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

EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR

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**

1.1. Motiváció 3

1.2. Leírás 3

2. Felhasználói dokumentáció **5**

2.1. Probléma megfogalmazása 5

2.2. Megoldás rövid leírása 6

2.3. Program használata 6

2.3.1. Kikötések 6

2.3.2. Fordítástól futásig 7

2.3.3. Felmerülő problémák 9

3. Fejlesztői dokumentáció **10**

3.1. Probléma specifikációja 10

3.2. Megoldás lépései 11

3.2.1. Class fájl felépítése 11

3.2.2. Az interpreter sajátosságai 24

3.3. Program szerkezete 28

3.4. Tesztelésről 29

3.4.1. Tesztelés lefuttatása 30

3.4.2. Saját teszteset hozzáadása 30

3.4.3. Tesztelés eredményei 31

3.5. Továbbiak 32

3.5.1. Továbbfejlesztési lehetőségek 32

3.5.2. Érdekességek a JVM specifikációból 35

4. Összegzés **37**

Köszönetnyilvánítás **38**

Irodalomjegyzék **38**

Ábrajegyzék **40**

Táblázatjegyzék **41**

Forráskódjegyzék **42**

1. fejezet

Bevezetés

1.1. Motiváció

Kifejezetten érdekel a számítógép hardveréhez közeli problémák megértése és megoldása. C programozói háttérel rendelkezve az x86-64, illetve ARM assemblyben való programozástól sem rémülök meg. Viszont amennyire azt tudom hogy azon assembly nyelvek hogyan működnek (regiszterek, `stack`, szubrutinok, ciklusok, feltételek), annyira nem tudom hogy egy magasabb, Objektumorientált programozási nyelv (azon belül is egy fordított és interpretált nyelv) hogyan "kommunikál a géppel".

Míg ha egy C++ interpretert akarok írni akkor valójában egy fordítóprogramot kell írnom, addig Java nyelvnél elegendő a köztes nyelv interpretálását megvalósítani, a fordítást pedig a beépített fordítóprogram hagyni. A Java nyelv így egy tökéletes köztes állapot a két architektúra között.

1.2. Leírás

A Java nyelvben írt programok fordításukat követően nem egy közvetlen 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 (fantázianevén *Jabyinja*: ***Ja**va **by**tecode **in**terpreter in **Ja**va*) 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) implementálásval 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 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 reflekcióra, 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 verizóját) 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 a 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 (ami nem a szakdolgozat megoldása), amely képes Java programokat futtatni.

2.1. Probléma megfogalmazása

A program alapvető problémának egy class fájl értelmezését tűzi ki, a fordítás az előbb leírtakban nem a feladat része.

A feladat megoldása egy futtatható class fájlt (vagy jar fájlt, a felhasználó szükségeinek megfelelően) eredményez, amely képes más Java class fájlokat interpretálni, azokban lévő különböző függvényeket meghívni, a `stack`, illetve lokális változókat helyesen változtatni.

A megoldáshoz be kell olvasnia a programnak egy class fájlt, helyesen értelmezni hogy mely része mi is, majd a megfelelő részeket szekvenciálisan értelmezni, az

utasításokat a specifikációnak megfelelően végrehajtani.

2.2. Megoldás rövid leírása

A megoldás erősen alapszik a JVM specifikációra, annak is a 7-es verziójára. Ebben a dokumentumban elsősorban le van írva a class fájl felépítése. Mik követik egymást, ezen belül fontos nekünk a `Constant Pool` és a `Methods` rész. A `Constant Pool` -ban találunk minden olyan információt amely szükséges ahhoz, hogy milyen osztályaink, metódusaink, metódus szignatúráink és változóértékeink vannak. A `Methods` részben találjuk a tényleges utasításokat, itt hívodnak meg a `Constant Pool`-ban leírt osztályoknak a metódusai, itt végezzük el az értelmezést.

A programban szerepelnek a szükséges adatszerkezetek: a `stack` és a lokális változók. Az interpretált kódot egy bájt tömbként értelmezi a program, amelyhez szükséges a jelenlegi indexet egy külön változóban eltárolnunk. Ez a változó azért is szükséges, hogy a különböző elágazások, ciklusok működni tudjanak, leegyszerűsítve a ciklus csak egy elágazás, amely többször hajtódik végre.

2.3. Program használata

2.3.1. 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 bátjkódban.)

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 bátjkód 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:

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

2.1. forráskód. invokedynamic utasítást tartalmazó Java kód

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.

2.3.2. Fordítástól futásig

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.

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):

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

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

```
$ 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:

```
$ mvn package
```

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

Futtattás

Ha a generált class fájllal 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 a korábbi parancsot)

A maven által készített jar fájllal 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:

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

Önfuttatás

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

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

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

2.3.3. 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 bájtkód 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. Probléma specifikációja

A probléma egy fájl beolvasásával kezdődik. Minden class fájl legelején szerepel a `CA FE BA BE` konstans bájt sorozat. Ez egy szükséges, viszont nem elégséges feltétel egy class fájl érvényességére. A konstans értéken felül számos más értéket is tartalmaz egy class fájl. Ha ezek a specifikációnak megfelelően vannak, és a fájl végén nincsen extra bájt, akkor érvényes ténylegesen a class fájl.

Ebben a class fájlban, ha érvényes, megpróbáljuk a belépési pont függvényét megkeresni. Ez Java programokban a `main` függvény.

Ehhez a függvényhez tartozik egy bájt sorozat a class fájlban, amely a függvény során végrehajtandó tényleges utasítások bájt sorozata. A probléma megoldására nekünk ezt a sorozatot szükséges értelmeznünk. Ha a bájt sorozat megköveteli, akkor más függvényeket kell meghívnunk, amelyek lehetnek ugyanabban a fájlban, viszont lehet hogy más fájlban vannak benne, ilyenkor szükséges egy másik class fájlt beolvasnunk, majd a megfelelő függvényt értelmeznünk.

A probléma megoldására során minden hibát szeretnénk a felhasználó felé megfelelően továbbítani, példa ezekre:

- A megadott class fájl nem érvényes
- A megadott class fájl nem tartalmaz érvényes belépési pontot
- A belépési pont futtatása során hiba keletkezett

Természetesen ugyanígy a sikeres futást is szeretnénk a felhasználó felé továbbítani. A sikeres futás itt azt jelenti, ha a kimenet megyezik a beépített `java` interpreter kimenetével.

3.2. Megoldás lépései

3.2.1. Class fájl felépítése

A specifikáció alapján a class fájl felépítése a következő:

- 4 bájt konstans bájt sorozat (`CA FE BA BE`)
- 2 bájt `minor` verziószám
- 2 bájt `mayor` verziószám
- 2 bájt `Constant Pool` darabszáma (+1, az 1-től való indexelés miatt)
- Változó bájt `Constant Pool Info` (praktikusan amíg darabszámnyit nem tudtuk beolvasni)
- 2 bájt hozzáférési zászlók
- 2 bájt a `this` osztály indexére
- 2 bájt a `super` osztály indexére (minden osztály valamely osztály leszármazottja)
- 2 bájt az interfészek darabszáma
- Változó bájt interfész információ
- 2 bájt a függvények darabszáma
- Változó bájt függvény információ
- 2 bájt az osztály attribútumainak darabszáma
- Változó bájt osztály attribútum információ

Class fájltól a benne levő metódus futtatásáig

A fő osztály a `ClassFile`, amely számos dologért felel. Többek között egy class fájl beolvasáért, amely során a megfelelő adattagok beállítja. A `ClassFile` osztálynak

egy konstruktora van, mégpedig:

```
1 public ClassFile(String fileName)
```

Tehát az egyetlen paraméter a beolvasandó class fájl neve.

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

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

Ez a függvény egy adott fájlnévre beolvassa a class fájlban tárolt adatokat a megfelelő változókba. (Ezen felül egy `VALID_CLASS_FILE` változót is beállít; feltételezük 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, String
  ↪  methodDescription)
```

Ez egy adott függvénynévre a megfelelő nevű metódust visszaadja a beolvasott fájlból (ha nem talál ilyet akkor `null`-t ad vissza).

Opcionális paraméterként megadhatjuk a metódus szignatúráját is, ehhez a JVM specifikációban szereplő típus szöveggé való kódolásának mintája használandó.

Kódolt típus	**Java Típus**
B	byte
C	char
D	double
F	float
I	int
J	long
L*classnév*;	reference (osztály)
S	short
Z	boolean
[	reference (tömb)

3.1. táblázat. Típust tartalmazó szöveg dekódolásá Java típusokká a JVM specifikáció alapján

A bemeneti paramétereket zárójelek választják el, ezután szerepel a visszatérési érték. Tehát az általános Java belépési függvény, a `main` java kódja:

```
public static void main(String[] args)
```

Ennek a függvény szignatúrájának kódolása a következő: `([Ljava/lang/String;)V`

Példa a belépési metódust megkereső függvény használatára:

```
ClassFile CLASS_FILE = new ClassFile("Main.class");
Method_Info method = CLASS_FILE.findMethodsByName("main",
↪  "([Ljava/lang/String;)V");
```

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

```
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 megfelelü függvény attribútumait is át kell adnuk, például:

```
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 ajanlott végigmenni, a `List` implementálja az `Iterable`-t, így egy for ciklussal elegánsan megtehetjük ezt:

```
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, Object[]
  ↪  args) Throwable
```

A reflekció miatt számos hibát dob vissza a függvény, ha nem helyes a kód formátuma akkor `IOException`-t dob a függvény, viszont egy `Throwable`-el az egészet le tudjuk kezelni; a `Throwable` az `ATHROW` Java bájtkód instrukció miatt egyébként szükséges (ekkor egy hibát dob vissza a metódusunk). Visszatérési értéke `Pair<Class<?>, Object>`, a számos `RETURN` utasítás miatt (ezeket a `stack`-en szükséges elhelyeznünk). Opcionálisan megadhatunk neki egy `args`-ot is, amely a függvény paramétere(i) egy `Object` tömbben. Példa a használatára:

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

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:

```
public class Interpreter {
  public static void main(String[] args) throws Throwable {
    ClassFile CLASS_FILE = new ClassFile("Main.class");
    Method_Info method = CLASS_FILE.findMethodsByName("main",
    ↪  "([Ljava/lang/String;)V");
    List<Attribute_Info> attributes =
    ↪  CLASS_FILE.findAttributesByName(method.attributes, "Code");

    for (Attribute_Info attribute : attributes) {
      Code_Attribute codeAttribute =
      ↪  Code_Attribute_Helper.readCodeAttributes(attribute);
      CLASS_FILE.executeCode(codeAttribute.code, null);
    }
  }
}
```

3.1. forráskód. Példa a Main.class interpretálására

Pár minta class fájl felépítésére

A legegyszerűbb class fájl ami értelmes, viszont nem futtattható:

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:

```

```

3.2. forráskód. Legegyszerűbb class fájl Java kódja

Elérhető a `src/test/java/com/zoltanbalazs/Simple.class` fájl formátumában 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: Az osztály hozzáférési zászlói ()
- 00 00: `This` osztály indexe a `Constant Pool`-ban
- 00 00: `Super` osztály indexe a `Constant Pool`-ban
- 00 00: Interfészek száma ($00\ 00_{16} = 0_{10}$ db)

- 00 00: Adattagok száma (00 00_{16} = 0_{10} db)
- 00 00: Függvények száma (00 00_{16} = 0_{10} db)
- 00 00: Osztály attribútumainak száma (00 00_{16} = 0_{10} db)

A legegyszerűbb class fájl amit a *Jabyinja* program le tud futtatni (a beépített `java` interpreter program nem képes ezt lefuttatni, mivel a `main` metódus szignatúrája nem megfelelő (a paraméterek kötelezőek), illetve nincsenek osztályok):

```
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:

```
public static void main() {
    return;
}
```

3.3. forráskód. Legegyszerűbb class fájl, amely a szakdolgozat programja által futtatható, Java kódja

Elérhető a `src/test/java/com/zoltanbalazs/Base.class` fájl formátumában

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, jelen esetben: 00 04_{16} = 4_{10}, 4 - 1 = 3 db)
- 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: 00 04_{16} = 4_{10} 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: 00 04_{16} = 4_{10} 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: 00 03_{16} = 3_{10} hosszú

    28 29 56: A `CONSTANT_Utf8` értéke: `()V`
- 00 21: Az osztály hozzáférési zászlói `(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_{16} = 0_{10} db)
- 00 00: Adattagok száma (00 00_{16} = 0_{10} db)
- 00 01: Függvények száma (00 01_{16} = 1_{10} db)
- 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: Függvények
  - 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 09: Hozzáférési zászlók `(Public, Static)`

    00 02: `Constant Pool`-ban lévő indexe a függvénynek: `main`

    00 03: `Constant Pool`-ban lévő indexe a függvény szignatúrájának: `()V`

    00 01: Függvény attribútumainak száma (00 01_{16} = 1_{10} db)

    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 (00 00 00 $0D_{16}$ = 13_{10} bájt)

      00 00 00 00 00 00 00 01 B1 00 00 00 00: Attribútum
      - 00 00 00 00 00 00 00 01 B1 00 00 00 00

        00 00: `Stack`-en lévő elemek maximális száma (00 00_{16} = 0_{10} db)

        00 00: Lokális változók száma (00 00_{16} = 0_{10} db)

        00 00 00 01: Kód hossza (00 00 00 01_{16} = 1_{10} bájt)

        B1: Kód `(B1 = return)`

        00 00: Kivételek száma (00 00_{16} = 0_{10} db)

        00 00: Attribútum attribútumainak száma (00 00_{16} = 0_{10} db)
- 00 00: Osztály attribútumainak száma (00 00_{16} = 0_{10} db)

A legegyszerűbb class fájl amelyet a már a beépített 17-es verziójú `java` interpretáló program is le tud futtatni, ez már teljesen megfelel a JVM specifikációnak (Fontos hogy a fájl neve `Main.class` legyen):

```
CA FE BA BE 00 00 00 2D 00 0C 0A 00 02 00 03 07
00 04 0C 00 05 00 06 01 00 10 6A 61 76 61 2F 6C
61 6E 67 2F 4F 62 6A 65 63 74 01 00 06 3C 69 6E
69 74 3E 01 00 03 28 29 56 07 00 08 01 00 04 4D
61 69 6E 01 00 04 43 6F 64 65 01 00 04 6D 61 69
6E 01 00 16 28 5B 4C 6A 61 76 61 2F 6C 61 6E 67
2F 53 74 72 69 6E 67 3B 29 56 00 21 00 07 00 02
00 00 00 00 00 02 00 01 00 05 00 06 00 01 00 09
00 00 00 11 00 01 00 01 00 00 00 05 2A B7 00 01
B1 00 00 00 00 00 09 00 0A 00 0B 00 01 00 09 00
00 00 0D 00 00 00 01 00 00 00 01 B1 00 00 00 00
00 00
```

Java kód megfelelője:

```
public class Main {
  public static void main(String[] args) {
    return;
  }
}
```

3.4. forráskód. Legegyszerűbb class fájl, amely a beépített interpreter által is futtatható, Java kódja

Elérhető a `src/test/java/com/zoltanbalazs/Main.class` fájl formátumában

class fájl formátumának magyarázata:

- CA FE BA BE: Mágikus szám, amely minden class fájl elején megtalálható
- 00 00 00 2D: class fájl `Minor` és `Major` verziószáma, egy táblázatnak megfelelően a felhasznált `javac` fordítóprogram verziója, jelen esetben az 1.0-ás verzió
- 00 0C: A `Constant Pool` mérete (+1, mivel 1-től indexelt, jelen esetben: 00 $0C_{16}$ = 12_{10}, 12 - 1 = 11 db)
- 0A 00 02 00 03 07 00 04 0C 00 05 00 06 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74 01 00 06 3C 69 6E 69 74 3E 01 00 03 28 29 56 07 00 08 01 00 04 4D 61 69 6E 01 00 04 43 6F 64 65 01 00 04 6D 61 69 6E 01 00 16 28 5B 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 29 56: `Constant Pool`

– 0A 00 02 00 03

  0A: `Constant Pool Info` érték: `(CONSTANT_Methodref)`

  00 02: `Constant Pool`-ban lévő indexe a függvényt tartalmazó osztálynak: `java/lang/Object`

  00 03: `Constant Pool`-ban lévő indexe a függvény nevének és szignatúrájának: `<init> ()V`
– 07 00 04

  01: `Constant Pool Info` érték: `(CONSTANT_Class)`

  00 04: `Constant Pool`-ban lévő indexe az osztály nevének: `java/lang/Object`
– 0C 00 05 00 06

  0C: `Constant Pool Info` érték: `(CONSTANT_NameAndType)`

  00 05: `Constant Pool`-ban lévő indexe a függvény nevének: `<init>`

  28 29 56: `Constant Pool`-ban lévő indexe a függvény szignatúrájának: `()V`
– 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74

  01: `Constant Pool Info` érték: `(CONSTANT_Utf8)`

  00 10: 00 10_{16} = 16_{10} hosszú

  6A 61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74: A `CONSTANT_Utf8` értéke: `java/lang/Object`
– 01 00 06 3C 69 6E 69 74 3E

  01: `Constant Pool Info` érték: `(CONSTANT_Utf8)`

  00 06: 00 06_{16} = 6_{10} hosszú

  3C 69 6E 69 74 3E: A `CONSTANT_Utf8` értéke: `<init>`
– 01 00 03 28 29 56

  01: `Constant Pool Info` érték: `(CONSTANT_Utf8)`

  00 06: 00 06_{16} = 6_{10} hosszú

  28 29 56: A `CONSTANT_Utf8` értéke: `()V`
– 07 00 08

  07: `Constant Pool Info` érték: `(CONSTANT_Class)`

  00 08: `Constant Pool`-ban lévő indexe az osztály nevének: `Main`
– 01 00 04 4D 61 69 6E

  01: `Constant Pool Info` érték: `(CONSTANT_Utf8)`

  00 04: 00 04_{16} = 4_{10} hosszú

  4D 61 69 6E: A `CONSTANT_Utf8` értéke: `Main`
– 01 00 04 43 6F 64 65

  01: `Constant Pool Info` érték: `(CONSTANT_Utf8)`

  00 04: 00 04_{16} = 4_{10} 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: 00 04_{16} = 4_{10} hosszú

  6D 61 69 6E: A `CONSTANT_Utf8` értéke: `main`
- 01 00 16 28 5B 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 29 56

  01: `Constant Pool Info` érték: `(CONSTANT_Utf8)`

  00 16: 00 16_{16} = 22_{10} hosszú

  28 5B 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 29 56: A `CONSTANT_Utf8` értéke: `([Ljava/lang/String;)V`

- 00 21: Hozzáférési zászlók `(Public, Super)`
- 00 07: `This` osztály indexe a Constant Pool-ban: `Main`
- 00 02: `Super` osztály indexe a Constant Pool-ban: `java/lang/Object`
- 00 00: Interfészek száma (00 00_{16} = 0_{10} db)
- 00 00: Adattagok száma (00 00_{16} = 0_{10} db)
- 00 02: Függvények száma (00 02_{16} = 2_{10} db)
- 00 01 00 05 00 06 00 01 00 09 00 00 00 11 00 01 00 01 00 00 00 05 2A B7 00 01 B1 00 00 00 00 00 09 00 0A 00 0B 00 01 00 09 00 00 00 0D 00 00 00 01 00 00 00 01 B1 00 00 00 00: Függvények
  - 00 01 00 05 00 06 00 01 00 09 00 00 00 11 00 01 00 01 00 00 00 05 2A B7 00 01 B1 00 00 00 00

    00 01: Hozzáférési zászlók `(Public)`

    00 05: `Constant Pool`-ban lévő indexe a függvénynek: `<init>`

    00 06: `Constant Pool`-ban lévő indexe a függvény szignatúrájának: `()V`

    00 01: Függvény attribútumainak száma (00 01_{16} = 1_{10} db)

    00 09 00 00 00 11 00 01 00 01 00 00 00 05 2A B7 00 01 B1 00 00 00 00: Attribútumok

    - 00 09 00 00 00 11 00 01 00 01 00 00 00 05 2A B7 00 01 B1 00 00 00 00

      00 09: `Constant Pool`-ban lévő indexe az attribútumnak: `Code`

      00 00 00 11: Attribútum hossza (00 00 00 11_{16} = 17_{10} bájt)

      00 01 00 01 00 00 00 05 2A B7 00 01 B1 00 00 00 00: Attribútum

      - 00 01 00 01 00 00 00 05 2A B7 00 01 B1 00 00 00 00

        00 01: `Stack`-en lévő elemek maximális száma (00 01_{16} = 1_{10} db)

        00 01: Lokális változók száma (00 01_{16} = 1_{10} db)

00 00 00 05: Kód hossza (00 00 00 05_{16} = 5_{10} bájt)

2A B7 00 01 B1: Kód `(2A = aload_0, B7 = invokespecial : 00 01 = java/lang/Object <init> ()V, B1 = return)`

00 00: Kivételek száma (00 00_{16} = 0_{10} db)

00 00: Attribútum attribútumainak száma (00 00_{16} = 0_{10} db)

- 00 09 00 0A 00 0B 00 01 00 09 00 00 00 0D 00 00 00 00 00 00 00 01 B1 00 00 00 00

00 09: Hozzáférési zászlók `(Public, Static)`

00 0A: `Constant Pool`-ban lévő indexe a függvénynek: `main`

00 0B: `Constant Pool`-ban lévő indexe a függvény szignatúrájának: `([Ljava/lang/String;)V`

00 01: Függvény attribútumainak száma (00 01_{16} = 1_{10} db)

00 09 00 00 00 0D 00 00 00 01 00 00 00 01 B1 00 00 00 00: Attribútumok

- 00 09 00 00 00 0D 00 00 00 01 00 00 00 01 B1 00 00 00 00

00 09: `Constant Pool`-ban lévő indexe az attribútumnak: `Code`

00 00 00 0D: Attribútum hossza (00 00 00 $0D_{16}$ = 13_{10} bájt)

00 00 00 01 00 00 00 01 B1 00 00 00 00: Attribútum

- 00 00 00 01 00 00 00 01 B1 00 00 00 00

00 00: `Stack`-en lévő elemek maximális száma (00 00_{16} = 0_{10} db)

00 01: Lokális változók száma (00 01_{16} = 1_{10} db)

00 00 00 01: Kód hossza (00 00 00 01_{16} = 1_{10} bájt)

B1: Kód `(B1 = return)`

00 00: Kivételek száma (00 00_{16} = 0_{10} db)

00 00: Attribútum attribútumainak száma (00 00_{16} = 0_{10} db)

- 00 00: Osztály attribútumainak száma (00 00_{16} = 0_{10} db)

Adatszerkezetek

A class fájlnak 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:

```
public List<Pair<Class<?>, Object>> stack = new ArrayList<>();
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 ($2^{16} = 65536$), alapból 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. Így a módosítás során 2 db 8-bites előjel nélküli számot olvasnak 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:

```
class CodeIndex {
    private int index = 0;

    public void Inc(int value) {
        index += value;
    }

    public int Next() {
        return index++;
    }

    public void Set(int value) {
        index = value;
    }

    public int Get() {
        return index;
    }
}
```

3.5. forráskód. Codeindex osztály, amely a kód bájttömb jelenlegi indexét tárolja

Az absztrakció oka, hogy függvényeknek átadva lehessen módosítani ezt a számot. Ez a szám a jelenlegi index a kódot reprezentáló bájt tömbben, ami 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.

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 bájttömb végén

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

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

```
public Pair<Class<?>, Object> executeCode(Code_Attribute attribute)
 ↪  throws Throwable {
  byte[] code = attribute.code;

  CodeIndex codeIndex = new CodeIndex();
  while (codeIndex.Get() < code.length) {
    byte opCode = code[codeIndex.Next()];

    switch (Opcode.opcodeRepresentation(opCode)) {
      ...
    }
  }
  throw new Throwable("Code did not contain a return statement");
}
```

3.6. forráskód. Interpretálásért felelős kódrészlet

3.2.2. Az interpreter sajátosságai

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	**java** interpreter		**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/Inheritence.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

Tesztfájl	**java** interpreter		**Jabyinja**	
	Memória	Futási idő	Memória	Futási idő
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
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

Tesztfájl	**java** interpreter		**Jabyinja**	
	Memória	Futási idő	Memória	Futási idő
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/Inheritence.class	41,6 MB	20,2 ms	47,7 MB	68,3 ms

3.2. táblázat. A beépített `java` és az interpreter közötti erőforrás különbségek

A program memóriamodellje

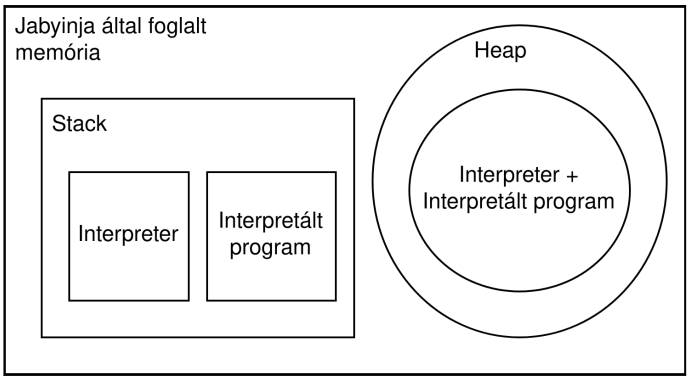
A heap memória nincsen implementálva a programban. Ez alapszik a standard Java interpreter implementációjára. Ide tartoznak többek között az objektumoknak a referenciái. Ebből következően a szemétgyűjtés (`garbage collector`) a beépített Java szemétgyűjtő algoritmust alkalmazza.

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 programban az alábbi memóriaeloszlás következik be:



3.1. ábra. Jabyinja program memóriamodellje

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
```

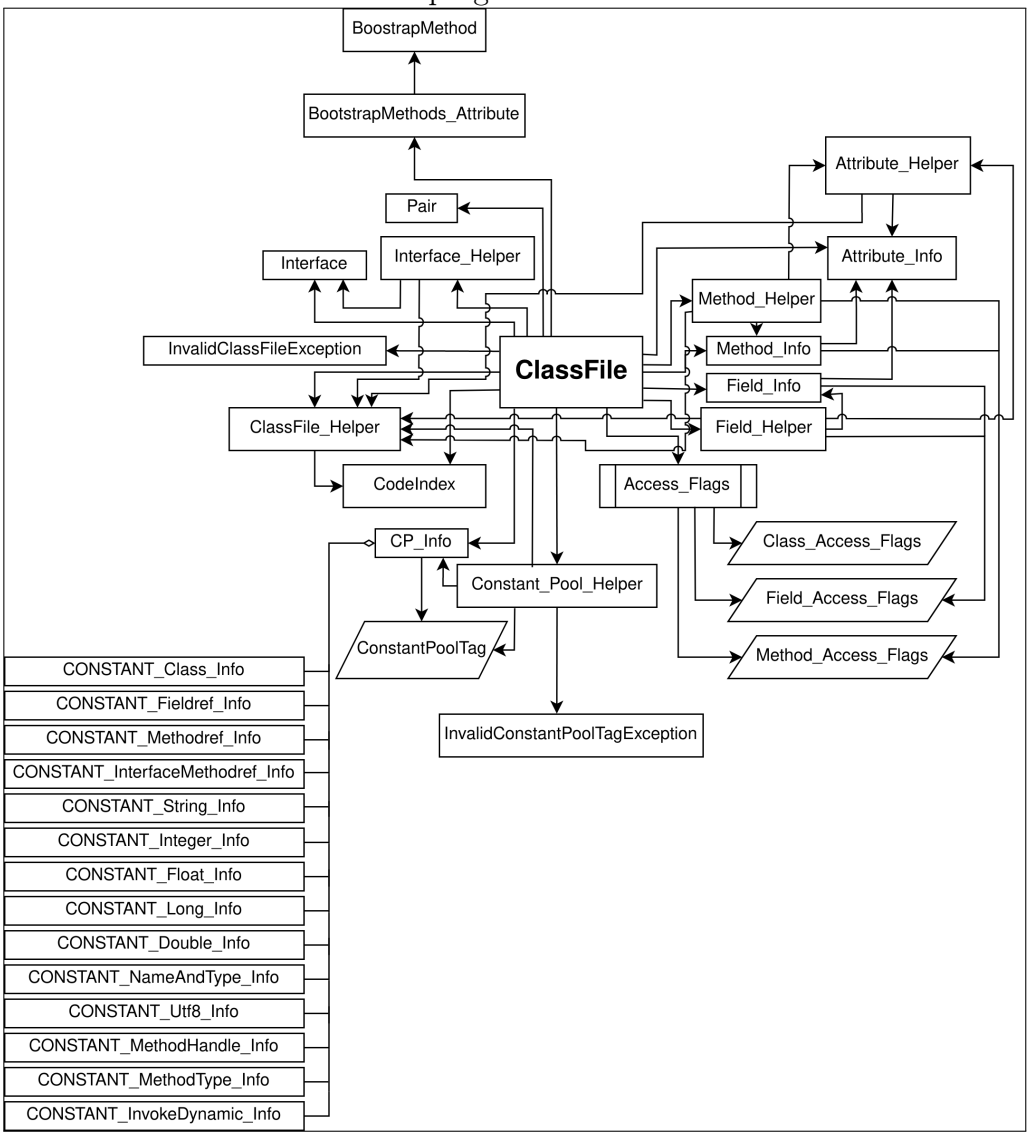
akkor igazából csak az interpreter `main` metódusáig interpretáljuk azt.

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

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

3.3. Program szerkezete

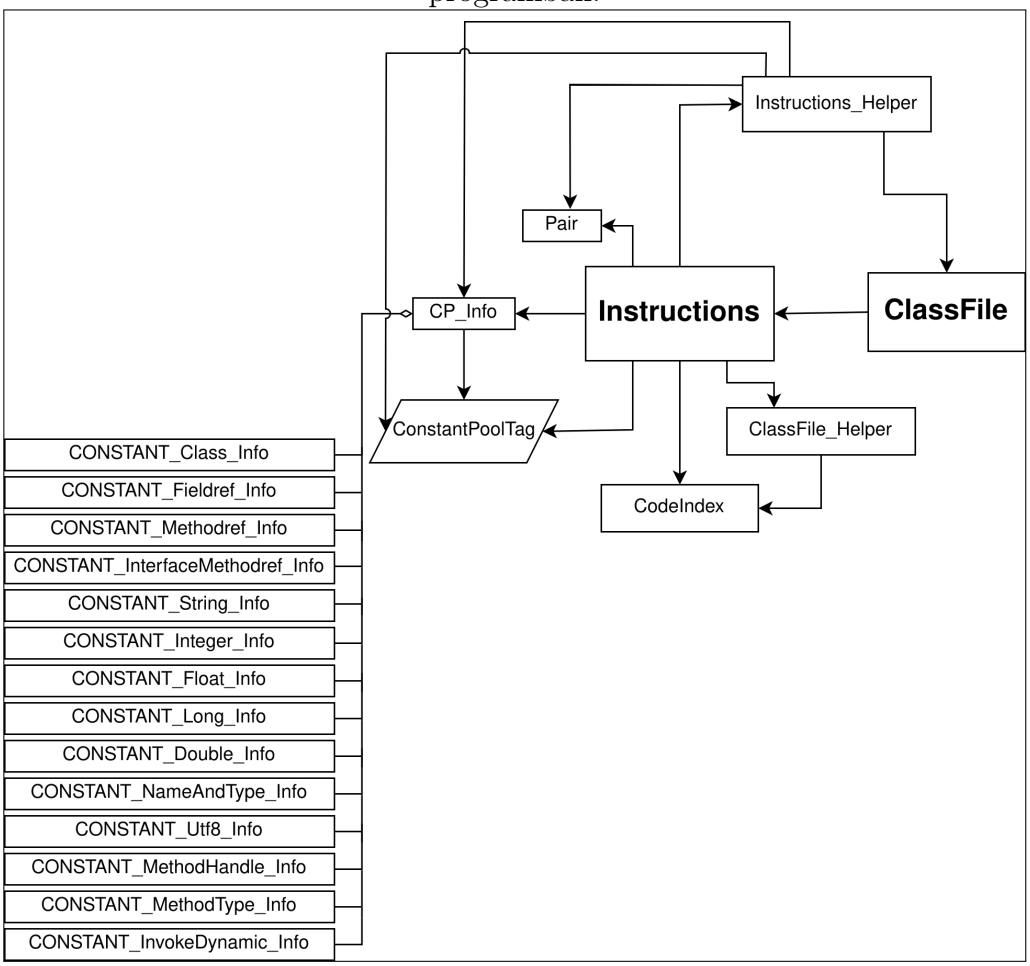
A class fájl beolvasása során a képen látható szerkezeti egység valósul meg a programban.



3.2. ábra. Class fájl beolvasása során a program szerkezete

Az ábrán látható, hogy a beolvasás feladata is egy komplex feladat. A különböző segédosztályok viszont megsegítik ennek a kisebb részek való lebontását, azoknak egyszerűbb megértését.

A bájt sorozat értelmezése során a képen látható szerkezeti egység valósul meg a programban.



3.3. ábra. Bájt sorozat értelmezése során a program szerkezete

Az ábrán látható, hogy némely osztály tagjait nem veszi figyelembe az értelmezés. Ennek oka a reflekció használata, amely során ezen adatokat elhanyagolhatjuk.

3.4. Tesztelésről

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 tudja. 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 fut le az interpret, ha nem, akkor helytelenül. Jelen esetben 3 teszteset van, amelyek nem mennek át a tesztelőn, ezek mindegyike tartalmaz `invokedynamic` utasítást.

3.4.1. Tesztelés lefuttatása

A tesztelőt nagyon egyszerűen meg lehet hívni a következő paranccsal:

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

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.

3.4.2. 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

3.4.3. Tesztelés eredményei

A teszteket lefuttatva az alábbi eredményt kell kapnunk:

```
~/Git/jabyinja main v1.0.0 v17.0.6
❯ python3 src/test/tester.py
Testing given files
 Own, basic test case(s):
  - com/zoltanbalazs/Own/Arithmetic:                PASSED
  - com/zoltanbalazs/Own/Arrayclass:                PASSED
  - com/zoltanbalazs/Own/Arraylist:                 PASSED
  - com/zoltanbalazs/Own/Athrow:                    PASSED
  - com/zoltanbalazs/Own/Dup2:                      PASSED
  - com/zoltanbalazs/Own/Functional:                FAILED
  - com/zoltanbalazs/Own/Inheritence:               PASSED
  - com/zoltanbalazs/Own/Instanceof:                PASSED
  - com/zoltanbalazs/Own/Multianewarray:            PASSED
  - com/zoltanbalazs/Own/Nested:                    PASSED
  - com/zoltanbalazs/Own/Ownclass:                  PASSED
  - com/zoltanbalazs/Own/SwitchAthrow:              PASSED
  - com/zoltanbalazs/Own/Template:                  PASSED
 PTI, basic test case(s):
  - com/zoltanbalazs/PTI/_01/Print:                 PASSED
  - com/zoltanbalazs/PTI/_02/_01/PointMain:         PASSED
  - com/zoltanbalazs/PTI/_02/_02/CircleMain:        PASSED
  - com/zoltanbalazs/PTI/_02/_03/CircleMain:        PASSED
  - com/zoltanbalazs/PTI/_02/_04/ComplexMain:       PASSED
  - com/zoltanbalazs/PTI/_02/_05/LineMain:          PASSED
  - com/zoltanbalazs/PTI/_03/IterletterMain:        PASSED
  - com/zoltanbalazs/PTI/_04/_02/DoubleVectorMain:  PASSED
  - com/zoltanbalazs/PTI/_05/_01/Swap:              PASSED
  - com/zoltanbalazs/PTI/_05/_02/IntegerMatrixMain: PASSED
  - com/zoltanbalazs/PTI/_05/_03/WildAnimalMain:    PASSED
  - com/zoltanbalazs/PTI/_05/_04/IntVectorMain:     PASSED
  - com/zoltanbalazs/PTI/_06/_02/AddByLine:         PASSED
  - com/zoltanbalazs/PTI/_06/_04/CircleMain:        PASSED
  - com/zoltanbalazs/PTI/_08/_01/BookMain:          PASSED
  - com/zoltanbalazs/PTI/_08/_02/CoffeeShop:        PASSED
  - com/zoltanbalazs/PTI/_09/_01/Divisors:          PASSED
  - com/zoltanbalazs/PTI/_09/_02/RemoveDiffer:      FAILED
  - com/zoltanbalazs/PTI/_09/_03/MinToFront:        FAILED
  - com/zoltanbalazs/PTI/_09/_04/MultiSetMain:      PASSED
  - com/zoltanbalazs/PTI/_10/_01/Extends:           PASSED
  - com/zoltanbalazs/PTI/_10/_02/BookMain:          PASSED
  - com/zoltanbalazs/PTI/_10/_03/BagMain:           PASSED
  - com/zoltanbalazs/PTI/_10/_04/Swap:              PASSED
  - com/zoltanbalazs/PTI/_11/_01/FlyingMain:        PASSED
  - com/zoltanbalazs/PTI/_11/_02/Main:              PASSED
  - com/zoltanbalazs/PTI/_11/_03/AnimalMain:        PASSED
  - com/zoltanbalazs/PTI/_11/_04/AnimalMain:        PASSED
  - com/zoltanbalazs/PTI/_12/Inheritence:           PASSED
 PTI, with argument test case(s):
  - com/zoltanbalazs/PTI/_01/GCD:                   PASSED
  - com/zoltanbalazs/PTI/_01/Greet:                 PASSED
  - com/zoltanbalazs/PTI/_01/Odd:                   PASSED
  - com/zoltanbalazs/PTI/_01/PerfectNum:            PASSED
  - com/zoltanbalazs/PTI/_01/PerfectNumRange:       PASSED
  - com/zoltanbalazs/PTI/_01/TwoNum:                PASSED
 PTI, with stdin test case(s):
  - com/zoltanbalazs/PTI/_01/Euler:                 PASSED
  - com/zoltanbalazs/PTI/_01/Factorial:             PASSED
  - com/zoltanbalazs/PTI/_01/Half:                  PASSED
  - com/zoltanbalazs/PTI/_01/Sqrt:                  PASSED
  - com/zoltanbalazs/PTI/_01/SquareRoot:            PASSED
  - com/zoltanbalazs/PTI/_04/_01/PointMain:         PASSED
  - com/zoltanbalazs/PTI/_06/_01/Calculator:        PASSED
 PTI, with stdin and argument test case(s):
  - com/zoltanbalazs/PTI/_06/_03/IsPartOf:          PASSED

Summary:
 PASSED: 53
 FAILED: 3
```

3.4. ábra. Jabyinja program tesztelésének eredménye

3.5. Továbbiak

3.5.1. Továbbfejlesztési lehetőségek

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 paramétereként), illetve a kiírás során szöveg(ek) és változó(k) konkatenációjánál is ez használt.

Az utóbbi egyszerűen kiküszöbölhető a `-XDstringConcat=inline` flag-gel való fordítással. Ezen flag használata 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.

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.

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 szerepele már a jelenleg hívandó konstruktor. Ha igen, akkor 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` (***L**ast **I**n **F**irst **O**ut*, 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 megtesszünk. 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 reflekciót újragondolva, azt elhagyva a sebesség növelhető lenne, erről részletesebben a "Reflekció újragondolása" szekcióban olvashatunk. Lényege, hogy a jelenlegi, beépített osztályokon levő reflekciót le kellene cserélni interpretálása. Az alapok ehhez megvannak, viszont ez is egy komolyabb programozási feladat.

További tesztelés

A szakdolgozat írása során megpróbáltam az alapos tesztelésre figyelni, ezért is vannak az alapvető instrukciók 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. 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 lefut-e 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 bájtkódot interpretálni, vagy csak limitált a Java programozási nyelvre. A témabejelentés alkalmával megfogalmazott terveimhez való igazodás miatt 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 a programban levő reflekció, mivel a program Java nyelven van írva, ezért nem találja a Clojure/Kotlin/Scala függvényeket. Helyette 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.

Reflekció újragondolása

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

A reflekció túlságos használata miatt a teljesítmény is rosszabb, amelynek javítása egy komoly optimalizálási feladatot eredményez.

A másik komoly probléma a szakdolgozatban a reflekció használatával, a 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 hagyatkoznánk a reflekcióra. Lehetne Java

nyelven egy univerzális interpretert írni, amely képes lenne Clojure/Kotlin/Scala fordítóprogramok által generált class fájlokat interpretálni, lefuttatni.

3.5.2. 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 bájtkód instrukciók viszonylag egyszerűek voltak, a 7-es verzióval lett az `invokedynamic` bevezetve, mely közelebbi ránézésre igazából egy "szuperinstrukció", ezzel az összes többi `invoke` instrukció implementálható, sőt, egy teljes Java bájtkód interpreter is megírható csak `invokedynamic` utasításokkal. Az `invokedynamic` utasítás a dinamikusság kérdésére volt a válasz. Lényege, hogy azon függvényhívásokat, amelyek paramatéreit fordítási időben nem tudjuk, egy hívási helynek (`CallSite`) megfelelően kerülnek a JVM-be bele, amely a hívás során a megfelelő paraméterekkel hívja meg azt.
- 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 fájlon belüli `Constant Pool`-ban. Ennek oka hogy minden `Constant Pool`-beli elem tényleges felhasználható adata 4 bájton van, kívéve a `Long` és `Double`, amelyek felhasználható adatai 8 bájton vannak. Így a fájlon 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.*", magyarul: "*Utólag visszagondolva, rossz döntés volt, hogy a 8 bájtos konstansok két konstans pool bejegyzést igényelnek.*" idézet áll.
- A `Long` és `Double` típusokhoz kapcsolódoan, egy hatalmas különbség a primitívek (`double, long`) és osztály (`java.lang.Double, java.lang.Long`) típusok kö-

zött, hogy lokális változókként a primitívek 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` primitívet, 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.

- A `Constant Pool` darabszámára 2 bájt áll rendelkezésre, így maximum $2^{2*8} = 65536$ érték szerepelhet a `Constant Pool`-ban egy class fájlon belül (ténylegesen a `Double` és `Long` értékek miatt ez lehet kevesebb). Ez azt jelenti, hogyha különösen nagy a forráskódot tartalmazó fájlunk, akkor lehet, hogy a fordítás során az információ nem fér bele egyetlen class fájlba.
- Az előző ponthoz kapcsolódoan, egy függvény maximális paramétereinek száma $2^{16} = 65536$ lehet, mivel a lokális változók darabszámát 2 bájton tárolja el a függvény attribútuma.

4. fejezet

Összegzés

A szakdolgozat célja egy Java bájtkód interpret írása volt. Habár a szakdolgozat témáját nem sikerült teljes egészében lefedni (`invokedynamic` utasítás), ennek ellenére elképesztően hasznos volt számomra a téma során végrehajtott kutatás. Alkalmam nyílt a JVM belső működését jobban megérteni, beleértve a class fájl felépítését.

A kódot törekedtem a jelenlegi tudásom szerint megfelelően felépíteni, jól dokumentálni, így mások számára is egyszerűen olvashatóra és érthetőre írni.

A `javap` program, amely a Java SDK-val jön (ennek a létezésére nem is tudtam ezelőtt) elképesztően hasznos volt számomra mind a szakdolgozat megírása során, és ezentúl is szerintem használni fogom.

Köszönetnyilvánítás

Szeretnék köszönetet mondani Kozsik Tamásnak, az ELTE Informatikai Karának dékánjának, a konzulensemnek, akivel az elfoglaltsága ellenére is nagyon sokat tudtam konzultálni és a félév során számos alkalommal a helyes irányba vezérelt engem.

Édesanyámnak, Boris Katalin Annának, aki nélkül az egyemetet el sem kezdtem volna, és végig ösztönzött engem. Nélküle nem sikerült volna a szakdolgozatom megírása.

Szeretnék külön köszönetet mondani Petes Mártonnak, az ELTE-IK Programtervező Informatikus BSc. szakának hallgatójának, aki az elakadásaim során elképesztően sokat segített abban hogy merre is keressem a problémámnak a megoldását. Meghallgatta a programmal való problémáimat.

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.

Ábrák jegyzéke

3.1. Jabyinja program memóriamodellje 27
3.2. Class fájl beolvasása során a program szerkezete 28
3.3. Bájt sorozat értelmezése során a program szerkezete 29
3.4. Jabyinja program tesztelésének eredménye 31

Táblázatok jegyzéke

3.1. Típust tartalmazó szöveg dekódolásá Java típusokká a JVM specifikáció alapján 12
3.2. Erőforrás különbségek 26

Forráskódjegyzék

2.1. invokedynamic utasítást tartalmazó Java kód 7
3.1. Példa a Main.class interpretálására 15
3.2. Legegyszerűbb class fájl Java kódja 15
3.3. Legegyszerűbb class fájl, amely a szakdolgozat programja által futtatható, Java kódja 16
3.4. Legegyszerűbb class fájl, amely a beépített interpreter által is futtatható, Java kódja 18
3.5. Codeindex osztály, amely a kód bájttömb jelenlegi indexét tárolja . . 22
3.6. Interpretálásért felelős kódrészlet 24