SWEN430 - Compiler Engineering

Lecture 11 - Java Bytecode

Erin Greenwood-Thessman

with thanks to Linsday Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

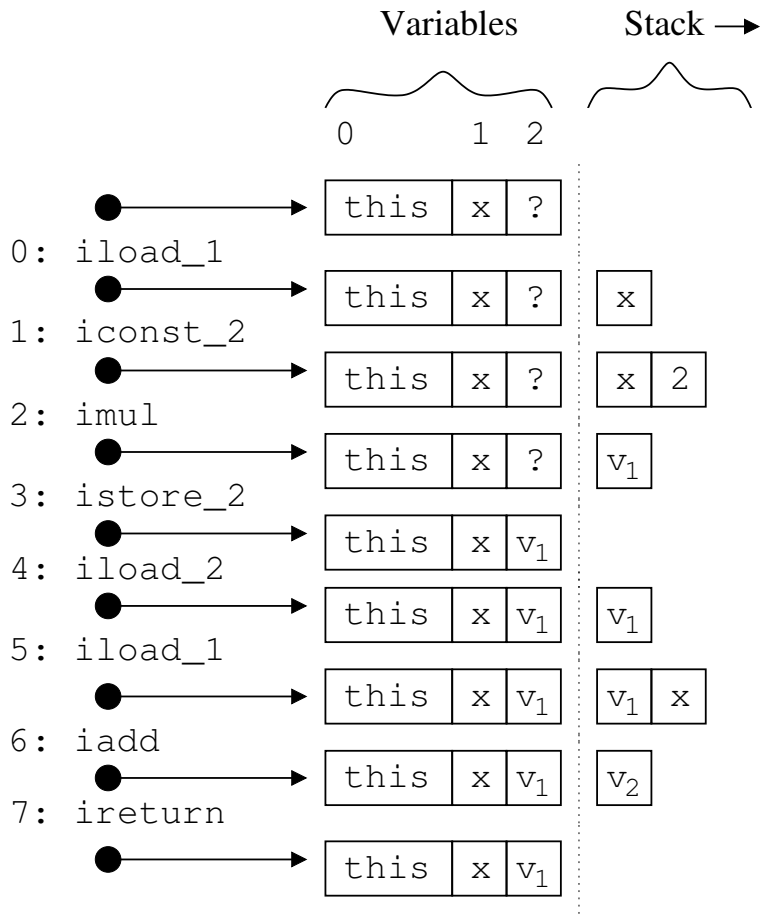
Java Bytecode

```
class Test {
 public int f(int x) {
  int y = x * 2;
  return y + x;
}}
```

```
public int f(int);
  Code:
   Stack=2, Locals=3
   0:   iload_1
   1:   iconst_2
   2:   imul
   3:   istore_2
   4:   iload_2
   5:   iload_1
   6:   iadd
   7:   ireturn
```

- A stack-based language, similar to machine code
- Can decompile Java programs with `javap`
- For details of bytecode instructions, see *JVM Specification*

A Detailed Example



- Here, `x` represents value of parameter `x` on entry
- v_1 and v_2 represent intermediate values

Some Bytecode Instructions

`aload X`	Push reference onto stack from variable X
`astore X`	Pop reference off stack into variable X
`iconst X`	Push **`int`** constant `i` onto stack
`ladd`	Take top two (long) items off stack, perform **`long`** addition, and push (long) result back on stack
`imul`	Take top two items off stack, perform **`int`** multiplication, and push result back on stack
`areturn`	Return top item on stack as reference
`goto` `X`	Goto location X
`ifeq X`	Take one item off stack; if equal top zero goto location X

More Bytecode Instructions

`pop`	Pop top item off stack and discard
`dup`	Duplicate top item on stack
`getfield F`	Pop reference off stack and load field `F` from it onto stack
`putfield F`	Pop top two items of stack; write first to field `F` in object referred by second
`checkcast C`	Pop item from stack, check instance of `C` and push back
`iinc X,c`	Increment local variable X by constant C

JVM Types

- JVM variables have simplistic types, compared with Java:
  `i` = integer, `l` = long, `f` = float, `d` = double, `a` = reference

`iload`	`lload`	`fload`	`dload`	`aload`
`istore`	`lstore`	`fstore`	`dstore`	`astore`
`iconst`	`lconst`	`fconst`	`dconst`	`aconst_null`
`ireturn`	`lreturn`	`freturn`	`dreturn`	`areturn`
`iadd`	`ladd`	`fadd`	`dadd`	
`imul`	`lmul`	`fmul`	`dmul`	
`idiv`	`ldiv`	`fdiv`	`ddiv`	
`ineg`	`lneg`	`fneg`	`dneg`	
`irem`	`lrem`	`frem`	`drem`	

- These all operate in essentially the same way, just for different types

Conditional Statements

```
class Test {
 int abs(int x) {
  if(x >= 0) {
   return x;
  } else {
   return -x;
}}}
```

```
int abs(int);
  Code:
   Stack=1, Locals=2
   0:   iload_1
   1:   iflt    6
   4:   iload_1
   5:   ireturn
   6:   iload_1
   7:   ineg
   8:   ireturn
```

- Control-flow implemented using *conditional branching*.

Looping Statements

```
public class Test {
 int count(int end) {
   int r = 0;
   for(int i=0;i!=end;++i) {
     r = r + i;
   }
   return r;
}}
```

```
int count(int);
  Code:
    Stack=2, Locals=4
    0:   iconst_0
    1:   istore_2
    2:   iconst_0
    3:   istore_3
    4:   iload_3
    5:   iload_1
    6:   if_icmpeq       19
    9:   iload_2
    10:  iload_3
    11:  iadd
    12:  istore_2
    13:  iinc    3, 1
    16:  goto    4
    19:  iload_2
    20:  ireturn
```

- Loops implemented using unconditional **backward branches**

JVM Type Conversions

```
long f(int x) {
  return x; // implict
}
```

```
long f(int);
  Code:
    Stack=2, Locals=2
    0:   iload_1
    1:   i2l    // explicit
    2:   lreturn
```

```
int f(int x, float y) {
  if(x == y) { return x; }
  return 0;
}
```

```
int f(int, float);
  Code:
    Stack=2, Locals=3
    0:   iload_1
    1:   i2f   // explicit
    2:   fload_2
    3:   fcmpl
    4:   ifne    9
    7:   ...
```

- Java permits **implicit conversions**
- JVM permits only **explicit conversions**

JVM Type Conversion Table

`i2l`	Convert **`int`** to **`long`**	
`i2f`	Convert **`int`** to **`float`**	(lossy)
`i2d`	Convert **`int`** to **`double`**	
`l2i`	Convert **`long`** to **`int`**	(lossy)
`l2f`	Convert **`long`** to **`float`**	(lossy)
`l2d`	Convert **`long`** to **`double`**	(lossy)
`f2i`	Convert **`float`** to **`int`**	(lossy)
`f2l`	Convert **`float`** to **`long`**	(lossy)
`f2d`	Convert **`float`** to **`double`**	

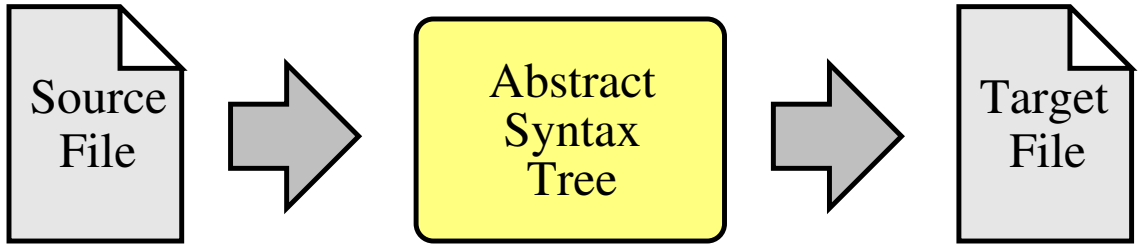
Long Types

```
public class Test {
  long f(long x, int y) {
    return x + y;
  }
}
```

```
long f(long, int);
  Code:
    Stack=4, Locals=4
    0:   lload_1
    1:   iload_3
    2:   i2l
    3:   ladd
    4:   lreturn
```

- "Long" types require **two variable slots**
  - i.e. **`long`** and **`double`** types
  - In example above, `x` occupies slots 1 + 2 and `y` occupies slot 3

Program Translation

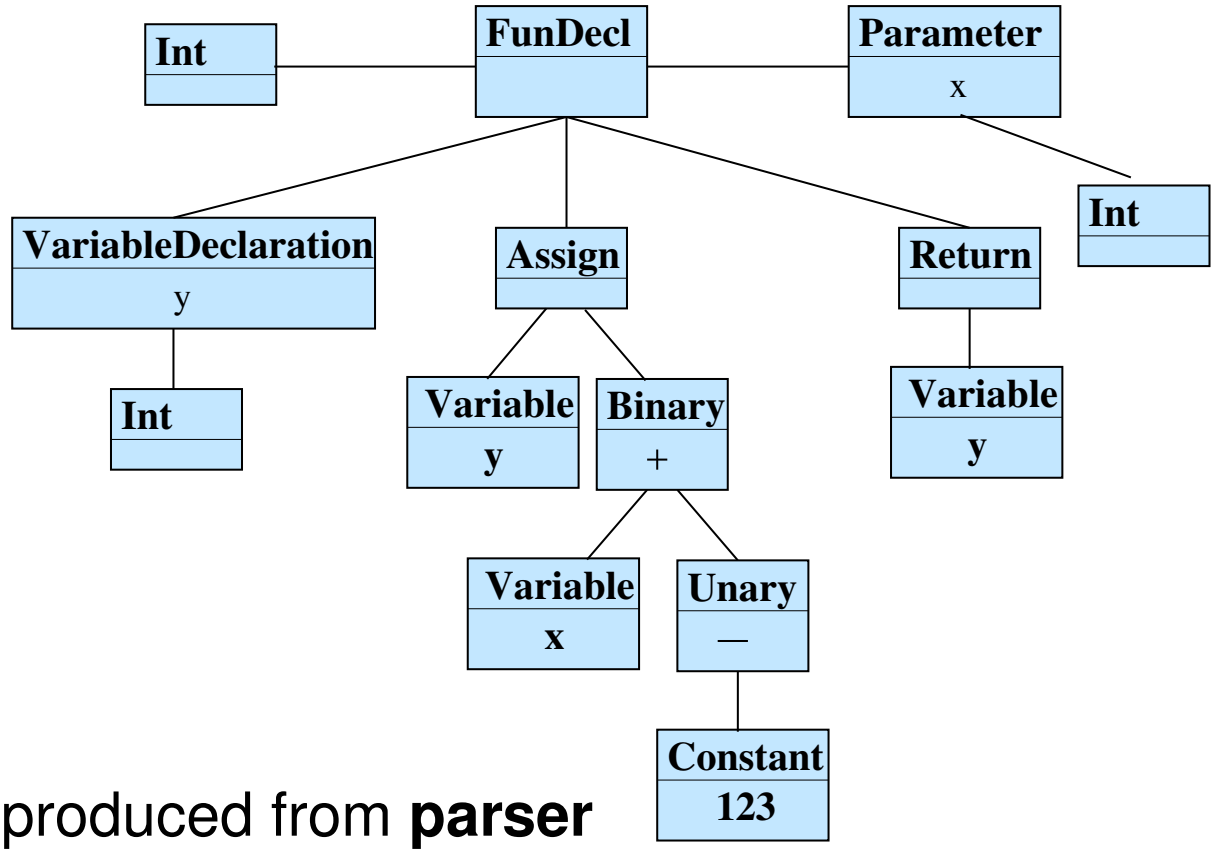


- For program in **source language**, construct “equivalent” program in **target language**.
- **Parse** the source program and construct an AST.
- **Traverse** the AST generating target code for each node.

$\Longrightarrow$ *Need to determine target code for each kind of AST node.*

Abstract Syntax Trees (AST)

```
int f(int x) {
    int y;
    y = x + -123;
    return y;
}
```



- Abstract syntax tree produced from **parser**
- Abstract syntax tree used for e.g. **type checking**
- Abstract syntax tree turned into **intermediate language** or **target code**

Compiler Backend

- Target language may be:
  - another programming language (C, JS),
  - virtual machine code (JVM, CLR, LLVM),
  - assembler language,
  - machine code.
- Each presents different **challenges**.

  E.g. translating to another programming language/assembler removes the need to determine addresses for variables and jumps.
- For now, we're considering JVM as the target (and X86 later).

References

1. “The Java Virtual Machine Specification, Java SE 7 Edition”, Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley.