Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matevž Fabjančič

Simulator izvajanja javanske zložne kode

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Simulator izvajanja javanske zložne kode:

V okviru diplomskega dela izdelajte didaktični pripomoček, ki bo služil kot simulator izvajanja javanske zložne kode. Med obstoječimi odprtokodnimi javanskimi navideznimi stroji izberite tistega, ki ga boste najlažje razširili, in mu dodajte zahtevano funkcionalnost. Vaš program naj omogoča prikaz vseh ukazov trenutne metode, med katerimi naj bo posebej izpostavljen trenutno izvajani ukaz. Poleg tega naj program prikazuje tudi vrednosti lokalnih spremenljivk ter spominskih lokacij na skladu. V primeru izvajanja večnitnega programa naj vaš program za prikaz uporabi več oken.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
JVM	java virtual machine	javanski navidezni stroj
JIT	just in time [compilation]	sprotno prevajanje
FIFO	first in first out	prvi pride, prvi gre
GUI	graphical user interface	grafični uporabniški vmesnik

Povzetek

Naslov: Simulator izvajanja javanske zložne kode

Avtor: Matevž Fabjančič

Java je eden najbolj razširjenih programskih jezikov. Študenti računalništva in programerji programski jezik Java dobro poznajo. Kljub zmogljivim prevajalnikom pa je koristno poznati tudi ozadje javanskega okolja, javanski navidezni stroj. To lahko trenutno najlažje spoznavamo z uporabo razhroščevalnikov. Da bi se lahko spustili v notranjost javanskega navideznega stroja, bomo v okviru te diplomske naloge izdelali simulator, ki bo jasno prikazoval notranje stanje javanskega navideznega stroja.

Ključne besede: java, jvm, zložna koda.

Abstract

Title: Java Bytecode Execution Simulator

Author: Matevž Fabjančič

Java is one of the most widely used programming languages used today. Even though programmers do not need to know the background of the Java Virtual Machine for their daily work, it is considered useful. To aid computer science students and programmers in the process of learning the fundamentals of the JVM, a visual simulator was created. The simulator displays internal components of the JVM and makes it easy to understand how bytecode and data flows through the JVM.

Keywords: java, jvm, bytecode.

Poglavje 1

Uvod

Trenutno na svetovnem spletu ni mogoče najti orodja, ki bi lahko služilo kot didaktični pripomoček pri poučevanju o delovanju javanskega navideznega stroja na nivoju zložne kode in omogočalo prikazovanje njegovega stanja. Orodje *BCEL Class Construction Kit* uporabniku omogoča pregledovanje in urejanje zložne kode, ne pa izvajanja programa [1]. Izvajanje zložne kode nam omogoča vtičnik za integrirano razvijalno okolje Eclipse, *Bytecode Outline plugin for Eclipse* [8].

Pomagamo si lahko z razhroščevalniki, ki jih ponujajo integrirana razvijalna okolja, kot sta orodji IntelliJ IDEA in Eclipse. Ker so ti razhroščevalniki narejeni za programerje z namenom, da bi jim olajšali razhroščevanje javanskih programov, nam ne nudijo pogleda v notranjost javanskega navideznega stroja. Spremenljivke in metode prikazujejo podobno, kot jih programer vidi v izvorni kodi programa.

Razlog, da takšnih orodij ni, tiči tudi v dejstvu, da nam specifikacija javanskega navideznega stroja podaja zelo grob opis njegovega delovanja. Večino podrobnosti implementacije prepusti programerjem in načrtovalcem konkretne implementacije. Specifikacija javanskega navideznega stroja natančno opiše le stvari, kot so struktura razredne datoteke in učinki ukazov zložne kode. Ostale podrobnosti delovanja konkretne implementacije javanskega navideznega stroja so opisane v tem diplomskem delu.

Cilj diplomske naloge je izdelati simulator, ki bo omogočal vpogled v izvajanje programov, napisanih v Javi. Ta bo služil kot didaktični pripomoček pri poučevanju delovanja javanskega navideznega stroja.

Do uresničitve cilja bomo prišli s predelavo odprtokodne različice javanskega navideznega stroja, PyJVM. Izdelali bomo vmesnik, ki bo omogočal vpogled v osnovne komponente javanskega navideznega stroja. Te bomo nato uporabili za izdelavo uporabniškega vmesnika, ki po prikazoval podatke o trenutnem stanju in omogočal premikanje po zložni kodi.

Delovanje simulatorja bomo preizkusili z izvajanjem in opazovanjem delovanja preprostih programov. Pri testiranju bomo pozorni tako na rezultate testnih programov kot tudi na stanja notranjih komponent javanskega navideznega stroja. Pomembno bo tudi testiranje uporabniškega vmesnika. Čeprav uporabniška izkušnja ne bo v ospredju, bo moral biti uporabniški vmesnik kar se da uporabniku prijazen.

Končano orodje bo objavljeno na spletu in prosto dostopno.

Poglavje 2

Javanski navidezni stroj

Programski jezik Java in z njim javanski navidezni stroj sta bila ustvarjena z namenom, da bi lahko programerji pisali programe, ki bi bili enostavno prenosljivi med računalniškimi arhitekturami. Da bi to omogočili, so razvijalci razvili navidezni stroj, ki različne računalniške arhitekture pripelje na isti nivo – na nivo, ki omogoča, da na njem izvajamo programe, prevedene za javanski navidezni stroj.

Trenutno aktualna različica javanskega navideznega stroja je različica 10, ki je izšla februarja 2018. To poglavje je napisano po specifikaciji javanskega navideznega stroja sedme različice. Razlog je uporaba implementacije PyJVM, ki ne podpira novejših različic Jave [11].

2.1 Arhitektura

Za potrebe tega diplomskega dela moramo poznati nekatere notranje komponente javanskega navideznega stroja.

Javanski navidezni stroj je skladovni računalnik. Vsaki niti navideznega stroja pripada sklad, na katerem se nahajajo klicni zapisi metod. Za pomikanje po zložni vsaka nit uporablja svoj programski števec. Po specifikaciji je to edini register javanskega navideznega stroja [7], implementacije pa lahko dodajo tudi druge registre [4].

Klicni zapisi metod hranijo vrednosti argumentov in lokalnih spremenljivk neke metode. Na nivoju zložne kode med tema dvema tipoma spremenljivk ne ločujemo. Klicni zapisi vsebujejo tudi operandni sklad, nad katerim se izvajajo računske operacije. Vsi elementi operandnega sklada imajo velikost ene pomnilniške besede. Preko klicnih zapisov se prenašajo tudi rezultati metod. Ob klicu metode se na vrhu sklada doda nov klicni zapis, ob zaključku metode pa se njen klicni zapis izbriše [7].

Različne implementacije javanskega navideznega stroja uporabljajo mnoge optimizacije, kar hitrost programov, ki tečejo na javanskih navideznih strojih, približa hitrosti programov, ki tečejo neposredno na procesorju. Eden najbolj razširjenih pristopov je uporaba prevajalnika JIT, ki s prevajanjem pogosto izvajanih ukazov in skupin ukazov zložne kode drastično izboljša čas izvajanja programa [6].

2.2 Razredna datoteka

Eno izmed glavnih polj razredne datoteke je polje *constant_pool*, ki vsebuje podatke o vseh konstantah, ki jih program potrebuje za izvajanje (razredi, globalne spremenljivke, literali, itd.). Na to polje se preko indeksa sklicujejo druga polja razredne datoteke.

Zložna koda metod je zapisana v enem izmed polj *attribute_info* znotraj polja *method_info*. To polje je označeno z imenom *Code* (oziroma z ustreznim indeksom v polju konstant) in mora biti prisotno. V nasprotnem primeru pride do napake.

2.3 Zložna koda

Zložno kodo sestavlja 205 ukazov, od tega se trije uporabljajo zgolj interno v javanskem navideznem stroju. Večina ukazov razpolaga zgolj z lokalnimi spremenljivkami in operandnim skladom, nekateri ukazi pa uporabljajo tudi operande, ki so jim podani kot del zložne kode.

Ukaza *tableswitch* in *lookupswitch* imata spremenljivo dolžino, ostali ukazi pa imajo stalno dolžino [5, 7].

Tipi, ki se pojavljajo v zložni kodi (skupaj z njihovimi kratkimi oznakami):

integer (i): celo število, na operandnem skladu zasede 4 bajte (javanski tip `int`),

long (l): celo število, na operandnem skladu zasede 8 bajtov (javanski tip `long`),

short (s): celo število, na operandnem skladu zasede 4 bajte (javanski tip `short`),

byte (b): celo število, na operandnem skladu zasede 4 bajte(javanski tip `byte`),

character (c): celo število (predstavitev znaka v sistemu UTF-16), dolžine dveh bajtov (javanski tip `char`),

float (f): število v plavajoči vejici po standardu IEEE 754, dolžine štirih bajtov (javanski tip `float`),

double (d): število v plavajoči vejici po standardu IEEE 754, dolžine osmih bajtov (javanski tip `double`),

reference (a): kazalec na objekt.

Pri nekaterih ukazih je predstavitev sklada pred izvršitvijo ukaza in po njej predstavljena z notacijo $N1, N2 \rightarrow N3$. To pomeni, da sta pred izvajanjem ukaza na vrhu sklada vrednosti $N1$ in $N2$, po izvršenem ukazu pa je na vrhu sklada vrednost $N3$.

2.3.1 Ukazi za branje in pisanje lokalnih spremenljivk

Za branje lokalnih spremenljivk se uporabljajo ukazi oblike *Tload* in *Tload_N*, za pisanje pa ukazi oblike *Tstore* in *Tstore_N*. *T* predstavlja tip vrednosti, ki jo premikamo, *N* pa indeks lokalne spremenljivke v klicnem zapisu. Tip je predstavljen s kratko oznako enega izmed tipov navedenih na strani 5.

Kadar je ukaz oblike *Tload*, mu mora slediti operand, ki predstavlja indeks lokalne spremenljivke, ki jo beremo ali pišemo.

2.3.2 Ukazi za delo s konstantami

Za nalaganje celih števil, ki so lahko predstavljena z enim bajtom, se uporablja ukaz *bipush*, za števila, ki so lahko predstavljena z dvema bajtoma pa ukaz *sipush*. V obeh primerih se binarna predstavitev števila razširi na štiri bajte.

Konstante, ki jih ni mogoče naložiti z ukazoma *bipush* in *sipush*, se nahajajo v razredni datoteki, v polju konstant. Za nalaganje teh števil se uporabljajo naslednji ukazi:

> *ldc*, ki zahteva, da je na naslednjem bajtu zapisan indeks konstante v polju konstant. To vrednost naloži na operandni sklad in jo po potrebi razširi na dolžino štirih bajtov,
>
> *ldc_w*, ki zahteva, da je na naslednjih dveh bajtih zapisan indeks v polju konstant. To vrednost naloži na operandni sklad in jo po potrebi razširi na dolžino štirih bajtov,
>
> *ldc2_w*, ki zahteva, da je na naslednjih dveh bajtih zapisan indeks v polju konstant. Na tem indeksu mora biti zapisana konstanta tipa `long` ali tipa `double`. Vrednost dolžine osmih bajtov naloži na operandni sklad.

Za pogosto uporabljene vrednosti obstajajo ukazi, ki ne potrebujejo operandov. Vrednost, ki jo nalagajo se prepozna iz operacijske kode. To so ukazi

aconst_null, ki na vrh sklada naloži vrednost `null`,

iconst_m1 (vrednost -1), *iconst_1*, *iconst_2*, *iconst_3*, *iconst_4* in *iconst_5* za nalaganje celih števil dolžine štirih bajtov,

lconst_0 in *lconst_1* za nalaganje celih števil dolžine osmih bajtov,

fconst_0, *fconst_1* in *fconst_2* za nalaganje števil v plavajoči vejici dolžine štirih bajtov,

dconst_0 in *dconst_1* za nalaganje števil v plavajoči vejici dolžine osmih bajtov.

2.3.3 Ukazi za manipulacijo operandnega sklada

Javanski navidezni podpira ukaze *pop* in *pop2* za brisanje operandov na vrhu sklada in *dup* in *dup2* za podvojitev operanda na vrhu sklada. Ukaza *dup_x1* in *dup2_x1* se uporabljata za podvojitev operandov na vrhu sklada (podvojena vrednost je vstavljena eno mesto nižje: $N1, N2 \rightarrow N2, N1, N2$). Podobna sta ukaza *dup_x2* in *dup2_x2* za podvojitev operandov na vrhu sklada (podvojena vrednost je vstavljena dve mesti nižje: $N1, N2, N3 \rightarrow N3, N1, N2, N3$). Ukaz *swap* zamenja zgornji dve vrednosti na skladu.

2.3.4 Ukazi za aritmetične operacije

Javanski navidezni stroj podpira ukaze za seštevanje (*Tadd*), odštevanje (*Tsub*), množenje (*Tmul*), deljenje (*Tdiv*) in modulo (*Trem*). T predstavlja tip števil, nad katerimi se izvaja aritmetično operacija. Ker javanski navidezni stroj na tem nivoju ne uporablja vrednosti krajših od štirih bajtov, je lahko tip T izbran izmed *i*, *l*, *f* in *d* (obrazložitve na strani 5). Ukazi uporabijo dva operanda z vrha opredandnega sklada in rezultat pustijo na vrhu sklada.

Za računanje nasprotne vrednosti števil se uporablja (*Tneg*). Ta izračuna nasprotno vrednost operanda, ki je na vrhu operandnega sklada. Tip T je izbran izmed *i*, *l*, *f* in *d* (obrazložitve na strani 5).

Za povečevanje lokalnih spremenljivk za 1 se uporablja ukaz *iinc*.

2.3.5 Ukazi za logične operacije

Pomikanje bitov

Javanski navidezni stroj podpira ukaza za pomik bitov v levo (*ishl* in *lshl*), ukaza za aritmetični pomik bitov v desno (*ishr* in *lshr*) in ukaza za logični pomik bitov v desno (*iushr* in *lushr*). Ukazi za pomike celih števil dolžine štirih bajtov (tip *i*) iz operandnega sklada preberejo dve vrednosti. Prva predstavlja vrednost, katere bite želimo pomakniti, druga pa število bitov za pomik. Ukazi za pomike celih števil dolžine osmih bajtov (tip *l*) iz operandnega sklada preberejo tri vrednosti. Prva predstavlja število bitov za pomik, druga in tretja (skupaj 8 bajtov) pa vrednost, katere bite želimo pomakniti.

Bitne operacije

Javanski navidezni stroj podpira ukaze za logični ali (*Tor*), logični in (*Tand*) in logični ekskluzivni ali (*Txor*). Ukazi uporabijo dva operanda z vrha sklada in rezultat pustijo na vrhu sklada.

2.3.6 Ukazi za delo z objekti

Za stvaritev novega objekta se uporablja ukaz *new*. Ta ukaz kot argument v zložni kodi prejme dva bajta dolgo število. To število je indeks, na katerem se v polju konstant nahajajo informacije o razredu novega objekta. Kazalec na nov objekt se postavi na vrh sklada. Če ima konstruktor argumente, se ti podajo kot argumenti ukazu *invokespecial* (2.3.9).

2.3.7 Ukazi za delo s tabelami

Za definiranje novih tabel se uporabljajo ukazi *newarray*, *anewarray* in *multinewarray*. Ukazi kazalce na kopico, kjer so definirane tabele, pustijo na vrhu operandnega sklada.

Ukaz *newarray* kot operand v zložni kodi prejme število, dolgo en bajt, ki predstavlja enega izmed osnovnih javanskih tipov, opisanih v poglavju 2.3. Dolžino tabele prebere z vrha operandnega sklada.

Z ukazom *anewarray* definiramo tabelo kazalcev na objekte. Kot operand v zložni kodi prejeme eno število, dolžine dveh bajtov. To število je indeks v polju konstant, na katerem se nahaja opis razreda, ki mu pripadajo elementi nove tabele. Dolžino tabele podamo kot število na vrhu operandnega sklada.

Za definiranje več dimenzionalnih tabel specifikacija definira ukaz *multinewarray*. Ta ukaz kot prvi operand v zložni kodi prejme število, dolgo dva bajta, ki predstavlja indeks v polju konstant. Na tem indeksu se nahaja opis razreda, s katerim je predstavljena večdimenzionalna tabela. Drugi operand v zložni kodi pa je enozložno število, s katerim navideznemu stroju povemo, koliko dimenzij ima nova tabela. Dolžine tabel po dimenzijah podamo kot števila na vrhu operandnega sklada, za vsako dimenzijo eno.

2.3.8 Ukazi za vejitve programa

Pogojni skoki

Ukazi za pogojne skoke v zložni kodi zasedejo tri bajte. Prvi bajt predstavlja operacijsko kodo. Naslednja dva bajta pa sestavita 16 bitno število. To število predstavlja odmik (od trenutne vrednosti programskega števca), na katerem se program nadaljuje, če je primerjava uspešna. V primeru neuspešne primerjave se program nadaljuje na naslednjem ukazu.

V nadaljevanju je za opis relacij med vrednostmi na operandnem skladu uporabljena notacija $A\ R\ B$, kjer R predstavlja relacijo, B število na vrhu sklada, A pa naslednje število na skladu.

Primerjanje kazalcev na objekte

Za primerjanje kazalcev se uporabljata ukaza *if_acmpeq* ($A = B$) in *if_acmpne* ($A \neq B$) [7, str. 460].

Primerjanje celih števil

Za primerjanje celih števil so na voljo ukazi *if_icmpeq* ($A = B$), *if_icmpne* ($A \neq B$), *if_icmplt* ($A < B$), *if_icmpge* ($A \leq B$), *if_icmpgt* ($A > B$) in *if_icmple* ($A \geq B$) [7, str. 461].

Primerjanje vrednosti s konstanto 0

Za primerjanje vrednosti s konstanto 0 so na voljo ukazi *ifeq* ($A = 0$), *ifne* ($A \neq 0$), *iflt* ($A < 0$), *ifge* ($A \leq 0$), *ifgt* ($A > 0$) in *ifle* ($A \geq 0$).

Ti ukazi z vrha sklada vzamejo le eno vrednost, vrednost A [7, str. 463].

Brezpogojni skoki

Za brezpogojne skoke se uporabljata ukaza *goto*, ki zahteva operand dolžine dveh bajtov in *goto_w*, ki zahteva operand dolžine štirih bajtov. Oba ukaza operand interpretirata kot odmik od trenutne vrednosti programskega števca. Na tem odmiku se nadaljuje izvajanje programa.

Če sta za brezpogojni skok uporabljena ukaza *jsr* ali *jsr_w*, se pred skokom naslov naslednjega ukaza shrani na operandni sklad. Po shranjevanju naslova se izračuna odmik po istem postopku kot pri ukazih *goto* in *goto_w*.

Ukaz *ret* se uporabi za skok na naslov, ki je shranjen v lokalni spremenljivki z indeksom, ki je ukazu *ret* podan kot operand. To pomeni, da je naslov, ki je bil shranjen z ukazom tipa *jsr*, pred uporabo ukaza *ret* potrebno prestaviti v eno izmed lokalnih spremenljivk.

Ukaza *tableswitch* (0xaa) in *lookupswitch* (0xab)

Ukaza *tableswitch* in *lookupswitch* imata spremenljivo dolžino. Uporabljata se ob prevajanju stikalnih stavkov (angl. *switch statements*). Njuna dolžina je odvisna od možnosti (angl. *case*) stikalnega stavka.

Oba ukaza na podlagi celoštevilske vrednosti, ki se nahaja na vrhu operandnega sklada, izračunata naslov odmika, ki je del ukaz. Ta odmik nato

prištejeta trenutni vrednosti programskega števca. Razlika je zgolj v načinu računanja naslova odmika.

V nadaljevanju V predstavlja vrednost na vrhu operandnega sklada, PC pa trenutno vrednost programskega števca.

Ukaz *tableswitch* izkorišča stikalne stavke, pri katerih so možnosti nadaljevanja označene z zaporednimi celimi števili. Ukaz ima tri stalne operande. Naj D predstavlja operand *default* (ta predstavlja odmik veje *default*), L operand *low* (odmik prva možnosti) in H operand *high* (odmik zadnje možnost). Če velja $L \leq V \leq H$, se odmik nahaja na indeksu $V - L$. Sicer se program nadaljuje na naslovu $PC + D$ [7, str. 560-561].

Ukaz *lookupswitch* dovoljuje, da v zaporedju možnosti niso zastopane vse vrednosti. To pa pomeni, da mora z binarnim iskanjem poiskati možnost, ki se ujema z vrednostjo V [7, str. 525].

2.3.9 Ukazi za klicanje metod in vračanje rezultatov

Ob klicanju metode objekta se izvrši ukaz *invokevirtual* ali *invokespecial*, če gre za klic metode razreda, ki ga trenutni razred razširja, metode z določilom `private` ali konstruktorji. Oba ukaza zahtevata dva enozložna operanda, ki sestavita indeks v polju konstant, na katerem so zapisane informacije o klicani metodi [7, str. 482, 489]. Če je ta metoda v razredu definirana kot implementacija vmesnika (angl. interface), se uporabi ukaz *invokeinterface*. Za klicanje statičnih metod razredov se uporabi ukaz *invokestatic* [7, str. 35].

Argumenti metod se prenašajo preko operandnega sklada. Če gre za klic nestatične metode, je prvi operand kazalec na objekt, nad katerim se neka metoda kliče [7, 12].

7. različica specifikacije javanskega navideznega stroja doda ukaz *invokedynamic*. Namenjen je lažji implementaciji dinamično tipiziranih programskih jezikov, ki se izvajajo na javanskem navideznem stroju [10].

Za vračanje iz metod se uporabljajo ukazi oblike *Treturn* in ukaz *return*, za metode ki ne vračajo rezultata. Pri ukazih oblike *Treturn* je s T označen tip rezultata metode (i, l, f, d in a). Rezultat je vzet iz vrha operandnega

sklada klicane metode in porinjen na sklad klicoče metode.

Poglavje 3

Opis delovanja PyJVM

PyJVM je odprtokodna implementacija javanskega navideznega stroja v programskem jeziku Python [11]. Ta implementacija je bila uporabljena kot osnova za orodje `pyjvmgui`, ki je rezultat te diplomske naloge.

Ker delovanje PyJVM ni dokumentirano, sem tok programa analiziral sam. S PyJVM sem pognal testne programe, ki jih je sestavil avtor PyJVM. Z uporabo razhroščevalnika in izpisovanja podatkov o vrednostih spremenljivk sem prišel do ugotovitev, ki so zapisane v tem poglavju.

3.1 Zagon navideznega stroja

Ob zagonu navideznega stroja PyJVM se ustvari 48 javanskih niti (večnitnost je opisana v poglavju 3.3), ki poskrbijo za inicializacijo navideznega stroja. Dvaindvajseta nit po končani inicializaciji navideznega stroja ostane v čakanju, ostale pa se zaključijo. Ta nit naj bi skrbela za objekte, ki implementirajo metodo `finalize()`. To metodo ta nit pokliče preden se objekt izbriše iz pomnilnika.

Zavoljo spremljanja delovanja te niti sem napisal testni razred `langfeatures.Finalizer` (programska koda 3.1, ki ustvari dva objekta z metodo `finalize`. Nad obema objektoma kliče metodo `message`, ki izpiše preprosto sporočilo. Povezava do objektov se nato izbriše. S klicem me-

tode `System.gc()` se javanskemu navideznemu stroju namigne, naj začne s čiščenjem pomnilnika. Da bi se čiščenje pomnilnika zgodilo z večjo gotovostjo, se nit programa s klicem metode `Thread.sleep(...)` za nekaj sekund ustavi.

Ker je v PyJVM pomnilnik predstavljen s strukturami, ki jih ponuja programski jezik Python, s pomnilnikom upravlja okolje Python. To je razlog, da ob zagonu testnega primera `langfeatures.Finalizer` ob klicu metode `System.gc()` pride do napake, saj metoda v PyJVM ni implementirana.

3.2 Izvajanje ukazov

Ukazi zložne kode so v PyJVM predstavljeni s funkcijami. Vsaka izmed njih je označena z dekoratorjem `@bytecode`, ki kot argument `code` sprejme operacijsko kodo nekega ukaza. Ob uvozu skripte, v kateri se nahajajo funkcije, se povezava na funkcije ukazov shranijo v slovar, kjer je ključ operacijska koda.

```
def bytecode(code):
    def cl(func):
        BYTECODE[hex(code)] = func
        return func

    return cl
```

Programska koda 3.2: Programska koda dekoratorja `@bytecode`

Ob izvajanju se v postopku dekodiranja ukaza iz slovarja `BYTECODE` pridobi povezava na funkcijo, ki predstavlja trenutni ukaz. Ta funkcija se nato pokliče, kot argument pa se ji poda povezava na klicni zapis, preko katere ima dostop do operandnega sklada in morebitnih operandov v zložni kodi. Če funkcija za nek ukaz ni implementirana, torej je ni v slovarju `BYTECODE`, pride do napake.

```
public class Finalizer {

    public static void main(String[] args) {
        Finalizer f = new Finalizer(); f.message();

        f = new Finalizer(); f.message();

        f = null;

        System.gc();

        try { Thread.sleep(1000); }
        catch (Exception e) { e.printStackTrace(); }
    }

    public void message() {
        System.out.println("hello");
    }

    @Override
    public void finalize() {
        System.out.println("Finalizing");
    }
}
```

Programska koda 3.1: Programska koda testnega razreda `langfeatures.Finalizer`

```
@bytecode(code=0x60)
def iadd(frame):
    value2 = frame.stack.pop()
    value1 = frame.stack.pop()
    result = value1 + value2
    result = cut_to_int(result)
    jassert_int(result)
    frame.stack.append(result)
```

Programska koda 3.3: Funkcija, ki izvrši ukaz *iadd* (celoštevilsko seštevanje)

Zložna koda metode in razredo, ki jih programerju ponuja javanska razredna zbirka, se preberejo iz datoteke rt.jar, ki jo PyJVM potrebuje za delovanje. Nativne metode so implementirane v Pythonu.

3.3 Večnitnost v PyJVM

Večnitnost je v navideznem stroju PyJVM implementirano z vrsto FIFO. V to vrsto se nove niti ob stvaritvi dodajajo preko metode `add_thread` iz razreda VM.

Vsaka nit je predstavljena z objektom razreda Thread. Pomembni atributi teh objektov so sklad klicnih zapisov in zastavice, ki nosijo informacijo o čakanju drugih niti (is_notified, waiting_notify).

Kadar v javanskem programu pride do čakanja na zaključek niti, PyJVM pokliče funkcijo `java_lang_Object_wait__J_V`, ki predstavlja nativno metodo `wait()` iz razreda java.lang.Object. Učinek te funkcije je odvisen od zastavic is_notified in waiting_notify objektov razreda Thread. Če nit še ne čaka, se ustrezno popravijo zastavice, nit pa se postavi v čakalno vrsto. V nasprotnem primeru se preveri, če je bila nit obveščena o zaključku kakšne druge niti

(zastavica `is_notified`).

Dodeljevanje procesorskega časa nitim poteka po naslednjem postopku. Niti se iz vrste jemljejo po principu FIFO. Naj bo nit, ki je na vrsti za izvajanje, imenovana *nit* T. Nit T lahko izvede največ 100 ukazov. Ko se izvajanje niti T ustavi (bodisi zaradi konca programa, bodisi zaradi čakanja na drugo nit), se preveri, če se je nit T končala. Če se je nit T končala, se o tem obvesti vse niti, ki so čakale na zaključek niti T.

Poglavje 4

Implementacija orodja pyjvmgui

Orodje `pyjvmgui`, rezultat te diplomske naloge, je simulator za spremljanje izvajanja javanskih programov. Orodje kot razširitev javanskega navideznega stroja PyJVM, ki je opisan v poglavju 3, ponuja pregled nad javansko zložno kodo, operandnim skladom in lokalnimi spremenljivkami metod. Programe lahko preko grafičnega uporabniškega vmesnika izvajamo postopoma.

4.1 Izbira primerne implementacije javanskega navideznega stroja

Ker implementacija javanskega navideznega stroja zahteva veliko časa, sem moral za nadgradnjo izbrati obstoječega. Odločal sem se med spodaj naštetimi implementacijami. Pri vrednotenju primernosti in končni izbiri sem se osredotočal na naslednje lastnosti:

- **Preprostost:** Ker je potrebna predelava javanskega navideznega stroja, do te mere da lahko nad njim prikažemo uporabniški vmesnik, more biti izvorna koda lahko razumljiva. Komponente navideznega stroja morajo biti v izvorni kodi čimbolj jasno razvidne.

- **Možnost izdelave uporabniškega vmesnika:** Da bi lahko nad navideznim strojem izdelal uporabniški vmesnik, ki bi uporabniku prikazoval notranje stanje navideznega stroja, je zaželeno, da je implementiran v programskem jeziku, ki omogoča enostavno izdelovanje uporabniškega vmesnika.

- **Aktualnost:** Da bi lahko javanski navidezni stroj podpiral današnje programske konstrukte, mora podpirati čim novejšo različico Jave.

4.1.1 JamVM

Javanski navidezni stroj JamVM je bil izdelan kot izjemno majhna implementacija javanskega navideznega stroja. Implementiran je v programskem jeziku C. Podpira izvajanje programov do različice Java 6. Omogoča tudi sprotno prevajanje v strojno kodo (JIT), kar bi lahko otežilo nadaljnje delo [9].

Za navidezni stroj JamVM se nisem odločil, ker je namenjen dejanski uporabi in je temu primerno tudi optimiziran. Optimizirana koda je težko berljiva, mnoge komponente javanskega navideznega stroja (npr. programskega števca) niso jasno vidne.

4.1.2 OpenJDK

OpenJDK je ena najbolj razširjenih implementacij javanskega razvojnega in izvajalnega okolja. Napisana je v kombinaciji programskih jezikov C in C++. OpenJDK podpirajo tudi najnovejše različice Jave [2].

Podobno kot JamVM je tudi OpenJDK optimiziran program in zato manj primeren za predelavo in izdelovanje uporabniškega vmesnika. Prednost izvajalnega okolja OpenJDK je v njegovi sodobnosti, saj so nove različice v koraku z novimi različicami Jave.

4.1.3 PyJVM

PyJVM je preprost javanski navidezni stroj, ki je v celoti implementiran v programskem jeziku Python (različica 2). Nudi podporo javanskim programom, ki so združljivi z Javo 7 [11].

Slabosti PyJVM izvirajo predvsem iz jezika, v katerem je ta javanski navidezni stroj implementiran. Ker Python ni statično tipiziran jezik je podpora v razvijalnih okoljih slabša. Tudi razumevanje izvorne kode je zato težje.

Python pa prinaša tudi prednosti, ki so bile povod za izbiro te implementacije. Programi so enostavno prenosljivi med računalniškimi arhitekturami in operacijskimi sistemi. Obstaja tudi širok nabor knjižnic za izdelovanje uporabniških vmesnikov, kar bo poenostavilo programiranje.

4.2 Implementacija potrebnih abstrakcij

4.2.1 Predstavitev zložne kode

Za branje in nalaganje razrednih datotek poskrbi PyJVM. Ker je zložna koda metod predstavljena kot seznam bajtov, za prikaz na uporabniškem vmesniku ni primerna.

PyJVM funkcije, ki izvršujejo ukaze, ob zagonu poveže z ustreznimi zložnimi kodami. Ob zagonu programa se uvozijo skripte v modulu pyjvm.ops. Ta vsebuje funkcije, označene z dekoratorjem @bytecode (glej 3.2). To nam omogoči, da na podlagi zložne kode dobimo povezavo do funkcije, ki izvrši ustrezen ukaz (programska koda 3.3).

Na podoben način sem pripravil skripti za generiranje objektov razreda Bytecode in razločevanje operandov iz zložne kode. Objekti razreda Bytecode nam omogočijo, da na enostaven način dostopamo do naslova, operacijske kode, imena in operandov poljubnega ukaza, kar je ključno za prikaz na uporabniškem vmesniku.

```
@to_bytecode(code=0x17)
def fload(loc, code):
    """
    :return (size, Bytecode)
    """
    division_arr = [1]
    operands = []
    offset = 1
    for op in division_arr:
        cur_op_slice = code[loc + offset : loc + offset
    + op]
        if op == 1:
            operands.append(unpack_byte(cur_op_slice))
        elif op == 2:
            operands.append(unpack_short(cur_op_slice))
        elif op == 4:
            operands.append(unpack_int(cur_op_slice))
        offset += op

    return (1 + 1, Bytecode(loc, 0x17, operands))
```

Programska koda 4.1: Funkcija, ki pretvori ukaz *fload* v objekt tipa Bytecode

4.2.2 Dostop do klicnih zapisov in niti

PyJVM evidenco niti vodi z vrsto FIFO v objektih razreda VM. Ker s tem nimamo direktnega dostopa do točno določene niti, sem razred VM prilagodil tako, da se evidenca niti vodi še v navadnem seznamu. S tem lahko preko zaporedne številke vedno dostopamo do točno določene niti.

Zavoljo lažje interakcije med uporabniškim vmesnikom in navideznim strojem sem nadzor izvajanja poenostavil z razredom ThreadExecutor, ki v konstruktorju prejme povezavo na nit, ki jo bo predstavljal. Objekti tipa ThreadExecutor nam ponujajo pet metod:

`get_frame_for_thread`: pridobi povezavo do trenutnega klicnega zapisa za podano nit,

`is_daemon`: preveri, če je podana nit demon,

`step_thread`: izvrši en ukaz za podano nit,

`step_thread_until_frame_over`: tekoče izvajanje zložne kode do zaključka trenutne metode,

`step_thread_until_done_or_blocked`: tekoče izvajanje niti, dokler se ta ne zaključi. Če pride do blokade niti (npr. zaradi čakanja na vhodno izhodne naprave), se nadzor vrne uporabniku.

4.3 Implementacija grafičnega uporabniškega vmesnika

4.3.1 Paket PySide2

Za programiranje uporabniškega vmesnika je bil uporabljen paket PySide2, izdelek projekta Qt for Python. Qt for Python je uradno podprt projekt organizacije Qt Group, ki razvija tudi okolje Qt [3]. Ponuja programski vmesnik, ki programerjem olajša programiranje komunikacije med programom in uporabniškim vmesnikom.

```
Button {
    // ...
    onClicked: {
        app.stepExecutor()
        bytecodeTable.selection.clear()
        var loc = app.getCurLoc()
        // ...
    }
}
```

Programska koda 4.2: Primer uporabe jezika JavaScript znotraj predloge QML

Paket PySide2 lahko namestimo preko portala PyPI, z uporabo ukaza `pip2 install pyside2`. Ko je paket nameščen, je razvijalno okolje že pripravljeno na razvoj uporabniškega vmesnika.

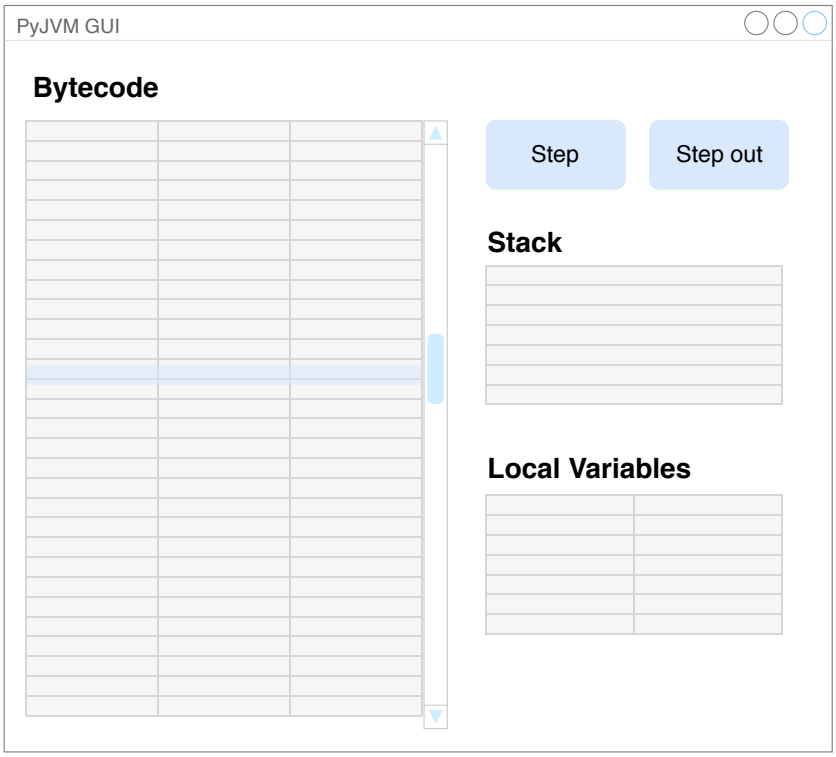
V okolju Qt lahko osnovni izgled in dogodke ob klikih na gumbe definiramo v predlogi QML. V njej v drevesni strukturi opišemo grafični uporabniški vmesnik. Akcije ob dogodkih definiramo v programskem jeziku JavaScript, ki ga lahko uporabljamo znotraj predolge QML.

4.3.2 Zaslonske maske

Javanski navidezni stroj ima več niti, ki so predstavljene na povsem enak način. Zato je smiselno, da za vsako nit prikažemo eno okno, ki prikazuje zložno kodo, operande in lokalne spremenljivke trenutnega okvirja (slika 4.2).

V tabeli zložne kode bo za vsak ukaz prikazan njegov naslov, ime ukaza in njegovi operandi.

Za spremljanje poteka programa bomo potrebovali tudi gumb, ki navideznemu stroju sporoči, da naj izvrši en ukaz. Funkcije, ki jih ponuja javanska razredna zbirka, pogosto kličejo druge funkcije, ki nas ne zanimajo. Da upo-



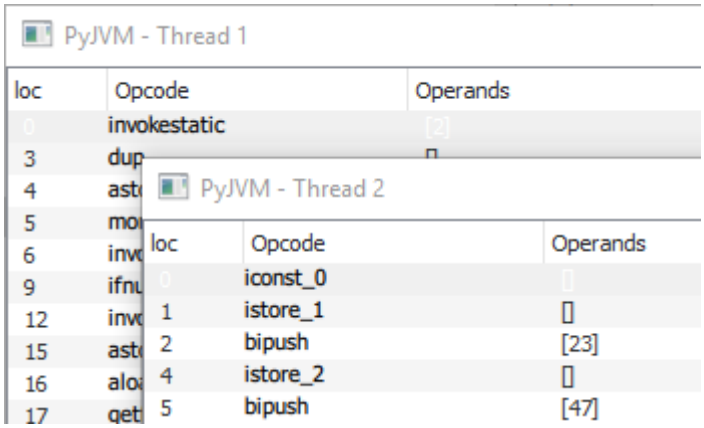
Slika 4.1: Zaslonska maska uporabniškega vmesnika

rabniku ne bi bilo treba spremljati poteka teh funkcij, bo na voljo tudi gumb, ki bo trenutno funkcijo brez interakcije z uporabnikom izvršil do konca.

4.3.3 Implementacija

Grafični vmesnik se sestavi v objektu razreda PyJvmGui. Ta razred vsebuje metode, do katerih lahko dostopamo iz predloge QML, ki predpisuje izgled grafičnega uporabniškega vmesnika.

Ker v predlogi QML ne moremo neposredno dostopati do spremenljivk in objektov, ki smo jih definirali v Pythonu, jih moramo zaviti v razrede, ki razširjajo razred QAbstractListModel, ki je del paketa PySide2. V teh razredih definiramo vlogo za vsak atribut objekta, ki ga zavijamo. Definiramo funkcijo *data*, ki kot argument prejme indeks elementa in vlogo (programska



Slika 4.2: Dve okni, za vsako nit eno

koda 4.3). Ta funkcija se implicitno kliče, kadar se v predlogi QML sestavljajo tabele. Kot argumenta, se funkciji podata zaporedna številka vrstice tabele, in vloga stolpca tabele (programska koda 4.4). Funkcija vrne niz znakov, ki predstavlja željeno vrednost.

Za pomikanje po programu se uporabljata dva gumba. Prvi gumb, z oznako "Step", ob kliku sproži funkcijo *step_thread* objekta tipa ThreadExecutor, ki izvrši en ukaz. Drugi gumb, z oznako "Step out", nam omogoči, da trenutno metodo izvršimo do konca.

Ob kliku na gumb se najprej pokliče metoda razreda PyJvmGui, ki je bila izpostavljena uporabniškemu vmesniku. Metode izpostavimo z dekoratorjem `@Slot()`. Argumenti dekoratorja predstavljajo tipe argumentov naše funkcije. Enega od operandov lahko označimo z imenom *result*. S tem okolju PySide2 povemo, da metoda vrača vrednost, ki je tipa *result* (programska koda 4.5). To vrednost lahko nato uporabimo za manipuliranje z uporabniškim vmesnikom v predlogi QML.

4.4 Objava orodja na spletnem portalu PyPI

PyPI, ali *Python Package Index*, je spletni portal, na katerem lahko brezplačno objavimo programe in knjižnice, napisane v Pythonu. Programe,

```
def data(self, q_modelindex, role=None):
    row = q_modelindex.row()

    if role == self.LOC_ROLE:
        return str(self.bytecodes[row].loc)

    if role == self.OP_ROLE:
        return str(self.bytecodes[row].name)

    if role == self.OPERAND_ROLE:
        return str(self.bytecodes[row].operands)
```

Programska koda 4.3: Funkcija data, ki predstavlja povezavo med predlogo QML in Pythonom za model "bytecode"

```
TableView {
    // ...

    TableViewColumn {
        role: "loc"
        // ...
    }

    // ...

    model: bytecode
}
```

Programska koda 4.4: Del predloge QML, ki definira stolpec z naslovi ukazov zložne kode

```
from PySide2.QtCore import Slot

@Slot(result=int)
def getCurLoc(self):
    return self.loc_to_idx.get(
        self.executor
            .get_frame_for_thread(self.thread_idx)
            .pc
    )
```

Programska koda 4.5: Funkcija, ki je izpostavljena grafičnemu uporabniškemu vmesniku.

objavljene v imeniku PyPI, lahko nameščamo in posodabljamo z orodjem `pip`. Orodje `pyjvmgui` je objavljeno v imeniku PyPI. Ta razdelek opisuje korake, ki vodijo do objave v imeniku PyPI.

Osnova za objavo v imeniku PyPI in nameščanje paketov je skripta *setup.py*, ki se nahaja v korenskem imeniku projekta. Ta mora vsebovati klic funkcije `setup`, ki jo ponuja paket `setuptools`, ki je del Pythona. Funkcija sprejme mnogo argumentov, s katerimi opišemo orodje: naziv, različico, avtorja, opis, idr.

4.4.1 Vsebina skripte *setup.py*

Skripta *setup.py* vsebuje informacije, ki so potrebne za namestitev. Večino teh podamo kot argumente funkciji `setup`. Kot argumente podamo ime paketa (`name`), različico (`version`), kratek opis paketa (`description`), sklice na kodo, ki jo želimo objaviti (argument `packages`).

Paketi so lahko tudi samostojni programi, ki jih lahko uporabnik požene iz ukazne vrstice. Če je naš paket samostojen program, kot argument podamo tudi slovar `entry_points`, ki vsebuje ključ `console_scripts`. Preko

tega ključa se v času namestitve prebere ime programa (ukaz, ki se bo uporabljal za zagon programa v ukazni vrstici) in sklic na funkcijo, ki predstavlja vstopno točko programa (v orodju `pyjvmgui` je to funkcija `main`).

Med argumenti podamo tudi seznam paketov, ki jih naš program potrebuje ob izvajanju. Te podamo v argumentu `install_requires`.

Namestitev lahko zahteva še kaj drugega kot le namestitev drugih paketov in namestitev skript. Kodo, za katero želimo, da se izvede, napišemo kot funkcijo `run` v razredu, ki razširja razred `install`, ki je del paketa `setuptools`. Podatke o novih razredih funkciji `setup` podamo v argumentu `cmdclass`. Ta argument je slovar, v katerem pod ključi, ki so imena osnovnih razredov (v tem primeru je to `install`), podamo sklice na nove razrede (tiste, ki implementirajo funcijo `run`).

4.4.2 Uporaba skripte *setup.py*

Skripto setup.py lahko uporabimo za sestavljanje paketov za objavo in tudi nalaganje sestavljenega paketa v imenik PyPI.

Objava paketa po priporočilih poteka v dveh korakih. V prvem koraku svoj paket objavimo v testnem imeniku PyPI. Testni imenik je sicer povsem enak glavnemu imeniku, namenjen pa je preizkušanju postopka objave in odpravljanju napak v skripti *setup.py*. Z ukazom `python setup.py sdist upload` svoj paket sestavimo kot distribucijo izvorne kode (skript, napisanih v Pythonu), in sestavljen paket naložimo v imenik. V testni fazi kot argument `-r` podamo še naslov testnega imenika, `https://test.pypi.org/legacy/`.

Ko smo prepričani, da bo objava uspešna, ponovimo zgornji postopek, le da izpustimo argument `-r`, s katerim smo podali naslov testnega imenika. Tokrat se naš paket naloži na glavni imenik PyPI.

Pri objavi ne smemo pozabiti, da se imena različic paketov v imeniku PyPI ne smejo podvajati. Pri preverjanju edinstvenosti se ne upoštevajo le različice paketov, ki so trenutno dostopne v imeniku PyPI, ampak se upošteva celotna zgodovina objav.

4.5 Uporaba orodja `pyjvmgui`

4.5.1 Namestitev

Orodje `pyjvmgui` je objavljeno v registru PyPI. Namestimo ga lahko z ukazom `pip2 install pyjvmgui`. Ukaz namesti orodje in vse pakete, ki jih `pyjvmgui` potrebuje.

Če imamo orodje `pyjvmgui` že nameščeno in ga želimo posodobiti, uporabimo ukaz `pip2 install pyjvmgui --upgrade`. Ob posodabljanju je priporočena uporaba zastavice `--no-deps`, ki prepreči, da bi se paket PySide2 še enkrat prenašal.

Namestitev v domačem imeniku uporabnika ustvari mapo .pyjvmgui, v katero shrani nekaj datotek, ki jih orodje potrebuje za delovanje.

Izvorna koda orodja `pyjvmgui` je izdana pod licenco GPL 3 in je dostopna na naslovu `https://github.com/MatevzFa/pyjvm`.

4.5.2 Prvi zagon

Orodje `pyjvmgui` za delovanje potrebuje javansko razredno zbirko sedme različice. Ob prvem zagonu programa se ta prenese s spletnega naslova `https://matevzfa.github.io/static/pyjvm/rt.jar` in shrani v mapo .pyjvmgui v domačem imeniku uporabnika. Ob vsakem zagonu programa se preveri če datoteka rt.jar obstaja. Če datoteka rt.jar ne obstaja ali je poškodovana, se prenese ponovno.

4.5.3 Uporaba orodja

Zagon orodja

Orodje, nameščeno z ukazom `pip2`, lahko uporabljamo neposredno iz ukazne vrstice. Javanski program mora biti preveden za Javo 7. Poženemo ga z ukazom `pyjvmgui -cp <IMENIK_RAZREDOV> <RAZRED>`, kjer `IMENIK_RAZREDOV` predstavlja pot do imenika z razredi našega programa, `RAZRED` pa ime razreda z metodo `main`, ki jo želimo pognati.

Ob zagonu se odpreta dve okni. Okno, ki je označeno s "Thread 1 (daemon)" je okno 22. niti PyJVM, ki je opisana v poglavju 3.1. Nit, ki izvede metodo `main` se prikaže v oknu, označenem s "Thread 2".

Pomikanje po programu

Za pomikanje po programu se uporabljajo trije gumbi na zgornji desni strani uporabniškega vmesnika, kot prikazuje slika 4.3.

Gumb z oznako "Step" izvrši ukaz, ki je osvetljen v seznamu zložne kode na levi strani okna.

Gumb z oznako "Step Out" povzroči tekoče izvajanje trenutne metode do njenega zaključka. Če je klicni zapis trenutna metoda edini na skladu niti, se ne zgodi nič.

Gumb z oznako "Step unitl thread done/blocked" povzroči tekoče izvajanje niti, dokler se ta ne zaključi. Če med izvajanjem pride do prekinitve, na primer zaradi izjeme ali klica metode `Thread.sleep(...)`, se izvajanje ustavi.

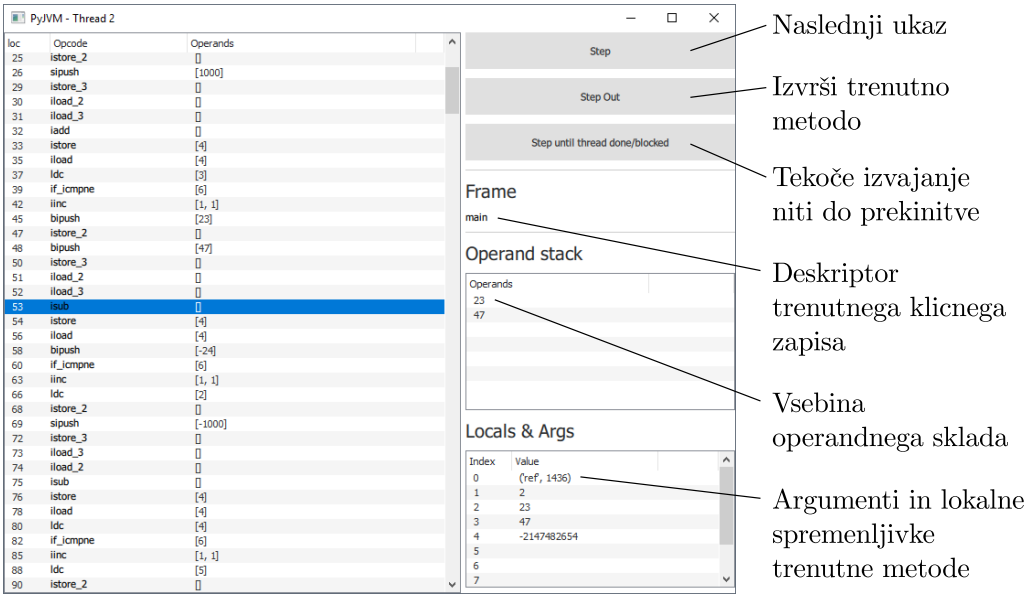
Če v programu, ki ga spremljamo pride do stvaritve novih niti, se ustvarijo nova okna orodja `pyjvmgui`.

Zaključek izvajanja

Ko se neka nit zaključi, se njeno okno zapre. Izjema je le okno "Thread 1 (daemon)", ki se zapre ob kliku na katerikoli gumb, če je to edina nit, ki še teče. Razlog za to izhaja iz upravljanja s pomnilnikom v PyJVM, ki je opisan v poglavju 3.3.

4.6 Orodje `class2json`

Z namenom, da bi bolje spoznal strukturo javanskih razrednih datotek, sem v programskem jeziku Rust napisal orodje, ki uporabniku omogoča, da razredno datoteko na zaslon izpiše na uporabniku prijazen način.



Slika 4.3: Okno orodja `pyjvmgui` med izvajanjem testnega primera `bytecode.CalcsTest`

Na podlagi knjige *The Java® Virtual Machine Specification: Java SE 7 Edition* sem sprva pripravil strukture, ki se pojavljajo v razrednih datotekah (cp_info, method_info, itd.) [7]. Te strukture se nato napolnijo s podatki, prebranimi iz podane razredne datoteke.

Ko je razredna datoteka predstavljena s strukturami, lahko njen prikaz spreminjamo z implementacijo funkcije `fmt`. V tej funkciji podamo obliko tekstovnega izpisa strukture, kot prikazuje programska koda 4.6.

Orodje `class2json` še ni povsem dokončano, saj razrednih datotek ne izpisuje v datoteke JSON. Vseeno pa nudi človeku razumljiv prikaz vsebine razrednih datotek. Orodje je odprtokodno in dostopno na spletnem naslovu `https://github.com/MatevzFa/class2json`.

```
fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    f.debug_struct("Utf8Info")
        .field("tag", &self.tag)
        .field("length", &self.length)
        .field(
            "bytes",
            &String::from_utf8((&self.bytes).clone()).
   unwrap()
        )
        .finish()
}
```

Programska koda 4.6: Funkcija `fmt` za prikazovanje strukture `Utf8Info`

Poglavje 5

Sklepne ugotovitve

Cilj te diplomske naloge je bil dosežen. Izdelano je bilo orodje `pyjvmgui`, ki lahko služi kot didaktični pripomoček pri poučevanju o delovanju javanskega navideznega stroja. Simulator na preprost način prikaže javanski navidezni stroj. Grafični uporabniški vmesnik orodja `pyjvmgui` nazorno prikaže niti programa in njihovo sočasno delovanje.

Orodje je odprtokodno, kar omogoča njegovim uporabnikom, da ga nadgradijo po lastnih željah in potrebah. Spremljanje izvorne kode programa, nadgradnja na tretjo različico programskega jezika Python in boljši uporabniški vmesnik so le nekatere izmed možnih nadgradenj orodja `pyjvmgui`.

Stranski produkt diplomskega dela je bilo tudi orodje class2json, opisano v poglavju 4.6. Orodje na človeku razumljiv način prikaže javansko razredno datoteko.

Poleg simulatorja lahko tudi drugo poglavje te diplomske naloge služi kot didaktični pripomoček. V njem so opisani ukazi javanske zložne kode in konkretna implementacija javanskega navideznega stroja, PyJVM.

Slike

Seznam izvlečkov programske kode

Literatura

[1] *BCEL Class Construction Kit*. Dosegljivo: `http://bcel.sourceforge.net/cck.html`. [Dostopano 11. september 2018].

[2] *OpenJDK*. Dosegljivo: `http://openjdk.java.net/`. [Dostopano 11. september 2018].

[3] *Qt for Python*. Dosegljivo: `https://www.qt.io/qt-for-python`. [Dostopano 11. september 2018].

[4] Tomaž Dobravec. *Sistemska programska oprema: Dinamično izvajanje programov*. Stran 24. Prosojnice pri predmetu sistemska programska oprema (63264).

[5] Stephen Gilmore and Javier Esparza. *CS1Bh Lecture Note 7 - Compilation I: Java Byte Code*. Dosegljivo: `http://www.dcs.ed.ac.uk/teaching/cs1/CS1/Bh/Notes/JavaByteCode.pdf`, 2003. [Dostopano 11. september 2018].

[6] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russel, and David Cox. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1), 2008.

[7] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java® Virtual Machine Specification: Java SE 7 Edition*. Dosegljivo: `https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf`, 2013. [Dostopano 11. september 2018].

[8] Andrey Loskutov. *Bytecode Outline plugin for Eclipse*. Dosegljivo: `http://andrei.gmxhome.de/bytecode/index.html`. [Dostopano 11. september 2018].

[9] Robert Lougher. *JamVM*. Dosegljivo: `http://jamvm.sourceforge.net/`. [Dostopano 11. september 2018].

[10] Francisco Ortin, Patricia Conde, Daniel Fernandez-Lanvin, and Raùl Izquierdo. The Runtime Performance of invokedynamic: An Evaluation with a Java Library. *IEEE Software, Software, IEEE, IEEE Software*, (4):1, 2014.

[11] Andrew Romanenco. *PyJVM*. Dosegljivo: `http://pyjvm.org/`. [Dostopano 11. september 2018].

[12] John Waldron and James Power. Comparison of Bytecode and Stack Frame Usage by Eiffel and Java Programs in the Java Virtual Machine. *Proceedings of the 2nd International Workshop on Computer Science and Information Technologies CSIT'2000*, 2000.