Introduction to Compiler Construction

JVM Code Generation: Preliminaries

Outline

1. Introduction

Introduction

Introduction

Once the AST has been fully analyzed, all variables and expressions have been typed, any necessary tree rewriting has been done, and a certain amount of setup needed for code generation has been accomplished

Introduction

Once the AST has been fully analyzed, all variables and expressions have been typed, any necessary tree rewriting has been done, and a certain amount of setup needed for code generation has been accomplished

The compiler is now ready to traverse the AST one more time to generate the Java Virtual Machine (JVM) code, ie, build the class file for the program

Introduction

Once the AST has been fully analyzed, all variables and expressions have been typed, any necessary tree rewriting has been done, and a certain amount of setup needed for code generation has been accomplished

The compiler is now ready to traverse the AST one more time to generate the Java Virtual Machine (JVM) code, ie, build the class file for the program

For example, consider the following very simple program

```
public class Square {
    public int square(int x) {
        return x * x;
    }
}
```

Introduction

Once the AST has been fully analyzed, all variables and expressions have been typed, any necessary tree rewriting has been done, and a certain amount of setup needed for code generation has been accomplished

The compiler is now ready to traverse the AST one more time to generate the Java Virtual Machine (JVM) code, ie, build the class file for the program

For example, consider the following very simple program

```
public class Square {
    public int square(int x) {
        return x * x;
    }
}
```

Compiling the program with our *j--* compiler

```
>_ ~/workspace/j--
$ ./bin/j-- Square.java
```

produces a class file `Square.class`

Introduction

Once the AST has been fully analyzed, all variables and expressions have been typed, any necessary tree rewriting has been done, and a certain amount of setup needed for code generation has been accomplished

The compiler is now ready to traverse the AST one more time to generate the Java Virtual Machine (JVM) code, ie, build the class file for the program

For example, consider the following very simple program

```
public class Square {
    public int square(int x) {
        return x * x;
    }
}
```

Compiling the program with our *j--* compiler

```
>_ ~/workspace/j--
$ ./bin/j-- Square.java
```

produces a class file `Square.class`

Running the `javap` program on the class file

```
>_ ~/workspace/j--
$ javap -verbose Square
```

produces the symbolic representation of the file shown in the next slide

Introduction

Introduction

```
public class Square extends java.lang.Object
  minor version: 0
  major version: 49
  Constant pool:
const #1 = Asciz    Square;
const #2 = class    #1; //  Square
const #3 = Asciz    java/lang/Object;
const #4 = class    #3; //  java/lang/Object
const #5 = Asciz    <init>;
const #6 = Asciz    ()V;
const #7 = NameAndType  #5:#6;//  "<init>":()V
const #8 = Method   #4.#7;  //  java/lang/Object."<init>":()V
const #9 = Asciz    Code;
const #10 = Asciz   square;
const #11 = Asciz   (I)I;

{
public Square();
  Code:
   Stack=1, Locals=1, Args_size=1
   0: aload_0
   1: invokespecial #8; //Method java/lang/Object."<init>":()V
   4: return

public int square(int);
  Code:
   Stack=2, Locals=2, Args_size=2
   0: iload_1
   1: iload_1
   2: imul
   3: ireturn

}
```

Introduction

Introduction

To emit JVM instructions, we firstly create a `CLEmitter` instance, which is an abstraction of the class file we wish to build, and then call upon `CLEmitter`'s methods for generating the necessary headers and instructions

Introduction

To emit JVM instructions, we firstly create a `CLEmitter` instance, which is an abstraction of the class file we wish to build, and then call upon `CLEmitter`'s methods for generating the necessary headers and instructions

For example, to generate the class header

```
public class Square extends java.lang.Object
```

we would invoke the `addClass()` method on `output`, an instance of `CLEmitter`

```
output.addClass(mods, "Square", "java/lang/Object", null, false);
```

Introduction

To emit JVM instructions, we firstly create a `CLEmitter` instance, which is an abstraction of the class file we wish to build, and then call upon `CLEmitter`'s methods for generating the necessary headers and instructions

For example, to generate the class header

```
public class Square extends java.lang.Object
```

we would invoke the `addClass()` method on `output`, an instance of `CLEmitter`

```
output.addClass(mods, "Square", "java/lang/Object", null, false);
```

As another example, the no-argument instruction `aload_1` may be generated by

```
output.addNoArgInstruction(ALOAD_1);
```

Introduction

Introduction

For a more involved example of code generation, consider the `Factorial` program from before

```
package pass;

import java.lang.System;

public class Factorial {
    // Two methods and a field

    public static int factorial(int n) {
        // position 1:
        if (n <= 0) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }

    public static void main(String[] args) {
        int x = n;

        // position 2:
        System.out.println(n + "! = " + factorial(x));
    }

    static int n = 5;
}
```

Introduction

Introduction

Running `javap` on `Factorial.class` produced by the *j--* compiler gives us

```
public class pass.Factorial extends java.lang.Object
  minor version: 0
  major version: 49
  Constant pool:
  ...
{
static int n;

public pass.Factorial();
  Code:
   Stack=1, Locals=1, Args_size=1
   0: aload_0
   1: invokespecial #8; //Method java/lang/Object."<init>":()V
   4: return

public static int factorial(int);
  Code:
   Stack=3, Locals=1, Args_size=1
   0: iload_0
   1: iconst_0
   2: if_icmpgt 10
   5: iconst_1
   6: ireturn
   7: goto  19
   10: iload_0
   11: iload_0
   12: iconst_1
   13: isub
   14: invokestatic #13; //Method factorial:(I)I
   17: imul
   18: ireturn
   19: nop
```

Introduction

Introduction

```
public static void main(java.lang.String[]);
  Code:
    Stack=3, Locals=2, Args_size=1
    0: getstatic #19; //Field n:I
    3: istore_1
    4: getstatic #25; //Field java/lang/System.out:Ljava/io/PrintStream;
    7: new   #27; //class java/lang/StringBuilder
    10: dup
    11: invokespecial    #28; //Method java/lang/StringBuilder."<init>":()V
    14: getstatic    #19; //Field n:I
    17: invokevirtual    #32; //Method java/lang/StringBuilder.append:
                              (I)Ljava/lang/StringBuilder;
    20: ldc  #34; //String ! =
    22: invokevirtual    #37; //Method java/lang/StringBuilder.append:
                              (Ljava/lang/String;)Ljava/lang/StringBuilder;
    25: iload_1
    26: invokestatic #13; //Method factorial:(I)I
    29: invokevirtual    #32; //Method java/lang/StringBuilder.append:
                              (I)Ljava/lang/StringBuilder;
    32: invokevirtual    #41; //Method java/lang/StringBuilder.toString:
                              ()Ljava/lang/String;
    35: invokevirtual    #47; //Method java/io/PrintStream.println:
                              (Ljava/lang/String;)V
    38: return

public static {};
  Code:
    Stack=2, Locals=0, Args_size=0
    0: iconst_5
    1: putstatic #19; //Field n:I
    4: return
}
```