



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSI NYELVEK ÉS FORDÍTÓPROGRAMOK

TANSZÉK

Java bytecode interpreter Javában

Témavezető:

Kozsik Tamás Dr.

egyetemi docens

Szerző:

Balázs Zoltán

programtervező informatikus BSc

Budapest, 2023

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:

Név: Balázs Zoltán

Neptun kód: HV56L5

Képzési adatok:

Szak: programtervező informatikus, alapképzés (BA/BSc/BProf)

Tagozat : Nappali

Belső témavezetővel rendelkezem

Témavezető neve: Kozsik Tamás Dr.

munkahelyének neve, tanszéke: ELTE IK, Programozási nyelvek és Fordítóprogramok Tanszék

munkahelyének címe: 1117, Budapest, Pázmány Péter sétány 1/C.

beosztás és iskolai végzettsége: egyetemi docens, programtervező matematikus

A szakdolgozat címe: Java bytecode interpreter Javában

A szakdolgozat témája:

(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását)

A Java nyelvben írt programok fordításuk során nem közvetlenül gépi kódra fordulnak, hanem egy hardver-független nyelvre, amit bytecode-nak neveznek.

Ezt a bytecode-ot az esetek többségében a JVM (Java Virtual Machine) interpreter-e hajtva végre, vagy futási időben fordul le a fordító gép hardverének gépi kódjára.

A szakdolgozat célja egy olyan Java bytecode interpreter fejlesztése, amely képes már előre, valamilyen Java fordító által, elkészített bytecode-ot interpreter-álni, ezt sikeresen (és helyesen) lefuttatni.

A fejlesztett interpreter-nek képesnek kell lennie az ELTE Programtervező Informatikus BSc szakán, különböző, Java-t használó tárgyakon (Programozási nyelvek, Konkurens programozás) elkészített beadandók és házi feladatok generált bytecode-ját interpreter-álni, ezeket helyesen futtatni.

Budapest, 2022. 11. 24.

Tartalomjegyzék

1. Bevezetés	3
2. Felhasználói dokumentáció	5
2.1. Kikötések	5
2.2. Fordítástól futásig	6
2.2.1. Minimum követelmények	6
2.2.2. Fordítás	6
2.2.3. Futtatás	7
2.2.4. Önfuttatás	8
2.3. Felmerülő problémák	8
3. Fejlesztői dokumentáció	9
3.1. Class fájl felépítése	9
3.1.1. Class fájlról a benne levő metódus futtatásáig	9
3.1.2. Pár minta class fájl felépítése	12
3.1.3. Adatszerkezetek	14
3.1.4. Interpretálás algoritmusai	16
3.1.5. Interpretálás működése	16
3.2. Az interpreter sajátosságai	17
3.2.1. Erőforrás igények	17
3.2.2. A program memóriamodellje	19
3.2.3. Az önfuttatásról	20
3.3. Tesztelő környezet	20
3.3.1. Tesztelés	21
3.3.2. Saját tesztet hozzáadása	21
3.4. Továbbfejlesztési lehetőségek	21
3.4.1. Invokedynamic utasítás	21
3.4.2. Java 7 előtti verziók támogatása	22

3.4.3. Optimalizálás	22
3.4.4. További tesztelés	23
3.4.5. Reflektió újragondolása	24
3.5. Érdekességek a JVM specifikációból	25
4. Összegzés	27
Köszönetnyilvánítás	28
Irodalomjegyzék	28
Táblázatjegyzék	30
Forráskódjegyzék	31

1. fejezet

Bevezetés

A Java nyelvben írt programok fordításukat követően nem egy közletlen futtatható állományra (gépi kódra) fordulnak (a fordítást általában a beépített `javac` program végzi el), hanem egy köztes nyelvre, Java bájtkódra, amelyet aztán különböző programokkal az adott architektúrán interpretáljuk. Legtöbb esetben az interpretálást a JVM (Java Virtual Machine, magyarul Java virtuális gép) interpretere hajtja végre (ez a beépített `java` program).

A szakdolgozat célja egy kiegészítő program (fantázianevét *Jabyinja: **Java** bytecode interpreter in **Java***) írása, amely ugyan hagyatkozik a `javac` és `java` programokra (az előbbire a fordítás, az utóbbira a futtatás miatt), de a tényleges futtatást a különböző Java bájtkód instrukciók (utasítások) implementálásával végzi el.

A program nincsen Java kód interpretálásához kötve, a Java bájtkód a neve ellenére nem csak a Java programozási nyelvnek a bájtkódja, más programozási nyelveknek is az alapja (ezek közül pár: Clojure, Kotlin, Scala), pontosabban azoknak amelyek a JVM-et használják fel, viszont a tesztelés csak Java kódból generált Java bájtkódra tér ki, ugyanis a szakdolgozat céljaként az ELTE Programtervező Informatikus BSc szakán elkészített Java programok lefordított class fájlainak interpretálását tűztem ki.

A programnak szükséges értelmeznie kell egy adott class fájlt (többet is ha egy külön fájlra is hivatkozunk), helyesen beolvasnia a benne lévő adatokat, majd a belépési (*main*) metódust lefuttatnia. A program erősen alapszik a Java nyelvbe beépített refleksióra, ezen felül saját stack és lokális változók implementálása is szükséges. Mivel a Java nyelvre épül a program, ezért saját heap megírására nincsen szükség, ez automatikusan kezelve lesz a standard Java interpreter által.

A program a JVM specifikációt[1] (ennek is a 7-es verzióját) követi, azt megpróbálja tökéletesen implementálni, minden egyes megoldási döntésével (jó vagy rossz) együtt.

2. fejezet

Felhasználói dokumentáció

A program elsődleges felhasználói fejlesztők, alapszintű tudás szükséges a Java nyelvről (vagy bármilyen olyan nyelvről amely Java bájtkódra fordul), a class fájlokról, illetve Java programok fordításáról.

Mivel az elkészített program csak interpretálni tud, a fordítást egy már elérhető Java fordítóprogrammal szükséges megtenni. Mivel a Java programok class fájlokra fordulnak, ezek futtatásához szükséges egy interpretáló program.

Alapvető esetben ez a fordítóprogrammal együtt telepítésre kerül. A szakdolgozat esetében a lefordított class fájl futtatásával képesek lehetünk más, már lefordított Java programot futtatni.

A mellékelt fájlok között elérhető egy jar fájl is, ennek a futtatásához ugyanúgy szükségünk van egy beépített interpretáló programra, amely képes Java programokat futtatni és nem a szakdolgozat maga.

2.1. Kikötések

A program csak Java 7-nél újabb fordítóprogrammal fordított Java programokat képes interpretálni, számos bájtkód instrukciót a Java 7-es verziójában elavulttá tettek (ezek: `ret`, `jsr`, `jsr_w`), nem fordulnak elő class fájlokban. A szakdolgozat ezeket az instrukciókat nem implementálta. (Implementálásukról részletesebben szó van a fejlesztői dokumentáció "Továbbfejlesztési lehetőségek" szekciójában, implementálásuk viszonylag triviális, viszont megértésük segít elmélyülni a Java bájtkódban.)

Ezen felül egy másik, a Java 7-nél újabb verziójú programokban elég gyakran előforduló instrukció is implementálatlan maradt (név szerint az `invokedynamic`), tehát

nem minden program futtatható. Ennek az indoka hogy ez az utasítás nagyon nagy szintű elmélyülést igényel a Java bájtkódban, értelmezése meghaladja egy szakdolgozat szintjét. Ez az instrukció önmagában használható arra hogy egy, a szakdolgozat témájához hasonló, Java bájtkód interpretert írjon az ember. Ha a class fájlok egyike tartalmazza ezt az instrukciót, akkor a program jelez a felhasználó számára. Akaratlanul is része lehet a programunknak ez az instrukció, amikor egy változót szöveggel együtt próbálunk kiírni:

```
1 public class InvokeDynamic {
2     public static void main(String[] args) {
3         String world = "world";
4         System.out.println("Hello " + world);
5     }
6 }
```

2.1. forráskód. Egyszerű példa invokedynamic utasítást tartalmazó Java kódra akkor a legtöbb fordítóprogram egy invokedynamic utasítást is elhelyez a programunkban.

Ez viszont elkerülhető, ha megfelelő flagekkel fordítjuk le a programunkat, mégpedig a `-XDstringConcat=inline` flag használatával az invokedynamic nem fog szerepelni a string konkatenációnál.

2.2. Fordítástól futásig

2.2.1. Minimum követelmények

A program fordításához legalább a Java 17-es verziója szükséges. Ez alatt a program fordulni sem képes, mivel pár olyan funkciót használ, amely csak a 17-es verzióban lett bevezetve.

A könnyebb fordítás (illetve egyszerűbb jar fájl készítés) érdekében a Maven fordítás automatizálási program telepítése ajánlott, ezen belül is a 3.9.0-ás verzió.

2.2.2. Fordítás

Ha nem akarunk Maven-t használni, akkor a fordítás menete a következő:

- Menjünk a `src/main/java` mappába (a `$` jel arra utal, hogy normál felhasználóként futtassuk a parancsot):


```
1 $ cd src/main/java/
```

- Fordítsuk le a `com/zoltanbalazs/Main.java` fájlt:

```
1 $ javac com/zoltanbalazs/Main.java
```

- Az elkészült class fájl a `src/main/java/com/zoltanbalazs` mappában lesz
- Maven-t használva ez a procedúra egyszerűbb:
- Futtassuk le a csomagoló parancsot:

```
1 $ mvn package
```

- Az elkészült class fájl a `target/classes/com/zoltanbalazs` mappában lesz, ezen felül a `target` mappában lesz egy futtatható jar fájl is

2.2.3. Futtatás

Ha a generált class fájlal akarjuk futtatni a programot, futtassuk le a `java com.zoltanbalazs.Main` parancsot a `src/main/java` mappában. (ha Maven-nel fordítottunk akkor a `target/classes` mappában futtassuk le az előző fenti parancsot)

A maven által készített jar fájlal való futtatáshoz, futtassuk le a `java -jar target/jabyinja-1.0.0.jar` parancsot a főmappában.

Mindkét esetben egy opcionális argumentumot (argumentum sorozatot ha a futtatandó programunk vár parancssori argumentumot) meg tudunk adni, ez a `main` metódust tartalmazó class fájl elérési útvonala. Alapvető esetben a program a futási mappában próbál meg egy `Main.class` fájlt futtatni.

Futásra egy példa:

```
1 $ java -jar target/jabyinja-1.0.0.jar  
↪ target/test-classes/com/zoltanbalazs/PTI/_01/Greet.class World
```

2.2.4. Önfuttatás

Az elkészült interpreter képes saját magát is futtatni, ehhez a futtatáshoz hasonlóan meg kell adni a programnak a saját class fájljának elérési útját, majd opcionálisan a többi paramétert.

Ez a futtatás jar fájl esetén így néz ki, a főkönyvtárból futtatva:

```
1 $ java -jar target/jabyinja-1.0.0.jar  
   ↪ target/classes/com/zoltanbalazs/Main.class  
   ↪ target/test-classes/com/zoltanbalazs/PTI/_01/Greet.class World
```

2.3. Felmerülő problémák

A futtatandó program futása során nem merül fel probléma (hacsak nincsen `invokedynamic` a generált class fájlban) amelyet a program okoz. Ha a futtatandó programunk hibát dob, akkor ezt az interpretáló program is ugyanúgy megteszi; viszont a hiba kiírása során nem biztos hogy ugyanazt a kimentet kapjuk mint a beépített interpreterrel.

Tehát ha a hibánk nem egy `try`, `catch` blokk-ban szerepel, akkor a kiírt üzenet nem biztos hogy ugyanaz lesz mint a beépített interpreterrel, az összes többi kiírt üzenet viszont ugyanaz kell hogy legyen.

3. fejezet

Fejlesztői dokumentáció

A program számos segítséget ad fejlesztőknek a kód egyszerű értelmezésére; a kód bővítésére. Vállalkozó szellemű embereknek a program egy teljes Java interpretert tud nyújtani viszonylag kevés továbbfejlesztéssel. A Java bájt kód instrukciók megértése során bármely Java fejlesztő jobban tudja értékelni hogy mit is csinál a Java fordítóprogram a háttérben. A program nagyon szorosan épül a JVM specifikációra, a specifikációt annak megfelelően próbálja implementálni.

3.1. Class fájl felépítése

3.1.1. Class fájl a benne levő metódus futtatásáig

A fő osztály a `ClassFile`, ez felel számos dologért, többek között egy class fájl beolvasásért, a megfelelő adattagok beállításával. A `ClassFile` osztálynak egy konstruktora van, mégpedig:

```
1 public ClassFile(String fileName, String[] mainArgs)
```

Tehát az első paraméter a beolvasandó class fájl neve, a második pedig a `main` metódusnak adott argumentumok.

Az implementáció alapján nem kötött a `main` metódus használata belépési pontként, tehát a 2. argumentum lehet `null` is.

A konstruktor meghívása egyidejűleg meghívja a `readClassFile` függvényt is:

```
1 public void readClassFile(String fileName)
```

Ez a függvény egy adott fájl névre beolvassa a class fájlban tárolt adatokat megfelelő változókba. (Ezen felül egy `VALID_CLASS_FILE` változót is beállít; feltételhezük hogy ha a mágikus szám (CA FE BA BE) megtalálható a fájl elején, akkor az adott fájl egy valid class fájl, ellenkező esetben egy `InvalidClassFileException`-t dob a beolvasó függvény.)

A beolvasás után (tehát az objektum létrehozása után) érdemes a belépési függvényt (általában `main`) megkeresni a `findMethodsByName` metódussal:

```
1 public Method_Info findMethodsByName(String methodName)
```

Ez egy adott függvény névre a megfelelő nevű metódust visszaadja a beolvasott fájlból (ha nem talál ilyet akkor `null`-t ad vissza). Egy példa a használatára:

```
1 ClassFile CLASS_FILE = new ClassFile("Main.class", null);  
2 Method_Info method = CLASS_FILE.findMethodsByName("main");
```

A függvény megtalálása után ajánlott a `code` attribútumot megtalálni, ebben, többek között, található a futtatandó Java bájtkód is. A segédfüggvény erre a `findAttributesByName`:

```
1 public List<Attribute_Info>  
    ↪ findAttributesByName(List<Attribute_Info> attributes, String  
    ↪ attributeName)
```

Mivel egy attribútumból több is lehet, egy listát kapunk vissza (a `code`-ból csak egy lesz), bemeneti paraméterként az attribútumnév mellett a megfelelő függvény attribútumait is át kell adnuk, például:

```
1 List<Attribute_Info> attributes =  
    ↪ CLASS_FILE.findAttributesByName(method.attributes, "Code");
```

(Ha nem talál ilyen nevezetű attribútumot akkor üres listát ad vissza.)

A megfelelően beolvasott attribútum után, a megtalált attribútumok között ajánlott végigmenni, a `List` implementálja az `Iterable`-t, így egy `for` ciklussal elegánsan megtehetjük ezt:

```
1 for (Attribute_Info attribute : attributes)
```

Mivel `Code` attribútumokról beszélünk, ezért a következő ajánlott dolog hogy ebből az attribútumból olvassuk be az adatokat. Ehhez a `Code_Attribute_Helper` osztály `readCodeAttributes` metódusa megfelelő:

```
1 public static Code_Attribute readCodeAttributes(Attribute_Info  
   ↪ attribute) throws IOException
```

A függvény egy attribútumot vár (például az előbbi kódrészlet `attribute` változóját), majd pedig beolvassa a specifikációnak megfelelően a `Code_Attribute`-ot, és visszaadja azt, ha valamiért nem sikerült a beolvasás akkor `IOException`-t dob a függvény.

```
1 Code_Attribute codeAttribute =  
   ↪ Code_Attribute_Helper.readCodeAttributes(attribute);
```

Ezt a beolvasott attribútumot a `ClassFile` osztály fel tudja használni az `executeCode` metódusával, mely egy `byte[]` változót vár bemeneti paraméterként, ami a `Code_Attribute` része:

```
1 public Pair<Class<?>, Object> executeCode(byte[] code) Throwable
```

A reflexió miatt számos hibát dob vissza a függvény, ha nem helyes a kód formátuma akkor `IOException`-t dob a függvény, a `Throwable` az `ATHROW` Java bájtkód instrukció miatt szükséges (ekkor egy hibát dob vissza a metódusunk). Visszatérési értéke `Pair<Class<?>, Object>`, a számos `RETURN` utasítás miatt (ezeket a stack-en szükséges elhelyezünk) Példa a használatára:

```
1 CLASS_FILE.executeCode(codeAttribute.code);
```

Ezzel el is jutottunk egy class fájl beolvasásától, az abban lévő adott függvény Java bájtkódjának futtatásáig, több teendőnk nincsen, a program az adott függvényben levő külön függvényhívásokat automatikusan elvégzi.

A teljes példakód:

```

1 public class Interpreter {
2     public static void main(String[] args) throws Throwable {
3         ClassFile CLASS_FILE = new ClassFile("Main.class", null);
4         Method_Info method = CLASS_FILE.findMethodsByName("main");
5         List<Attribute_Info> attributes = CLASS_FILE.
            findAttributesByName(method.attributes, "Code");
6
7         for (Attribute_Info attribute : attributes) {
8             Code_Attribute codeAttribute = Code_Attribute_Helper.
                readCodeAttributes(attribute);
9             CLASS_FILE.executeCode(codeAttribute.code);
10        }
11    }
12 }

```

3.1. forráskód. Egyszerű példakód a Main.class interpretálására

3.1.2. Pár minta class fájl felépítése

A legegyszerűbb class fájl ami értelmes, viszont nem futatható:

```

CA FE BA BE 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00

```

Java kódban ennek megfelelője az üres fájl:

```

1

```

Class fájl formátumának magyarázata:

- CA FE BA BE: Mágikus szám, amely minden Class fájl elején megtalálható
- 00 00 00 00: Class fájl Minor és Major verziószáma, egy táblázatnak megfelelően a fordítóprogram verziója
- 00 00: A Constant Pool mérete (+1, mivel 1-től indexelt, itt nem számít)
- 00 00: Hozzáférési zászlók ()
- 00 00: This osztály indexe a Constant Pool-ban
- 00 00: Super osztály indexe a Constant Pool-ban

- 00 00: Interfészek száma
- 00 00: Adattagok száma
- 00 00: Függvények száma
- 00 00: Osztály attribútumainak száma

A legegyszerűbb class fájl amit a *Jabyinja* program le tud futtatni (a beépített java program nem képes ezt lefuttatni, mivel nincsenek benne osztályok, a JVM specifikáció alapján az osztályok elhanyagolhatóak):

```
CA FE BA BE 00 00 00 00 00 04 01 00 04 43 6F
64 65 01 00 04 6D 61 69 6E 01 00 03 28 29 56
00 21 00 00 00 00 00 00 00 00 00 01 00 09 00
02 00 03 00 01 00 01 00 00 00 0D 00 00 00 00
00 00 00 01 B1 00 00 00 00 00 00
```

Java kód megfelelője:

```
1 public static void main() {
2     return;
3 }
```

Class fájl formátumának magyarázata:

- CA FE BA BE: Mágikus szám, amely minden class fájl elején megtalálható
- 00 00 00 00: Class fájl Minor és Major verziószáma, egy táblázatnak megfelelően a javac fordítóprogram verziója
- 00 04: A Constant Pool mérete (+1, mivel 1-től indexelt)
- 01 00 04 43 6F 64 65 01 00 04 6D 61 69 6E 01 00 03 28 29 56: Constant Pool
 - 01 00 04 43 6F 64 65
 - 01: Constant Pool Info érték (CONSTANT_Utf8)
 - 00 04: 4 hosszú
 - 43 6F 64 65: A CONSTANT_Utf8 értéke: Code
 - 01 00 04 6D 61 69 6E
 - 01: Constant Pool Info érték (CONSTANT_Utf8)
 - 00 04: 4 hosszú
 - 6D 61 69 6E: A CONSTANT_Utf8 értéke: main
 - 01 00 03 28 29 56
 - 01: Constant Pool Info érték (CONSTANT_Utf8)
 - 00 03: 3 hosszú
 - 28 29 56: A CONSTANT_Utf8 értéke: ()V

- 00 21: Hozzáférési zászlók (Public, Super) - elhanyagolhatóak ebben az esetben
- 00 00: This osztály indexe a Constant Pool-ban
- 00 00: Super osztály indexe a Constant Pool-ban
- 00 00: Interfészek száma
- 00 00: Adattagok száma
- 00 01: Függvények száma
- 00 09 00 02 00 03 00 01 00 01 00 00 00 0D 00 00 00 00 00 00 01 B1 00 00 00 00: Függvények
 - 00 09 00 02 00 03 00 01 00 01 00 00 00 0D 00 00 00 00 00 00 01 B1 00 00 00 00
- 00 09: Hozzáférési zászlók (Public, Static)
- 00 02: Constant Poolban lévő indexe a függvénynek: main
- 00 03: Függvény leírása (bemeneti paraméterek, visszatérési érték): ()V
- 00 01: Függvény attribútumainak száma
- 00 01 00 00 00 0D 00 00 00 00 00 00 00 01 B1 00 00 00 00: Attribútumok
 - 00 01 00 00 00 0D 00 00 00 00 00 00 00 01 B1 00 00 00 00
 - 00 01: Constant Pool-ban lévő indexe az attribútumnak: code
 - 00 00 00 0D: Attribútum hossza (0D = 13 bájt)
 - 00 00 00 00 00 00 00 01 B1 00 00 00 00: Attribútum
 - 00 00 00 00 00 00 01 B1 00 00 00 00
 - 00 00: Stack mérete
 - 00 00: Lokális változók száma
 - 00 00 00 01: Kód hossza
 - B1: Kód (B1 = return)
 - 00 00: Kivételek száma
 - 00 00: Attribútum attribútumainak száma
- 00 00: Osztály attribútumainak száma

3.1.3. Adatszerkezetek

A class fájlban megfelelően a két legfontosabb adattag a `stack` és a `local` (lokális) változók. A különböző instrukciók az ezeken lévő adatokkal dolgoznak, erre/ebbe helyeznek el megfelelő adatokat.

Az egyszerűség kedvéért a `stack` reprezentációjában az osztály típusát is elmentjük, a két adattag Java reprezentációja a `ClassFile` osztályban:


```
1 public List<Pair<Class<?>, Object>> stack = new ArrayList<>();  
2 public Object[] local = new Object[65536];
```

(A `Pair` egy egyedi osztály, mely két adattagot tud eltárolni, más nyelvekben `tuple`-ként is ismeretes.)

A lokális változók maximális mennyiségét előre tudjuk, ez nem lehet több mint egy 16-bites előjel nélküli szám ($2^{16} = 65536$), alából ennek az értéke egy 8-bites előjel nélküli szám ($2^8 = 256$) lenne, mivel a `store` és `load` utasításokat csak egy 8-bites előjel nélküli szám (az `index`) követi, viszont a `wide` utasítással a `store` és `load` utasítások módosíthatóak, hogy 2 db 8-bites előjel nélküli számot olvassanak be, tehát lényegében egy 16-bites előjel nélküli számot.

Gyakorlatban ez a szám csökkenthető lenne, tudhatjuk hogy futási időben mennyi lokális változója (illetve a `stack` nagyságát is tudhatjuk, tehát tömbként is reprezentálhatnánk) van egy metódusnak. Ez bővebben le van írva a továbbfejlesztési lehetőségekben.

Kényelmi szempontból létezik a `CodeIndex` osztály, amely lényegében egy `int` szám absztrakciója:

```
1 class CodeIndex {  
2     private int index = 0;  
3  
4     public void Inc(int value) {  
5         index += value;  
6     }  
7  
8     public int Next() {  
9         return index++;  
10    }  
11  
12    public void Set(int value) {  
13        index = value;  
14    }  
15  
16    public int Get() {  
17        return index;  
18    }  
19 }
```

3.2. forráskód. A Codeindex osztály, amely a kód bájtömb jelenlegi indexét tárolja

Az absztrakció oka hogy függvényeknek átadva lehessen módosítani ezt a számot; a szám a jelenlegi index a kódot reprezentáló `byte` tömbben, megmondja hogy a tömbben lévő melyik indexen levő instrukciót kell végrehajtani.

Az absztrakció különösen észrevehető amikor az `if` és `goto` utasításokat hajtjuk végre, a `ClassFile` objektumunk lokális változója módosítható az `Instructions` osztály metódusain keresztül. Mivel a Java érték szerint adja át a paramétereket, ez egy sima `int` számmal nem lehetne megoldani.

3.1.4. Interpretálás algoritmusa

Az 1. algoritmus egy általános elágazás és korlátozás algoritmust (*Branch and Bound algorithm*) mutat be. A 3. lépésben egy megfelelő kiválasztási szabályt kell alkalmazni. Példa forrása: Acta Cybernetica (ez egy hiperlink).

1. algoritmus A general interval B&B algorithm

Funct $IBB(S, f)$

```

1: Set the working list  $\mathcal{L}_W := \{S\}$  and the final list  $\mathcal{L}_Q := \{\}$ 
2: while (  $\mathcal{L}_W \neq \emptyset$  ) do
3:   Select an interval  $X$  from  $\mathcal{L}_W$                                 ▷ Selection rule
4:   Compute  $lb f(X)$                                               ▷ Bounding rule
5:   if  $X$  cannot be eliminated then                                ▷ Elimination rule
6:     Divide  $X$  into  $X^j$ ,  $j = 1, \dots, p$ , subintervals          ▷ Division rule
7:     for  $j = 1, \dots, p$  do
8:       if  $X^j$  satisfies the termination criterion then          ▷ Termination rule
9:         Store  $X^j$  in  $\mathcal{L}_W$ 
10:      else
11:        Store  $X^j$  in  $\mathcal{L}_W$ 
12:      end if
13:    end for
14:  end if
15: end while
16: return  $\mathcal{L}_Q$ 

```

3.1.5. Interpretálás működése

Az algoritmus alapján az interpretálás viszonylag egyszerűen működik.

1. Olvassuk be a jelenlegi indexen lévő utasítást

2. A specifikáció alapján olvassuk be a megfelelő darabszámú extra paramétert
3. Végezzük el az utasításnak megfelelő műveletet; módosítsuk a lokális változókat és a stack-et
4. Ismételjük az 1. pontot amíg nem vagyunk a kódot tartalmazó bájtömb végén

Természetesen valóságban egy kicsit komplikáltabb ennél a működés (bár nem sokkal). Az utasítások nagy része ténylegesen leírható ezzel az egyszerű 4 lépéses "algoritmussal". A komplikáltabb utasítások viszont a `invoke*`, `put*`, `get`. Ezeknél a stack-nek megfelelően kell lekérnünk az adatokat, majd a Java nyelv sajátossága miatt (illetve a reflexió használata miatt) le kell kérnünk a megfelelő metódust, és a paraméterekkel együtt meghívni / interpretálni a függvényt az interpreterrel.

A fő interpretálást végrehajtó kód a `ClassFile` osztályban az `executeCode` metódus.

```
1 public Pair<Class<?>, Object> executeCode(Code_Attribute attribute)
   throws Throwable {
2     byte[] code = attribute.code;
3
4     CodeIndex codeIndex = new CodeIndex();
5     while (codeIndex.Get() < code.length) {
6         byte opCode = code[codeIndex.Next()];
7
8         switch (Opcode.opcodeRepresentation(opCode)) {
9             ...
10        }
11    }
12    throw new Throwable("Code did not contain a return statement");
13 }
```

3.3. forráskód. Az interpretálásért felelős kódrészlet

3.2. Az interpreter sajátosságai

3.2.1. Erőforrás igények

A Linux operációs rendszeren beépített `time` programot (illetve a `hyperfine` programot) használva az erőforrás igények a tesztfájlokra az alábbiak (a tesztelt számítógép releváns specifikációi: Intel Core i7-8700k processzor 4.7 GHz-en, 16 GB DDR4 memória 2133 MT/s sebességgel):

Tesztfájl	/usr/bin/java		Jabyinja	
	Memória	Futási idő	Memória	Futási idő
Own/Arithmetic.class	37,1 MB	21,4 ms	47,6 MB	92,8 ms
Own/Arrayclass.class	34,9 MB	20,9 ms	54,6 MB	130,8 ms
Own/Arraylist.class	37,2 MB	21,4 ms	49,6 MB	87,1 ms
Own/Athrow.class	34,6 MB	20,4 ms	46,9 MB	61,7 ms
Own/Dup2.class	36,3 MB	20,3 ms	39,2 MB	45,6 ms
Own/Inheritance.class	34,8 MB	20,7 ms	51,9 MB	98,1 ms
Own/Instanceof.class	38,8 MB	20,2 ms	43,1 MB	64,2 ms
Own/Multianewarray.class	34,4 MB	20,6 ms	47,7 MB	62,9 ms
Own/Nested.class	39,3 MB	22,1 ms	47,2 MB	80,5 ms
Own/Ownclass.class	37,5 MB	20,2 ms	61,9 MB	135,5 ms
Own/SwitchAthrow.class	36,8 MB	20,5 ms	40,5 MB	43,9 ms
Own/Template.class	39,2 MB	21,2 ms	51,9 MB	86,5 ms
PTI/_01/Euler.class	39,5 MB	22,1 ms	47,5 MB	66,4 ms
PTI/_01/Factorial.class	39,6 MB	21,8 ms	50,7 MB	68,4 ms
PTI/_01/GCD.class	38,9 MB	20,6 ms	51,6 MB	67,8 ms
PTI/_01/Greet.class	38,8 MB	20,8 ms	50,8 MB	56,8 ms
PTI/_01/Half.class	39,6 MB	22,7 ms	50,7 MB	69,7 ms
PTI/_01/Odd.class	38,5 MB	20,7 ms	37,8 MB	69,7 ms
PTI/_01/PerfectNum.class	38,8 MB	20,7 ms	51,2 MB	57,1 ms
PTI/_01/PerfectNumRange.class	37,8 MB	20,6 ms	70,6 MB	87,6 ms
PTI/_01/Print.class	41,1 MB	21,4 ms	51,2 MB	67,2 ms
PTI/_01/Sqrt.class	35,7 MB	22,3 ms	50,9 MB	70,3 ms
PTI/_01/SquareRoot.class	37,6 MB	22,4 ms	43,1 MB	71,5 ms
PTI/_01/TwoNum.class	35,1 MB	20,6 ms	47,7 MB	75,8 ms
PTI/_02/_01/PointMain.class	41,5 MB	22,3 ms	55,6 MB	122,7 ms
PTI/_02/_02/CircleMain.class	41,5 MB	21,3 ms	53,1 MB	82,5 ms
PTI/_02/_03/CircleMain.class	41,3 MB	22,7 ms	55,1 MB	93,7 ms
PTI/_02/_04/ComplexMain.class	41,5 MB	21,2 ms	58,4 MB	125,7 ms
PTI/_02/_05/LineMain.class	41,4 MB	20,7 ms	56,7 MB	102,5 ms
PTI/_03/IterletterMain.class	41,7 MB	20,8 ms	78,9 MB	180,5 ms
PTI/_04/_01/PointMain.class	39,6 MB	22,7 ms	58,8 MB	126,7 ms

Tesztfájl	/usr/bin/java		Jabyinja	
	Memória	Futási idő	Memória	Futási idő
PTI/_04/_02/DoubleVectorMain.class	41,3 MB	21,1 ms	68,5 MB	176,2 ms
PTI/_05/_01/Swap.class	41,2 MB	20,1 ms	51,9 MB	60,5 ms
PTI/_05/_02/IntegerMatrixMain.class	41,2 MB	20,7 ms	46,8 MB	70,1 ms
PTI/_05/_03/WildAnimalMain.class	41,0 MB	21,2 ms	73,4 MB	183,5 ms
PTI/_05/_04/IntVectorMain.class	43,2 MB	21,0 ms	53,9 MB	93,9 ms
PTI/_06/_01/Calculator.class	39,1 MB	21,1 ms	51,6 MB	64,1 ms
PTI/_06/_02/AddByLine.class	41,2 MB	21,6 ms	53,0 MB	92,3 ms
PTI/_06/_03/IsPartOf.class	39,3 MB	22,4 ms	51,8 MB	73,9 ms
PTI/_06/_04/CircleMain.class	41,3 MB	22,0 ms	64,7 MB	124,8 ms
PTI/_08/_01/BookMain.class	41,4 MB	21,4 ms	74,3 MB	197,6 ms
PTI/_08/_02/CoffeeShop.class	41,1 MB	21,2 ms	60,8 MB	123,7 ms
PTI/_09/_01/Divisors.class	41,2 MB	20,2 ms	52,2 MB	85,4 ms
PTI/_09/_04/MultiSetMain.class	41,4 MB	21,6 ms	67,7 MB	175,4 ms
PTI/_10/_01/Extends.class	41,4 MB	20,6 ms	52,6 MB	82,4 ms
PTI/_10/_02/BookMain.class	41,1 MB	22,3 ms	85,0 MB	254,5 ms
PTI/_10/_03/BagMain.class	41,4 MB	21,4 ms	72,7 MB	175,2 ms
PTI/_10/_04/Swap.class	41,6 MB	20,3 ms	53,6 MB	101,0 ms
PTI/_11/_01/FlyingMain.class	41,4 MB	21,5 ms	54,3 MB	92,0 ms
PTI/_11/_02/Main.class	41,3 MB	20,8 ms	71,6 MB	153,7 ms
PTI/_11/_03/AnimalMain.class	41,6 MB	20,5 ms	52,1 MB	78,3 ms
PTI/_11/_04/AnimalMain.class	41,3 MB	21,7 ms	64,5 MB	152,3 ms
PTI/_12/Inheritance.class	41,6 MB	20,2 ms	47,7 MB	68,3 ms

3.1. táblázat. A beépített java és az interpreter közötti erőforrás különbségek

3.2.2. A program memóriamodellje

A heap memória nincsen implementálva a programban. Ez alapszik a standard Java interpreter implementációjára. Ide tartoznak többek között az objektumoknak a referenciái. Ebből következően a szemétgyűjtés (*garbage collector*) a beépített Java szemétgyűjtő algoritmust alkalmazza.

A stack memória fontos építőeleme a programnak, az implementációja `ArrayList` osztállyal van megoldva.

```
1 public List<Pair<Class<?>, Object>> stack = new ArrayList<>();
```

Ez az `ArrayList` egy `Class<?>`-t és `Object`-et tartalmazó `Pair`-eket (tuple) foglal magába. Az `Object` a konkrét érték amit a stack-en tárolni szeretnénk, a `Class<?>` egy kényelmi megoldás miatt a tárolt értéknek a típusa.

Az implementációból következően lényegében a tényleges program-ban az alábbi memóriaeloszlás következik be:

3.2.3. Az önfuttatásról

Habár a program tényleg képes önmagát lefuttatni, a Java programozási nyelvet mélyebben értő emberek észrevehetik, hogy ez egy erősen kikötéses állítás. Mivel a beépített osztályokat nem interpretálja a program, ezért amikor egy olyan parancsot hívunk meg, mint például:

```
1 $ java -jar target/jabyinja-1.0.0.jar
   ↪ target/classes/com/zoltanbalazs/Main.class
   ↪ target/test-classes/com/zoltanbalazs/PTI/_01/Greet.class World
```

akkor igazából csak az interpreter `main` metódusáig interpretáljuk azt.

A java beépített interpreter működése miatt amikor egy `jar` fájlt futtatunk, minden, a `jar` fájlban lévő, osztály beépített lesz. Az eddig leírtak alapján pedig a beépített osztályok metódusai pedig reflexióval vannak meghívva.

Természetesen a `main` függvényben is megfelelően kell felépíteni a különböző adat-tagokat, tehát az interpreter tényleg saját magát futtatja, viszont nem látunk akkora erőforrásbeli különbséget mint ami a beépített interpretert és a megírt program között van.

3.3. Tesztelő környezet

A tesztelő környezet egy átlagos Java tesztelő (pl. JUnit) helyett egy saját Python szkript. Az előnye ennek az, hogy a tesztelésnél többet is tud ez a szkript; például

a memóriahasználatot, és a futási idő különbséget is mérni. A tesztelő szkript a `src/test/tester.py` fájlban elérhető. A szkript futtatásához legalább Python 3-as verzió szükséges. A tesztelő működésileg megnézi hogy a beépített Java interpreter, és a megírt interpreter kimenetei megegyeznek-e, ha igen akkor helyesen futott le az interpret, ha nem, akkor helytelenül. Jelen esetben 3 teszteset van, amelyek nem mennek át a tesztelőn, ezek mindegyike tartalmaz `invokedynamic` utasítást.

3.3.1. Tesztelés

A tesztelőt nagyon egyszerűen meg lehet hívni a

```
1 $ python3 src/test/tester.py
```

paranccsal

Fontos: Ha mindegyik teszteset megbukik akkor nincsen fordított állomány! A tesztelő feltételezi hogy jar fájlként van fordítva a program, amely a `target` mappában van elhelyezve.

3.3.2. Saját teszteset hozzáadása

Nagyon egyszerűen adható saját teszteset hozzá a szkripthez.

1. Helyezzük el a tesztesetet a `src/test/java/zoltanbalazs/Own` mappába
2. Nyissuk meg a tesztelő szkriptet
3. Adjuk hozzá a megfelelő `own_` névvel kezdődő listához a saját tesztesetünket
4. Ha van, határozzuk meg a bemeneti argumentum(ok) és/vagy standard bemenetről kapott értékek mennyiségét és típusát
5. A következő futtatásnál az új teszteset le fog futni

3.4. Továbbfejlesztési lehetőségek

3.4.1. Invokedynamic utasítás

Az egyik legszembetűnőbb hiány a szakdolgozatban az egyik nem implementált utasítás, az `invokedynamic`, hiánya. Ez az utasítás számos helyen előfordul Java

programokban, leginkább a lambda kifejezésekben (ezen belül is a Konkurens programozás tárgyon megismert `Executor` osztály paramétereiként), illetve a kiírás során szöveg(ek) és változó(k) konkatenációjánál is ez használt.

Az utóbbi egyszerűen kiküszöbölhető a `-XDstringConcat=inline` flag-gel való fordítás során az `invokedynamic` utasítás lecserélődik `StringBuilder`-en keresztül lévő `invokevirtual` és `invokespecial` hívásokra.

Az előző viszont sajnos jelen állapotban nem megoldott, és nem is oldható meg egyszerűen. Ahhoz hogy lambda függvények működjenek, az `invokedynamic`-ot implementálni kell. Ehhez már az alapvető előkészület megvan, a class fájlban lévő `bootstrap` metódusok egy külön adattag elemeiként el vannak helyezve. A továbbfejlesztés során csak a megfelelő `CallSite` helyet, illetve a class fájlban lévő `constant pool` általi `index`-eken levő metódusokkal (illetve paraméterekkel) kell meghívni az éppen leírt függvényt.

3.4.2. Java 7 előtti verziók támogatása

Viszonylag egyszerűen továbbfejleszthető a program hogy Java 7 előtti verzióval fordított class fájlokat is támogasson.

A hiányzó utasítások a `ret`, `jsr`, `jsr_w`, ezek mindegyikéhez csak az szükséges, hogy a megfelelő index-re ugorjunk, a `jsr` és `jsr_w` utasítások során a visszatérési címet pedig a `stack`-re helyezzük.

Természetesen mindegyik utasítás során a megfelelő index-et is be kell olvasnunk a class fájlból, amely a lokális változó megfelelő indexére (`ret`), vagy egy adott számot (`jsr`, `jsr_w`) határoz meg, amely a visszatérési cím, illetve az ugrási cím.

3.4.3. Optimalizálás

A futási idő táblázata alapján látható hogy a program exponenciálisan lassabb, mint a beépített `java` interpretáló program. A program több memóriát is igényel mint szükséges lenne. Ezeknek számos oka is van, ezek közül pár:

- A nem beépített osztályok megfelelő konstruktorait minden egyes alkalommal a program egyesével keresi ki a program. Ez a limitáció nagyon szembetűnő ha sok saját osztállyal dolgozunk. Ilyenkor a futási idő exponenciálisan lassul. Egy lehetséges megoldás erre hogy a megfelelő konstruktorokat elmenti a program, majd keresés előtt az elmentett konstruktorok között megnézzük hogy szerepel-

e már a jelenleg hívandó konstruktor. Ha igen, akkor egyértelmű módon nem keressük ki, hanem felhasználjuk azt, ha nem, akkor pedig megkeressük.

- A `local` változóknak maximális értéket (65536) foglal a program minden nem beépített függvény meghívása során, viszont a class fájlban ennek a maximális értéke le van írva a megfelelő függvény attribútumaként. Tehát a megoldás erre viszonylag egyszerű, függvényfuttatás előtt kellene a lokális változók mennyiségét beállítani. Ennek a problémának a másik verziója, hogy habár a stack egy LIFO (*Last In First Out*, amely adat utoljára kerül bele, az kerül elsőnek ki belőle) adatszerkezet, a maximális mérete ennek is a class fájlban a függvény egy megfelelő attribútumaként jelen van. A már megírt függvényekkel viszonylag triviális ezt lekérdezni. (A megfelelő függvény a `ClassFile` osztályban található `findAttributesByName`)
- A különböző class fájlok beolvasásának eredménye nincs elmentve, ha egy fájlt be kell olvasnunk, akkor azt minden egyes alkalommal külön-külön megteszünk, ha az eredményt elmentenénk akkor drasztikusan lehetne a sebességen gyorsítani. Ennek megoldásaként a `Main` osztályban egy listában fel lehetne venni a beolvasott class fájlokat, egy nem beépített class fájlban levő metódus interpretálása előtt pedig végig lehetne menni a már beolvasott fájlokon. Ezzel jóval kevesebb fájl beolvasás műveletet végeznénk, cserébe a listán való végigmenés lehet hogy kevés class fájl használata esetén negatív hatással lenne.
- A refleksiót újragondolva, azt elhagyva a sebesség növelhető lenne, erről részletesebben a "Refleksió újragondolása" szekcióban van írva. Lényege, hogy a jelenlegi, beépített osztályokon levő refleksiót le kellene cserélni interpretálása. Az alapok ehhez megvannak, viszont ez is egy komolyabb programozási feladat.

3.4.4. További tesztelés

A szakdolgozat írása során megpróbáltam az alapos tesztelésre figyelni, ezért is vannak az alapvető instrukciót egyesével tesztelve (minden tesztfájl-ban külön-külön instrukciók szerepelnek). Viszont a tökéletes program nem létezik, elképzelhető hogy valahol nincs megfelelően a `stack` törölve, vagy valamely instrukció mégsem helyes. Tehát a programban elképzelhető a probléma. Ezt a még alaposabb teszteléssel minél

inkább meg lehetne cáfolni.

Ehhez egy példa még több tesztfájl mellékelése. A tesztelő környezetbe (Python szkript) viszonylag egyszerűen be lehet helyezni új teszt fájlokat, amely leellenőrzi hogy megfelelő-e a program futása. Továbbfejlesztésként lehet Java programokat írni, majd ezeket a tesztelő környezethez hozzáadni, és ellenőrizni hogy jól fut-e le a program.

Egy másik érdekes továbbfejlesztés, hogy ne csak Java programokat írjunk, hanem más JVM-et felhasználó nyelvekben is írjunk programokat. Ezekkel ellenőrizhetjük hogy a program tényleg képes-e általános Java bájtódot interpretálni, vagy csak limitált a Java programozási nyelvre. A témabejelentés miatt én az ezekkel a nyelvekkel való tesztelést kihagytam. Egy egyszerű "Hello, World!"-el való tesztelés során (Clojure/Kotlin/Scala nyelven) a program hibát fog dobni, ennek az indoka hogy a programban levő reflexió, mivel a program Java nyelven van írva, nem találja a Clojure/Kotlin/Scala függvényeket, ehelyett érdemes lenne minden egyes fájlt interpretálni, nem csak a nem beépített osztályok fájlait. Természetesen ez megoldható ha a megfelelő nyelv standard könyvtárát be tudjuk másolni a `cp` parancssori zászlóval.

3.4.5. Reflexió újragondolása

A reflexió egy viszonylag negatív hírrel rendelkező programozási folyamat, melynek számos indoka van, legtöbbször a fő indok az, hogy a reflexióval megoldott problémát valamilyen elegánsabb módon is meg lehetne oldani.

Sajnos a szakdolgozat programjában is észrevehető hogy a reflexió nem annyira tökéletes megoldás erre a problémára, lehetne elegánsabb megoldást is találni. A reflexió túlságos használata miatt a teljesítmény is rosszabb, egy komoly optimalizálási feladat a programban lévő reflexiót lecserélni.

A másik komoly probléma a szakdolgozatban a reflexió használatával, az előbb említett képtelenség arra, hogy Java nyelven kívüli programokat futtatni tudjon. Habár a program készen áll arra, hogy ezeket futtatni tudja, a jelenlegi állapotában nem lehet. Ha minden egyes függvényt interpretálnánk, nem hagynánk a reflexióra, akkor egy univerzális interpretert lehetne írni, tehát egy Java nyelven íródott program képes lenne Clojure/Kotlin/Scala fordítóprogramok által generált class fájlokat interpretálni, lefuttatni.

3.5. Érdekességek a JVM specifikációból

A specifikációt olvasva számos érdekességre bukkanhat az ember, ezek között vannak tervezési anomáliák, jó megoldások és trükkök. Ezek közül pár:

- A Java 7-es verzió előtt a Java bájt kód instrukciók viszonylag egyszerűek voltak, a 7-es verzióval lett az `invokedynamic` bevezetve, mely közelebbi ránézésre igazából egy "superinstrukció", ezzel az összes többi `invoke` instrukció implementálható, sőt, egy teljes Java bájt kód interpret is megírható csak `invokedynamic` utasításokkal. Az `invokedynamic` utasítás a dinamikuság kérdésére volt a válasz. Lényege, hogy azon függvényhívásokat, amelyek paramétereit fordítási időben nem tudjuk, egy hívási helynek (`CallSite`) megfelelően kerülnek a JVM-be bele, amely az a hívás során a megfelelő paraméterekkel hívja meg.
- Elméletileg a lokális változók száma nem kéne hogy 65536 legyen. Ha megnézi az ember, akkor minden `store` és `load` utasítás egy darab 8 bites előjel-mentes számot olvas be paraméterként. Ez azt jelenti hogy összesen $2^8 = 256$ lehetséges index kellene hogy legyen. A `wide` módosító utasítás miatt viszont ezek az utasítások beolvashatnak két darab 8 bites előjel-mentes számot, tehát lényegében egy darab 16 bites előjel-mentes szám lesz a paraméterük. Ez azt jelenti hogy 256 lehetőség hirtelen $2^{16} = 65536$ -ra ugrik fel.
- Minden `Long` és `Double` típusú adattag két (egymást követő) helyet foglal el a class fájlban belüli `Constant Pool`-ban. Ennek oka hogy minden `Constant Pool` belüli elem tényleges felhasználható adata 4 byte-on van, kivéve a `Long` és `Double`, amelyek felhasználható adatai 8 byte-on vannak. Így a fájlban belüli folytonosság miatt a specifikáció készítői jobbnak látták hogy inkább két helyet foglaljon el. A specifikációban azóta is a *In retrospect, making 8-byte constants take two constant pool entries was a poor choice.* - Utólag visszagondolva, rossz döntés volt, hogy a 8 bájtos konstansok két konstans pool bejegyzést igényelnek. idézet áll.
- A `Long` és `Double` típusokhoz kapcsolódóan, egy hatalmas különbség a primitívek (`double`, `long`) és osztály (`java.lang.Double`, `java.lang.Long`) típusok között, hogy lokális változóként a primitívek (csak a `double` és `long`) két helyet foglalnak el a lokális változók között; míg az osztály típusok csak egy helyet. Ez azt jelenti, hogyha függvény paraméterként adunk át egy `double` primití-

vet, akkor valójában 2 helyet foglalunk el a lokális változók között, míg ha egy `java.lang.Double` értéket adunk át, csak egy 1 helyet fogunk elfoglalni.

4. fejezet

Összegzés

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In eu egestas mauris. Quisque nisl elit, varius in erat eu, dictum commodo lorem. Sed commodo libero et sem laoreet consectetur. Fusce ligula arcu, vestibulum et sodales vel, venenatis at velit. Aliquam erat volutpat. Proin condimentum accumsan velit id hendrerit. Cras egestas arcu quis felis placerat, ut sodales velit malesuada. Maecenas et turpis eu turpis placerat euismod. Maecenas a urna viverra, scelerisque nibh ut, malesuada ex.

Aliquam suscipit dignissim tempor. Praesent tortor libero, feugiat et tellus portitor, malesuada eleifend felis. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nullam eleifend imperdiet lorem, sit amet imperdiet metus pellentesque vitae. Donec nec ligula urna. Aliquam bibendum tempor diam, sed lacinia eros dapibus id. Donec sed vehicula turpis. Aliquam hendrerit sed nulla vitae convallis. Etiam libero quam, pharetra ac est nec, sodales placerat augue. Praesent eu consequat purus.

Köszönetnyilvánítás

Petes Márton (ELTE IK PTI BSc): Az elektadásaim során elképesztően sok segítséget nyújtott, nélküle nem tudom hogy meglett volna-e a szakdolgozat

Irodalomjegyzék

- [1] Gilad Bracha Tim Lindholm Frank Yellin és Alex Buckley. „The Java Virtual Machine Specification”. (2013. febr.). URL: <https://docs.oracle.com/javase/specs/jvms/se7/html/index.html>.

Táblázatok jegyzéke

3.1. Erőforrás különbségek	19
--------------------------------------	----

Forráskódjegyzék

2.1. Egyszerű példa invokedynamic utasítást tartalmazó Java kódra	6
3.1. Egyszerű példakód a Main.class interpretálására	12
3.2. A Codeindex osztály, amely a kód bájtömb jelenlegi indexét tárolja .	15
3.3. Az interpretálásért felelős kódrészlet	17