Bytecode

Contents

Befehle

Lesen und Schreiben von Variablen

x - Nummer von Variable a

?- steht für typ. i - integer, a - reference, l - long, f - float, d - double, c-char

```
?load_x   // lokale Variable a laden
?store_x  // Oberster Wert auf Stack in lokaler Variable a speichern
?const_10 // Konstante 10 laden
bipush 10 // Konstante 10 laden
(iload, istore ... für integer)
```

Arithmetische Berechnungen

?- steht für typ. i - integer, a - reference, l - long, f - float, d - double, c-char

a auf dem Stack, dann b auf dem Stack, dann sub

-> (a - b) auf dem Stack, a, b und sub - nicht mehr

Allgemein: old - new - op -> (old op new)

```
?add // Addition der zwei letzten Werte auf Stack
?sub // Subtraktion der zwei letzten Werte auf Stack
?mult // Multiplikation der zwei letzten Werte auf Stack
?div  // Teilen der zwei letzten Werte auf Stack
?neg  // Negieren Zahl
(iadd, imult, idiv, ineg... für integer)


iload_1 //(a)
iload_2 //(b)
sub // --> a - b, div genau so
```

Sprungbefehle: 1 Parameter

x - das letzte Wert auf dem Stack

```
goto label //Sprung zu Sprungmarke label
ifeq label // x = 0 dann Sprung
ifge label // x >= 0 dann Sprung
ifgt label // x > 0 dann Sprung
ifle label // x <= 0 dann Sprung
iflt label // x < 0 dann Sprung
```

Sprungbefehle: compare two parameters

zuerst wurde x geladen, dann wurde y geladen

i - integer, cmp - compare, eq - equal, g - greater, l - less, e - equal, t - than

```
if_icmpeq label // x == y dann Sprung
if_icmpge label // x >= y dann Sprung
if_icmpgt label // x > y dann Sprung
if_icmple label // x <= y dann Sprung
if_icmplt label // x < y dann Sprung
```

Sprungbefehle: nullwerte

```
ifnotnull label // Wenn nicht null dann sprung
ifnull label // Wenn null dann sprung
```

Sprungbefehle: return

```
return  // Return void
?return //Return Wert aus dem Stack
(ireturn - return integer)
```

Methodenaufrufe

Objekt-Pointer soll auf dem Stack liegen, dann soll Parameter liegen, dann kommt invoke
Annahme: die Funktion foo() steht in Konstantenpool mit Nummer 12.

```
aload_x // this. Pointer laden
invokevirtual #12 // Funktion an lokaler Konstanten Stelle x laden mit
invokestatic #12  // Statische Funk. an lokaler Konstanten Stelle x laden
invokespecial #12  // Konstruktor aufrufen
```

Objekterzeugung

WURDE IN VORLESUNG UND ÜBUNG NICHT BEHANDELT

```
new #x // Neue Referenz von Typ x
dup // Oberster Stack Wert duplizieren wofür wird dup benutzt? -> Lerngruppe
fragen
invokespecial #x // Konstruktor an Stelle x aufrufen
areturn // Referenz zurückgeben (a - reference)
```

Sonstige Befehle

```
putfield FIELD:TYPE // Schreibe das Wert aus dem Stack in FIELD von TYPE
// Objekt-Pointer muss auf dem Stack liegen
// wird fast immer mit this.FIELD gemeint

getField FIELD:TYPE // Lese Wert, analog zu putfield
```

```
iinc v,n // inkrementiere die Variable mit der Adresse v um n (das neue Wert wird
automatisch in v gespeichert)
--> v = v + n
iinc 3,1 // inkrementiere die Variable mit der Adresse 3 um 1
```

FAQ

Funktionen

Wie kann man eine Funktion this.f aufrufen?

Annahmen:
this-Pointer ist in Variable 0 gespeichert,
Im Konstantenpool steht an Stelle 42 die Information für die nicht-stat Methode f.
f braucht keine Argumente

```
aload_0
invokevirtual #42
//dann liegt Ergebnis von f auf dem Stack
```

Aufruf mit parameter?

Annahmen:
Signatur f(int arg)
Argument liegt in Variable a mit Addresse 95

```
aload_0
iload_95
invokevirtual #42
//this laden -> parameter laden -> funktion
```

Aufruf mit parameter, der inline berechnet wird?

Annahmen:
f(a * (x - 1) + c)
a - 1, x - 2, c - 3

```
aload_0
iload_2
iconst_1
isub
iload_1
imult
iload_3
iadd
invokevirtual #42
```

Muss man ein „this“ hinzufügen, wenn es nicht explizit steht?

Ja, immer

IFs und Sprungbefehle

In welcher Rheihenfolge müssen Variablen auf Stack liegen?

„logische“ Rheihenfolge:
if (a > b) jump to x else y
a - 1, b - 2

```
iload 1
iload 2
if_icmpgt x   // integer compare greater than
goto y
```

Code generation

Minimale vs nichtminimale Stackverbrauch

Neg (Add (Const 1) (Add (Const 2) (Var 3)))
-(1+(2+x))
Nichtminimal:

```
bipush 1 //
bipush 2 //
iload_3 //
iadd
iadd
ineg
//es liegen 3 werte auf dem stack vor erstem iadd
```

Minimal:

```
bipush 2
iload_3
iadd
bipash 1
iadd
ineg
//max 2 werte
```

Code

Objekt erstellen

WURDE IN VORLESUNG UND ÜBUNG NICHT BEHANDELT

Java:

```
class Test {
  Test foo() {
    return new Test();
  }
}
```

Tabelle:

```
#1 java/lang/Object.<init>()V
#2 Test
#3 Test.<init>()V
```

Test() ausführen;

```
aload_0
invokespecial #1;
return
```

Test foo();

```
new #2;
dup
invokespecial #3;
areturn
```

Kurzauswertung und Negation

Immer von Links nach Rechts parsen und mehrmals die Labels überprüfen.

Logische Negation

Negation: Kein zusätzlicher Bytecode Befehl `not` o.ä., sondern: Vertauschung der Sprungziele. Beispiel:

```
if (x<y && y<3) {..} else {..}
        iload 1
        iload 2
        if_icmplt leftTrueLabel
        goto elseLabel
leftTrueLabel:
        iload 2
        iconst_3
        if_icmplt thenLabel
        goto elseLabel
thenLabel:
        ...
        goto afterLabel
elseLabel:
        ...
afterLabel:
```

```
if (!(x<y && y<3)) {..} else {..}
        iload 1
        iload 2
        if_icmplt leftTrueLabel
        goto thenLabel
leftTrueLabel:
        iload 2
        iconst_3
        if_icmplt elseLabel
        goto thenLabel
thenLabel:
        ...
        goto afterLabel
elseLabel:
        ...
afterLabel:
```

Aufgaben

|| , && und not

```
if (((a < b) || !((a < c) || (c < b))) && !(c < 0)) {
  c = b + a;
}
```

Hinweis: Um eine Bedingung der Form !*cond* zu übersetzen, genügt es, *cond* zu übersetzen und die Sprungziele zu tauschen.

Byte-Code:

```
     iload_0
     iload_1
     if_icmplt condition4
     iload_0
     iload_2
     if_icmplt afterif
     iload_2
     iload_1
     if_icmplt afterif
     goto condition4 // optional
 condition4:
     iload_2
     iconst_0
     if_icmplt afterif
     goto thenbranch // optional
 thenbranch:
     iload_0
     iload_1
     iadd
     istore_2
 afterif:
```

ByteCode <--> Java Code

Bytecode

Gegeben sei folgender Java-Bytecode einer Methode f:

```
public void f(int a, int b):
      0:  iconst_0
      1:  istore_3
      2:  iload_1
      3:  iload_2
      4:  if_icmplt 17
      7:  iload_1
      8:  iload_2
      9:  isub
      10: istore_1
      11: iinc 3, 1
```

```
14: goto 2
17: aload_0
18: iload_3
19: putfield x:I
22: aload_0
23: iload_1
24: putfield y:I
27: return
```

Java Code

Rekonstruieren Sie ein gültiges Java-Programm, für das dieser Bytecode erzeugt wird. Ergänzen Sie dazu in der rechts angegebenen Klassendefinition den Rumpf von `f`.

Sie dürfen dabei folgende Annahmen machen:

- `a` und `b` sind als lokale Variablen 1 und 2 verfügbar.
- Die lokale Variable 3 heißt `c`.

Beispiellösung:

```
class C {
    int x;
    int y;

    public void f(int a, int b) {
        int c = 0;
        while (a >= b) {
            a -= b;
            ++c;
        }
        x = c;
        y = a;
    }
}
```