Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Luka Šveigl

Prevajanje programskega jezika PINS v javansko zložno kodo

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2023

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Kandidat: Luka Šveigl
Naslov: Prevajanje programskega jezika PINS v javansko zložno kodo
Vrsta naloge: Diplomska naloga na visokošolskem programu prve stopnje Računalništvo in informatika
Mentor: doc. dr. Boštjan Slivnik

Opis:
Izdelajte zadnji del prevajalnika za programski jezik PINS, ki omogoča generiranje javanske zložne kode in s tem izvajanje programov v javanskem navideznem stroju.

Title: Compiling the PINS programming language to Java bytecode

Description:
Implement the backend of the compiler for the PINS programming language. It should generate Java Bytecode and thus enable the execution of programs on the Java virtual machine.

Za vso podporo in spodbudo se zahvaljujem družini in dobrim prijateljem, ki so me na študijski poti vedno podpirali. Prav tako se zahvaljujem svojemu mentorju, doc. dr. Boštjanu Slivniku, za strokovno svetovanje, korektno mentorstvo in pomoč pri diplomski nalogi.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
JVM	Java virtual machine	javanski navidezni stroj
LIFO	last-in-first-out	zadnji-noter-prvi-ven
PC	program counter	programski števec
JIT	Just-In-Time	sprotno prevajanje
AST	abstract syntax tree	abstraktno sintaksno drevo

Povzetek

Naslov: Prevajanje programskega jezika PINS v javansko zložno kodo

Avtor: Luka Šveigl

V diplomskem delu je predstavljen postopek dopolnitve prevajalnika za programski jezik PINS tako, da se ta prevaja v javansko zložno kodo. Programski jezik PINS je učni programski jezik, katerega prevajalnik smo implementirali pri predmetu Prevajalniki in navidezni stroji. V diplomskem delu so opisani registrski in skladovni procesorji, javanski navidezni stroj, posebej pa se posvetimo tudi sami nadgradnji prevajalnika za programski jezik PINS. Delovanje novega prevajalnika za programski jezik PINS je preverjeno na različnih testnih primerih, prav tako pa zložno kodo preveri tudi javanski navidezni stroj. Izmerjena je tudi hitrost delovanja novo prevedenih programov in primerjana z osnovnim prevajalnikom PINS ter Javo.

Ključne besede: prevajalniki, zložna koda, Java.

Abstract

Title: Compiling PINS to Java bytecode

Author: Luka Šveigl

This thesis presents a procedure for enhancing the compiler for the PINS programming language by compiling the source code into Java bytecode. PINS is an educational programming language, and its compiler was implemented as part of the Compilers and Virtual Machines course. The thesis describes register and stack machines, the Java vritual machine and focuses particularly on the upgrade of the compiler for the PINS programming language. The functionality of the new compiler for the PINS programming language is tested using various test cases, and the Java Virtual Machine also verifies the bytecode. Additionally, the performance of the newly translated programs is measured and compared to the basic PINS compiler and Java.

Keywords: compilers, bytecode, Java.

Poglavje 1

Uvod

Prevajalniki predstavljajo ključno orodje za programerje in so nepogrešljiv del razvojnega procesa pri ustvarjanju programske kode. Sposobnost preoblikovanja visokonivojskih programskih jezikov v strojne ukaze omogoča razvijalcem, da se osredotočajo na svoje zamisli, algoritme in logiko, ne pa na nizkonivojske podrobnosti. Naloga prevajalnikov je, da prevedejo visokonivojsko, človeku razumljivo kodo v računalniku razumljive strojne ukaze.

V tem diplomskem delu smo se osredotočili na dopolnitev prevajalnika za programski jezik PINS, s čimer smo omogočili prevajanje PINS programov v javansko zložno kodo. Tekom tega diplomskega dela splošno opišemo sestavo prevajalnikov, registrske in skladovne procesorje, podrobneje pa se posvetimo tudi javanskemu naveideznem storju, strukturi javanske zložne kode in dopolnjenemu prevajalniku za programski jezik PINS.

Poglavje 2

Pregled sorodnih del

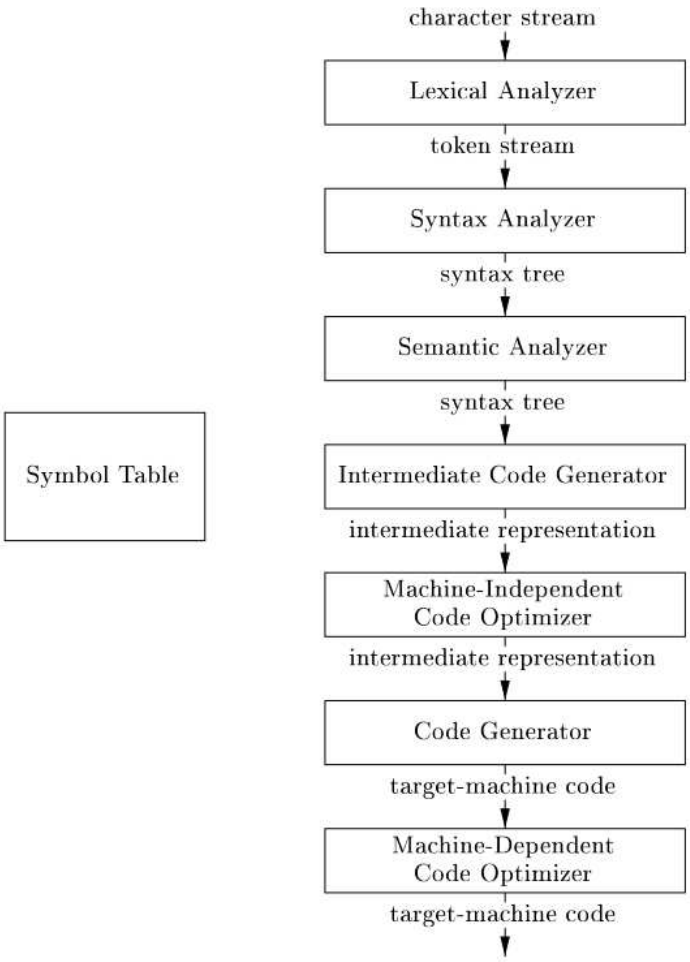
2.1 Prevajalniki

Prevajalniki so kosi programske opreme, katerih naloga je branje programa v enem jeziku - *izvorni* jezik (angl. *source language*) - in njegova prevedba v nek drug jezik - *ciljni* jezik (angl. *target language*) [12].

Sodobni prevajalniki so organizirani v več različnih faz, katere vsaka deluje na drugi stopnji abstrakcije jezika. Na sliki 2.1 je prikazan splošen diagram faz prevajalnika.

V splošnem postopek prevajanja delimo na dva glavna dela: *analiza* in *sinteza*. Pri postopku analize prevajalnik program pregleda in poskusi razumeti njegovo obliko ter pomen, pri postopku sinteze pa ga pretvori v ciljni jezik [11]. Analiza programa se deli na naslednje korake:

1. **Leksikalna analiza** (angl. *lexical analysis* ali *scanning*)
2. **Sintaksna analiza** (angl. *syntax analysis* ali *parsing*)
3. **Semantična analiza** (angl. *semantic analysis*)



Slika 2.1: Struktura prevajalnika [12]

Sinteza programa se ponavadi deli na naslednje korake:

1. **Generiranje vmesne kode** (angl. *intermediate code generation*)
2. **Optimizacija kode** (angl. *code optimization*)
3. **Generacija ciljnega programa** (angl. *code generation*)

V tej diplomski nalogi smo se osredotočali predvsem na postopek sinteze programske kode, analize ni bilo potrebno spreminjati.

2.2 Registrski in skladovni procesorji

Ko prevajalnik konča proces prevajanja, je njegov rezultat prevedena izvršljiva koda. Za izvrševanje ciljne kode je potreben nek stroj ali procesor, zmožen izvajanja takšne kode. Ko govorimo o registrskih in skladovnih procesorjih, govorimo o dveh osnovnih arhitekturah za izvrševanje ukazov in upravljanje podatkov.

Ti arhitekturi se primarno razlikujeta v načinu hranjenja podatkov, dostopu do operandov ukazov, naboru ukazov, velikosti kode in hitrosti izvajanja. V prihodnjih podpoglavjih so opisane ključne razlike med pristopoma.

2.2.1 Registrski procesorji

Registrski procesorji so procesorji, ki za shranjevanje podatkov običajno uporabljajo omejen nabor registrov. Registri so majhne podatkovne regije znotraj procesorja, ki omogočajo hiter dostop do podatkov.

Registrski procesorji za dostop do operandov ne potrebujejo posebnih nalagalnih ukazov, saj ukazi že sami vsebujejo podatke o izvoru in ponoru operandov. Izračun `a = b + c` lahko torej v registrski kodi predstavimo kot `add a, b, c` (ukaz je vzet kot primer iz zbirnega jezika ARM).

Ta pristop omogoča dobro preglednost pretoka izvajanja in s tem boljše priložnosti za optimizacijo prevedene kode, kot je dodeljevanje registrov

(angl. *register allocation*). Prav tako je v primerjavi s skladovno kodo potencialno potrebnih manj ukazov za prevedbo istega ukaza. Pri tem pristopu pa lahko pride do težav z omejenim številom registrov v procesorju, kar pomeni, da je potrebno več nalagalnih ukazov za dostopanje do vrednosti [5].

Večina procesorjev trenutno uporablja registrsko arhitekturo, najbolj znane arhitekture danes so x86, amd64, ARM, MMIX, System V in PowerPC.

2.2.2 Skladovni procesorji

Skladovni procesorji so procesorji, ki za delovanje uporabljajo sklad. Sklad je LIFO podatkovna struktura, kar pomeni da je zadnji podatek, ki je vanj vstopil, prvi, ki iz njega izstopi; direkten dostop do drugih podatkov ni možen.

Vsi podatki, ki so uporabljeni v skladovnih procesorjih, so na sklad naloženi z nalagalnimi operacijami kot sta *push* ali *load*, s sklada pa jih odstranjujejo ostale operacije. Pogosto so rezultati operacij vrnjeni nazaj na sklad. Primer `a = b + c` iz prejšnjega podpoglavja lahko v skladovni kodi predstavimo kot zaporedje ukazov `iload c, iload b, iadd, istore a` (primer je vzet iz javanske zložne kode).

Kot lahko vidimo v zgornjem primeru, ukazi v skladovni kodi pogosto implicitno operirajo nad vrhom sklada. To omogoča, da je skladovna koda bolj kompaktna, saj operacije pobirajo operande direktno iz vrha sklada, kar pa oteži razumevanje pretoka podatkov skozi izvajanje.

Ko govorimo o skladovnih arhitekturah so najbolj znane JVM, .NET VM, WASM in Forth.

2.3 Javanski navidezni stroj

Javanski navidezni stroj (angl. *Java Virtual Machine* ali JVM) je abstraktni računski stroj. Ko govorimo o JVM lahko govorimo o abstraktni specifikaciji, konkretni implementaciji in o tekoči instanci, saj vsak javanski program deluje na tekoči instanci konkretne implementacije specifikacije [13]. V priho-

dnjih poglavjih smo se osredotočili na abstraktno specifikacijo, ki je opisana v knjigi *The Java Virtual Machine Specification* [10].

Tako kot resnični računski stroji ima tudi JVM svoj nabor ukazov in zmožnost manipuliranja mnogih regij spomina med izvajanjem programov. Prednost JVM je, da ni odvisen od podrobnosti strojne opreme, prav tako pa ne pozna nobenih podrobnosti o programskem jeziku Java, čeprav sta obliki obeh, Jave in JVM povezani. Pozna le specifičen binarni format, imenovan razredni format (angl. *class file format*). Razredni format opisuje strukturo javanske razredne datoteke, katera vsebuje zložno kodo in ukaze za javanski navidezni stroj [10].

JVM skrbi tudi za varnost izvedenih programov. To dela tako, da ob procesu nalaganja razredne datoteke preveri tudi stroge sintaktične in strukturne omejitve kode.

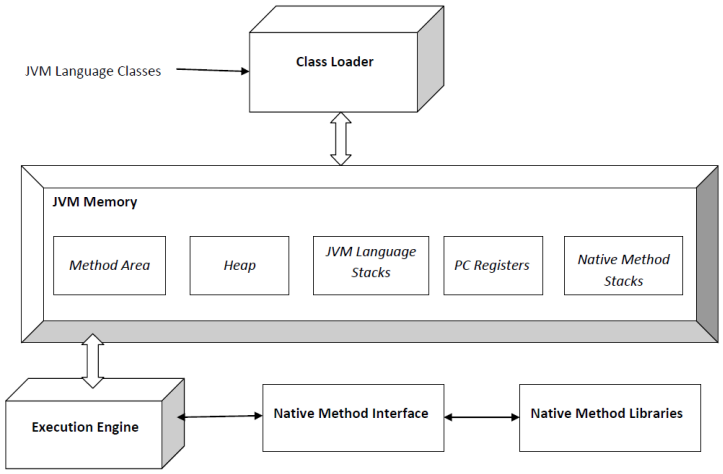
Javanski navidezni stroj kot platformo za izvajanje uporablja več programskih jezikov, med katerimi so najbolj poznani Java, Scala, Kotlin, Groovy in Clojure.

2.3.1 Struktura JVM

Specifikacija javanskega navideznega stroja ne predpisuje nobene dejanske implementacije, ampak le predpiše konstrukte, ki jih mora implementirati vsaka konkretna implementacija.

Kot je razvidno na sliki 2.2, JVM sestoji iz razrednega nalagalnika, raznih podatkovnih regij, ki jih dodeli navidezni stroj, izvajalnega stroja in vmesnika za izvorne metode.

- **Razredni nalagalnik** (angl. *class loader*) je podsistem javanskega navideznega stroja, ki je zadolžen za dinamično nalaganje razrednih datotek v javanski navidezni stroj. To omogoča, da izvajalni sistem ne potrebuje znanja o datotekah in datotečnih sistemih. Ob zagonu JVM se sprožijo trije razredni nalagalniki: bootstrap nalagalnik, nalagalnik razširitev in sistemski nalagalnik. Bootstrap nalagalnik skrbi za nalaganje jedrnih javanskih knjižnic in je edini nalagalnik, ki je napisan v



Slika 2.2: Struktura JVM [7]

strojni kodi. Nalagalnik razširitev naloži kodo iz razširitvenih direktorijev, sistemski nalagalnik pa naloži kodo iz *CLASSPATH*. Razredni nalagalnik ni del specifikacije JVM [6].

- **Področje metod** (angl. *method area*) je področje pomnilnika v JVM, ki shranjuje podatkovne strukture, ki pripadajo določenemu razredu. Te strukture so bazen konstant, bazen polj in podatki o metodah in konstruktorjih.

- **Kopica** (angl. *heap*) je področje pomnilnika v JVM, ki je deljeno med vsemi nitmi. Njen namen je, da se na njej dodelijo vse instance razredov in tabel. Pomnilnik, dodeljen na kopici, je počiščen z uporabo čistilca spomina.

- **JVM skladi** (angl. *JVM language stacks*) so skladi, ki pripadajo določeni niti. Vsak sklad hrani javanske klicne zapise in je analogen npr. skladu jezika C. Na skladu so shranjene lokalne spremenljivke, delni rezultati, prav tako pa igra vlogo pri klicu metod.

- **Registri programskih števcev** (angl. *PC registers*) so registri, ki

hranijo vrednost programskega števca. Vsaka nit ima svoj lasten programski števec.

- **Skladi izvornih metod** (angl. *native method stacks*) so konvencionalni skladi, poznani tudi pod imenom "C skladi", ki služijo podpori klicev izvornih metod, ki se izvajajo v strojni kodi procesorja, na katerem teče JVM.

- **Izvajalni stroj** (angl. *execution engine*) je podsistem JVM, ki je zadolžen za izvajanje naložene zložne kode. Vsebuje virtualni procesor, interpreter in JIT prevajalnik (angl. *Just-In-Time compiler*). Tudi izvajalni stroj ni del specifikacije JVM.

2.3.2 Javanska razredna datoteka

Prevedena koda, ki jo lahko izvaja javanski navidezni stroj je predstavljena s formatom, ki je neodvisen od strojne in programske opreme in se imenuje razredni format (angl. class file format). Vsaka taka datoteka natančno definira predstavitev natanko enega razreda ali vmesnika.

Razredna datoteka sestoji iz zaporedja bajtov. Vse 16-, 32- in 64-bitne strukture so sestavljene z branjem zaporednih 8-bitnih struktur, večbajtni podatki pa so vedno shranjeni v formatu debelga konca (angl. *big endian*) [10].

Notacija razredne datoteke

Deli razredne datoteke so predstavljeni z uporabo psevdostruktur, zapisanih v notaciji, podobni jeziku C, ki je razvidna spodaj. V razredni datoteki so vsi elementi zapisani zaporedno, brez podlaganja in poravnave:

```
ime_strukture {
    podatkovni_tip ime_elementa;   // Enostavni elementi
    podatkovni_tip ime_elementa[]; // Tabele
}
```

Osnovni podatkovni tipi pri zapisu struktur so sledeči:

- **u1:** nepredznačeni 1-bajtni podatek.
- **u2:** nepredznačeni 2-bajtni podatek.
- **u4:** nepredznačeni 4-bajtni podatek.

Poleg osnovnih podatkovnih tipov, uporabljenih v notaciji, javanska razredna datoteka vsebuje tudi tabele. Tabele so elementi, veliki 0 ali več elementov različnih velikosti. Predstavljene so z notacijo, podobno tabelam v programskem jeziku C. Ker tabele lahko vsebujejo elemente različnih velikosti, to pomeni, da pretvorba indeksa direktno v bajtni odmik ni mogoča [10].

Struktura razredne datoteke

Sama razredna datoteka je predstavljena z naslednjo strukturo:

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
```

```
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

- **magic** je število, ki identificira razredno datoteko; njegova vrednost je vedno 0xCAFEBABE.
- **minor_version, major_version** sta števili, ki identificirata verzijo prevajalnika in navideznega stroja. Skupaj ti števili predstavljata verzijo `v = major_version.minor_version`.
- **constant_pool_count** je število vnosov v bazen konstant + 1. Indeks v bazen konstant je veljaven če je večji od 0 in manj kot `constant_pool_count`.
- **constant_pool**[] je tabela struktur, ki vsebuje vse konstante, potrebne za izvajanje. Te konstante so naprimer črkovni nizi, imena razredov, spremenljivk in vmesnikov, ter velika števila.
- **access_flags** je število, ki je pridobljeno z izračunom maske števil, katera predstavljajo dostopna dovoljenja do razreda. Dostopna dovoljenja so prikazana v tabeli 2.1.
- **this_class** je število, ki predstavlja indeks v bazen konstant. Vnos v bazenu konstant mora biti `CONSTANT_Class_info` struktura, ki predstavlja razred ali vmesnik, katerega definira ta razredna datoteka.
- **super_class** je število, ki predstavlja indeks v bazen konstant. Vnos v bazenu konstant mora biti `CONSTANT_Class_info` struktura, ki predstavlja direktni starševski razred trenutnega razreda.
- **interfaces_count** je število, ki predstavlja število vnosov v seznamu vmesnikov.
- **interfaces**[] je seznam števil, kjer mora biti vsako število indeks v bazenu konstant. Vnos v bazenu konstant mora biti

`CONSTANT_Class_info` struktura, ki predstavlja starševski vmesnik trenutnega razreda.

- **fields_count** je število, ki predstavlja število vnosov v seznamu polj.
- **fields**[] je seznam `field_info` struktur, ki vsebujejo popoln opis polj v trenutnem razredu.
- **methods_count** je število, ki predstavlja število vnosov v seznamu metod.
- **methods**[] je seznam `method_info` struktur, ki vsebujejo popoln opis metod v trenutnem razredu.
- **attributes_count** je število, ki predstavlja število vnosov v seznamu atributov.
- **attributes**[] je seznam `attribute_info` struktur v trenutnem razredu.

ime	**vrednost**	**pomen**
ACC_PUBLIC	0x0001	Javni razred (dovoljen dostop iz zunanjega paketa).
ACC_FINAL	0x0010	Končni razred (ni podrazredov).
ACC_SUPER	0x0020	Metode nadrazreda so obravnavane drugače.
ACC_INTERFACE	0x0200	Vmesnik, ne razred.
ACC_ABSTRACT	0x0400	Abstraktni razred (instanca razreda ne more obstajati).
ACC_SYNTHETIC	0x1000	Sintetični razred (ni prisoten v strojni kodi).
ACC_ANNOTATION	0x2000	Anotacijski tip.
ACC_ENUM	0x4000	Enumeracijski tip.

Tabela 2.1: Dostopne zastavice razreda [10]

Bazen konstant

Ukazi javanskega navideznega stroja za svoje delovanje uporabljajo simbolične informacije, shranjene v bazenu konstant. Vsi zapisi v bazenu konstant sledijo splošnem zapisu, razvidnem spodaj:

```
cp_info {
    u1 tag;
    u1 info[];
}
```

Vsak zapis v bazenu konstant se začne z enobajtno oznako, ki nakazuje tip zapisa. Vsebina *info* tabele je odvisna od vrednosti oznake. Veljavne oznake in njihove vrednosti so prisotne v tabeli 2.2.

tip konstante	**vrednost**
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1
CONSTANT_MethodHandle	15
CONSTANT_MethodType	16
CONSTANT_InvokeDynamic	18

Tabela 2.2: Oznake elementov v bazenu konstant [10]

V sklopu te diplomske naloge smo uporabili sledeče strukture, ki se navezujejo na oznake:

- **CONSTANT_Class**:

```
CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}
```

  Element `name_index` predstavlja veljaven indeks v bazen konstant, kjer se nahaja `CONSTANT_Utf8_info` struktura, v kateri je shranjeno ime razreda ali vmesnika.

- **CONSTANT_Fieldref**, **CONSTANT_Methodref** in **CONSTANT_InterfaceMethodref**:

```
CONSTANT_Fieldref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT_Methodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT_InterfaceMethodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

  Element `class_index` predstavlja veljaven indeks v bazen konstant, kjer se nahaja `CONSTANT_Class_info` struktrura razreda, ki si lasti ta

element. `name_and_type_index` predstavlja indeks v bazen konstant kjer se nahaja `CONSTANT_NameAndType_info` struktura, ki povezuje ime in deskriptor polja ali metode.

- **CONSTANT_String**:

```
CONSTANT_String_info {
    u1 tag;
    u2 string_info;
}
```

Ta struktura predstavlja konstantne objekte tipa niz. Element `string_info` predstavlja indeks v bazen konstant, kjer se nahaja `CONSTANT_Utf8_info` struktura, ki vsebuje zaporedje Unicode znakov, na podlagi katerih je niz inicializiran.

- **CONSTANT_Integer** in **CONSTANT_Float**:

```
CONSTANT_Integer_info {
    u1 tag;
    u4 bytes;
}

CONSTANT_Float_info {
    u1 tag;
    u4 bytes;
}
```

Element `bytes` predstavlja 4 bajte, ki sestavljajo to število, v primeru števila v plavajoči vejici se vrednost izračuna po standardu IEEE 754.

- **CONSTANT_Long** in **CONSTANT_Double**:

```
CONSTANT_Long_info {
    u1 tag;
```

```
    u4 high_bytes;
    u4 low_bytes;
}

CONSTANT_Double_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
```

Strukturi predstavljata 8 bajtne numerične konstante. Vrednost konstante se izračuna na podlagi elementov `high_bytes` in `low_bytes` po enačbi

$$((long)high\_bytes << 32) + low\_bytes.$$

- **CONSTANT_NameAndType**:

```
CONSTANT_NameAndType_info {
    u1 tag;
    u2 name_index;
    u2 descriptor_index;
}
```

Struktura predstavlja polje ali metodo, brez podatka o pripadajaočem razredu. Element `name_index` predstavlja indeks v bazen konstant, na katerem se nahaja `CONSTANT_Utf8_info` struktura, ki predstavlja ime polja ali metode, element `descriptor_index` pa predstavlja indeks v bazen konstant, kjer se nahaja `CONSTANT_Utf8_info` struktura, ki predstavlja deskriptor polja ali metode.

- **CONSTANT_Utf8**:

```
CONSTANT_Utf8_info {
    u1 tag;
```

```
    u2 length;
    u1 bytes[length];
}
```

Struktura predstavlja konstanten niz Unicode znakov. Element `length` predstavlja število bajtov v tabeli `bytes`. Tabela `bytes` predstavlja bajte, ki gradijo niz.

- **CONSTANT_MethodType**:

```
CONSTANT_MethodType_info {
    u1 tag;
    u2 descriptor_index;
}
```

Struktura predstavlja tip metode. Element `descriptor_index` predstavlja indeks v bazenu konstant, na katerem se nahaja `CONSTANT_Utf8_info` struktura, ki vsebuje deskriptor metode.

Bazen polj

Vsako polje v razredni datoteki je opisano z spodnjo strukturo. Nobeni dve polje ne smeta imeti istega imena in deskriptorja.

```
field_info {
    u2             access_flags;
    u2             name_index;
    u2             descriptor_index;
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

- Element **access_flags** je število, izračunano na podalgi maske zastavic, ki predstavlja dostopna dovoljenja in lastnosti tega polja. Dostopne zastavice polja so prikazane v tabeli 2.3.

ime	vrednost	pomen
ACC_PUBLIC	0x0001	Javno polje (dovoljen dostop iz zunanjega paketa).
ACC_PRIVATE	0x0002	Zasebno polje (dovoljen dostop le znotraj razreda).
ACC_PROTECTED	0x0004	zaščiteno polje (dovoljen dostop le znotraj podrazredov).
ACC_STATIC	0x0008	Statično polje.
ACC_FINAL	0x0010	Končno polje (po stvarjenju objekta prireditev ni več mogoča).
ACC_VOLATILE	0x0040	Polje ne more biti shranjeno v predpomnilnik.
ACC_TRANSIENT	0x0080	Polje ni dostopano s strani upravljalca objektov.
ACC_SYNTHETIC	0x1000	Polje ni prisotno v izvorni kodi.
ACC_ENUM	0x4000	Polje je element enumeracije.

Tabela 2.3: Dostopne zastavice polja [10]

- Element **name_index** mora biti veljaven indeks v bazen konstant, na katerem se nahaja `CONSTANT_Utf8_info` struktura, v kateri se nahaja ime polja.

- Element **descriptor_index** mora biti veljaven indeks v bazen konstant, na keterem se nahaja `CONSTANT_Utf8_info` struktura, v kateri se nahaja deskriptor polja.

- Element **attributes_count** je število vnosov v tabelo atributov polja.

- Element **attributes**[] je tabela opcijskih atributov polja, sestavljena iz struktur, predstavljenih v poglavju atributi (2.3.2).

Bazen metod

Vsaka metoda, vključno z inicializacijsko metodo, je v javanski razredni datoteki opisana s spodnjo strukturo. Tako kot polja, nobeni dve metodi ne smeta imeti istega imena in deskriptorja.

```
method_info {
    u2             access_flags;
    u2             name_index;
    u2             descriptor_index;
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Pomeni elementov v strukturi metod imajo enak pomen kot elementi v strukturi polj, predstavljeni v prejšnjem poglavju (2.3.2). Edina razilka je v dostopnih zastavicah. Pomembnejše so predstavljene v tabeli 2.4.

ime	vrednost	pomen
ACC_PUBLIC	0x0001	Javna metoda (dovoljen dostop iz zunanjega paketa).
ACC_PRIVATE	0x0002	Zasebna metoda (dovoljen dostop le znotraj razreda).
ACC_PROTECTED	0x0004	Zaščitena metoda (dovoljen dostop le znotraj podrazredov).
ACC_STATIC	0x0008	Statična metoda.

Tabela 2.4: Dostopne zastavice metod [10]

Atributi

Atributi v javanski datoteki so prisotni pri razredih, poljih in metodah. Vsi atributi sledijo splošnemu formatu:

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

- Element **attribute_name_index** mora biti veljaven 16-bitni indeks v bazen konstant, na katerem se nahaja `CONSTANT_Utf8_info` struktura, ki vsebuje veljavno ime atributa.
- Element **attribute_length** predstavlja dolžino prihodnjih informacij v bajtih. Dolžina ne vsebuje začetnih 6 bajtov `attribute_info` strukture.

Javanske razredne datoteke poznajo veliko različnih prej-definiranih atributov, v naši diplomski nalogi pa smo se osredotočili le na "Code" atribut. Code atribut je atribut variabilne dolžine. Pripada `attributes` tabeli `method_info` strukture in vsebuje ukaze JVM, ki sestavljajo metodo.

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {   u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
```

```
    attribute_info attributes[attributes_count];
}
```

Pri atributu Code elementa `max_stack` in `max_locals` predstavljata maksimalno velikost skladu metode in maksimalno število lokalnih spremenljivk, ki jih metoda rabi. Element `code_length` predstavlja število bajtov, ki sestavljajo `code` tabelo. `code` tabela predstavlja dejansko zaporedje bajtov ukazov, ki metodo gradi. Ostala polja v strukturi v sklopu diplomske naloge niso bila pomembna.

2.3.3 Ukazi v javanski zložni kodi

Ukazi v javanski zložni kodi delujejo podobno kot zbirni jezik za navidezni stroj. Vsi ukazi so predstavljeni v sledeči obliki

```
menmonic
operand1
operand2
...
```

kjer je mnemonik človeku razumljivo ime ukaza, operandi pa so vrednosti, podane ukazu direktno. Poleg operandov ukazi operirajo večinoma tudi s skladom, iz katerega vrednosti pobirajo in na katerega vračajo rezultate. Poleg mnemonika ima vsak ukaz tudi operacijsko kodo (angl. *opcode*). Operacijska koda je unikatna šestnajstiška vrednost, ki predstavlja določen ukaz.

V javanski razredni datoteki mnemoniki niso prisotni, prisotne so le operacijske kode in operandi ukazov. Vzemimo na primer ukaz `ldc`, ki iz bazena konstant naloži vrednost na sklad. Ukaz `ldc` je v specifikaciji predstavljen kot

```
ldc
index
```

kjer je operacijska koda ukaza `ldc` enaka `0x12`. Zapis ukaza `ldc 15` v javanski razredni datoteki je torej `12 0F`.

Celoten nabor ukazov za javanski navidezni stroj je na voljo v knjigi *The Java Virtual Machine Specification* [10], poglavje 6.

Poglavje 3

Prevod v javansko zložno kodo

Pri razvoju prevajalnikov se lahko odločamo med mnogimi implementacijami zadnjega dela prevajalnika. Implementiramo lahko svoj interpreter, prevajalnik v strojno kodo, zanesemo pa se lahko tudi na uveljavljene platforme. Najbolj znane izmed teh so LLVM, CLR in JVM.

LLVM je zelo zrela infrastruktura, ki podpira implementacijo sprednjih delov za katerikoli jezik in zadnjih delov za katerokoli procesorsko arhitekturo. Poleg tega, LLVM skrbi tudi za mnoge optimizacije prevedenih programov [9].

Druga možnost je uporaba *Common Language Runtime* ali *CLR*. CLR je navidezni stroj, ki je del Microsoft .NET ogrodja, zadolžen za izvajanje .NET programov. Najbolj znani jeziki, ki uporabljajo CLR so C#, Visual Basic in F# [1].

V sklopu diplomske naloge pa smo se odločili za uporabo javanskega navideznega stroja. Ta odločitev ima mnoge prednosti. Med njimi so neodvisnost od ciljne platforme, ogromen javanski ekosistem, zrelost navideznega stroja, performanca zaradi JIT in odprtokodne implementacije, kot je OpenJDK.

Za implementacijo prevajalnika, katerega ciljna platforma je javanski navidezni stroj imamo na voljo več odprtokodnih knjižnic. Najpopularnejši sta knjižnici ASM in BCEL [2]. V glavnem se ti knjižnici razlikujeta v nivoju abstrakcije na katerem delujeta. ASM deluje na najnižjem nivoju in upo-

rabniku omogoča zelo granularen pregled in nadzor nad zložno kodo, BCEL (Byte Code Engineering Library) pa deluje na višjem nivoju kot ASM, za ceno slabše fleksibilnosti. Odločili smo se, da za svoj prevajalnik ne bomo uporabili nobenih knjižnic ampak bomo za celoten postopek prevajanja poskrbeli sami. Ta pristop zmanjša odvisnosti projekta od zunanjih faktorjev, prav tako pa predstavlja boljšo učno priložnost.

Novi zadnji del prevajalnika smo strukturirali v 3 glavne faze, kjer vsaka skrbi za svoj nivo abstrakcije delovanja. Te faze so podrobno opisane v prihodnjih poglavjih:

1. Referenčna analiza
2. Generiranje vmesne zložne kode
3. Pretvorba in zapisovanje zložne kode

Poleg novih faz prevajalnika pa so v prihodnjih poglavjih predstavljene tudi specifične funkcije jezika, ki so potrebovale posebno obravnavo.

3.1 Referenčna analiza

Programski jezik PINS podpira 2 funkciji, ki v javanskem navideznem stroju nista prisotni in sicer, kazalci ter gnezdene funkcije. Da smo lahko implementirali kazalce, smo jih simulirali z uporabo referenc, bolj natančno, tabel. Isti mehanizem smo uporabili tudi pri gnezdenih funkcijah. Sam mehanizem je bolj podrobno opisan v poglavjih 3.5 in 3.6. Pomembno je, da je za delovanje mehanizma potrebno vedeti, katere spremenljivke morajo biti referenčnega tipa.

Za identifikacijo spremenljivk, ki morajo biti referenčnega tipa, skrbi faza referenčne analize. V tej fazi se prevajalnik sprehodi po abstraktnem sintaksnem drevesu. ki predstavlja program in hrani sklad funkcij. Med sprehodom prevajalnik pazi na 2 scenarija:

1. Prevajalnik naleti na imenski izraz (`AstNameExpr`), pri katerem je globina trenutne funkcije večja od 1 in spremenljivka, ki definira ta izraz ni definirana znotraj te funkcije.

2. Prevajalnik naleti na imenski izraz (`AstNameExpr`), katerega starš je prefiksni izraz (`AstPreExpr`) z referenčnim operatorjem.

V prvem primeru prevajalnik spremenljivko, ki definira ime v imenskem izrazu, označi kot kandidata za pokritje. V drugem primeru pa prevajalnik spremenljivko označi kot kandidata za referenco.

3.2 Generiranje vmesne zložne kode

Druga faza novega zadnjega dela prevajalnika je generiranje vmesne zložne kode. Vhod te faze so abstraktna sintaksna drevesa, ki gradijo program, izhod pa so razredi, ki vsebujejo polja in metode. Prevajalnik se v tej fazi sprehodi po AST v 2 obhodih:

1. V prvem obhodu prevajalnik obišče vse globalne spremenljivke PINS programa in na podlagi tega generira polja razreda.

2. V drugem obhodu prevajalnik obišče vse definicije funkcij PINS programa in na podlagi tega generira zaporedje vmesnih ukazov.

Razlog za dva obhoda je ta, da mnogokrat ukazi v funkcijah uporabljajo globalne spremenljivke, zato morajo biti te generirane prve. Vse spremenljivke in metode so zapisane kot javne in statične. Razlog za to odločitev je ta, da je, če se v prihodnosti odločimo za nadgradnjo jezika z vnosnimi stavki (angl. *import statements*), potrebnih manj prilagoditev zadnjega dela, saj so spremenljivke avtomatsko dostopne.

Vmesni ukazi v tej fazi so analogni dejanskim ukazom, ki jih uporablja javanski navidezni stroj. Vsak ukaz ima svojo operacijsko kodo, operande in bajtni odmik od začetka metode. Vzemimo za primer nalagalne ukaze v javanskem navideznem stroju. To so ukazi `lload`, `iload` in podobni. V vmesni

kodi so predstavljeni kot en ukaz in sicer `BtcLOAD`. `BtcLOAD` ukaz v vmesni kodi prejme bajtni odmik, indeks lokalne spremenljivke in tip spremenljivke. Na podlagi teh dejavnikov se ukazu dodeli pravilna operacijska koda, ki je uporabljena v naslednji fazi prevajanja. Po tem principu delujejo vsi vmesni ukazi, edina izjema je ukaz `ldc` in njegove različice, `ldc_w` in `ldc2_w`.

Ukazi `ldc` so posebni zato, ker za svoje delovanje potrebujejo informacije o bazenu konstant, ki pa v tej fazi še niso na voljo. Bolj natančno, ta ukaz za delovanje potrebuje indeks v bazenu konstant, na katerem se nahaja konstanta, ki jo želimo naložiti na sklad. Zaradi tega v tej fazi ta ukaz dobi le vrednost konstante in njen tip, operacijska koda in indeks pa se nastavita šele v naslednji fazi, ko je indeks konstante v bazenu konstant znan.

3.3 Zapisovanje zložne kode

Zadnja faza novega zadnjega prevajalnika je zapisovanje zložne kode. Vhod te faze prevajanja je vmesna koda, ki predstavlja razrede in metode, ki sestavljajo programe, izhod pa je javanska razredna datoteka, ki vsebuje javansko zložno kodo. To fazo sestavljata naslednji 2 podfazi:

1. Pretvorba vmesne kode: Ta podfaza najprej pretvori vsa polja v razredu, nato pa vse metode.

2. Zapisovanje zložne kode: Ta podfaza vzame pretvorjeno kodo in jo zapiše v datoteko.

3.3.1 Pretvorba polj

Kot smo opisali v poglavju Bazen polj, so polja v javanski razredni datoteki predstavljena z določeno strukturo `field_info`. Ta struktura razredu pove karakteristike polja, kar pa za dostop do polja ni dovolj. Za dostop do polja sta potrebni še strukturi `NameAndType_info` in `Fieldref_info`. Struktura `NameAndType_info` povezuje ime nekega elementa (`Utf8_info`) v javanski

zložni kodi z njegovim deskriptorjem (`Utf8_info`), struktura `Fieldref_info` pa povezuje razred (`Class_info`) z nekim poljem (`NameAndType_info`).

Če z pregledovalnikom razrednih datotek `javap` pogledamo preprost javanski program 3.1, lahko vidimo izpis 3.1. Vrstice, obarvane z rdečo prikazujejo potrebne strukture za dostop polja, nepomembni podatki pa so odstranjeni.

```
public class fieldDemo {
    static long x;
    public static void main(String[] args){
        x = 1;
    }
}
```

Program 3.1: Primer razreda z poljem

V podfazi pretvorbe polj prevajalnik torej poskrbi za kreiranje strukture polja `field_info` in dodajanje ustreznih struktur za dostop do polja v bazen konstant.

```
Constant pool:
   #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #5:#6          // "<init>":()V
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Fieldref           #8.#9          // fieldDemo.x:J
   #8 = Class              #10            // fieldDemo
   #9 = NameAndType        #11:#12        // x:J
  #10 = Utf8               fieldDemo
  #11 = Utf8               x
  #12 = Utf8               J
  #13 = Utf8               Code
  #14 = Utf8               main
  #15 = Utf8               ([Ljava/lang/String;)V
{
  static long x;
    descriptor: J
    flags: (0x0008) ACC_STATIC
```

Slika 3.1: Izpis javap programa 3.1

3.3.2 Pretvorba metod

Podobno kot polja, tudi metode za pravilno delovanje potrebujejo mnoge podporne vnose v bazen konstant. Ti vnosi so sledeči:

- `Utf8_info` z imenom metode.
- `Utf8_info` z deskriptorjem metode.
- `NameAndType_info`, ki povezuje ime in deskriptor metode.
- `Methodref_info`, ki povezuje razred, kateremu metoda pripada in `NameAndType_info` metode.
- `Utf8_info`, ki vsebuje niz "Code". Ta niz potrebuje `Code_attribute` metode, ki vsebuje vse ukaze, ki metodo sestavljajo.

Vzemimo za primer preprost program 3.2, kjer metoda `setX` nastavi globalno spremenljivko iz prejšnjega primera na vrednost 1. V izpisu 3.2 programa `javap` je prikazan bazen konstant, polja in metode, ki razred sestavljajo, nepomembni podatki o razredu pa so za preglednost odstranjeni. Vrstice, obarvane z rdečo prikazujejo elemente, pomembne za pravilno delovanje metode `setX`.

```
public class methodDemo {
    static long x;
    public static void main(String[] args) {
        setX();
    }

    public static void setX() {
        x = 1;
    }
}
```

Program 3.2: Primer metod v razredu

```
Constant pool:
   #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #5:#6          // "<init>":()V
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Methodref          #8.#9          // methodDemo.setX:()V
   #8 = Class              #10            // methodDemo
   #9 = NameAndType        #11:#6         // setX:()V
  #10 = Utf8               methodDemo
  #11 = Utf8               setX
  #12 = Fieldref           #8.#13         // methodDemo.x:J
  #13 = NameAndType        #14:#15        // x:J
  #14 = Utf8               x
  #15 = Utf8               J
  #16 = Utf8               Code
  #17 = Utf8               main
  #18 = Utf8               ([Ljava/lang/String;)V
{

  ...podatki o poljih, konstruktorjih in metodah...

  public static void setX();
    descriptor: ()V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=0, args_size=0
         0: lconst_1
         1: putstatic     #12                 // Field x:J
         4: return
}
```

Slika 3.2: Izpis javap programa 3.2

V podfazi pretvorbe metod prevajalnik torej v bazen konstant doda vse potrebne vnose, ki omogočajo pravilno delovanje metod. Poleg tega prevajalnik metodi doda tudi `Code_attribute`, v katerem se nahajajo pretvorjeni ukazi, ki metodo gradijo.

3.4 Prevajanje tabel

Tabele v programskem jeziku PINS so zelo podobne tabelam v programskem jeziku C. To pomeni, da ko je tabela definirana, lahko vanjo začnemo pisati, kot je razvidno v programu 3.3. Za razliko od PINS in C, pa je v Javi in posledično tudi v javanski zložni kodi, tabele pred uporabo potrebno alocirati in inicializirati, kot je razvidno v programu 3.4.

```
fun main(): void = ({
    x[1] = 3;
}
where
    var x: [5]int;
);
```

Program 3.3: Primer tabel v jeziku PINS

```
public static void main(String[] args) {
    long[] x = new long[5];
    x[1] = 3;
}
```

Program 3.4: Primer tabel v jeziku Java

Dejstvo, da se pristopa v jezikih razlikujeta pomeni, da je bilo potrebno implementirati inicializacijo tabel brez, da bi spremenili sam programski jezik. Tabele v programskem jeziku PINS se lahko pojavijo v 2 scenarijih:

- tabela kot globalna spremenljivka in
- tabela kot lokalna spremenljivka.

V primeru, da je tabela globalna spremenljivka, vključimo njeno inicializacijo na čisti začetek metode `main`. Ker je metoda `main` vstopna točka programa, tako zagotovimo, da bodo vse globalne tabele zagotovo inicializirane pred uporabo drugje v programu.

V primeru, da je tabela lokalna spremenljivka, vključimo njeno inicializacijo na čisti začetek metode, kateri pripada. Tako kot pri globalnih tabelah tako zagotovimo, da je tabela inicializirana pred njeno uporabo.

Ta pristop je viden na programu 3.3 in njegovem izpisu 3.3 z orodjem `javap`, kjer so z rdečo obarvane inicializacijske vrstice.

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC
  Code:
    stack=256, locals=256, args_size=1
       0: ldc2_w        #16                 // long 5l
       3: l2i
       4: newarray       long
       6: astore        0
       8: aload         0
      10: lconst_1
      11: l2i
      12: ldc2_w        #18                 // long 3l
      15: lastore
      16: return
```

Slika 3.3: Izpis javap programa 3.3

3.5 Prevajanje kazalcev

Tako kot jeziki C, C++ in C# tudi programski jezik PINS omogoča uporabo kazalcev. Kazalci so osnovni gradniki mnogih programskih jezikov, posebno jezikov, ki omogočajo ročno upravljanje s pomnilnikom. Ko govorimo o kazalcih, v bistvu govorimo o spremenljivkah, ki namesto konkretnih vrednosti hranijo pomnilniške naslove. Omogočajo indirekten dostop do podatkov shranjenih na lokaciji, na katero kazalec kaže.

Pri prevajanju kazalcev naletimo na omejitve pri izbiri javanskega navideznega stroja kot ciljne platforme. Za razliko od platform kot so WASM in CLR, Java in javanski navidezni stroj ne podpirata ročnega upravljanja s pomnilnikom in posledično tudi uporabe kazalcev. To dejstvo pomeni, je bilo kazalce potrebno simulirati. Ker javanski navidezni stroj podpira reference, del katerih so tudi tabele, smo težavo rešili z uporabo le teh.

V programskem jeziku PINS lahko kazalce ustvarimo na 2 načina:

- direktna alokacija pomnilnika in
- referenciranje druge spremenljivke.

Oba primera sta vidna v programu 3.5.

```
fun main(): void = ({
    base = 5;
    refPointer = ^base;
    allocPointer = new 10;
}
where
    var base: int;
    var refPointer: ^int;
    var allocPointer: ^int;
);
```

Program 3.5: Primer kazalcev v jeziku PINS

V primeru, da kazalcu direktno dodelimo pomnilnik z uporabo ključne besede `new`, je implementacija kazalcev trivialna. V zložni kodi preprosto ustvarimo tabelo prave velikosti in jo shranimo v spremenljivko, kot je razvidno v izpisu 3.4, če z programom `javap` pogledamo program 3.5.

```
public static void main(java.lang.String[]);
  Code:
     0: ldc2_w        #16                 // long 1l
     3: l2i
     4: newarray       long
     6: astore        0
     8: ldc2_w        #18                 // long 1l
    11: l2i
    12: newarray       long
    14: astore        1
    16: ldc2_w        #20                 // long 1l
    19: l2i
    20: newarray       long
    ... ukazi ...
    40: astore        1
    42: aload         2
    44: iconst_0
    45: ldc2_w        #24                 // long 10l
    48: l2i
    49: newarray       long
    51: astore        2
    53: return
```

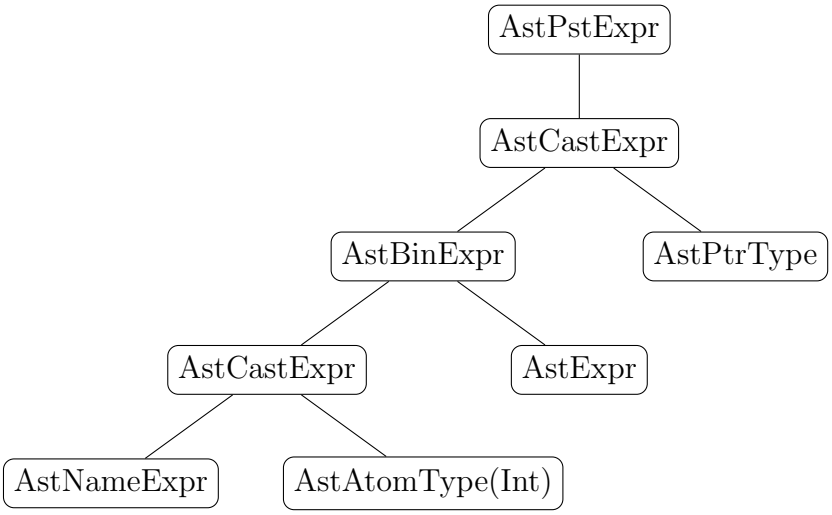
Slika 3.4: Izpis javap programa 3.5

Ta pristop omogoča, da lahko drugi kazalci brez težav dobijo referenco na tako inicializirani kazalec in spreminjajo njegovo vsebino. Poleg tega lahko s

takšnim pristopom implementiramo kazalčno aritmetiko, pri kateri namesto računanja odmika v pomnilniku računamo odmik indeksa. To je razvidno v programu 3.6, kjer kazalec pretvorimo v celoštevilski tip, mu prištejemo 2 in ga pretvorimo nazaj v kazalec. Na ta način vrednost shranimo v kazalec (tabelo) na indeksu 2. Ker v javanskem navideznem stroju tabel ne moremo pretvarjati v primitivne tipe mora prevajalnik prepoznati AST, razviden na sliki 3.5 in iz nje izluščiti pomembne informacije, kot so ime spremenljivke in odmik.

```
fun main(): void = ({
    allocPointer = new 10;
    ((allocPointer : int) + 2 : ^int)^ = 2;
}
where
    var allocPointer: ^int;
);
```

Program 3.6: Primer kazalčne aritmetike v jeziku PINS



Slika 3.5: AST dostopa do vrednosti v kazalcu

V primeru, da kazalec inicializiramo z referenciranjem druge spremenljivke, pa je implementacija veliko bolj kompleksna, saj ni dovolj, da le ka-

zalec predstavimo kot tabelo, ampak je potrebno kot tabelo predstaviti tudi spremenljivko, na katero kazalec kaže. To je potrebno zato, da obe spremenljivki referencirata isti objekt in tako tudi dostopata do in spreminjata iste podatke. V ta namen smo razvili fazo referenčne analize, ki skrbi za identifikacijo takih spremenljivk.

```
fun main(): void = ({
    base = 3;
    ptr = ^base;
}
where
    var base: int;
    var ptr: ^int;
);
```

Program 3.7: Primer referenciranja v jeziku PINS

Če za primer vzamemo program 3.7 in njegov prevod pregledamo z programom `javap`, lahko vidimo izpis 3.6. Na začetku metode najprej v obe spremenljivki shranimo tabelo velikosti 1. Nato v prvo tabelo (spremenljivka `base`) shranimo vrednost 3, kar je v izpisu predstavljeno z modro barvo, zatem pa v kazalec shranimo referenco na prvo tabelo, kar je predstavljeno z rdečo. Nekaj generiranih ukazov med označenima deloma je odveč, kar kaže na napako v prevajalniku. K sreči ta napaka ne vpliva na pravilnost delovanja, ampak le na prostorsko in časovno zahtevnost.

Implementacija kazalcev se je izkazala za eno izmed težjih funkcij programskega jezika PINS. V njej so še prisotne napake, a pristop deluje dovolj dobro. Drug pristop, ki bi ga lahko izbrali je uporaba ovijalskih razredov (angl. *wrapper classes*) namesto tabel. Ta pristop bi bil morda bolj fleksibilen, zaradi možnosti implementiranja obnašanja ovijalskih razredov. Po premisleku se je izkazalo, da dodana fleksibilnost ni vredna veliko večje kompleksnosti implementacije, zato smo ta pristop opustili.

```
public static void main(java.lang.String[]);
  Code:
     0: ldc2_w        #16                 // long 1l
     3: l2i
     4: newarray       long
     6: astore        0
     8: ldc2_w        #18                 // long 1l
    11: l2i
    12: newarray       long
    14: astore        1
    16: aload         0
    18: iconst_0
    19: ldc2_w        #20                 // long 3l
    22: lastore
    23: aload         1
    25: iconst_0
    26: aload         0
    28: iconst_0
    29: laload
    30: aload         0
    32: astore        1
    34: return
```

Slika 3.6: Izpis javap programa 3.7

3.6 Prevajanje gnezdenih funkcij

Poleg kazalcev je ena izmed funkcionalnosti jezika PINS, ki ni podprta s strani javanskega navideznega stroja tudi gnezdenje funkcij. Gnezdene funkcije (angl. *nested functions*), poznane tudi pod imenom notranje funkcije (angl. *inner functions*) so funkcije, ki so definirane znotraj telesa neke druge funkcije in imajo dostop do spremenljivk in območja vidnosti starševske funkcije. Glavne lastnosti gnezdenih funkcij so:

- **Območje vidnosti**: Gnezdene funkcije lahko dostopajo do spremenljivk in parametrov iz zunanje (angl. *outer* ali *enclosing*) funkcije. To je poznano tudi pod imenom leksikalno določanje obsega (angl. *lexical scoping*) ali pokritje (angl. *closure*) in omogoča funkciji, da si "zapomni" stanje znanje funkcije ob klicu.
- **Enkapsulacija**: Notranje funkcije lahko uporabimo za enkapsuliranje funkcionalnosti. Če je funkcija uporabljena le znotraj konteksta druge funkcije, jo lahko definiramo kot gnezdeno funkcijo, kar izboljša berljivost in ohranja kodo bolj organizirano in modularno.
- **Berljivost**: Gnezdenje funkcij omogoča gručenje funkcionalnosti v logične programske enote, kar izboljša berljivost in vzdržljivost kode.
- **Omejena vidnost**: Tipično je gnezdena funkcija vidna le v območju vidnosti starševske funkcije in ni dostopna zunaj nje. To preprečuje imenske konflikte in skrbi za čistost globalnega območja imen (angl. *global namespace*).

Preprost primer gnezdenih funkcij v programskem jeziku je viden v programu 3.8.

```
fun outerFunction(): void = ({
    x = 3;
    innerFunction();
}
```

```
where
    var x: int;
    fun innerFunction(): void = {
        putInt(x);
    };
);
```

Program 3.8: Primer gnezdenih funkcij v jeziku PINS

Pri implementaciji gnezdenih funkcij bi se lahko poslužili veliko različnih pristopov, od katerih vsak ima svoje prednosti in slabosti. Ti pristopi so:

- **Anonimni razred**: Najboljši približek gnezdenim funkcijam v programskem jeziku Java so pokritja. Pokritja v Javi dosežemo z uporabo anonimnih razredov, kjer ob definiciji notranje funkcije definiramo razred, kateremu ta funkcija pripada. Ta pristop je viden v programu 3.9. Kot lahko vidimo, je pri tem pristopu potrebno dodati še vmesnik in ob definiciji instance tega objekta implementirati njegove metode. To pa oteži sam proces prevajanja, saj je potrebno generirati dodatno javansko razredno datoteko, v kateri se nahaja ta vmesnik.

```
interface Inner {
    void run();
}

class anonymousDemo {
    public void outer() {
        Inner inner = new Inner() {
            @Override
            public void run() {
                System.out.println(this.
                    getClass().getName());
            }
        }
```

```
            inner.run();
        }
    }
```

Program 3.9: Primer anonimnih razredov v jeziku Java

- **Vmesnik Runnable**: Drug način za implementacijo gnezdenih funkcij je javanski vmesnik `Runnable`. Vmesnik `Runnable` se navadno uporablja pri implementiranju večnitnih aplikacij. Ta vmesnik lahko uporabimo na 2 načina, kot je prikazano v programu 3.10.

```
public class runnableDemo {
    public static void main(String[] args) {
        // Example 1: Anonymous class
        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                System.out.println("anonymous");
            }
        };

        // Example 2: Lambda expression
        Runnable r2 = () -> {
            System.out.println("lambda");
        };

        r1.run();
        r2.run();
    }
}
```

Program 3.10: Primer vmesnika Runnable jeziku Java

Kot je razvidno v zgornjem programu, lahko vmesnik `Runnable` inicializiramo z uporabo anonimnih razredov in lambda izrazov. V primeru, da uporabimo anonimne razrede, zopet naletimo na iste težave kot pri prvem pristopu. V primeru, da uporabimo lambda izraze pa je potrebno gereriranje atributa `BootstrapMethods`, ki vsebuje klice na `java/lang/invoke/LambdaMetaFactory.metafactory` in mnoge druge metode. Tudi to precej poveča kompleksnost implementacije, zato se tega pristopa nismo poslužili.

- **Lambda dvigovanje**: Zadnji pristop, ki smo ga pogledali je lambda dvigovanje. Lambda dvigovanje je tehnika, uporabljena v kontekstu lambda računa, kadar imamo opravka z gnezdenimi funkcijami ali pokritji [8]. Tehnika deluje tako, da notranjo funkcijo spremeni v funkcijo v najvišjem nivoju (angl. *top-level function*), zdraven pa ji doda še dodatne parametere, ki zajamejo okolje starševske funkcije. V programu 3.11 je prikazan primer lambda dvigovanja v psevdokodi. Ta pristop za prevajanje gnezdenih funkcij uporablja na primer programski jezik Scala [3]. To lahko vidimo tudi z izpisom faz v prevajalniku `scalac` in sicer z ukazom `scalac -Xshow-phases`.

```
fun outer():
    var1 = 3;
    fun inner(par1):
        print(var1, par1);
    inner(2);

# After lambda lifting
fun outer():
    var1 = 3;
    inner‘(var1, 2);

fun inner‘(var1, par1):
```

```
print(var1, par1);
```

Program 3.11: Primer lambda dvigovanja

Po temeljitem premisleku smo se odločili, da bomo za implementacijo gnezdenih funkcij uporabili tehniko lambda dvigovanja. Do te odločitve smo prišli zato, saj je ta pristop konceptualno zelo preprost in testiran v dejanskih prevajalnikih, kljub dejstvu da moderni prevajalniki za Haskell in OCaml uporabljajo tehniko pretvorbe pokritij (angl. *closure conversion*) [4].

V sklopu implementacije je bilo potrebno poskrbeti za naslednji dve funkcionalnosti:

- dvigovanje funkcij in
- podajanje dodatnih parametrov.

Za dvigovanje funkcij smo poskrbeli tako, da ko prevajalnik naleti na funkcijo, katere globina je več kot 1, jo v javanski zložni kodi predstavi kot vrhnjo funkcijo, njenemu imenu pa na začetek doda ime prejšnje funkcije in znak `$`. To lahko vidimo v izpisu zložne kode 3.7 programa 3.12.

```
fun main(): void = ({
    inner();
}
where
    fun inner(): void = none;
);
```

Program 3.12: Primer lambda dvigovanja v jeziku PINS

Druga funkcionalnost, ki jo je bilo potrebno implementirati v sklopu gnezdenih funkcij je bila podajanje dodatnih parametrov. Ko prevajalnik naleti na definicijo gnezdene funkcije, ustvari metodni objekt na najvišjem nivoju. Ko je objekt ustvarjen, prevajalnik vse lokalne spremenljivke in parametre starševske metode novi metodi poda kot parametre. Ta pristop je razviden v izpisu podpisov funkcij 3.8 programa 3.13.

```
  public static void main(java.lang.String[]);
    Code:
       0: invokestatic  #21  // Method tmptest.main$inner:()V
       3: return

  public static void main$inner();
    Code:
       0: aconst_null
       1: return
```

Slika 3.7: Izpis javap programa 3.12

```
fun main(): void = ({
    x = 2;
    inner(1);
}
where
    var x: int;
    fun inner(y: int): void = putInt(x + y);
);

fun putInt(i: int): void = none;
```

Program 3.13: Primer podajanja argumentov v jeziku PINS

```
public final class tmptest {
  public static void main(java.lang.String[]);
  public static void main$inner(long[], long);
  public static void putInt(long);
}
```

Slika 3.8: Izpis javap programa 3.13

Iz zgornjega izpisa je razvidno, da funkcija `inner` (`main$inner`) prejme 2 argumenta. Prvi argument je tabela, torej referenčni tip, drugi element pa je primitivnega tipa long. Ker morajo notranje funkcije biti zmožne spreminjati vrednosti v zunanjih funkcijah, je podane parametre, ki predstavljajo okolje zunanje funkcije, potrebno podati kot referenčne tipe. To je razlog, da je prvi argument funkcije `inner` podan kot tabela tipa long. Prav tako je tudi spremenljivka `x` v zunanji funkciji predstavljena z tabelo, tako kot to storimo pri kazalcih. Tudi tu nam pri identifikaciji takšnih spremenljivk pomaga faza referenčne analize.

Tudi prevajanje gnezdenih funkcij se je izkazalo za eno izmed težjih funkcionalnosti programskega jezika PINS. V implementaciji so še prisotne napake, specifično pri senčenju spremenljivk, a pristop deluje dovolj dobro in bi z več časa lahko bil implementiran boljše.

Poglavje 4

Meritve in testiranje

Delovanje novega prevajalnika smo testirali na treh algoritmih: quicksort, kadanov algoritem in fibonacci. Poleg delovanja prevajalnika smo testirali tudi hitrost prevedenih programov z in brez uporabe JIT prevajalnika v JVM. Poleg tega smo na istih algoritmih izmerili hitrost tudi enakim algoritmom, implementiranim v PINS in Javi (z in brez JIT). Vsi algoritmi, pognani na javanskem navideznem stroju, so bili pognani z zastavico `-Xverify:none`.

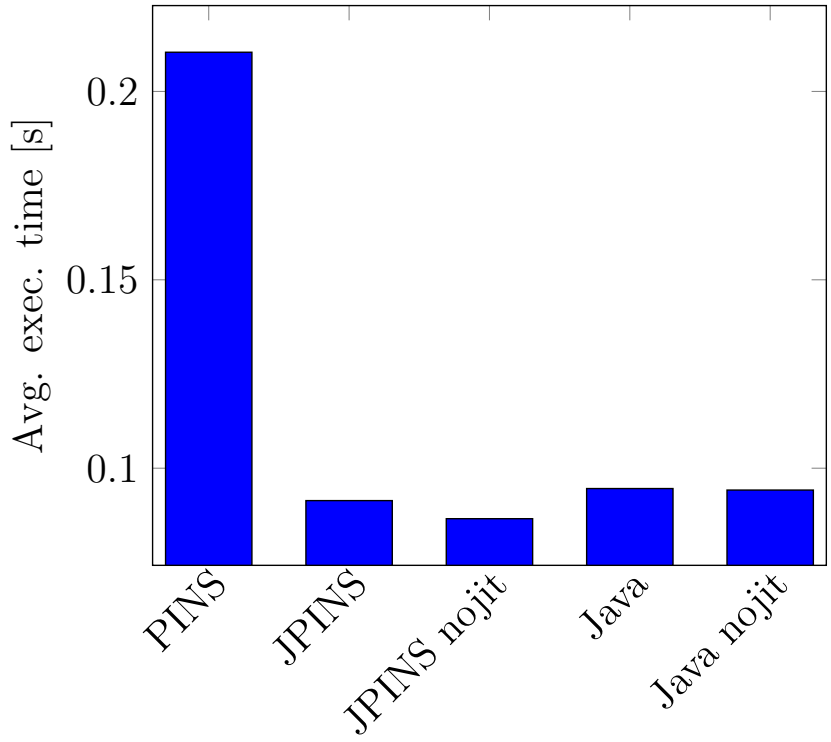
4.1 Hitro urejanje

S hitrim urejanjem (angl. *quicksort*) smo želeli preveriti klice funkcij, podajanje tabel kot parametrov, kontrolne strukture in gnezdene funkcije. Algoritem smo izvajali na tabeli z 20 števili

```
{5,2,7,3,9,1,8,4,6,11,0,24,3,1,5,12,1,9,13,69}.
```

Da smo zagotovili čim manjšo možnost, da osamelci pokvarijo rezultate meritev, smo program izvedli 10.000-krat in izračunali povprečni čas trajanja. Rezultati so vidni na sliki 4.1.

Kot pričakovano, je bila najpočasnejša interpretirana implementacija programskega jezika PINS. Zanimivo je, da je čista javanska implementacija algoritmov rahlo počasnejša od kode, prevedene z novim prevajalnikom. Dejstvo

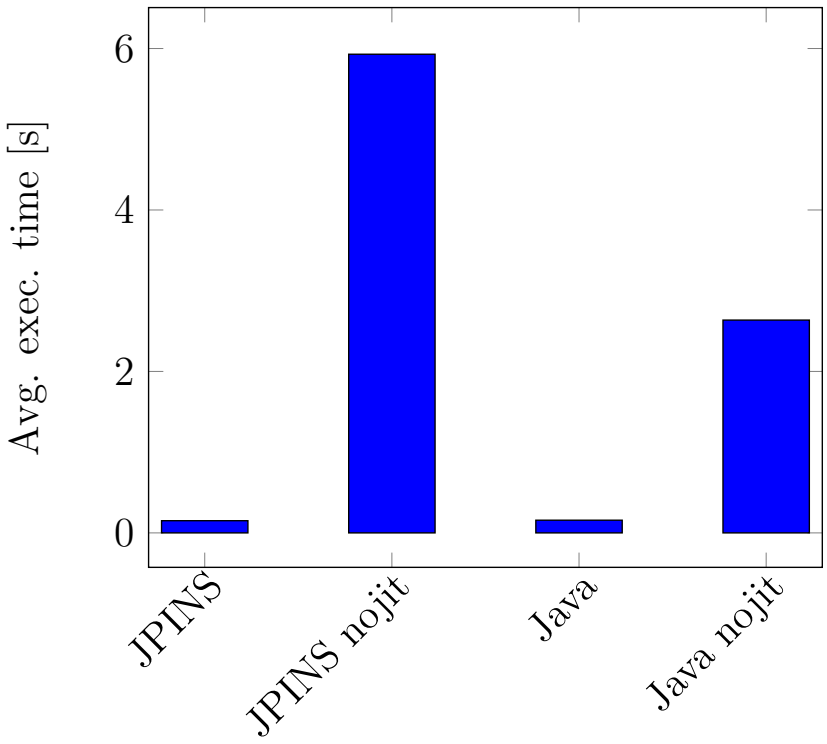


Slika 4.1: Quicksort algoritem

za to lahko pojasnimo z razlikami v algoritmih. Še ena zanimivost, ki jo opazimo v rezultatih, je dejstvo, da je izvedba programa brez JIT prevajalnika v primeru Jave in PINS rahlo hitrejša. Razlog za to je najverjetneje ogrevalni čas (angl. *warm-up time*) JIT prevajalnika. JIT prevajalnik v tako kratkem algoritmu tudi nima vpliva, tako da njegova uporaba program le upočasni.

Kot zanimivost smo izmerili čas trajanja tudi za quicksort, ki operira nad tabelo 100.000 naključnih elementov, saj smo tako zagotovili, da se bo JIT prevajalnik aktiviral. Rezultati so vidni na sliki 4.2.

Kot lahko vidimo, je razlika med izvajalnim časom algoritmov, pognanih z in brez JIT prevajalnika ogromna. Zanimivo je, da je PINS implementacija algoritma z uporabo JIT prevajalnika rahlo hitrejša od javanske različice, v primeru, da algoritem izvajamo brez uporabe JIT, pa je javanska različica približno dvakrat hitrejša.



Slika 4.2: Quicksort algoritem z 100.000 števili

4.2 Kadanov algoritem

Kadanov algoritem je algoritem, ki se uporablja za reševanje problema največje podtabele (angl. *maximum subarray problem*). Problem največje tabele je naloga iskanja podtabele z največjim seštevkom elementov znotraj neke enodimenzionalne tabele.

S Kadanovim algoritmom smo tako kot s quicksortom preverjali podajanje tabel kot parametrov (kazalci), kontrolne strukture in klice ter vračanje vrednosti funkcij. Tako kot quicksort, smo tudi kadanov algoritem izvajali na tabeli z 20 števili

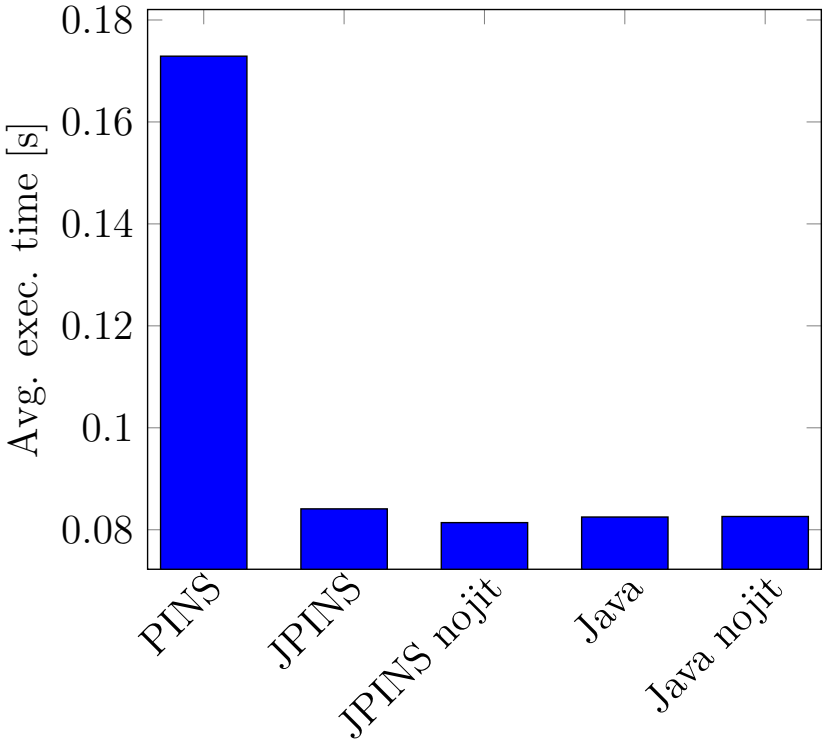
```
{-2,-1,-3,4,1,-1,6,-5,1,0,-3,-1,3,2,-1,-5,5,3,-4,0}.
```

Tudi kadanov algoritem smo izvedli 10.000-krat in tako zagotovili čim manjše odstopanje od realnosti. Rezultati so vidni na sliki 4.3.

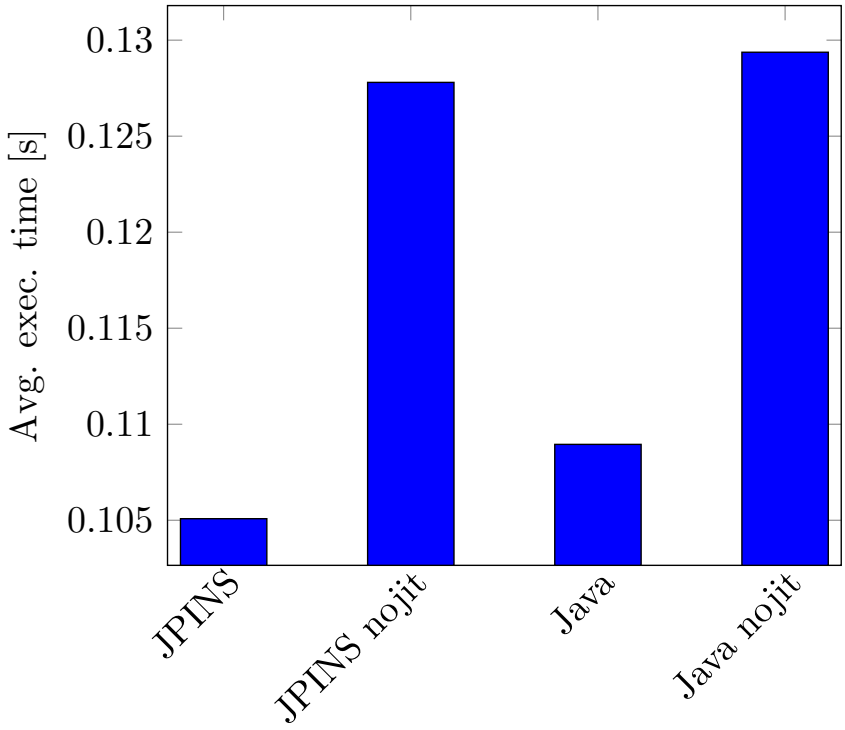
Kot lahko vidimo na grafu, je interpretirana različica PINS zopet močno počasnejša od ostalih. Za razliko od quicksort algoritma pa se javanska različica PINS in čista javanska implementacija občutno manj razlikujeta.

Tudi pri kadanovem algoritmu smo izmerili čas trajanja pri operiranju nad tabelo z 100.000 naključnih elementov. Rezultati so vidni na sliki 4.4.

Kot je razvidno na grafu 4.4, je tudi v primeru kadanovega algoritma razlika izvajalnega časa izvedbe algoritmov z in brez uporabe JIT ogromna. V obeh primerih opazimo, da je naša PINS implementacija rahlo hitrejša od javanske, kar je lahko posledica raznih manjših razlik v algoritmih in števil v tabeli.



Slika 4.3: Kadanov algoritem



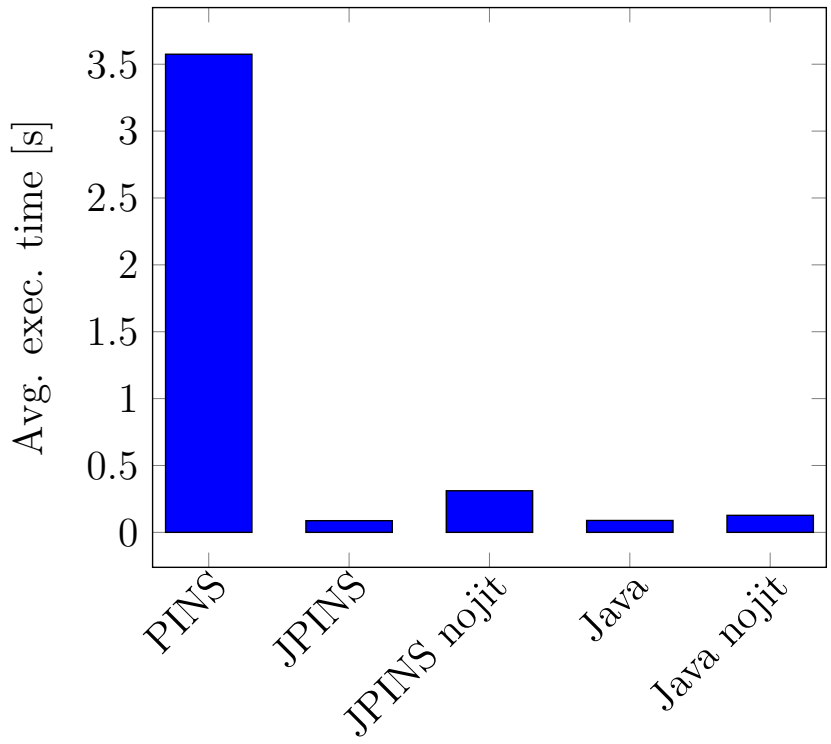
Slika 4.4: Kadanov algoritem z 100.000 števili

4.3 Fibonacci

Z izračunom n-tega fibonaccijevega števila smo dodatno testirali rekurzijo, poleg tega pa smo potrebovali tudi dolg iterativen ali rekurziven test, da smo lahko merili razliko v zagonu javanskega navideznega stroja z in brez JIT prevajalnika. Fibonaccijevo število smo računali po rekurzivni enačbi $fib(0) = 1; fib(1) = 1; fib(n) = fib(n-1) + fib(n-2)$. Računali smo 30. fibonaccijevo število.

Rezultati meritev so predstavljeni na sliki 4.5.

Kot pričakovano je interpretirana različica PINS močno počasnejša od ostalih, kar pa je zanimivo da je razlika še bolj občutna, kot pri drugih primerih. Ker izračun 30. fibonaccijevega števila traja občutno dlje, kot druga predstavljena algoritma, lahko opazimo močno razliko pri zagonu z uporabe JIT. Do tega pride zato, ker ima JIT prevajalnik dovolj časa za



Slika 4.5: 30. fibonaccijevo število

popoln zagon in moč njegovih optimizacij prevesi ceno njegovega zagona. V tem programu se tudi vidi, da je čista javanska implementacija hitrejša od novejšega prevajalnika.

Poglavje 5

Zaključek

V diplomskem delu smo predstavili prevajalnike in njihovo zgradbo, registrske in skladovne procesorje, podrobno pa smo se posvetili tudi javanskemu navideznemu stroju, njegovi zgradbi in uradni specifikaciji. Opisan je tudi glavni del diplomske naloge, prevajanje programskega PINS v javansko zložno kodo, v sklopu katerega je opisan postopek prevajanja in zanimivejše funkcionalnosti, ki jih je bilo potrebno implementirati.

Cilj diplomske naloge je bil razširiti prevajalnik za programski jezik PINS tako, da je njegov izhod izvorna koda, prevedena v javansko zložno kodo. V sklopu prevajalnika smo morali implementirati 3 faze, katere so referenčna analiza, generiranje vmesne zložne kode in pretvorba in zapisovanje zložne kode. Menimo, da smo zastavljeni cilj dosegli, saj prevajalnik in koncepti, uporabljeni pri razvoju delujejo dovolj dobro. Kljub temu pa se zavedamo, da so v implementaciji še vedno prisotni nekateri hrošči, katere v prihodnosti nameravamo odpraviti.

V sklopu dela na prevajalniku za programski jezik PINS je mogočih še ogromno izboljšav, tako pri implementaciji jezika, kot pri implementaciji zadnjega dela. V prihodnosti lahko v prevajalnik dodamo razne optimizacije, nadgradimo jezik PINS z mnogimi sintaksnimi ali semantičnimi konstrukt, poleg tega pa nam javanski navidezni stroj omogoča izrabo zelo zrelega javanskega ekosistema.

Poglavje 6

Priloge

6.1 PINS

6.1.1 Quicksort

```
fun main(): void = ({
    fill(^array, 20);
    print(^array, 20);
    quick(^array, 0, 19);
    print(^array, 20);
}
where
    var array : [20]int;
);

fun quick(array : ^[20]int, low : int, high : int) :
    void = ({
    if (low < high) then
        mid = partition(array, low, high);
        quick(array, low, mid - 1);
        quick(array, mid + 1, high);
```

```
    end;
}
where
    var mid : int;
);

fun partition(array : ^[20]int, low : int, high :
   int) : int = ({
    pivot = array^[high];
    i = low - 1;
    j = low;
    while (j < high) do
        if (array^[j] <= pivot) then
            i = i + 1;
            swap(array, i, j);
        end;
        j = j + 1;
    end;
    swap(array, i + 1, high);
    i + 1;
}
where
    var i : int;
    var j : int;
    var pivot : int;

    fun swap(array : ^[20]int, i : int, j : int) :
       void = ({
        temp = array^[i];
        array^[i] = array^[j];
        array^[j] = temp;
```

```
    }
    where
        var temp : int;
    );
);


fun fill(array : ^[20]int, size : int) : void = ({
    array^[0] = 5;
    array^[1] = 2;
    array^[2] = 7;
    array^[3] = 3;
    array^[4] = 9;
    array^[5] = 1;
    array^[6] = 8;
    array^[7] = 4;
    array^[8] = 6;
    array^[9] = 11;
    array^[10] = 0;
    array^[11] = 24;
    array^[12] = 3;
    array^[13] = 1;
    array^[14] = 5;
    array^[15] = 12;
    array^[16] = 1;
    array^[17] = 9;
    array^[18] = 13;
    array^[19] = 69;
}
where
    var i : int;
```

```
    var rand : int;
);

fun print(array : ^[20]int, size : int) : void = ({
    i = 0;
    while (i < size) do
        putInt((array^)[i]);
        putChar('␣');
        i = i + 1;
    end;
    putChar(endl());
}
where
    var i : int;
);


#{ Utilities }#

fun endl(): char = (10 : char);


#{ Forward declarations: }#

fun putInt(i: int): void = none;
fun getInt(): int = 0;

fun putChar(c: char): void = none;
fun getChar(): char = 'a';
```

Program 6.1: Quicksort

6.1.2 Kadanov algoritem

```
fun main(): void = ({
    array[0] = -2;
    array[1] = -1;
    array[2] = -3;
    array[3] = 4;
    array[4] = 1;
    array[5] = -1;
    array[6] = 6;
    array[7] = -5;
    array[8] = 1;
    array[9] = 0;
    array[10] = -3;
    array[11] = -1;
    array[12] = 3;
    array[13] = 2;
    array[14] = -1;
    array[15] = -5;
    array[16] = 5;
    array[17] = 3;
    array[18] = -4;
    array[19] = 0;

    result = kadane(^array);

    putInt(result);
}
where
    var result: int;
    var array: [20]int;
);
```

```
fun kadane(arr: ^[20]int): int = ({
    current_max = 0;
    max_ending = 0;

    i = 0;

    while i < 20 do
        max_ending = max_ending + arr^[i];

        if max_ending < 0 then
            max_ending = 0;
        end;

        if current_max < max_ending then
            current_max = max_ending;
        end;

        i = i + 1;
    end;

    current_max;
}
where
    var i: int;
    var current_max: int;
    var max_ending: int;
);

fun putInt(i: int): void = none;
```

Program 6.2: Kadanov algoritem

6.1.3 N-to fibonaccijevo število

```
fun main(): void = {
    fib(30);
    none;
};

fun fib(n: int): int = ({
    if (n <= 1) then
        ret = n;
    else
        ret = fib(n - 1) + fib(n - 2);
    end;

    ret;
}
where
    var ret: int;
);
```

Program 6.3: Fibonacci

6.2 Java

6.2.1 Quicksort

```
public class quicksort {
    public static void main(String[] args) {
        long[] array = {5, 2, 7, 3, 9, 1, 8, 4, 6,
           11, 0, 24, 3, 1, 5, 12, 1, 9, 13, 69};
        print(array);
        quick(array, 0, 19);
```

```
        print(array);
    }

    static void quick(long[] array, long low, long
       high) {
        if (low < high) {
            long pivot = partition(array, low, high)
               ;
            quick(array, low, pivot - 1);
            quick(array, pivot + 1, high);
        }
    }

    static long partition(long[] array, long low,
       long high) {
        long pivot = array[(int) high];
        long i = low - 1;
        for (long j = low; j < high; j++) {
            if (array[(int) j] <= pivot) {
                i++;
                swap(array, i, j);
            }
        }
        swap(array, i + 1, high);
        return i + 1;
    }

    static void swap(long[] array, long i, long j) {
        long temp = array[(int) i];
        array[(int) i] = array[(int) j];
        array[(int) j] = temp;
```

```
    }

    static void print(long[] array) {
        for (long i : array) {
            System.out.print(i + "␣");
        }
        System.out.println();
    }
}
```

Program 6.4: Quicksort

6.2.2 Kadanov algoritem

```
public class kadane {
    public static void main(String[] args) {
        long[] array = {-2, -1, -3, 4, 1, -1, 6, -5,
            1, 0, -3, -1, 3, 2, -1, -5, 5, 3, -4,
           0};
        long result = kadanes(array);
        System.out.println(result);
    }

    static long kadanes(long[] array) {
        long currentMax = 0, maxEnding = 0;

        long i = 0;
        while (i < 20) {
            maxEnding = maxEnding + array[(int) i];

            if (maxEnding < 0) {
                maxEnding = 0;
```

```
            }

            if (currentMax < maxEnding) {
                currentMax = maxEnding;
            }

            i++;
        }

        return currentMax;
    }
}
```

Program 6.5: Kadanov algoritem

6.2.3 N-to fibonaccijevo število

```
public class fibonacci {
    public static void main(String[] args) {
        fib(30);
    }

    static long fib(long n) {
        if (n <= 1) {
            return n;
        } else {
            return fib(n - 1) + fib(n - 2);
        }
    }
}
```

Program 6.6: Fibonacci

Literatura

[1] Don Box in Chris Sells. *Essential. Net: the common language runtime*. Zv. 1. Addison-Wesley Professional, 2003.

[2] Eric Bruneton, Romain Lenglet in Thierry Coupaye. "ASM: a code manipulation tool to implement adaptable systems". V: *Adaptable and extensible component systems* 30.19 (2002).

[3] Iulian Dragos. "Compiling Scala for Performance". Doktorska disertacija. EPFL, 2010.

[4] Sebastian Graf in Simon Peyton Jones. "Selective Lambda Lifting". V: *Proc. ACM Program. Lang. 1.* 2019.

[5] David Gregg in sod. "The case for virtual register machines". V: *Science of Computer Programming* 57.3 (2005). Advances in Interpreters, Virtual Machines and Emulators, str. 319–338.

[6] Benjamin J Evans, Jason Clark in David Flanagan. *Java in a Nutshell, 8th Edition*. O'Reilly Media, Inc., 2023. ISBN: 9781098131005.

[7] *Java virtual machine*. URL: `https://en.wikipedia.org/wiki/Java_virtual_machine` (pridobljeno 13. 8. 2023).

[8] Thomas Johnsson. "Lambda lifting: Transforming programs to recursive equations". V: *Functional Programming Languages and Computer Architecture*. Springer Berlin Heidelberg, 1985, str. 190–203.

[9] C. Lattner in V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". V: *International Symposium on Code Generation and Optimization*. 2004, str. 75–86.

[10] Tim Lindholm in sod. *The Java™ Virtual Machine Specification*. Java SE 7 Edition. Oracle America, Inc., feb. 2012. URL: `http://docs.oracle.com/javase/specs/`.

[11] Bashir Salisu Abubakar in sod. "An Overview of Compiler Construction". V: *International Journal of Research in Engineering and Technology* 8 (mar. 2021), str. 587–590.

[12] Alfred V. Aho in sod. *Compilers: Principles, Techniques, & Tools, Second Edition*. Addison-Wesley, 2007.

[13] Bill Venners. *Inside the Java Virtual Machine*. USA: McGraw-Hill, Inc., 1996. ISBN: 0079132480.