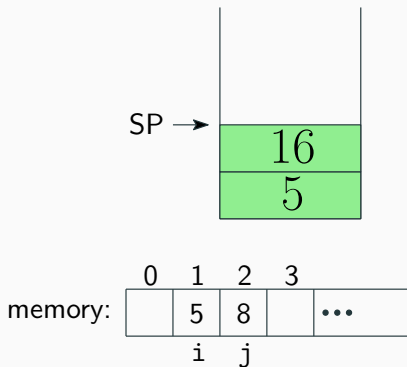


Java Bytecode

6: // ...
9: iload_1
10: iconst_2
11: iload_2
PC → 12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...

Java Statement: $i += 2 * j + 1;$

Stack Machine Execution Example

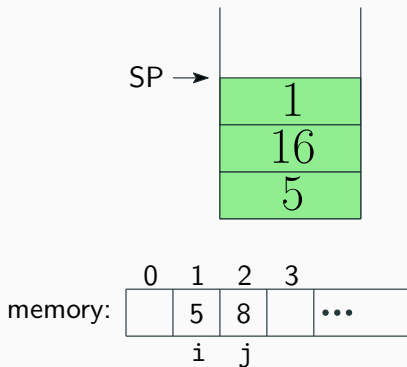


Java Bytecode

6: // ...
9: iload_1
10: iconst_2
11: iload_2
12: imul
PC → 13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...

Java Statement: $i += 2 * j + 1;$

Stack Machine Execution Example



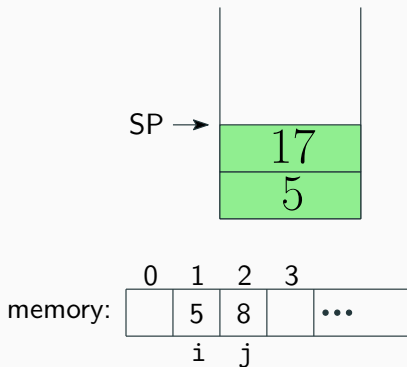
Java Bytecode

```
6: // ...
9: iload_1
10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...
```

PC →

Java Statement: `i += 2 * j + 1;`

Stack Machine Execution Example



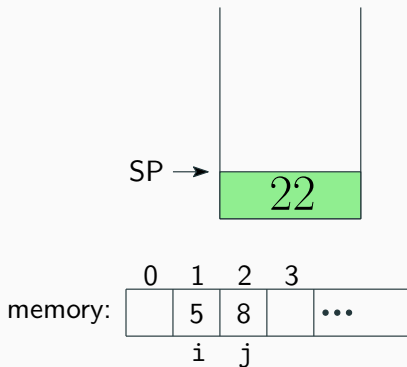
Java Bytecode

```
6: // ...
9: iload_1
10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...
```

PC →

Java Statement: $i += 2 * j + 1;$

Stack Machine Execution Example



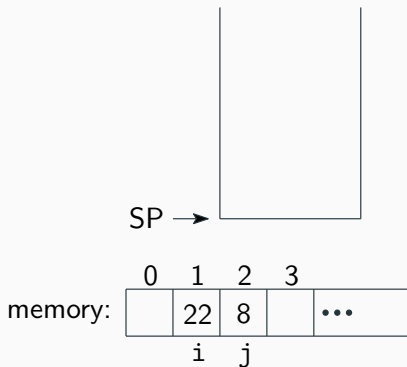
Java Bytecode

```
6: // ...
9: iload_1
10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...
```

PC →

Java Statement: `i += 2 * j + 1;`

Stack Machine Execution Example



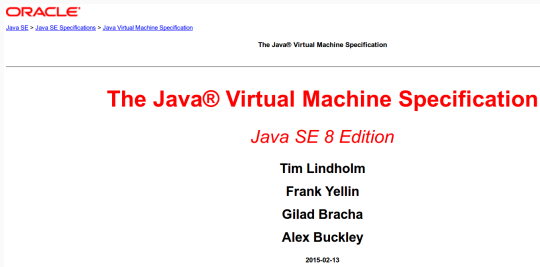
Java Bytecode

```
6: // ...
9: iload_1
10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...
```

PC →

Java Statement: `i += 2 * j + 1;`

- Separate for each type, including
 - integer types (`iadd`, `imul`, `iload`, `istore`, `bipush`)
 - reference types (`aload`, `astore`)
- Why are they separate if not in e.g. x86?
 - Memory safety
 - Each reference points to a valid allocated object
- Conditionals and jumps
- Further high-level operations
 - array operations
 - object method and field access



<http://docs.oracle.com/javase/specs/jvms/se8/html/index.html>

- Use `javac -g *.java` to compile
- Use `javap -c -l ClassName` to explore

Selected Instructions

<code>iload_x</code>	Loads the integer value of the local variable in slot x on the stack. $x \in \{0, 1, 2, 3\}$
<code>iload X</code>	Loads the value of the local variable pointed to by index X on the top of the stack.
<code>iconst_x</code>	Loads the integer constant x on the stack. $x \in \{0, 1, 2, 3, 4, 5\}$
<code>bipush X</code>	Like <code>iconst</code> , but for arbitrarily large X
<code>istore_x</code>	Stores the current value on top of the stack in the local variable in slot $x \in \{0, 1, 2, 3\}$
<code>istore X</code>	Stores the current value on top of the stack in the local variable indexed by X .
<code>ireturn</code>	Method return statement (note that the return value has to have been put on the top of the stack beforehand).
<code>iadd</code>	Pop two (integer) values from the stack, add them and put the result back on the stack.
<code>isub</code>	Pop two (integer) values from the stack, subtract them and put the result back on the stack.

Selected Instructions

<code>imult</code>	Pop two (integer) values from the stack, multiply them and put the result back on the stack.
<code>idiv</code>	Pop two (integer) values from the stack, divide them and put the result back on the stack.
<code>irem</code>	Pop two (integer) values from the stack, put the result of $x_1 \% x_2$ back on the stack.
<code>ineg</code>	Negate the value on the stack.
<code>iinc x, y</code>	Increment the variable in slot <code>x</code> by amount <code>y</code> .
<code>ior</code>	Bitwise OR for the two integer values on the stack.
<code>iand</code>	Bitwise AND for the two integer values on the stack.
<code>ixor</code>	Bitwise XOR for the two integer values on the stack.
<code>ifXX L</code>	Pop one value from the stack, compare it zero according to the operator <code>XX</code> . If the condition is satisfied, jump to the instruction given by label <code>L</code> . $XX \in \{\text{eq, lt, le, ne, gt, ge, null, nonnull}\}$

Selected Instructions

<code>if_icmpXX L</code>	Pop two values from the stack and compare against each other. Rest as <code>ifXX L</code> .
<code>goto L</code>	Unconditional jump to instruction given by the label <code>L</code> .
<code>pop</code>	Discard word currently on top of the stack.
<code>dup</code>	Duplicate word currently on top of the stack.
<code>swap</code>	Swaps the two top values on the stack.
<code>aload_x</code>	Loads an object reference from slot <code>x</code> .
<code>aload X</code>	Loads an object reference from local variable indexed by <code>X</code> .
<code>iaload</code>	Loads onto the stack an integer from an array. The stack must contain the array reference and the index.
<code>iastore</code>	Stores an integer in an array. The stack must contain the array reference, the index and the value, in that order.

Example: Twice

```
class Expr1 {  
    public static int twice(int x) {  
        return x*2;  
    }  
}
```

```
> javac -g Expr1.java; javap -c -l Expr1
```

```
public static int twice(int);
```

Code:

```
0: iload_0 // load int from var 0 to top of stack  
1: iconst_2 // push 2 on top of stack  
2: imul // replace two topmost elements with their product  
3: ireturn // return top of stack
```

Example: Area

```
class Expr2 {  
    public static int cubeArea(int a,int b,int c) {  
        return (a*b + b*c + a*c) * 2;  
    }  
}
```

```
> javac -g Expr2.java; javap -c -l Expr2
```

LocalVariableTable:

Start	Length	Slot	Name	Signature
0	14	0	a	I
0	14	1	b	I
0	14	2	c	I

```
public static int  
    cubeArea(int,int,int);
```

Code:

```
0: iload_0  
1: iload_1  
2: imul  
3: iload_1  
4: iload_2  
5: imul  
6: iadd  
7: iload_0  
8: iload_2  
9: imul  
10: iadd  
11: iconst_2  
12: imul  
13: ireturn
```