CSCI 742 - Compiler Construction

Lecture 28

A Primer on Jasmin

Instructor: Hossein Hojjat

April 4, 2018

Recap: Code Generation for Expressions

$[\![e_1 + e_2]\!] =$

```
    ⟦e₁⟧
    ⟦e₂⟧
    iadd
```

$[\![e_1 * e_2]\!] =$

```
    ⟦e₁⟧
    ⟦e₂⟧
    imul
```

Jasmin

- Java class files use binary format
- We use an equivalent of assembly language for JVM bytecode
- Jasmin assembles human readable assembly code to java bytecode
  - `http://jasmin.sourceforge.net/`

Assembling

- To assemble Jasmin file test.j containing Jasmin code:

```
> java -jar jasmin.jar test.j
```

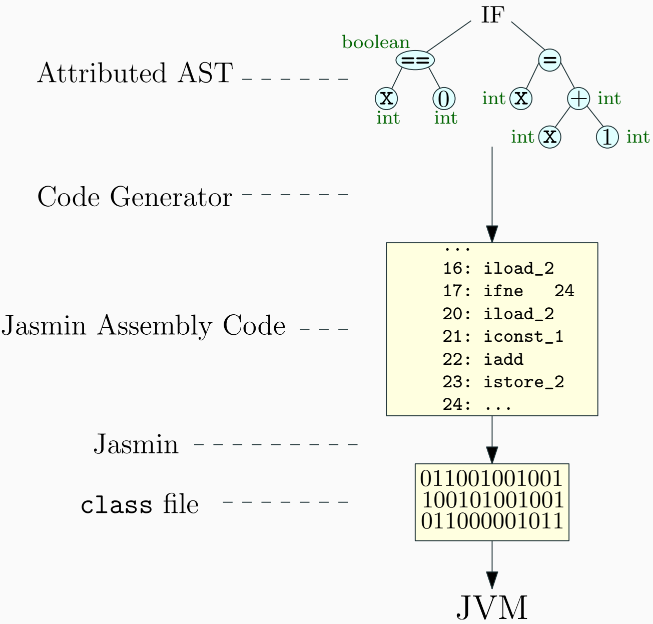
- This produces `test.class` which can be run by java interpreter

Disassembling:

```
> javap -c test.class
```

- Prints an assembler version in (almost) Jasmin syntax

Code Generation



Attributed AST

Code Generator

Jasmin Assembly Code

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

Jasmin

class file

```
011001001001
100101001001
011000001011
```

JVM

Jasmin Syntax

- One statement per line
- Inline comments started by “;”
- Assembly structure:

$$\begin{array}{c}\text{Options}\\ \text{Method}_1\\ \cdots\\ \text{Method}_n\end{array}$$

Options

- **.class**: the name of the `class` file to be created (required)
  - e.g.: `.class public Test`
- **.super**: Superclass of resulting java class (required)
  - always: `.super java/lang/Object`
  - (unless the class inherits from another class)
- **.field**: Specify fields of class
  - e.g.: `.field public my_field I`

JVM Types

Java Type	**Type Signature**
`boolean`	`Z`
`byte`	`B`
`char`	`C`
`double`	`D`
`float`	`F`
`int`	`I`
`long`	`J`
`short`	`S`
`void`	`V`
Reference type t	`L`t
Array of type a	`[`a
Function of type $a \to b$	$(a)b$

Method Structure

```
.class public Main
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit locals <number of local variables>
.limit stack <maximum stack depth>
<generated code>
return
.end method
```

Maximum number of local variables
Default value: 1

Maximum size of the operand stack

- Make an over-approximation of operand stack size
- Typical stack-based microprocessor can hold only a few elements
- Data elements can always be moved in and out of memory

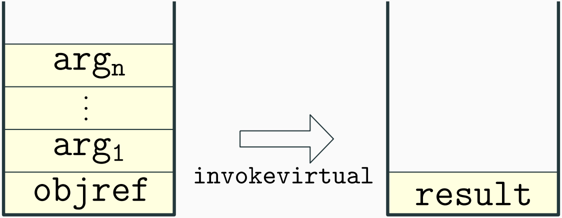
non-static method calls

- Call a method of an object on stack

  `invokevirtual <class ID>/<method signature>`
- Requires parameters and object on stack

Example

- method: `public int myMethod(int a)`
- signature: `myMethod(I)I`
- invocation: `invokevirtual MyClass/myMethod(I)I`



- `invokestatic` does not need the object reference on stack

Method Calls

Invoking methods

- `invokestatic`: for static methods
- `invokevirtual`: for ordinary instance methods
- `invokespecial`: for constructor (`<init>`), `private`, or superclass methods

Returning value from methods

- `ireturn`:
  - Pop an integer from stack and push it onto stack of invoker
- `areturn`:
  - Pop a reference from stack and push it onto stack of invoker
- `return`: return from a method returning `void`

Example: Static Method Invocation

```
.method static add(II)I
.limit stack 2
.limit locals 2
  iload_0
  iload_1
  iadd
  ireturn
.end method
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 1
  iconst_2
  iconst_5
  invokestatic Main/add(II)I
  pop
  return
.end method
```

```
public class Main {
  static int add(int x, int y) {
    return x + y;  }
  public static void main(String argv[]) {
    add(2, 5); }
}
```

Note: code does not contain initialization lines

Example: Instance Method Invocation

```java
public class Main {
  int add(int x, int y) {
    return x + y; }
  public static void main(String argv[]) {
    Main m = new Main();
    m.add(2, 5); }
}
```

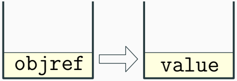
```
.method add(II)I
.limit stack 2
.limit locals 3
  iload_1
  iload_2
  iadd
  ireturn
.end method
.method public static main([Ljava/lang/String;)V
.limit stack 3
.limit locals 2
  new Main ; Make a Main object and leave a reference to it on stack
  dup      ; Duplicate the object reference
  invokespecial Main/<init>()V ; Invoke object initializer
  astore_1 ; Store the objectref in local variable 1
  aload_1
  iconst_2
  iconst_5
  invokevirtual Main/add(II)I
  pop
  return
.end method
```

Note: code does not contain initialization lines

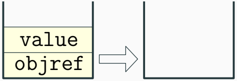
Field Access

- Retrieve the value of a field, Set the value of a field

```
getfield <class ID>/<field ID> <type>
```



```
putfield <class ID>/<field ID> <type>
```



Example

- field: `public int my_field`
- signature: `myClass/my_field I`
- assignment: `putfield myClass/my_field I`

System.out.println

1. Push `PrintStream` object onto stack

```
getstatic java/lang/System/out Ljava/io/PrintStream;
```

2. Push value onto stack (`iload`, `aload`, etc.)
3. Invoke matching `PrintStream` method

```
invokevirtual java/io/PrintStream/print(I)V
invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
```

Example: Field Assignment

```
class Point {
  public int xCoord, yCoord;
};
```

- Java statement:

```
p.xCoord = 0;
```

- Corresponding JVM bytecode:

```
aload_1                       ; push object in local variable 1
                              ; (which is p) onto stack
iconst_0                      ; push 0 onto stack
putfield Point/xCoord I       ; set value of integer field
                              ; p.xCoord to 0
```

Read More

- Read Jasmin User Guide to understand syntax and rules
- `http://jasmin.sourceforge.net/guide.html`
- Read Jasmin instruction reference manual to understand instructions
- `http://jasmin.sourceforge.net/instructions.html`