Praktikum Compilerbau
Sitzung 9 – Java Bytecode

Lehrstuhl für Programmierparadigmen
Universität Karlsruhe (TH)

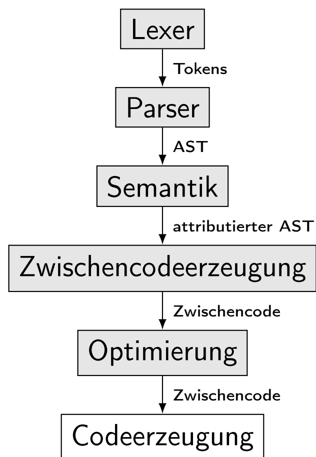
24. Juni 2009

1. Letzte Woche
2. Java Bytecode
3. Jasmin Bytecode Assembler
4. Sonstiges

Letzte Woche

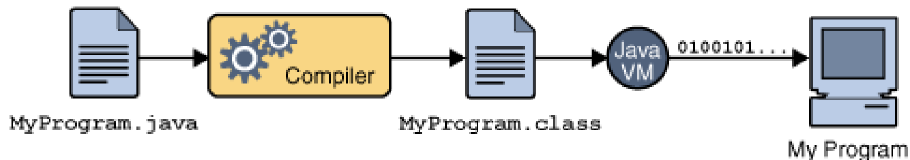
- Was waren die Probleme?
- Hat soweit alles geklappt?

Compilerphasen



1. Letzte Woche
2. Java Bytecode
3. Jasmin Bytecode Assembler
4. Sonstiges

Java Technologie

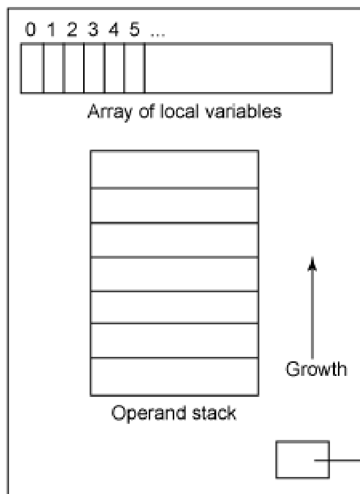


- Zwischensprache: Bytecode (Tipp: `javap -c [-verbose] <Class>`)
- Virtuelle Maschine (Stackmaschine)
- Umfangreiche Bibliothek (API)
- Laufzeitsystem
- *The Java Virtual Machine Specification*
  `http://java.sun.com/docs/books/jvms/`

Virtuelle Maschine - Laufzeitsystem

- Global nur 1x:
  - **Heap**: Daten der Objektinstanzen. (shared memory → Synchronisation)
  - **Method Area**: Je Klasse eine Konstantentabelle sowie Bytecode für Methoden / Konstruktoren
- Je Thread 1x:
  - **Program Counter**: Adresse des aktuell ausgeführten Bytecode-Befehls
  - **JVM Stack**: Activation Records der Methodenaufrufe (Rücksprungadresse, alter Framepointer, ...)
- Je aufgerufener Methodeninstanz 1x: **Methodenframe**

Virtuelle Maschine - Frame einer Methode



- Array mit lokalen Variablen und Parametern
  - $0 \leftrightarrow X$ Methoden Parameter (wobei $0 =$ `this`, $1 =$ 1. Parameter, ...)
  - $(X + 1) \leftrightarrow (X + Y)$ lokale Variablen (Ziel: möglichst wenige)
- Stack für die Operanden

Constant pool

Bytecode

Für jeden Bytecode Befehl gilt:

- Operanden (Parameter) auf dem Stack. (Reihenfolge: 1. Parameter $\rightarrow$ Top of Stack, ...)
- Rückgabewerte werden auch auf dem Stack abgelegt.

Es gibt u.a. Befehle für (`?` $\leftarrow$ Typ)

- Auslesen und Schreiben von lokalen Variablen (`?load <x>`, `?store <x>`, ...)
- Auslesen und Schreiben von Feldern (`getfield`, `putfield`, ...)
- Sprungbefehle (`ifeq`, `ifnull`, `tableswitch`, ...)
- Methodenaufrufe (`invokevirtual`, `invokestatic`, ...)
- Objekterzeugung (`new`, `newarray`, ..)
- Arithmetische Berechnungen (`?mul`, `?add`, ..)

Beispiel: Ausruck berechnen

```
void calc(int x) {
  int y;
  int z;
  ...
  x = x + y * z;
}
```

```
1  // Lade y
2  iload_2
3  // Lade z
4  iload_3
5  // y * z
6  imul
7  // Lade x
8  iload_1
9  // x + (y * z)
10 iadd
11 // Speichere x
12 istore_1
```

Beispiel: Ausruck berechnen

```
void calc(int x) {
  int y;
  int z;
  ...
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	6	...

Stack: | ⊥ | | |

Befehl: | |

```
1  // Lade y
2  iload_2
3  // Lade z
4  iload_3
5  // y * z
6  imul
7  // Lade x
8  iload_1
9  // x + (y * z)
10 iadd
11 // Speichere x
12 istore_1
```

Beispiel: Ausruck berechnen

```
void calc(int x) {
  int y;
  int z;
  ...
  x = x + y * z;
}
```

```
1  // Lade y
2  iload_2
3  // Lade z
4  iload_3
5  // y * z
6  imul
7  // Lade x
8  iload_1
9  // x + (y * z)
10 iadd
11 // Speichere x
12 istore_1
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	6	...

Stack: | **5** | ⊥ | |

Befehl: `iload_2`

Beispiel: Ausruck berechnen

```
void calc(int x) {
  int y;
  int z;
  ...
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	6	...

Stack: | **6** | 5 | ⊥ |

Befehl: `iload_3`

```
1  // Lade y
2  iload_2
3  // Lade z
4  iload_3
5  // y * z
6  imul
7  // Lade x
8  iload_1
9  // x + (y * z)
10 iadd
11 // Speichere x
12 istore_1
```

Beispiel: Ausruck berechnen

```java
void calc(int x) {
  int y;
  int z;
  ...
  x = x + y * z;
}
```

```
1  // Lade y
2  iload_2
3  // Lade z
4  iload_3
5  // y * z
6  imul
7  // Lade x
8  iload_1
9  // x + (y * z)
10 iadd
11 // Speichere x
12 istore_1
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	6	...

Stack: | 30 | ⊥ | |

Befehl: imul

Beispiel: Ausruck berechnen

```
void calc(int x) {
  int y;
  int z;
  ...
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	6	...

Stack: | **7** | 30 | ⊥ |

Befehl: iload_1

```
1  // Lade y
2  iload_2
3  // Lade z
4  iload_3
5  // y * z
6  imul
7  // Lade x
8  iload_1
9  // x + (y * z)
10 iadd
11 // Speichere x
12 istore_1
```

Beispiel: Ausruck berechnen

```
void calc(int x) {
  int y;
  int z;
  ...
  x = x + y * z;
}
```

```
1  // Lade y
2  iload_2
3  // Lade z
4  iload_3
5  // y * z
6  imul
7  // Lade x
8  iload_1
9  // x + (y * z)
10 iadd
11 // Speichere x
12 istore_1
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	6	...

Stack:

37	⊥	

Befehl: iadd

Beispiel: Ausruck berechnen

```
void calc(int x) {
  int y;
  int z;
  ...
  x = x + y * z;
}
```

```
1  // Lade y
2  iload_2
3  // Lade z
4  iload_3
5  // y * z
6  imul
7  // Lade x
8  iload_1
9  // x + (y * z)
10 iadd
11 // Speichere x
12 istore_1
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	**37**	5	6	...

Stack:

⊥		

Befehl: `istore_1`

Beispiel: Fibonnaci-Berechnung

```
static void fib() {
  long num = 1;
  long a = 1;
  long b = 1;
  for (;;) {
    num = a + b;
    a = b;
    b = num;
  }
}
```

```
1  lconst_1 // constant 1
2  lstore_0 // set num
3  lconst_1 // constant 1
4  lstore_1 // set a
5  lconst_1 // constant 1
6  lstore_2 // set b
7  lload_1 // load a
8  lload_2 // load b
9  ladd // add a+b
10 lstore_0 // set num
11 lload_2 // load b
12 lstore_1 // set a
13 lload_0 // load num
14 lstore_2 // set b
15 goto 7 // loop forever
```

Methodenaufrufe

1. Bezugsobjekt auf den Stack (falls nicht **static**)
2. Parameter auf den Stack
3. **invokevirtual** / **invokestatic** ausführen:
   Folgendes passiert vor / nach dem Aufruf automatisch:
   1. Array für Parameter und lokale Variablen anlegen (Größe ist angegeben)
   2. Returnadresse (Program Counter+1) und alten Framepointer sichern
   3. Neuen Framepointer setzen
   4. **this** Pointer und Parameter vom Stack ins Parameter Array kopieren
   5. Zu Methodenanfang springen und Code ausführen
   6. Returnwert auf den Stack
   7. Alten Framepointer setzen und zur Returnadresse springen
4. Returnwert vom Stack holen und weiterverarbeiten

Beispiel: Methodenaufruf

```
int bar() {
  return foo(42);
}

int foo(int i) {
  return i;
}
```

Konstantenpool

#2	Method	#3.#16
#3	class	#17
#11	Asciz	foo
#12	Asciz	(I)I
#16	NameAndType	#11:#12
#17	Asciz	Test

```
1 int bar();
2    aload_0
3    bipush 42
4    invokevirtual #2
5    ireturn
6
7 int foo(int);
8    iload_1
9    ireturn
```

Deskriptoren

Namen von Klassen, Feldern und Methoden müssen einem festgelegtem Schema entsprechen. (siehe JVMS 4.3)

- Klassennamen: `java.lang.Object` → `Ljava/lang/Object;`
- Typen: `int` → `I`, `void` → `V`, `boolean` → `Z`
- Methoden: `void foo(int, Object)` → `foo(ILjava/lang/Object;)V`
  Deskriptor: `(` *Parametertypen* `)` *Rückgabetyp*
  Identifiziert über "*Name* × *Deskriptor*"
- Felder: `boolean b` → `b:Z`
  Identifiziert nur über "*Name*"
- Konstruktoren: Name ist `<init>`, Static Initializer `<clinit>`

Objekt erzeugen & initialisieren

1. Objekt anlegen → Speicher reservieren
2. Objekt initialisieren → Konstruktor aufrufen

Hinweis: Jede Klasse braucht einen Konstruktor (Defaultkonstruktor)!

```
class Test {

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

}
```

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

```
1  Test();
2    aload_0
3    invokespecial #1;
4    return
5
6  Test foo();
7    new #2;
8    dup
9    invokespecial #3;
10   areturn
```

1 Letzte Woche

2 Java Bytecode

3 Jasmin Bytecode Assembler

4 Sonstiges

Jasmin Assembler

Der "bessere" Bytecode:

- `http://jasmin.sourceforge.net/`
- An Bytecode angelehnte Assemblersprache
- Leichter lesbar → debuggen
- Befehle sehr ähnlich der Ausgabe von `javap`
- Sprungmarken → keine Bytecodepositionen
- Automatischer Aufbau des Konstantenpools
- Einfache Installation: Es reicht `jasmin.jar`
- Aufruf: `java -jar jasmin.jar <Datei>`

Jasmin Klasse

- Header:

```
1  .class <Modifier> <ClassName>
2  .super <SuperClass>
```

- Methode:

```
1  .method <Modifier> <Name and Deskriptor>
2          <Code>
3  .end method
```

- Feld:

```
1  .field <Modifier> <FieldName> <Descriptor> [= <Value>]
```

Beispiel: Jasmin Code

```
.class Test
.super java/lang/Object

.method public <init>()V
  aload_0
  invokenonvirtual java/lang/Object/<init>()V
  return
.end method

.method foo()LTest;
 .limit locals 1
 .limit stack 2
  new Test
  dup
  invokespecial Test/<init>()V
  areturn
.end method
```

Kontrollfluss mit Sprungmarken

```java
void foo(int z) {
  int i = 0;
  while (i < z) {
    i = i + 1;
  }
}
```

```
1  .method foo(I)V
2      .limit locals 2
3      .limit stack 3
4      iconst_0
5      istore_2
6  l1:
7      iload_2
8      iload_1
9      if_icmpge l2
10     iload_2
11     iconst_1
12     iadd
13     istore_2
14     goto l1
15 l2:
16     return
17 .end method
```

1. Letzte Woche
2. Java Bytecode
3. Jasmin Bytecode Assembler
4. Sonstiges

Zum Schluss

Hinweise:

- Befehle sind getypt: Der richtige Typ muss gefunden werden
- Spezialversionen gängiger Befehle (schneller):
  z.B. `istore_1` ↔ `istore 1` aber nicht `istore_231`
- Übungsblatt beachten: Jasmin Programme schreiben

Noch Fragen?

- Anmerkungen?
- Probleme?
- Fragen?