CS142B Language Processor Construction

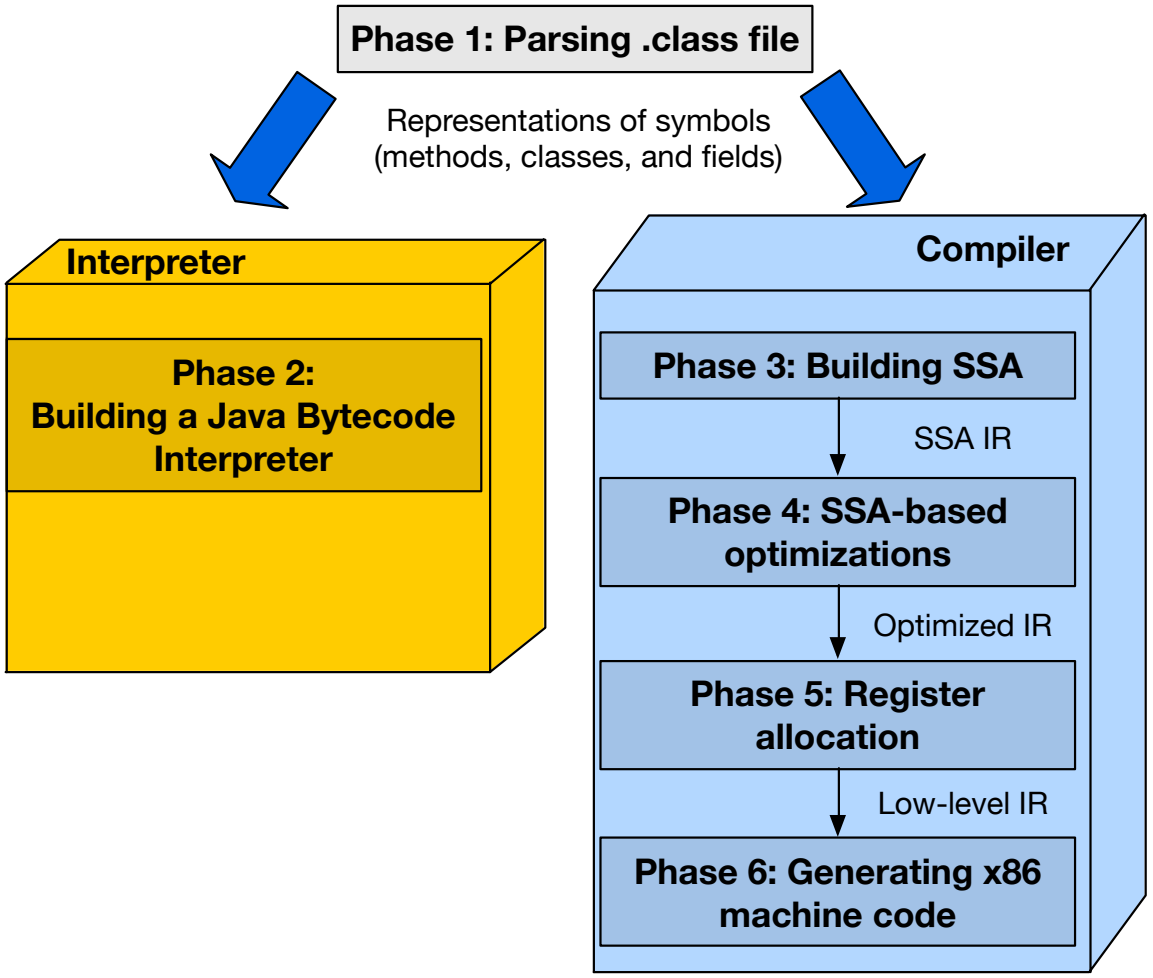
Java Bytecode Interpreter

Yeoul Na

UCI

April 9, 2019

Recap the project



Recap parsing the .class file

- Parse Constant table
- Methods
  - Method name
  - Get bytecode instructions from the “Code” attribute

Bytecode interpreter

- Dispatching the instruction
- Accessing the operands
- Performing the computation

Stack machine vs. Register machine

Stack-based bytecode	Register-based bytecode
iload_1	move r10, r1
iload_2	move r11, r2
iadd	iadd r10, r10, r11
istore_3	move r3, r10

- Shi, Yunhe, et al. "Virtual machine showdown: Stack versus registers." ACM Transactions on Architecture and Code Optimization (TACO) 4.4 (2008): 2.

Stack machine vs. Register machine

- Stack machine – JVM, CPython
  - Operands are on stack
  - Results are pushed to stack
  - No need to specify the operand's address in instruction
    - Simpler bytecode format
    - Simpler implementation
- Register machine – Lua, SpiderMonkey
  - Need to specify the address of the operands
    - Bigger per instruction size
    - Require fewer number of instructions

Java Virtual Machine (JVM)

- An abstract computing machine
  - has an instruction set and manipulates various memory areas at run time
- Stack-based machine
- Knows nothing of the java programming language, only of a particular binary format, the class file format

JVM Data types

- JVM distinguishes its operand types by using instructions intended to operate on values of specific types
  - E.g., iadd, ladd, fadd, and dadd. Each is specialized for its operand type: int, long, float and double.
- Int : 32-bit signed integers

JVM Run-Time Data Areas

- Java Virtual Machine stack
  - Stores frames
  - Analogous to the stack of languages like C
    - Holds local variables and partial results
    - Plays a part in method invocation and return
- Run-time constant pool
  - Loaded from .class file
  - Serves a function similar to that of a symbol table

Frames

- Allocated from the Java Virtual Machine stack
- Is used to store data and partial results, as well as return values / pass arguments for methods
- A new frame is created each time a method is invoked
- A frame is destroyed when its method invocation completes
- Has its own array of local variables, its own operand stack

Frames – Local Variables and Operand Stacks

- Max sizes are determined at compile-time
- Local Variables
  - Are used to pass parameters
  - Addressed by indexing
  - JVM uses local variables to pass parameters on method invocation
    - Starting from local variable 0 for static methods
- Operand Stacks (last-in-first-out)
  - Are used to store temporary results and return values
  - JVM instructions take their operands from the operand stack, operate on them, and push the result back onto the operand stack

Format of instruction description

- Format (zero or more operands)

  mnemonic
  operand1
  operand2
  ...

- Representation in the bytecode stream
  - Each line is 1 byte (8-bit) value
  - mnemonic = opcode
- Operand Stack

  ..., *value1*, *value2* →
  ..., *value3*

https://docs.oracle.com/javase/specs/jvms/se12/html/jvms-6.html

How to interpret Java bytecode

- E.g. *iload_0*
  - Operand Stack

    ... →
    ..., *value*

  - Description*:* The *value* of the local variable at <*n*> is pushed onto the operand stack.

How to interpret Java bytecode

- E.g. *if_icmpeq*
  - Format

    ```
    if_icmp<cond>
    Branchbyte1
    branchbyte2
    ```

  - Operand Stack

    ..., *value1*, *value2* →
    ...
  - Description
    - if_icmpeq succeeds if and only if value1 = value2
    - If succeeds, (branchbyte1 << 8) | branchbyte2 constructs a signed 16-bit offset
    - Execution proceeds at that offset from the address of this instruction

Bytecode Instructions

- We are interested in…
  - Load/Store : *iconst_<i>, iload_<i>, istore_<i>*
  - Arithmetic : *iadd, iinc, isub, imul, ishl, ishr*
  - Control Transfer : *if_icmpne, if_icmpeq, if_icmpgt,if_cmpge, if_icmplt, if_icmple, ifeq, ifne, ifgt, ifge, iflt, ifle*
  - *goto*
  - *bipush*
  - *invokestatic, invokevirtual (only for println)*
  - *return, ireturn*
- Load and Store Instructions
  - Load a local variable/constant on to the operand stack
  - Store a value from the operand stack into a local variable

Example Implementation

```
while (pc < end_addr) {
    switch (pc[0]) {
        case iload_0: {
            Frame.push(Frame.getLocal(0)); pc += 1;
            break;
        }
        case if_cmpeq: {
            value2 = Frame.pop();
            value1 = Frame.pop();
            if (value1 == value2) {
                offset = signext((pc[1] << 8) | pc[2]);
                pc += offset;
            } else {
                pc += 3;
            }
            break;
        }
        . . .
    }
}
```

Handling of Methods

- We are handling *static* methods only
- Interpret the main method
  - Check if the method name matches “main”
  - If so, start interpreting bytecode instructions of the main method
- *Invokevirtual #index*
  - Look up the method name from ConstantTable[index]
  - Check if the name matches “println”
  - Call C++ cout or C printf instead with the argument on the stack top

Tips

```
terminal> javac Test1.java
terminal> javap –v Test1.class > Test1.txt

== Test1.txt ==
Constant pool:
    #4 = Methodref   #5.#23 // Test1.printInt:(I)V
    #11 = Utf8    printInt
    #12 = Utf8    (I)V
    #13 = Utf
    #23 = NameAndType    #11:#12    // printInt:(I)V

Public static void main(…);
    Code:
        0: iconst_0
        1: istore_1
        …
```