Analizator statyczny bajtkodu Javy

Michał Ziemba

20 października 2010

Plan prezentacji

O czym jest praca?

- Celem jest napisanie narzędzia, umożliwiającego prostą analizę statyczną kodu wykonywalnego Javy
- Narzędzie będzie zintegrowane ze środowiskiem Umbra
- Analiza będzie obejmować wykrywanie dereferencji wskaźników, które mogą być *NULL*, ...

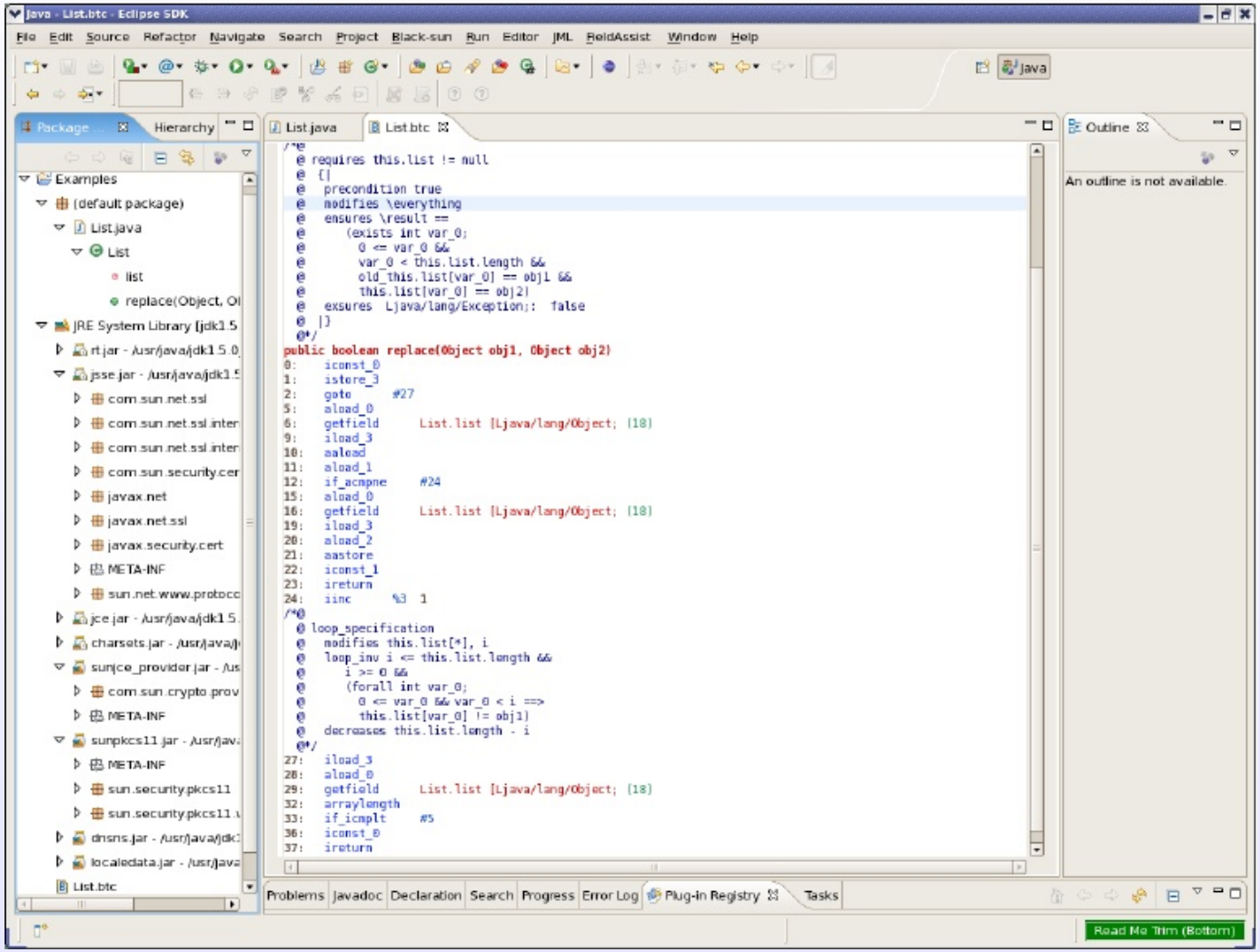
Motywacja

- Bajtkod składa się z 200 instrukcji
- Możliwa jest faktoryzacja tych instrukcji do 12 grup instrukcji, co znacznie ułatwia analizę
- Nie ma potrzeby zajmowania się kodem źródłowym

Umbra

- Plugin do Eclipse, umożliwiający generowanie bajtkodu z pliku *.class*
- Jest w fazie rozwojowej
- Umożliwia pisanie specyfikacji w BML

Umbra


Java - List.btc - Eclipse SDK
File Edit Source Refactor Navigate Search Project Black-sun Run Editor JML FieldAssist Window Help
Java
Package Hierarchy
List.java List.btc
Outline
An outline is not available.
Examples
(default package)
List.java
List
list
replace(Object, Ol
JRE System Library [jdk1.5
rt.jar - /usr/java/jdk1.5.0
jsse.jar - /usr/java/jdk1.5
com.sun.net.ssl
com.sun.net.ssl.inter
com.sun.net.ssl.inter
com.sun.security.cer
javax.net
javax.net.ssl
javax.security.cert
META-INF
sun.net.www.protoco
jce.jar - /usr/java/jdk1.5.
charsets.jar - /usr/java/j
sunjce_provider.jar - /us
com.sun.crypto.prov
META-INF
sunpkcs11.jar - /usr/java
META-INF
sun.security.pkcs11
sun.security.pkcs11.
dnsns.jar - /usr/java/jdk
localedata.jar - /usr/java
List.btc
@ requires this.list != null
@ []
@ precondition true
@ modifies \everything
@ ensures \result ==
@ (exists int var_0;
@ 0 <= var_0 &&
@ var_0 < this.list.length &&
@ old_this.list[var_0] == obj1 &&
@ this.list[var_0] == obj2)
@ exsures (Ljava/lang/Exception;) false
@ []
@*/
public boolean replace(Object obj1, Object obj2)
0: iconst_0
1: istore_3
2: goto #27
5: aload_0
6: getfield List.list [Ljava/lang/Object; (18)
9: iload_3
10: aaload
11: aload_1
12: if_acmpne #24
15: aload_0
16: getfield List.list [Ljava/lang/Object; (18)
19: iload_3
20: aload_2
21: aastore
22: iconst_1
23: ireturn
24: iinc %3 1
/*@
@ loop_specification
@ modifies this.list[*], i
@ loop_inv i <= this.list.length &&
@ i >= 0 &&
@ (forall int var_0;
@ 0 <= var_0 && var_0 < i ==>
@ this.list[var_0] != obj1)
@ decreases this.list.length - i
@*/
27: iload_3
28: aload_0
29: getfield List.list [Ljava/lang/Object; (18)
32: arraylength
33: if_icmplt #5
36: iconst_0
37: ireturn
Problems Javadoc Declaration Search Progress Error Log Plug-in Registry Tasks
Read Me Trim (Bottom)

Bajtkod

```
outer:
for (int i = 2; i < 1000; i++) {
    for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}
```

```
0:   iconst 2
1:   istore 1
2:   iload 1
3:   sipush  1000
6:   if icmpge       44
9:   iconst 2
10:  istore 2
11:  iload 2
12:  iload 1
13:  if icmpge       31
16:  iload 1
17:  iload_2
18:  irem
19:  ifne    25
22:  goto    38
25:  iinc    2, 1
28:  goto    11
31:  getstatic       #84; //Field java/lang/System.out:Ljava/io/PrintStream;
34:  iload 1
35:  invokevirtual   #85; //Method java/io/PrintStream.println:(I)V
38:  iinc    1, 1
41:  goto    2
44:  return
```

Dziedziny semantyczne

Podstawowa forma pojedynczego kroku:

$$P \vdash h,\ ts \rightarrow h',\ ts'$$

Struktury semantyczne:

$$Heap = [Loc \times ThrId \rightarrow_{fin} (Cnames \times Monitor \times [Fnames \rightarrow_{fin} Val])]$$
$$Thread = ThrId \times ThreadStatus \times EvalState \times FrameStack$$
$$FrameStack = MethodFrame^*$$
$$MethodFrame = Cnames \times Mnames \times LVals \times OpStack \times PC$$

$load(k, n)$

Instrukcja, która odczytuje lokalne wartości i wkłada je na stos. Typy dzielimy na $32 - bitowe$ ($Cat1^{\bullet}$) i $64 - bitowe$ ($Cat2$).

$$\frac{lv_{ts} = (k, m) \quad ostck' = (k, m) \cdot ostck_{ts} \quad pc' = next(cmthd, pc_{ts}) \quad P@pc_{ts}.mnm_{ts}.cnm_{ts} = load(k, n) \quad k \in Cat1^{\bullet} \quad est_{ts} = null}{P \vdash h, ts \rightarrow h, ts\,[ostck \leftarrow ostck']\,[pc \leftarrow pc']}$$

$$\frac{lv_{ts}(n) = (k, m_1) \quad lv_{ts}(n+1) = (k, m_2) \quad ostck' = (k, k(m_1, m_2)) \cdot ostck_{ts} \quad pc' = next(cmthd, pc_{ts}) \quad P@pc_{ts}.mnm_{ts}.cnm_{ts} = load(k, n) \quad k \in Cat2 \quad est_{ts} = null}{P \vdash h, ts \rightarrow h, ts\,[ostck \leftarrow ostck']\,[pc \leftarrow pc']}$$

$store(k, n)$

Instrukcja, która zdejmuje wartość ze stosu i umieszcza ją w tablicy zmiennych lokalnych

$$\frac{\begin{array}{c} lv' = lv_{ts}\,[n \leftarrow (k, m)] \\ ostck_{ts} = (k, m) \cdot ostck' \quad pc' = next(cmthd, pc_{ts}) \\ P@pc_{ts}.mnm_{ts}.cnm_{ts} = store(k, n) \quad k \in Cat1^{\bullet} \quad est_{ts} = null \end{array}}{P \vdash h, ts \rightarrow h, ts\,[ostck \leftarrow ostck']\,[pc \leftarrow pc']\,[lv \leftarrow lv']}$$

$$\frac{\begin{array}{c} lv' = lv_{ts}\,[n \leftarrow (k, m_1)]\,[n+1 \leftarrow (half, m_2)] \\ ostck' = (k, k(m_1, m_2)) \cdot ostck_{ts} \quad pc' = next(cmthd, pc_{ts}) \\ P@pc_{ts}.mnm_{ts}.cnm_{ts} = store(k, n) \quad k \in Cat2 \quad est_{ts} = null \end{array}}{P \vdash h, ts \rightarrow h, ts\,[ostck \leftarrow ostck']\,[pc \leftarrow pc']}$$

$stackop(op)$

Instrukcja, która używa wyłącznie stosu. $kinds_{stackop}(op)$ jest zbiorem trójek: lista typów wejściowych, funkcja f i lista typów wyjściowych. Na przykład

$$kinds_{stackop}(iadd) = \{([int, int], f_{iadd}, [int])\}$$

$$\frac{\begin{array}{c}(l, f, l') \in kinds_{stackop}(op) \quad ostck_{ts} = s \cdot r \\ check(s, l) \quad ostck' = f(s) \cdot r \quad pc' = next(cmthd, pc_{ts}) \\ P@pc_{ts}.mnm_{ts}.cnm_{ts} = stackop(op) \quad est_{ts} = null\end{array}}{P \vdash h, ts \rightarrow h, ts\,[ostack \leftarrow ostack']\,[pc \leftarrow pc']}$$

$cond(op, d)$

Instrukcja, która może wpływać na przebieg sterowania w programie, ale nie modyfikuje stosu ramek (m.in. skoki warunkowe i bezwarunkowe)
Funkcja f zwraca zmodyfikowany stos oraz nową wartość PC

$$\frac{\begin{array}{c}(l, f, l') \in kinds_{cond}(op) \quad ostck_{ts} = s \cdot r \\ check(s, l) \quad (s', pc') = f(d, s, next(cmthd, pc_{ts})) \quad ostck' = s' \cdot r \\ P@pc_{ts}.mnm_{ts}.cnm_{ts} = cond(op, d) \quad est_{ts} = null\end{array}}{P \vdash h, ts \rightarrow h, ts\,[ostack \leftarrow ostack']\,[pc \leftarrow pc']}$$

$iinc(n, c)$

Tylko jedna instrukcja($iinc$), używająca wyłącznie tablicy zmiennych lokalnych

$$\frac{lv_{ts}(n) = (int, m) \quad lv' = lv_{ts}\,[n \leftarrow (int, m +_{int} c)] \quad pc' = next(cmthd, ps_{ts}) \quad P@pc_{ts}.mnm_{ts}.cnm_{ts} = iinc(n, c) \quad est_{ts} = null}{P \vdash h, ts \rightarrow h, ts\,[lv \leftarrow lv']\,[pc \leftarrow pc']}$$

$get(op, d)$

Instrukcja odczytująca wartość z *Heap* i wstawiająca ją na *OpStack*

$$\frac{(l, f, l') = kinds_{get}(op, d) \quad ostck_{ts} = s \cdot r \quad check(s, l) \quad s' = f(d, s, h) \quad s' \in OpStack \quad ostck' = s' \cdot r \quad est_{ts} = null \quad pc' = next(cmthd, ps_{ts}) \quad P@pc_{ts}.mnm_{ts}.cnm_{ts} = get(op, d)}{P \vdash h, ts \rightarrow h, ts\,[lv \leftarrow lv']\,[pc \leftarrow pc']}$$

$$\frac{(l, f, l') = kinds_{get}(op, d) \quad ostck_{ts} = s \cdot r \quad check(s, l) \quad e = f(d, s, g) \quad e \in Loc^{\bullet} \quad P@pc_{ts}.mnm_{ts}.cnm_{ts} = iinc(n, c) \quad est_{ts} = null}{P \vdash h, ts \rightarrow h, ts\,[est \leftarrow e]}$$

$put(op, d)$

Instrukcja odczytująca wartość z *OpStack* i wstawiająca ją na *Heap*

$$\frac{(l, f, l') = kinds_{put}(op, d) \quad ostck_{ts} = s \cdot r \quad check(s, l) \quad ostck' = r \quad h' = f(d, s, h) \quad h' \in Heap \quad pc' = next(cmthd, ps_{ts}) \quad P@pc_{ts}.mnm_{ts}.cnm_{ts} = put(op, d) \quad est_{ts} = null}{P \vdash h, ts \rightarrow h', ts\,[ostck \leftarrow ostck']\,[pc \leftarrow pc']}$$

$$\frac{(l, f, l') = kinds_{put}(op, d) \quad ostck_{ts} = s \cdot r \quad check(s, l) \quad ostck' = r \quad e = f(d, s, h) \quad e \in Loc^{\bullet} \quad P\ pc_{ts}.mnm_{ts}.cnm_{ts} = put(op, d) \quad est_{ts} = null}{P \vdash h, ts \rightarrow h, ts\,[est \leftarrow e]}$$

$new(op, d)$

Instrukcja przydzielająca pamięć tworzonej strukturze. W odróżnieniu od *put*, *new* tworzy nowe lokacje na stosie, zamiast modyfikować już istniejące

$$\frac{\begin{array}{c}(l, f, l') \in kinds_{new}(op, d) \quad ostck_{ts} = s \cdot r \quad check(s, l) \\ pc' = next(cmthd, ps_{ts}) \quad ostck' = s' \cdot r \\ (s', h') = f(d, s, h) \quad s' \in OpStack \quad h' \in Heap \\ P@pc_{ts}.mnm_{ts}.cnm_{ts} = new(op, d) \quad est_{ts} = null\end{array}}{P \vdash h, ts \rightarrow h', ts\,[lv \leftarrow lv']\,[pc \leftarrow pc']}$$

$$\frac{\begin{array}{c}(l, f, l') = kinds_{new}(op, d) \quad ostck_{ts} = s \cdot r \quad check(s, l) \\ e = f(d, s, h) \quad e \in Loc^{\bullet} \quad P@pc_{ts}.mnm_{ts}.cnm_{ts} = new(op, d) \\ est_{ts} = null\end{array}}{P \vdash h, ts \rightarrow h, ts\,[est \leftarrow e]}$$

monitor(*op*)

Instrukcja modyfikująca stan wątków poprzez zajęcie ($op = enter$) lub zwolnienie ($op = exit$) monitora

$$\frac{\begin{array}{c}(l, f, l') \in kinds_{new}(op, d) \quad ostck_{ts} = s \cdot r \quad check(s, l) \\ ostck' = s' \cdot r \\ (s', h') = f(d, s, h) \quad s' \in OpStack \quad h' \in Heap \\ pc' = next(cmthd, ps_{ts}) \quad P@pc_{ts}.mnm_{ts}.cnm_{ts} = new(op, d) \\ est_{ts} = null\end{array}}{P \vdash h, ts \rightarrow h', ts\,[lv \leftarrow lv']\,[pc \leftarrow pc']}$$

monitor(*op*) *cd*.

$$\frac{(l, f, l') = kinds_{new}(op, d) \quad ostck_{ts} = s \cdot r \quad check(s, l) \quad e = f(d, s, h) \quad e \in Loc^{\bullet} \quad P@pc_{ts}.mnm_{ts}.cnm_{ts} = new(op, d) \quad est_{ts} = null}{P \vdash h, ts \longrightarrow h, ts\,[est \leftarrow e]}$$

invoke(*mode*, *cnm*, *mnm*)

- Instrukcja modyfikująca stos i stos ramek
- *mode* może być jednym z `interface`, `special`, `static` lub `virtual`
- Funkcja `dispatch` między innymi sprawdza czy flagi metody nie są sprzeczne z `mode` oraz czy prawa dostępu są zachowane (`private`, `protected`)
- Funkcja `initlv` umieszcza wartości ze stosu w tablicy zmiennych lokalnych

$invoke(mode, cnm, mnm)\ cd.$

$$\frac{\begin{array}{c}(l, l') = kindsof_{invoke}(mode, cnm, mnm) \quad ostck_{ts} = s \cdot r \quad check(s, l) \\ cmn' = dispatch(mode, cnm, mnm, s, h) \quad tfs' = tfs_{ts}\,[ostck \leftarrow r] \\ lv' = initlv(lvlength(P@mnm.cnm), s) \\ tfs'' = \langle cnm', mnm, lv', [\,], 0\rangle \cdot tfs' \\ P@pc_{ts}.mnm_{ts}.cnm_{ts} = invoke(mode, cnm, mnm) \quad est_{ts} = null\end{array}}{P \vdash h, ts \rightarrow h, ts\,[tfs \leftarrow tfs'']}$$

$$\frac{\begin{array}{c}(l, l') = kindsof_{invoke}(mode, cnm, mnm) \quad ostck_{ts} = s \cdot r \quad check(s, l) \\ e = dispatch(mode, cnm, mnm, s, h) \quad e \in Loc \\ P@pc_{ts}.mnm_{ts}.cnm_{ts} = invoke(mode, cnm, mnm) \quad est_{ts} = null\end{array}}{P \vdash h, ts \rightarrow h, ts\,[est \leftarrow e]}$$

$return(l)$

Instrukcja usuwająca aktualną ramkę ze stosu oraz modyfikująca poprzednią ramkę

$$\frac{\begin{array}{c} ostck_{ts} = s \cdot r \quad check(s, l) \\ tfs_{ts} = f_1 \cdot \langle cnm', mnm', lv', ostck', pc' \rangle \cdot tfs^{tail} \quad f_1 \in MethodFrame \\ tfs' = \langle cnm', mnm', lv', vsc(s) \cdot ostck', next(P@mnm'.cnm', pc') \rangle \cdot tfs^{tail} \\ P@pc_{ts}.mnm_{ts}.cnm_{ts} = return(l) \quad est_{ts} = null \end{array}}{P \vdash h, ts \rightarrow h, ts\,[tfs \leftarrow tfs']}$$

$return(l)\ cd.$

$$\frac{ostck_{ts} = s \cdot r \quad check(s, l) \quad tfs_{ts} = [f] \quad f \in MethodFrame \quad tfs' = [] \quad tstatus' = TERMINATED \quad P@pc_{ts}.mnm_{ts}.cnm_{ts} = return(l) \quad est_{ts} = null}{P \vdash h, ts \rightarrow h, ts\,[tfs \leftarrow tfs']\,[tstatus \leftarrow tstatus']}$$

throw

Instrukcja nie mająca argumentów, odczytująca i usuwająca lokację wyjątku ze stosu i zmieniająca stan wykonania aktualnego wątku
Do kolejnego stanu można przejść albo wykonując instrukcję `throw`, albo rzucając wyjątek

$$\frac{ostck_{ts} = e \cdot r \quad e \in Loc^{\bullet} \qquad P@pc_{ts}.mnm_{ts}.cnm_{ts} = throw \quad est_{ts} = null}{P \vdash h, ts \rightarrow h, ts\,[est \leftarrow e]}$$

$$\frac{ostck' = [e] \quad (pc_{ts}, h(e)@cnm) \in dom(P@etable.nmn_{ts}.cnm_{ts}) \qquad pc' = P@etable.mnm_{ts}.cnmts(pc_{ts}, h(e)@cnm) \quad est_{ts} = e \in Loc^{\bullet}}{P \vdash h, ts \rightarrow h, ts\,[ostck \leftarrow ostck']\,[pc \leftarrow pc']\,[est \leftarrow null]}$$

throw cd.

$$\frac{(pc_{ts}, h(e)@cnm) \notin dom(P@etable.nmn_{ts}.cnm_{ts}) \quad tfs_{ts} = f_1 \cdot f_2 \cdot tfs_{ts}^{tail} \quad f_1, f_2 \in MethodFrame \quad est_{ts} = e \in Loc^{\bullet}}{P \vdash h, ts \rightarrow h, ts \left[tfs \leftarrow f_2 \cdot tfs_{ts}^{tail} \right]}$$

$$\frac{(pc_{ts}, h(e)@cnm) \notin dom(P@etable.nmn_{ts}.cnm_{ts}) \quad tfs_{ts} = [f] \quad f \in MethodFrame \quad est_{ts} = e \in Loc^{\bullet}}{P \vdash h, ts \rightarrow h, ts \, [tfs \leftarrow [\,]] \, [tstatus \leftarrow TERMINATED]}$$

NonNull

- Niektóre języki obiektowe i funkcyjne nie pozwalają odwoływać się do obiektów w czasie ich tworzenia, jednak w Javie nie ma tego ograniczenia
- Potrzebujemy annotacji opisujących obiekty, które mogą być `NULL`, a także obiekty które mogą być częściowo zainicjalizowane
- Dzięki takim annotacjom w kodzie można przeprowadzić podstawową analizę

```
class A {
  /*@ non_null @*/ string name;

  public A (/*@ non_null @*/ string s) {
    this.name = s;
    this.m(55);
  }

  virtual void m(int x) {...}
}

class B extends A {
  /*@ non_null @*/ string path;

  public B (/*@ non_null @*/ string p; /*@ non_null @*/ string s) {
    super();
    this.path = p;
  }

  override void m(int x) {
    ...
    this.path
    ...
  }
}
```

- Analizator statyczny bajtkodu Javy może być przydatnym narzędziem, gdy chcemy wykonać proste sprawdzenie kodu bez oglądania i parsowania kodu źródłowego
- Dzięki uproszczeniu bajtkodu do 12 grup instrukcji analiza będzie łatwiejsza
- Początkowo przewidziana jest tylko analiza w zakresie dreferencji `NULLi`, ale będzie możliwość rozszerzenia narzędzia o inne sprawdzenia

Bibliografia

1. J. Chrząszcz, P. Czarnik, A. Schubert, *A dozen instructions make Java bytecode*
2. M. Fähndrich, K. R. M. Leino, *Declaring and Checking Non-null Types in an Object-Oriented Language*
3. `http://www.mimuw.edu.pl/~alx/umbra/bml-bytecode.pdf`
4. `http://java.sun.com/docs/books/jvms/second_edition/html/Overview.doc.html`