SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD

Dekompajliranje aplikacija pisanih u programskom jeziku Kotlin

Filip Todorić

Mentor:
doc. dr. sc. Stjepan Groš

Zagreb, lipanj 2020.

Sadržaj

1. Uvod ... **1**

2. Arhitektura Android aplikacije i programski jezik Kotlin ... **3**

2.1. Struktura aplikacije i proces njezinog nastanka ... 3

2.2. Značajke programskog jezika Kotlin ... 4

2.2.1. Tipovi podataka ... 4

2.2.2. Kontrola toka ... 6

2.2.3. Klase i objekti ... 6

2.3. Java izvršna datoteka i bajt kod ... 7

3. Proces dekompajliranja aplikacije ... **11**

3.1. Priprema izvršnih datoteka ... 11

3.2. Analiza Java bajt koda ... 11

3.2.1. Deklaracije i definicije varijable ... 12

3.2.2. Instrukcije aritmetičkih operacija ... 13

3.2.3. Analiza kontrole toka ... 14

3.2.4. Klase i metode ... 16

3.3. Faze dekompajliranja ... 17

3.3.1. Obrada ulazne datoteke ... 18

3.3.2. Analiziranje procedura ... 18

4. Programsko ostvarenje dekompajlera ... **23**

5. Zaključak ... **31**

6. Literatura ... **32**

Sažetak ... **34**

Summary ... **35**

1. Uvod

Razvojem računarske znanosti i inženjerstva stvoreni su brojni računalni sustavi i programi koji nam olakšavaju obavljanje svakodnevnih poslova. Od korištenja osobnih računala i mobitela do upravljanja automatskih pogona i robotike automobilskih ili prehrambenih industrija. Svi izvršni programi koji obavljaju određene zadatke svojih korisnika nazivaju se aplikacije. Aplikacije mogu biti izgrađene za računala, za mobitele ili za razne strojeve koji se koriste u različite svrhe.

Kao što vrijedi i za većinu stvari, korištenje aplikacija treba biti sigurno i zloupotrebe trebaju biti spriječene. To se može postići izgradnjom sigurne arhitekture novoosmišljene aplikacije i analiziranjem stranih i nepoznatih aplikacija kako osoba koja se želi koristiti njome ne pretrpi nikakvu štetu. Nakon proboja sigurnosti aplikacije šteta može biti mala, poput krađe resursa iz igrice, ili može biti većih razmjera poput krađe novca s računa bankovnih aplikacija.

Svaka se aplikacija može analizirati u potrazi za mjestima ranjivosti. Mjesta ranjivosti traže se radi njihovog otklanjanja i poboljšavanja sigurnosti arhitekture aplikacije. Traženjem mjesta ranjivosti želi se dobiti pristup nekom dijelu aplikacije, određenim podacima za koje se nema dozvola ili se pokušava onemogućiti daljnji rad aplikacije. Takav postupak naziva se *penetracijsko testiranje*, a tijekom tog ispitivanja ispitivač se ponaša kao napadač. Za vrijeme penetracijskog ispitivanja ispitivač iskorištava svoje znanje i alate, a jedni od značajnih alata su programi zvani *dekompajleri*.

Svaki programski jezik ima pripadni program zvan *prevoditelj* (*kompajler*) koji analizira izvorni kod napisan u jeziku više razine i prevodi ga u binarni kod koji se sastoji od niza nula i jedinica kako bi računalo moglo učitati i izvršiti napisani program. Suprotnost procesu kompajliranja je proces dekompajliranja, koji pokušava prevesti binarni kod u izvorni kod odnosno obaviti rekonstrukciju. Ako se ne poznaje izvorni kod i imamo na raspolaganju samo izvršnu datoteku koja sadrži binarni kod potreban je dekompajler jezika izvršne datoteke. Dekompajler je jedan od alata reverznog inženjerstva. Još jedan primjer alata reverznog inženjerstva, na kojeg se često nailazi, je i *debugger*, koji služi za traženje grešaka u kodu u razvojnim okruženjima [1].

Tijekom procesa dekompajliranja nailazi se na probleme koji ponekad predstavljaju dovoljnu prepreku zbog koje dekompajleri ne proizvode uvijek precizan izvorni kod. Binarni kod je optimiziran i minimiziran kako bi bio spreman za što efikasnije učitavanje i izvršavanje. Tijekom procesa kompajliranja, optimizacijom i restrukturiranjem izvornog koda u izvršni, izostavljaju se podaci koji nisu potrebni za vrijeme izvođenja [2]. Zadaća dekompajlera je stoga, metodama analiziranja binarnog koda rekonstruirati što bolji i što čitljiviji izvorni kod za ciljni jezik.

Kako je programski jezik Java bio popularan programski jezik i najbolji izbor za izgradnju aplikacija za operacijski sustav Android, mnogi su se dekompajleri gradili upravo za jezik Javu. U današnje vrijeme sve je popularniji jezik Kotlin koji zamjenjuje Javu u izgradnji Android aplikacija. Kotlin je postao najprihvaćeniji jezik za razvoj mobilnih aplikacija od 2019. zbog svojih povoljnijih značajki i nadogradnjom nad jezikom Java [3][4]. Danas ne postoje

dekompajleri za jezik Kotlin sa zadovoljavajućim mogućnostima poput mnoštva dovoljno dobrih dekompajlera za jezik Java [5], prvenstveno jer je Kotlin nov jezik.

Kako bi se proveo postupak dekompajliranja izvršnog programa aplikacije u čitljiv izvorni kod jezika Kotlin, moraju se opisati struktura android aplikacije, izvršnih datoteka i upoznati se sa značajkama i sa sintaksom jezika Kotlin. Kada se opišu glavne strukture i kada su poznata mjesta izvršnih datoteka, dekompajler se programski ostvaruje koristeći pritom različite alate i postupke. Dekompajliranje se provodi u nekoliko faza i, konačno, proizvodi se datoteka koja sadržava izvorni kod ciljnog jezika, u ovom slučaju jezika Kotlin. Izvorni kod se onda može koristiti za dobivanje uvida u logiku ponašanja i izgled podatkovnih struktura programa, a takav uvid olakšava analiziranje i ispitivanje sigurnosnog stanja programa te cjelokupnog sustava.

Cilj ovoga rada je istražiti mogućnosti dekompajliranja aplikacija pisanih u jeziku Kotlin. U poglavlju 2. opisuje se arhitektura Android aplikacije i osnove programskog jezika Kotlin. U poglavlju 3. razmatrat će se općeniti proces dekompajliranja u kojemu će se opisati proučavanje izvršnog koda kao skup instrukcija i proučavat će se na koji način instrukcije predstavljaju složene strukture i izraze izvornog koda. U poglavlju 4. opisuje se programsko ostvarenje dekompajlera u sklopu ovoga rada. Implementacija dekompajlera upotrijebit će prikupljena znanja prijašnjih poglavlja, a dobiveni rezultati tj. dekompajlirani izvorni kodovi jednostavnijih programa i aplikacija bit će pojašnjeni i prokomentirani.

2. Arhitektura Android aplikacije i programski jezik Kotlin

Aplikacije se najčešće razvijaju u razvojnim okruženjima gdje se napisani izvorni kodovi prevode u izvršne datoteke i, zajedno s ostalim generiranim podacima i korištenim bibliotekama, tvore funkcionalnu cjelinu. Aplikacije su zatim spremne za distribuciju, instalaciju i izvršavanje unutar izvršnih okolina operacijskih sustava. Izgled i nastanak Android aplikacije opisan je u poglavlju 2.1.

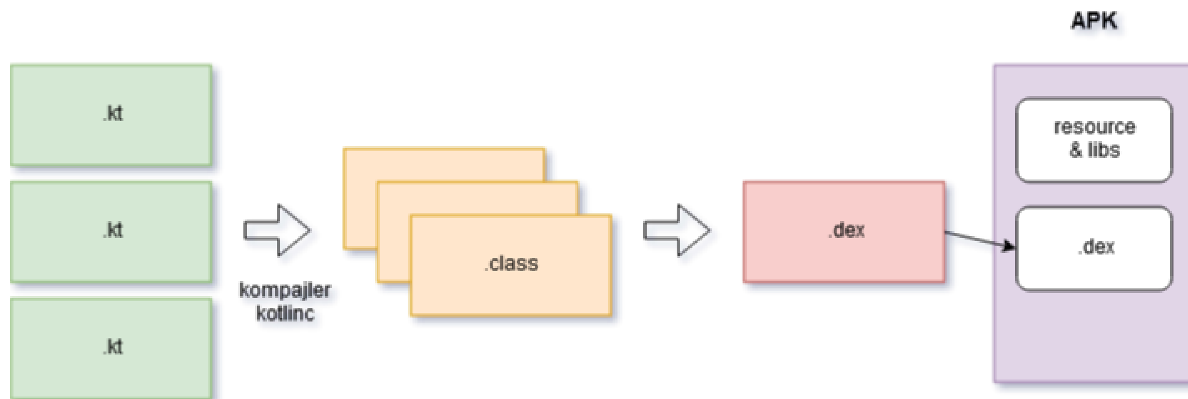
Jezik Kotlin ima sintaksu prilagođeniju lakšem učenju jezika, tipove podataka iste kao i jezik Java, te isti ciljni virtualni stroj kao jezik Java. Značajke i format izvršne datoteke jezika Kotlin opisani su u poglavljima 2.2. i 2.3.

2.1. Struktura aplikacije i proces njezinog nastanka

Aplikacije napisane za izvršavanje na operacijskom sustavu Android prenose se i instaliraju u obliku arhive odnosno paketa zvanog *Android application package* (APK). Arhiva sadrži razne kompajlirane podatke, metapodatke i biblioteke potrebne za učitavanje i izvršavanje unutar izvršnog okruženja operacijskog sustava Android. Prije verzije 4.4 Android operacijskog sustava imena „KitKat“, sve su se aplikacije izvršavale u *Dalvik* virtualnom stroju, dok se u novijim verzijama Android sustava koristi izvršno okruženje *Android Runtime* (ART) [6]. Izvršna datoteka APK arhive naziva se *classes.dex* i u njoj se nalazi *Dalvik* izvršni kod (*Dalvik Executable* skr. DEX). Takav format izvršne datoteke razumljiv je Dalvik virtualnom stroju i ART-u.

Nastanak Dalvik izvršne datoteke iz datoteka s izvornim kodom programskog jezika Kotlin, prikazan na slici 2.1, provodi se kroz dvije faze. U prvoj fazi, prevoditelj jezika u kojemu je napisan izvorni kod prevodi izvorni kod u izvršnu datoteku tog jezika. Izvršna datoteka sadrži binarni kod kojega računalo može učitati u memoriju i izvršavati pa se izvršna datoteka jezika Kotlin može izvršavati pomoću svog virtualnog stroja. Tako će, naprimjer, izvorni kod sadržan u datoteci ekstenzije *.kt* jezika Kotlin biti preveden Kotlin kompajlerom u izvršni kod pohranjen u datoteci ekstenzije *.class*.

Međutim, izvršni kod jezika Kotlin nije razumljiv izvršnom okruženju Android operacijskog sustava zbog drugačije organizacije podataka i instrukcija te datoteke. Stoga se, u drugoj fazi, instrukcije nastalog izvršnog koda trebaju pretvoriti u instrukcije Dalvik izvršnog koda. Takav proces nije prevođenje iz višeg programskog jezika u niži, što obavljaju prevodioci, već preslikavanje različitih vrsta binarnih kodova približno jednakih razina [7].



Slika 2.1 Prevođenje izvornog koda jezika Kotlin u izvršni kod Android aplikacije.

Razvijanje Android aplikacije u integriranom razvojnom okruženju *Android Studio* započinje stvaranjem projekta. Jedan projekt obuhvaća skupinu osnovnih biblioteka i programa kao i osnovni predložak u kojem se piše kod aplikacije. Razvoj aplikacije dijeli se u dva osnovna područja: programiranje dizajna aplikacije („*frontend*“) i programiranje pozadinskih procesa i baze podataka („*backend*“). Dizajn aplikacije obuhvaća oblikovanje i raspoređivanje već dostupnih komponenta ili stvaranje svojih u kojima se prikazuju sadržaji u različitim oblicima. Pozadinski procesi su procesi poput obrade podataka iz baze podataka ili drugih izvora, upravljanja kanalima notifikacija itd. Pozadinski procesi su povezani s dizajnom aplikacije na način da se obrađeni podaci mogu prikazivati unutar komponenti ili se mogu prikazivati notifikacije korisniku. Opisni kod dizajna aplikacije najčešće je pohranjen u datoteci *activity_main.xml* i sl., dok su početni pozadinski procesi i logika ponašanja aplikacije opisani datoteci *MainActivity.kt*. Datoteka *activity_main.xml* i ostale datoteke koje sadrže opisni jezik dizajna aplikacije mogu se pronaći u APK datoteci u direktoriju *layout*.

2.2. Značajke programskog jezika Kotlin

Tvrtka JetBrains 2011. godine pokreće projekt Kotlin, a u veljači 2016. godine izlazi prva stabilna verzija. Jezik Kotlin objektno je orijentiran jezik i razvijen je za izvršavanje unutar virtualnog stroja *Java Virtual Machine* (JVM) koji je ujedno i izvršni stroj jezika Java. Pisan je kao jezik nadogradnje Jave i zbog sličnosti s Javom potpuno je interoperabilan s njom [8]. Jezik se odlikuje osobinama poput slobodnije sintakse (npr. opcionalnost sintaksnih znakova poput znaka točka zarez i vitičastih zagrada) i automatskim zaključivanjem tipova podataka (engl. type inference). Takve osobine također omogućuju lakše učenje jezika.

2.2.1. Tipovi podataka

Primitivni tipovi podataka u jeziku Kotlin su: cjelobrojni brojevi, brojevi s pomičnom točkom, znakovni i logički tipovi. Od složenijih postoje nizovi, klase, sučelja itd. Tipovi podataka ne moraju se eksplicitno navesti prilikom deklaracija varijabli. Varijable se deklariraju kao nepromjenjive i promjenjive varijable. U ispisu 2.1 definirana je jedna nepromjenjiva varijabla

i dvije promjenjive. Nepromjenjiva varijabla (engl. read-only) definira se ključnom riječi *val* (engl. value), a promjenjiva ključnom riječi *var* (engl. variable).

```
val broj1: Byte = 1
var broj2 = 202
var broj3 = 32768
```

Ispis 2.1 Deklariranje i automatsko zaključivanje tipova podatka.

Cjelobrojni tipovi prikazani su s određenim brojem okteta u memoriji i imaju određen raspon mogućih vrijednosti. Popis tipova prikazan je u tablici 2.1. Nad varijablama kojima tip nije eksplicitno definiran provodi se automatsko zaključivanje tipa podatka (engl. type inference). Varijabla s cjelobrojnom vrijednošću bez navedenog tipa uvijek će imati zaključeni tip podatka *Int* dok vrijednost varijable ne pređe mogući raspon, a tada je tip *Long*. U ispisu 2.1 sve definirane cjelobrojne varijable imaju tip podatka *Int*, osim varijable `broj1` koja je definirana tipom *Byte*.

Tip	Veličina (okteti)	Najmanja vrijednost	Najveća vrijednost
Byte	8	-128	127
Short	16	-32768	32767
Int	32	-2,147,483,648	2,147,483,647
Long	64	-2^{63}	$2^{63}-1$

Tablica 2.1 Tipovi cjelobrojnih varijabli u jeziku Kotlin.

Tipovi podataka s pomičnom točkom su: *Float* i *Double*. Zaključeni tip podatka varijable vrijednosti broja s pomičnom točkom je tip *Double*. Da bi se eksplicitno specificirao cjelobrojni tip ili tip s pomičnom točkom moguće je dodati sufikse na literale: *L* za definiranje tipa *Long*, *f* za definiranje tipa *Float*.

Varijable logičkog tipa *Boolean* poprimaju vrijednosti *true* ili *false*.

Znakovni tipovi su *Char* i *String*. Tip *Char* se ne koristi direktno kao brojevni tip. Znakovi se definiraju pomoću jednostrukih navodnika poput izraza: `var c: Char = '1'`. Podatak tipa *String* nepromjenjiv je, indeksiran i definira se pomoću dvostrukih navodnika. Vrijednost varijable tipa *String* se sastoji od znakova i svakom znaku unutar te varijable se može pristupiti svojim indeksom.

Složeni tip podatka niz reprezentiran je parametriziranom klasom *Array* ili klasama *ByteArray*, *ShortArray*, *IntArray* itd., koje predstavljaju nizove primitivnih tipova. Primjer korištenja niza prikazan je u ispisu 2.3.

2.2.2. Kontrola toka

Kontrola toka obuhvaća naredbene strukture poput: *if*, *when*, *for*, *while*. Kontrola toka služi za ispitivanje uvjeta i određivanje sljedećih naredbi na temelju istinitosti uvjeta, iteriranje po elementima itd. Sintaksa je donekle slična sintaksi jezika Java, ali postoje razlike. Jednostavan primjer *if* i *for* izraza dan je u ispisu 2.2. Program ispisuje parne brojeve između brojeva 1 i 10. Izraz *when*, prikazan u ispisu 2.3, zamjenjuje funkcionalnost izraza *switch* u Javi.

```
for (i in 1..10) {
      if (i%2==0) println(i)
}
```

Ispis 2.2   Jednostavna kontrola toka: petlja.

```
val auti: Array<String> = arrayOf("Kia", "Audi", "Jaguar")
for (auto in auti) {
      when (auto) {
      "Audi" -> println("Njemačka")
      "Jaguar" -> println("UK")
      "Kia" -> println("Južna Koreja")
      }
}
```

Ispis 2.3   Varijabla tipa *Array* i kontrola toka *when*.

2.2.3. Klase i objekti

Klasa se sastoji od imena klase, zaglavlja i tijela okruženog vitičastim zagradama. Zaglavlje klase nalazi se između definicije imena klase i tijela klase. Primarni konstruktor se navodi u zaglavlju klase, a ključna riječ *constructor* smije biti izostavljena. Klasa može sadržavati konstruktore, varijable, metode i ugniježđene klase. Instanciranjem klase nastaje objekt. Primjer definicije klase i kreiranje objekta prikazano je u ispisu 2.4. U ispisu, poslije imena klase, u zaglavlju je definiran konstruktor ključnom riječi *constructor,* koji prima dva argumenta tipa *String* i *Int* pri stvaranju instance klase. Svaka klasa ima mogućnost nasljeđivanja druge klase čime nasljeđuje i njezine varijable i metode. Klasa koja nasljeđuje drugu klasu zove se klasa *dijete*, dok se nasljeđivana klasa naziva i *roditeljska* klasa. Jedna klasa može imati više roditeljskih klasa, kao što i jedan roditelj može imati više djece. Mogu se nasljeđivati i *sučelja* (*interface*). Sučelja su klase koje sadrže neimplementirane metode. Nasljeđivanjem sučelja sve se neimplementirane metode moraju implementirati. Sučelja služe za stvaranje objekta čije se funkcionalnosti mogu implementirati na različite načine.

```
class Car constructor(brand: String, maxspeed: Int) {
      private var myBrand = brand
      private var myMaxspeed = maxspeed

      init {
            println("You bought %s with maximum speed of %d km/h."
            .format(myBrand, myMaxspeed))
      }

      fun unlock() {
            println("%s is now unlocked.".format(myBrand))
      }
}

fun main() {
    val mycar = Car("Audi", 250)
    mycar.unlock()
}
```

Ispis 2.4 Klasa *Car* i objekt *mycar* tipa *Car*.

2.3. Java izvršna datoteka i bajt kod

Svaka izvršna datoteka, stvorena za izvršavanje opisanog programa unutar memorije, ima jedinstveni format rasporeda podataka zapisanih u binarnom obliku. Tako Java izvršna datoteka ima svoj format slijeda podataka pohranjen u datoteku ekstenzije *.class*. Ciljana izvršna okolina izvršne datoteke jezika Kotlin je JVM. Zato je izvršna datoteka jezika Kotlin istog formata kao i izvršna datoteka jezika Jave za učitavanje i izvršavanje unutar okoline JVM. U izvršnim datotekama pohranjeni su nizovi instrukcija koje virtualni stroj učitava i skupine podataka koji su referencirani instrukcijama. Osim po ekstenziji *.class*, Java izvršna datoteka može se prepoznati i po prva četiri jedinstvena okteta (*„magic number“*), heksadekadski zapisano, `0xcafebabe`. Java izvršna datoteka i JVM koriste *big-endian* zapis u memoriju. To označava da su značajniji okteti pohranjeni na nižim adresama memorije. Sekcije Java izvršne datoteke redom su prikazane u tablici 2.2.

Sekcije	Opis
„magic number“	`0xcafebabe`
minor version	Verzije datoteke označavaju verzije formata datoteke različitih verzija Java SE platforme [9].
major version	
constant pool count	Tablica konstanti sadrži konstante kao što su brojevi, znakovni nizovi, reference na korištene objekte i metode itd.
constant pool table	
access flags	Pristupne zastavice sadrže dozvole pristupa odnosno modifikatore trenutne klase poput: *public* ili *final*.
this class	Sekcija od dva okteta, a dva okteta sačinjavaju indeks tablice konstanti gdje je pohranjeno ime trenutne klase.
super class	Sekcija sadržava indeks tablice konstanti gdje je pohranjeno ime roditeljske klase. Ako klasa roditelj ne postoji vrijednost je 0. Valja napomenuti kako sve klase koje nemaju eksplicitno definiranog roditelja implicitno nasljeđuju klasu *Object*. Klasa *Object* jedina je klasa koja nema roditelja.
interface count	Tablica sučelja sadržava indekse tablice konstanti gdje su definirana imena sučelja koja trenutna klasa implementira.
interface table	
field count	Tablica atributa razreda sadrži popis definiranih atributa deklariranih u trenutnoj klasi, a ne sadrži popis naslijeđenih atributa deklariranih u roditeljskim klasama ili sučeljima.
field table	
method count	Tablica metoda sadrži broj i opis pojedinih metoda sadržanih u klasi i niz instrukcija svake metode.
method table	
attribute count	Tablica atributa izvršne datoteke nije, kao što bi ime sugeriralo, popis atributa trenutne klase već popis atributa trenutne izvršne datoteke [10].
attribute table	

Tablica 2.2 Slijed zapisa podataka Java izvršne datoteke.

Skup instrukcija napisanih za virtualni stroj *Java Virtual Machine* (JVM) naziva se Java *bajt kod*. Svaka instrukcija predstavljena je jednim oktetom ili operacijskim kodom i dodatnim opcionalnim oktetima koji predstavljaju indekse tablice konstanti ili zasebne vrijednosti operanda. Svaka instrukcija obavlja operacije koristeći stog i lokalne varijable JVM-a. Pošto je instrukcija zapisana jednim oktetom skup se instrukcija sastoji se od ukupno 2^8 (256) instrukcija. Dio skupa instrukcija prikazan je u tablici 2.3 [11]. Instrukcije se dijele u skupine prema radnjama koje obavljaju:

- instrukcije za učitavanje i spremanje operanda
- instrukcije za obavljanje aritmetičkih i logičkih operacija
- instrukcije za pretvaranje tipova podataka
- instrukcije za stvaranje i manipulaciju objektima
- instrukcije za upravljanje stogom
- instrukcije kontrole toka
- instrukcije za pozivanje i vraćanje iz metoda

operacijski kod	**<tip>**				
<sufiks>ime	*integer*	*long*	*float*	*double*	*reference*
<tip>const_0	iconst_0	lconst_0	fconst_0	dconst_0	-
<tip>load	iload	lload	fload	dload	aload
<tip>store	istore	lstore	fstore	dstore	astore
<tip>add	iadd	ladd	fadd	dadd	-
<tip>sub	isub	lsub	fsub	dsub	-
<tip>mul	imul	lmul	fmul	dmul	-

Tablica 2.3 Dio skupa instrukcija bajt koda za stvaranje vrijednosti 0, učitavanje i spremanje podataka i aritmetičkih operacija.

JVM je izvršni stroj sa stogovnom i registarskom arhitekturom. Podatkovne strukture su stog, registri tj. lokalne varijable indeksa 0-4 i indeksirane varijable. Valja napomenuti da učitavanje vrijednosti operanda znači učitavanje vrijednosti iz lokalnih varijabla i stavljanje na vrh stoga. Naravno, vrijednost može biti i stvorena dodatnim oktetom instrukcije i zatim stavljena na stog. A spremanje znači skidanje vrijednosti s vrha stoga i pohranjivanje u lokalnu varijablu. Stog predstavlja privremenu memoriju u koju vrijednosti dolaze i odlaze. Posljedica toga je da sve aritmetičke i logičke operacije, kao i druge slične instrukcije, koriste operande koji su pohranjeni na vrhu stoga. Tako će, naprimjer, instrukcija zbrajanja *iadd* uzeti dva operanda s vrha stoga, zbrojiti ih i pohraniti rezultat natrag na stog. Isto tako, instrukcije za upravljanje stogom uzeti će vrijednosti s vrha stoga i pohraniti neke druge vrijednosti na stog. Jedna takva instrukcija je instrukcija *pop*. Ona uzima vrijednost s vrha stoga i odbacuje ju, ne sprema ju nigdje. Instrukcije za upravljanje kontrolom toka imaju sljedeća dva okteta za definiranje adresu skoka. Skok se izvršava ako je uvjet ispunjen. Također, postoji i instrukcija bezuvjetnog skoka *goto*.

Prefiksi i sufiksi određenim nazivima instrukcija definiraju tip podatka nad kojim instrukcija operira. Tipovi podataka i njihovi prefiksi odnosno sufiksi instrukcija su:

- *integer – i*
- *long – l*
- *short – s*
- *byte – b*
- *character – c*
- *float – f*
- *double – d*
- *reference – a*

Tipovi podatka izvornog jezika tijekom procesa prevođenja zamjenjuju se tipom podatka korištenih u Java bajt kodu. Pa će se zbog toga, naprimjer, sa svim brojevnim tipovima podataka izvornog koda koji nisu *long*, *float* ili *double* računati s tipom *integer*. Primjeri takvih tipova podataka u jeziku Kotlin su *Boolean*, *Byte*, *Char* i *Short*.

Sve metode datoteke izvornog koda se prevode u niz bajt kod instrukcija, a te instrukcije sadržane se u tablici metoda (*method table*).

3. Proces dekompajliranja aplikacije

U ovom poglavlju opisan je proces općenitog stvaranja dekompajlera i njegov rad u različitim fazama, dok će se u poglavlju 4. opisati konkretna implementacija dekompajlera u sklopu ovog rada. U poglavlju 3.1. proučava se ulazni skup podataka aplikacije za dekompajler. Ulazni podaci su datoteke koje sadrže informacije koje dekompajlera upotrebljava i ne temelju čega generira konačan produkt odnosno izvorni kod. U poglavlju 3.2. proučava se semantika instrukcija Java bajt koda i slijed izvođenja instrukcija kako bi se odgonetnule naredbene strukture koje te instrukcije predstavljaju. U poglavlju 3.3. opisuje se konceptualni proces dekompajliranja izvršne datoteke.

Isto tako, treba definirati opseg mogućnosti koje bi dekompajler trebao posjedovati. Jednostavan dekompajler trebao bi imati mogućnosti dekompajliranja deklaracija i definicija varijabli, rukovanje aritmetičkim operacijama i njihovim operandima, dekompajliranja jednostavnijih oblika kontrole toka, rukovanje s najčešćim objektima i njihovim metodama.

3.1. Priprema izvršnih datoteka

Ulazni podatak dekompajlera je binarni kod jedne izvršne datoteke ili više njih. Dekompajler aplikacije može analizirati Dalvik izvršnu datoteku ili izvršnu datoteku s Java bajt kodom. Za analizu izvršne datoteke s Java bajt kodom, prvo se binarni kod Dalvik datoteke preslikava u binarni kod Java izvršne datoteke. Za translaciju Dalvik koda u bajt kod postoje programi poput alata *dex2jar* [12].

Dalvik izvršna datoteka sadrži sustav povezanih klasa, ponekad i jako opsežnih, koje su grupirane i smještene u zasebne pakete. Rezultat postupka preslikavanja Dalvik izvršne datoteke je stablo direktorija koje reprezentira pakete klasa i u svakom krajnjem direktoriju nalazi se mnoštvo Java izvršnih datoteka. Ponekad će postojati više Dalvik izvršnih datoteka. Više se DEX datoteka dobiva u slučajevima kada broj metoda u aplikaciji i bibliotekama koje aplikacija referencira premašuje iznos od 65536 [13]. U slučaju kada postoji više Dalvik datoteka, postupak translacije se provodi za oba dvije datoteke. Dekompajler pojedinačno učitava dobivene Java izvršne datoteke i generira njezin izvorni kod.

3.2. Analiza Java bajt koda

Kako bi se programski ostvario dekompajler, prvo je bilo potrebno proučiti što se točno događa prilikom procesa prevođenja i u kojem se obliku organizira binarni kod izvršne datoteke. Proučavanje nizova nula i jedinica ili heksadekadskih znamenki bilo bi mukotrpno, stoga se proučavaju Java bajt kod instrukcije. Jedan od načina prevođenja binarnog koda u simbolički jezik instrukcija obavlja Java disasembler dostupan naredbom `javap -c` u naredbenom retku [14]. Java disasembler je sastavni dio *Java Development Kit* (JDK) okruženja u kojemu se također nalazi i JVM. Primjer dobivanja niza instrukcija svake metode izvršne datoteke prikazan je u ispisu 3.1.

```
C:\KotlinWorkspace\out\production\KotlinWorkspace>javap -c
HelloworldKt.class

Compiled from "helloworld.kt"
public final class HelloworldKt {
  public static final void main();
    Code:
       0: ldc           #11         // String Hello World!
       2: astore_0
       3: iconst_0
       4: istore_1
       5: getstatic     #17         // Field
java/lang/System.out:Ljava/io/PrintStream;
       8: aload_0
       9: invokevirtual #23      // Method
java/io/PrintStream.println:(Ljava/lang/Object;)V
      12: return

public static void main(java.lang.String[]);
    Code:
       0: invokestatic  #9      // Method main:()V
       3: return
}
```

Ispis 3.1 Dobivanje instrukcija Java bajt koda Java disasemblerom. Program ispisuje „Hello World!“ na standardni izlaz.

U ispisu 3.1 raspoznaju se razne instrukcije unutar pojedinih metoda s indeksom bajta na početku svake linije. Cilj je upoznati se sa semantikom pojedinih instrukcija i analizirati logiku kompajliranja našega izvornog koda u skup takvih instrukcija. Nakon poznavanja instrukcija analizira se raspored instrukcija i organizacija pojedinih instrukcija u cjeline na koje često nailazimo. Potrebno je proučiti kako se instrukcijama predstavljaju tipovi podataka, deklariranje i definiranje varijabla, kontrole toka, deklariranje i definiranje metoda i funkcija, te klase i objekata.

3.2.1. Deklaracije i definicije varijable

Deklaracijom varijable pridajemo varijabli određeni tip vrijednosti kojega varijabla može sadržavati ili, ako je tip izostavljen u jeziku Kotlin, automatski se zaključuje. Međutim, deklaracija sama po sebi ništa ne znači dok se ne definira vrijednost varijable. Deklaracija varijable ne zadaje nikakav zadatak za izvršavanje stoga prevoditelj traži vrijednost koju bi smjestio u tablicu konstanti (*ConstantPool*). Bez definirane vrijednosti varijabla koja je samo deklarirana izostavlja se i ne prikazuje instrukcijama. To je svojevrsna optimizacija koju prevoditelj obavlja. Jednom kada je varijabla deklarirana i definirana s određenom vrijednosti svojega tipa, prevoditelj pretvara deklaraciju i definiciju u dvije instrukcije. Prva instrukcija dohvaća vrijednost iz tablice konstanti ili stvara vrijednost pomoću jednog ili dva okteta i smješta tu vrijednost na stog. Druga instrukcija pohranjuje vrijednost sa stoga u lokalnu varijablu ili, ako nema slobodne lokalne varijable, u pomoćne indeksirane varijable. Opisane dvije instrukcije mogu se vidjeti u ispisu 3.2. Ispis 3.2 opisuje izvorni kod s jednom naredbom

`var car = „Audi"`. Može se vidjeti da je konstanta instrukcije indeksa 0 brojevnog tipa *byte* po prefiksu „b“, a konstanta se na stog pohranjuje kao vrijednost tipa *integer* što je vidljivo po prefiksu „i“. Druga konstanta instrukcije indeksa 3 znakovnog je tipa *string* učitana instrukcijom *ldc*. Međutim, ne može se vidjeti radi li se o nepromjenjivoj ili promjenjivoj varijabli jezika Kotlin. Vrste instrukcija za učitavanje konstante iz tablice konstanti i njihovo pohranjivanje u lokalne varijable ovisit će o tipu konstante. Imajući na umu navedeno, na nailaženje na dvije navedene instrukcije i analizom vrste instrukcije dobivaju se informacije o tipu podatka i vrijednosti varijable pa se može rekonstruirati izraz koji predstavlja deklaraciju i definiciju varijable izvornog koda. Kako se vrijednosti varijabli spremaju u tablicu konstanti i dohvaćaju instrukcijama pomoću indeksa tablice, ime varijable u izvornom kodu predstavlja redundantan podatak kojega prevoditelj nakon izrade tablice konstanti odbacuje. To je, također, svojevrsna optimizacija koju provodi prevoditelj.

```
public static final void main();
    Code:
        0: bipush        100
        2: istore_0
        3: ldc           #11                 // String Audi
        5: astore_1
        6: return
```

Ispis 3.2 Primjer definiranja dviju varijabla bajt kod instrukcijama.

3.2.2. Instrukcije aritmetičkih operacija

Različite aritmetičke, logičke i relacijske operacije izvornog koda prevode se u niz instrukcija koje predstavljaju pojednostavljene operacije. Složene aritmetičke operacije svode se na primitivne operacije poput zbrajanja, oduzimanja, bitovnog pomaka, a time nastaje i veći broj instrukcija. Međutim, mnoštvo jednostavnijih instrukcija predstavljenih jednim oktetom pogodnije su za procesiranje i time se dobiva na brzini izvođenja programa.

Bajt kod instrukcije koje obavljaju aritmetičke, logičke i relacijske operacije uzimaju dvije vrijednosti sa stoga i stavljaju rezultat natrag na stog. Neposredno prije toga, vrijednosti operandi učitavaju se na stog iz lokalnih varijabli ili kao novostvorene konstante. Operatori su definirani za različite tipove podataka. Nakon prevođenja izraza izvornog koda koji sadrži aritmetičke, logičke i druge operacije, dobivaju se instrukcije za dohvaćanje vrijednosti iz lokalnih varijabli, stvaranje novih vrijednosti, instrukcije pripadnih aritmetičkih i logičkih operacija, instrukcije konverzije tipova i instrukcije spremanja u lokalne varijable. Redoslijed instrukcija određuje redoslijed operacija i instrukcija spremanja vrijednosti u lokalnu varijablu određuje skladištenje rezultata u novu varijablu.

Pri proučavanju aritmetičkih operacija uzima se u obzir i mogućnost nadovezivanja operanda s različitima aritmetičkim operatorima kao u primjeru ispisa 3.3. Također treba uzeti u obzir da se kao operand može koristiti novostvorena konstanta ili vrijednost varijable. Instrukcija spremanja vrijednosti sa stoga u lokalnu varijablu predstavlja pohranu rezultata dosadašnjih

operacija i stvaranje nove varijable. U ispisu 3.4 instrukcije indeksa 0 i 1 predstavljaju pohranu vrijednosti 1 varijable *a* izvornog koda. Instrukcije indeksa 2 i 5 predstavljaju definiciju varijable *b*. Od instrukcije indeksa 6 do instrukcije indeksa 12 obavlja se aritmetika izraza pridruživanja varijabli *c* izvornog koda, a instrukcija *dstore_3* indeksa 12 predstavlja završetak izraza čiji se rezultat sprema u novu varijablu. Ostale instrukcije do instrukcije *return* predstavljaju pridruživanje varijabli *d*.

```
fun main() {
    val a = 1
    val b = 0.1
    val c = a + 3 - b
    val d = b - c;
}
```

Ispis 3.3 Primjer različitih aritmetičkih operacija izvornog koda.

```
public static final void main();
    Code:
       0: iconst_1
       1: istore_0
       2: ldc2_w        #10    // double 0.1d
       5: dstore_1
       6: iload_0
       7: iconst_3
       8: iadd
       9: i2d
      10: dload_1
      11: dsub
      12: dstore_3
      13: dload_1
      14: dload_3
      15: dsub
      16: dstore        5
      18: return
```

Ispis 3.4 Bajt kod instrukcije prevedenog izvornog koda u ispisu 3.3.

Međutim, primarni zadatak dekompajlera nije spremiti rezultat u novu varijablu već pokušati rekonstruirati izvorni kod koji će pridružiti izraz s operandima i aritmetičkim operatorima novonastaloj varijabli. Uz praćenja stanja stoga i dobivenog rezultata aritmetičke operacije, moraju se poznavati koje varijable su učitane s naredbama učitavanja kako bi se nad njima obavile računske operacije.

3.2.3. Analiza kontrole toka

Kontrole toka izvornog koda poput naredbi *if-else*, *for* i *when* prevoditelj prevodi u skupinu instrukcija koje imaju različite sljedove izvođenja ovisno o postavljenome uvjetu. Provjeru

uvjeta obavljaju instrukcije kontrole toka koje učitavaju dva operanda sa stoga i evaluiraju relacijski izraz s ta dva operanda u istinu ili neistinu. Dobivenom se evaluacijom određuje sljedeća instrukcija za izvođenje. To može biti instrukcija neposredno iza instrukcije kontrole toka ili može biti na odredišnoj adresi skoka. Adresa skoka određuje se s dva okteta neposredno nakon instrukcije kontrole toka u binarnom kodu. Poredak instrukcija uvjetnih i bezuvjetnih skokova određuje naredbenu strukturu izvornog koda. Dok se skokovi u instrukcijama *if-else* strukture odvijaju samo unaprijed, nizovi instrukcija *for* i *while* struktura posjeduju skokove i prema unazad dok se ne ispuni uvjet za nastavak programa. Jednostavan primjer *if* naredbe prikazan je u ispisu 3.5. Za analizu instrukcijskih cjelina koje predstavljaju složenije kontrole toka potrebne su unaprijed poznate informacije o redoslijedu izvršavanja pojedinih skupina instrukcija. Postoje različiti načini obrade kontrole toka. Jedan od načina naveden je u poglavlju 3.3.2. pomoću tzv. *osnovnih blokova*.

```
public static final void main();
    Code:
       0: new           #11     // class java/util/Scanner
       3: dup
       4: getstatic     #17
       7: invokespecial #21
      10: astore_0
      11: aload_0
      12: invokevirtual #25
      15: istore_1
      16: iload_1
      17: iconst_5
      18: if_icmplt     36
      21: ldc           #27            // String >= 5
      23: astore_2
      24: iconst_0
      25: istore_3
      26: getstatic     #31
      29: aload_2
      30: invokevirtual #37
      33: goto          48
      36: ldc           #39            // String < 5
      38: astore_2
      39: iconst_0
      40: istore_3
      41: getstatic     #31
      44: aload_2
      45: invokevirtual #37
      48: return
```

Ispis 3.5 Primjer programa koji provjerava je li uneseni broj veći ili jednak od ili manji od broja 5. Izvorni kod dan je u ispisu 3.6.

```
fun main() {
    var sc = Scanner(System.`in`)
    var a = sc.nextInt()
    if(a >= 5) println(">= 5")
    else println("< 5")
}
```

Ispis 3.6 Primjer jednostavne kontrole toka.

U ispisu 3.5 instrukcija kontrole toka je *if_icmplt* na indeksu 18. Sufiks te instrukcije je *lt* i označava relaciju „manje od" („*less than*"). Ako je broj manji od učitane konstante 5 skače se na odredišnu adresu 36 koja je početak tijela *else* naredbe. Ako se pak izvrši tijelo *if* naredbe, naredbom *goto* bezuvjetno se skače na odredišnu adresu 48. Instrukcijom *return* završava se trenutna metoda odnosno vraća se iz nje.

3.2.4. Klase i metode

Definirane klase sadrže varijable i svoje pripadne metode. Varijable klase pohranjuju se u tablicu atributa (*field table*), a metode se pohranjuju u tablicu metoda izvršne datoteke. Postoje instrukcije za stvaranje i rukovanje objektima pojedinih klasa, te pozivanje njihovih definiranih metoda. Primjer stvaranja objekta *Scanner* prikazan je u ispisu 3.5. Učitavanje statičkih atributa klasa obavlja instrukcija *getstatic*. Česti objekti koji se stvaraju su objekti poput tipova *Scanner, StringBuilder*, a primjer česte metode je metoda za ispis na standardni izlaz *println* koja se poziva nad objektom *out* klase *System*. Instrukcije za stvaranje objekta i pozivanje metoda učitavaju imena klase i imena metoda iz tablice konstanti na stog. U ispisu 3.5 instrukcija *new* indeksa 0 stvara objekt učitavajući ime klase *Scanner* indeksom tablice konstanti. Za rekonstrukciju izvornog koda potrebni su podaci o imenu metode i imenu objekta nad kojim se metoda poziva. Ta imena se učitavaju iz tablice konstanti na stog te se zatim parsiraju za dobivanje izraza izvornog koda. Poziv definiranih statičkih funkcija tj. metodi obavlja instrukcija *invokestatic* koja pomoću dodatnog bajta koji predstavlja indeks tablice konstanti dohvaća naziv te metode. Zatim se kao argumenti pozvane metode uzimaju vrijednosti sa stoga, koje se neposredno prije pozivanja metode dohvaćaju i pohranjuju na stog. Tada, započinje izvršavanje instrukcija pozvane metode a rezultat se tj. povratna vrijednosti, ako postoji, vraća pomoću stoga. Isto vrijedi i za ostale instrukcije pozivanja metodi. Instrukcije za pozivanje metoda nad objektima dodatno zahtijevaju i referencu objekta nad kojim je metoda pozvana, a ta referenca se učitava sa stoga nakon učitanih argumenata. Za prenošenje i vraćanje vrijednosti između metoda unutar JVM koristi se *okvir* (engl. frame). Okvir je skup podataka na vrhu stoga kojega čine: lista lokalnih varijabli trenutne metode, učitani argumenti metode i referenca na tablicu konstanti trenutne klase. Novi okvir stvara se svaki put kada se pozove nova metoda. Nakon povratka iz metode na stogu se nalazi samo povratna vrijednost i vraća se u posljednje aktivan okvir stoga odnosno nastavlja se daljnje izvođenje programa.

3.3. Faze dekompajliranja

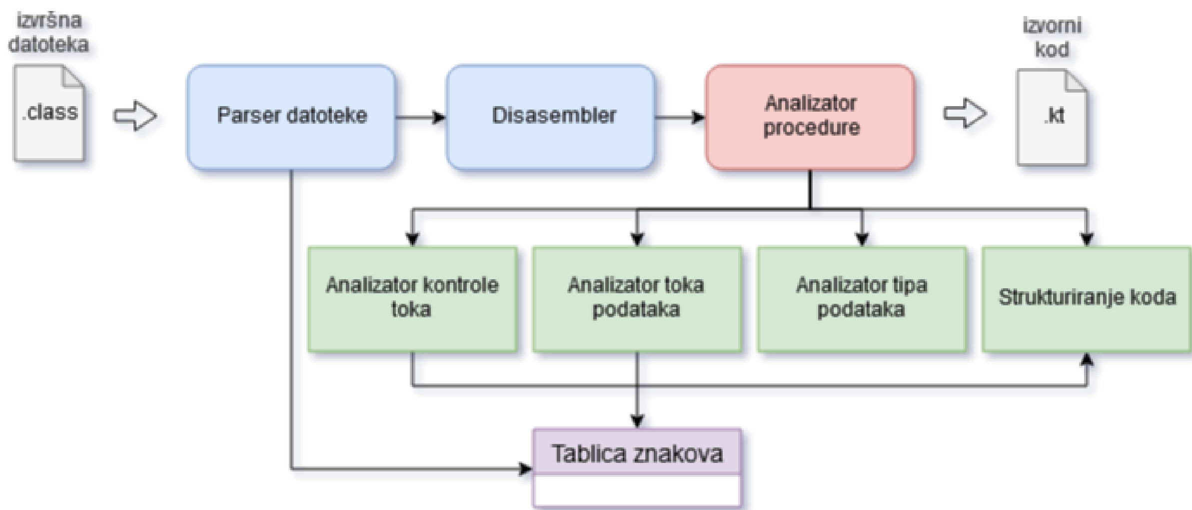
U ovom poglavlju opisuje se jedna od mogućih arhitektura dekompajlera i zadaci pojedinih faza dekompajliranja. Faze dekompajliranja opisuju se za općenite slučajeve, dok se u poglavlju 4. proučava programsko ostvarenje dekompajlera ovog rada koji posjeduje neke, ne sve, značajke i faze navedene arhitekture.

Bez obzira na ciljni jezik dekompajliranja, dekompajler uvijek nailazi na tri vrste problema iliti zadataka koje treba riješiti. To su:

1. izrazi i funkcije,
2. djelokruzi varijabla,
3. tipovi podataka i definicije funkcija.

Dekompajliranju se može pristupiti na takav način da se svaka analizirana instrukcija doslovno prevede u ekvivalentnu naredbu izvornog jezika. Rekonstruirajući tako izvorni kod nismo dobili nikakav bolji uvid u logiku programa nego što je to već prikazano izvršnim jezikom npr. Java bajt kodom. Time bi se, naprimjer, sve kontrole toka svele na najjednostavnije oblike ali bi se i povećalo broj nastalih izraza izvornog koda. Takav pristup je nepovoljniji što je ciljni program veći. Ako se pak veći broj instrukcija pretvara u jedan složeni izrazi ekvivalente radnje tada se moraju imati bolje metode analiziranja pojedinih skupova ili idioma instrukcija. Što je bolja analiza idioma instrukcija to se povećava broj uspješno pronađenih idioma i pronalazak optimalnog uzorka izvornog koda koji će predstavljati tu cjelinu.

Jedan od načina organizacije modula dekompajlera je prikazan na slici 3.1. Svaki modul obavlja specifičnu analizu nad učitanim ulaznim podacima, a povezanošću s ostalim dijelovima sustava osigurava rezultate drugim modulima. Moduli dekompajlera mogu učitanu izvršnu datoteku proći jednom ili više puta, prikupljajući pri tom što više podataka. Dekompajler sastavljen od jednog modula s jednim prolazom kroz instrukcije ima značajno manje mogućnosti pošto ne koristi naprednije tehnike analize postupaka. U dekompajleru s jednim modulom za analizu sve se suštinske faze odvijaju u tom modulu.



Slika 3.1 Moduli dekompajlera zaduženi za različite vrste analiza ulaznih podataka.

3.3.1. Obrada ulazne datoteke

Učitanu izvršnu datoteku potrebno je raščlaniti na sekcije koje ta datoteka sadržava, te iz takvih sekcija učitati sve podatke. Izvršne datoteke sadrže binarni zapis ne samo slijeda instrukcija već i drugih skladištenih podataka poput različitih tablica s podacima kao u Java izvršnoj datoteci. Stoga se prvo pronalaze podaci pohranjeni u obliku indeksiranih tablica te se zatim traži adresa zapisa prve instrukcije programa tzv. ulazna točka programa (*entry point*). U ovoj fazi osim otkrivanja adrese ulazne točke programa potrebno je i učitati sve potrebne podatke iz binarnog zapisa u za to prikladne strukture podataka. Takvi podaci će tijekom daljnje analize biti ili referencirani instrukcijama ili sadrže podatke o programu, npr. koje dodatne biblioteke i sučelja program koristi. Nakon učitanih struktura podataka i niza instrukcija, instrukcije u obliku binarnog koda trebaju se raspoznati i svakoj instrukciji dodijeliti njezin posao koji izvršava odnosno *disasemblirati*. Na taj način pratimo vrijednosti koje instrukcije stvaraju, učitavaju i prenose. Također pratimo i broj okteta koji neposredno slijede određenu instrukciju za određivanje referenci ili vrijednosti, jer ne predstavljaju svi okteti samo operacijske kodove instrukcija.

3.3.2. Analiziranje procedura

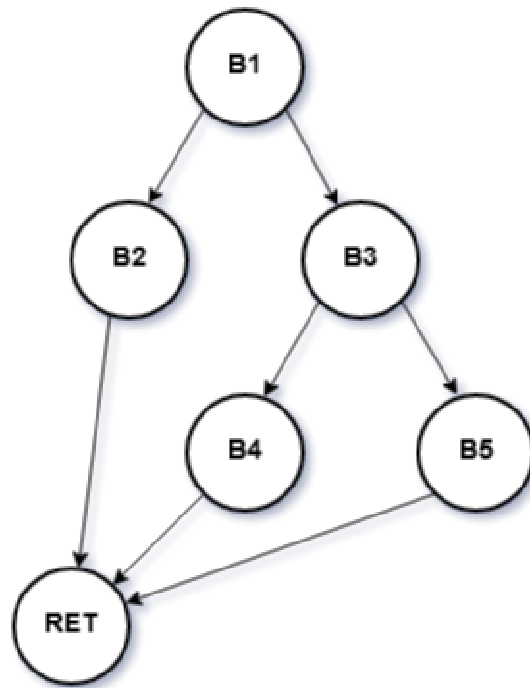
Nailaskom na pojedine instrukcije potrebno je odabrati akciju koja će se obaviti na temelju znanja o tipu podatka s kojim instrukcija operira, o poziciji u kodu i podacima iz referentnih tablica. Na taj način će se neki skup instrukcija odabrati kao idiom i započeti strukturiranje izraza izvornog koda kojega idiom predstavlja. Analizator kontrole toka zadužen je za izgradnju podatkovnih struktura koje pomažu strukturiranju koda. Te podatkovne strukture su najčešće grafovi opisani kasnije u ovom poglavlju. Analizator toka podataka koristi metode za određivanje djelokruga varijabla, propagacije vrijednosti varijabla, definicije dosega vrijednosti za novu varijablu itd. [15]. Podatkovna struktura u koju se učitavaju podaci iz tablica izvršne datoteke te skladišteni rezultati drugih analizatora na slici 3.1 prikazana je

imenom *tablica znakova.* Pohranjeni rezultati drugih analizatora mogu biti npr. imena i djelokruzi varijabli analizatora toka podataka.

Među prvim analizama koje se izvršavaju nakon učitavanja i disasembliranja ulazne datoteke je analiza kontrole toka. Analizom kontrole toka pokušavaju se pronaći mjesta neprekinutih nizova instrukcija i grade se tzv. *osnovni blokovi* (engl. basic blocks). Osnovni blok definiran je kao niz instrukcija u kojemu je prva instrukcija jedini mogući ulazak u izvođenje bloka, a zadnja instrukcija osnovnog bloka jedina je instrukcija s kojom se izlazi odnosno završava izvođenje bloka [16]. Posljedica toga je da pojedini osnovni blok započinje odredišnom adresom bilo kojih instrukcija skokova, a završava ili s bilo kojom instrukcijom skoka ili s instrukcijom koja se nalazi neposredno prije sljedećeg osnovnog bloka. Osnovnim blokovima predstavlja se niz instrukcija koje se zasigurno izvršavaju slijedno i neprekinuto. Nadalje, osnovne blokove možemo povezati na način da odredimo prethodnike i sljedbenike svakog osnovnog bloka. *Prethodnik* (engl. predecessor) trenutnom osnovom bloku je blok koji skače na početnu instrukciju trenutnog bloka. *Sljedbenik* (engl. successor) trenutnog bloka je onaj blok na kojega skače zadnja instrukcija trenutnog bloka. Skakanje ne mora nužno biti preskakivanje niza instrukcija već može biti i izvođenje neposredno sljedeće instrukcije. Povezivanjem osnovnih blokova s njihovim prethodnicima i sljedbenicima dobiva se *graf kontrole toka* (engl. control-flow graph, CFG). CFG predstavlja sve moguće putanje koda koje se mogu proći tijekom izvođenja programa odnosno sve moguće redoslijede izvođenja instrukcija. U ispisu 3.7 dan je niz instrukcija grupiranih u pripadne osnovne blokove, a rezultirajući CFG prikazan je grafom na slici 3.2.

```
0: new             #11      B1
3: dup
4: getstatic       #17
7: invokespecial #21
10: astore_0
11: aload_0
12: invokevirtual #25
15: istore_1
16: iload_1
17: iconst_5
18: if_icmple       36
21: ldc             #27     B2
23: astore_2
24: iconst_0
25: istore_3
26: getstatic       #31
29: aload_2
30: invokevirtual #37
33: goto            68
36: iload_1                 B3
37: iconst_5
38: if_icmpne       56
41: ldc             #39     B4
43: astore_2
44: iconst_0
45: istore_3
46: getstatic       #31
49: aload_2
50: invokevirtual #37
53: goto            68
56: ldc             #41     B5
58: astore_2
59: iconst_0
60: istore_3
61: getstatic       #31
64: aload_2
65: invokevirtual #37
68: return                  RET
```

Ispis 3.7 Osnovni blokovi instrukcija. Program učitava jedan broj tipa *Int* i provjerava je li broj veći, jednak ili manji od broja 5. Naredbena struktura u izvornom kodu je *if* - *else if* - *else*.

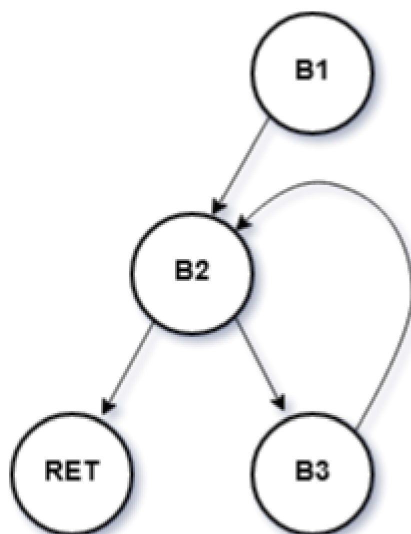


Slika 3.2 CFG niza instrukcija u ispisu 3.7.

Pomoću CFG dobivamo informacije o kakvoj se kontroli toka radi. CFG sadrži sve moguće nizove instrukcija, pa će tako *if* kontrolom toka svaka instrukcija biti posjećena samo jednom tj. niz instrukcija nema ponovljene prijašnje instrukcije. Dok će u slučaju kad se javljaju petlje, neki dijelovi koda posjetiti više nego jednom. CFG će tada imati povratnu vezu prema nekom od prijašnjih osnovnih blokova. Jednostavni primjer programske petlje prikazan je u ispisu 3.8, a pripadni CFG grafom na slici 3.3.

Code	Blok
0: iconst_1 1: istore_0 2: bipush 10 4: istore_1	B1
5: iload_0 6: iload_1 7: if_icmpgt 25	B2
10: iconst_0 11: istore_2 12: getstatic #15 15: iload_0 16: invokevirtual #21 19: iinc 0, 1 22: goto 5	B3
25: return	RET

Ispis 3.8 Petlja u kojoj se ispisuju cijeli brojevi 1-10.



Slika 3.3 CFG niza instrukcija u ispisu 3.8.

4. Programsko ostvarenje dekompajlera

U ovom poglavlju opisana je programska implementacija dekompajlera. U implementaciji je korištena dodatna biblioteka i dekompajler je organiziran u dva modula. Također su prokomentirani dobiveni rezultati dekompajliranja nekih jednostavnijih programa.

Za provođenje postupka pretvorbe Dex izvršnih datoteka u stablo direktorija Java izvršnih datoteka koristi se translator *dex2jar*. Translator prima kao argument u naredbenom retku datoteku s Dalvik izvršnim kodom (classes.dex). Rezultat je Java arhiva (engl. Java archive, JAR). Java arhiva sadrži stablo direktorija s Java izvršnim datotekama. Primjer stabla direktorija jednostavne Android aplikacije dobivenim alatom *dex2jar* prikazan je u ispisu 4.1. Java izvršne datoteke se pojedinačno dekompajliraju i generira se izvorni kod u izlaznu datoteku.

```
|classes-dex2jar.jar
|-android
|--...
|-androidx
|--...
|-com
|--example
|---first_app
|----MainActivity.class
|----BuildConfig.class
|----R.class
|----...
|-kotlin
|--...
|-kotlinx
|--...
|-lib
|--...
|-org
|--...
```

Ispis 4.1 Paketi klasa Android aplikacije. Rezultat prevođenja Dalvik izvršne datoteke *classes.dex* alatom *dex2jar*. Aplikacija naziva *first_app* ispisuje korisniku na zaslon tekst „Hello World“.

Dekompajler jezika Kotlin, koji se razmatra u ovom poglavlju, programski je ostvaren u razvojnom okruženju IntelliJ IDEA tvrtke JetBrains, a za jezik izgradnje dekompajlera odabran je programski jezik Java. Unutar razvojne okoline nalazi se *Maven* projekt s pripadnim konfiguracijskim datotekama [17].

Učitavanje Java izvršne datoteke obavljeno je pomoću dodatne biblioteke *Apache Commons BCEL* koja je dodana kao priključak Maven projektu [18]. Commons BCEL je biblioteka klasa

koje služe za učitavanje, parsiranje i rukovanje Java bajt kodom. Za otvaranje izvršne datoteke i učitavanje njezinih sekcija podataka navedenih u poglavlju 2.3. koristi se klasa *ClassParser* paketa *org.apache.bcel.classfile*. Metodom *parse* objekta ClassParser dobiva se objekt tipa *JavaClass* istoimenog paketa. Neki od dostupnih metoda za dohvaćanje atributa klase *JavaClass* prikazani su u tablici 4.1.

Metoda	**Opis**
getFileName()	Vraća naziv učitane datoteke.
getConstantPool()	Vraća objekt tipa *ConstantPool* koji predstavlja tablicu konstanti Java izvršne datoteke.
getMethods()	Vraća niz objekata tipa *Method*, gdje svaka metoda sadrži podatke o sebi i niz svojih bajt kod instrukcija.
getClassName()	Vraća ime definirane klase u datoteci.
getSuperClasses()	Vraća niz objekata tipa *JavaClass*; objekti predstavljaju roditeljske klase drugih Java izvršnih datoteka.

Tablica 4.1 Metode klase *JavaClass* biblioteke Apache Commons BCEL.

Klasa *Method* metodom *getCode* vraća objekt tipa *Code* u kojemu je sadržan niz instrukcija metode, međutim trebamo niz instrukcija u čitljivom obliku. Klasa *Code* metodom *getCode* vraća niz tipa *Byte* koji predstavlja niz operacijskih kodova instrukcija zajedno s dodatnim oktetima svake instrukcije.

Dekompajler je organiziran u dva modula odnosno dvije klase. Prvi modul *ClassParser* je zadužen za parsiranje ulazne datoteke pomoću funkcionalnosti biblioteke *Commons BCEL*. Drugi modul sadrži glavni program u kojemu se inicijalizira objekt *ClassParser* i dobivaju potrebni podaci za analizu. Podaci koji se dobivaju su ime učitane datoteke, tablica konstanti, imena metoda i niz instrukcija svake metode. Modul s glavnim programom obavlja sve analize nad bajt kodom. Isto tako, glavni modul ovog dekompajlera cjelokupnu analizu obavlja u jednom prolazu. Također, otvara se i izlazna datoteka u koju će se strukturirati dekompajlirani izvorni kod. Ime izlazne datoteke dobiva se na temelju imena učitane datoteke (*getFileName*).

Nakon dobivanja niza okteta koji predstavljaju Java bajt kod svake metode, iterira se po svim definiranim metodama i analiziraju se nizovi instrukcija. Prije analize niza instrukcija svake metode, potrebno je pronaći podatak o kojoj se metodi radi odnosno pronaći ime metode. Ime metode zajedno s njezinim modifikatorima vidljivosti i definiranim parametrima dostupni su pomoću objekta *Method* metodom *toString*. Cjelokupni naziv metode treba prilagoditi zapisu

definicije metode u programskom jeziku Kotlin stoga naziv metoda treba parsirati i generirati naziv i parametre u izlaznu datoteku kao što je prikazano u ispisu 4.2.

Pošto se instrukcije nalaze u binarnom obliku, potrebno je za svaki oktet raspoznati radi li se o operacijskom kodu instrukcije ili o dodatnim oktetima indeksa i vrijednosti. Početna instrukcija metode je zasigurno operacijski kod. Kako bi se pratio protok podataka svaka instrukcija treba obavljati svoj posao, a da bi obavljala svoj posao potrebno je definirati neke podatkovne strukture koje će instrukcije koristiti. Instrukcije koriste stog i lokalne varijable 0-4, a ako lokalne varijable nisu dostupne koriste se i indeksirane varijable gdje se u oktetu nakon operacijskog koda instrukcije navodi indeks varijable iz koje se učitava ili u koju se sprema. Za svaku učitanu instrukciju, nakon obavljanja njezine operacije, analiziraju se daljnje mogućnosti postupka generiranja izvornog koda. Ako se ne pronađe mogući postupak prelazi se na sljedeću instrukciju.

Pri analizi deklaracije i definicije varijabla koristi se sljedeći idiom:

- instrukcija za učitavanje vrijednosti iz tablice konstanti ili za stvaranje vrijednosti
- instrukcija za spremanje vrijednosti

Tako će spremanje vrijednosti pokrenuti postupak strukturiranja koda odnosno zapisivanja izraza u izlaznu datoteku izvornog koda. Primjer je prikazan u ispisu 4.2.

```
fun main() {
      var int0: Int = 1
      var str0: String = "Audi"
      var double0: Double = 0.1
}

fun printHey() {
      var str0: String = "Hey"
      var int0: Int = 0
      println(str0)
}
```

Ispis 4.2 Dekompajlirani izvorni kod izvršne datoteke *firstKt.class*, originalni izvorni kod je prikazan u ispisu 4.3.

```
fun main() {
    var a = 1
    var car = "Audi"
    var num = 0.1
}

fun printHey() {
    println("Hey")
}
```

Ispis 4.3 Originalni izvorni kod izvršne datoteke *firstKt.class*. Primjer definicija varijabli i definiranje dodatne metode.

U generiranoj datoteci s izvornim kodom u ispisu 4.2. primjećuje se više različitih stvari u odnosu na originalni izvorni kod 4.3. Prvo, imena varijabla nisu ista. Prevoditelj za vrijeme prevođenja skladišti sve definirane konstante i druge vrijednosti u tablicu konstanti i dodjeljuje sustav indeksiranja svakoj od tih vrijednosti. Instrukcije zato konstantama pristupaju pomoću indeksa. U tom slučaju imena varijabli postaju suvišan podatak. Ako imena više nisu dostupna moraju se dodijeliti proizvoljna. Zbog toga se praćenje imena varijabli ostvaruje internim brojačem imena koji se uvećava za imena varijabla pojedinih tipova. Druga stvar koja se primjećuje u kodu 4.2. je da se stvara razina indirekcije kod prenošenja vrijednosti varijabli u ostale varijable ili kao argumente metoda. Drugim riječima u funkciji *printHey* prvo se inicijalizira varijabla tipa *String*, dodjeljuje joj se vrijednost „Hey", zatim se ta varijabla predaje kao argument funkcije *println*. Uzrok tome je dekompajliranje deklaracije i definicije varijable pomoću već navedenog idioma. Instrukcija će prvo učitati vrijednost „Hey" iz tablice konstanti na stog, zatim će ju spremiti s vrha stoga u lokalnu varijablu. Time će nastati definicija nove varijable onoga tipa nad kojim operira instrukcija koja dohvaća konstantu. U slučaju funkcije *printHey* instrukcija *ldc* učitava konstantu „Hey" njenim indeksom tablice konstante, zatim instrukcija *astore_0* pohranjuje učitanu vrijednost u lokalnu varijablu. Treća stvar koja se razlikuje od originalnog izvornog koda je postojanje dodatne varijable *int0*. Za stvaranje te varijable očito postoji idiom para instrukcija. Prevoditelj radi niz optimizacija prilikom prevođenja izvornog programa. Optimizacije se rade restrukturiranjem izvornog koda u optimalni redoslijed instrukcija. Tako će, primjerice, *if* petlju izvornog koda koja sadrži uvjet koji je uvijek evaluiran u istinu prevoditelj zamijeniti nizom instrukcija bez instrukcije kontrole toka jer će se tijelo *if* petlje uvijek izvršavati. Postupkom takvih optimizacija nastaju i druge varijable i vrijednosti koje prate stanja nekih objekta ili metoda pa se pohrane neke vrijednosti koje nisu nužno znane dekompajleru. Primjer takve varijable je varijabla *int0*. Koja je nekakav produkt procesa prevođenja.

Pri dekompajliranju aritmetičkih i logičkih izraza nad varijablama treba imati na umu nadovezivanje operanda u jedan izraz čiji se rezultat pohranjuje u neku varijablu. Takav primjer opisan je u poglavlju 3.2.2. Takav izraz zahtijeva analiziranje toka podataka. Međutim, u dekompajleru s jednim modulom tj. jednom klasom analizatora to može biti problem pogotovo ako se datoteka učitava i prolazi samo jednom kao u ovom slučaju. Međutim, ako primijenimo sljedeći princip može se dobiti ulančavanje izraza. Princip se sastoji od niza podataka u koji se pohranjuju poznate dosadašnje varijable s njihovim definiranim vrijednostima. Ta nam struktura isto omogućuje praćenje koje su sve varijable već deklarirane i definirane. Pri

uzimanju vrijednosti operanda s vrha stoga traže se varijable koje sadrže tu vrijednost operanda. Također, primarni cilj nije izračunati vrijednost aritmetičke operacije već generirati izraz koji se pohranjuje u novu varijablu. Stoga, aritmetička operacija, npr. *iadd,* strukturira izraz pridruživanja aritmetičkih operacija jednoj varijabli. Primjer aritmetičke operacije prikazan je u ispisu 4.4.

```
fun main() {
      var int0: Int = 1
      var double0: Double = 0.1
      var double1: Double = int0 + 3 - double0
      var double2: Double = double0 - double1
}
```

Ispis 4.4 Dekompajlirani izvorni kod s različitim aritmetičkim operacijama. Originalni kod je prikazan u ispisu 4.5.

```
fun main() {
    var a = 1
    var b = 0.1
    var c = a + 3 - b
    var d = b - c;
}
```

Ispis 4.5 Originalni izvorni kod s različitim aritmetičkim operacijama.

U ispisu 4.4. također vidimo da je rezultirajuća varijabla onoga tipa čiji je najveći prioritet pretvorbe. *Int* se implicitno pretvara u *Double*. Pretvorba tipa podataka se postiže instrukcijama za konverziju tipova. U ovom slučaju to je instrukcija *i2d*. Pošto se dogodi takva konverzija automatski će se zahtijevati instrukcija *dadd* za zbrajanje brojeva tipa *double* stoga se izvršava i generiranje nove varijable koja je tipa *Double* na temelju toga. Posljedica toga je održavanje ispravnog tipa podataka odnosno bajt kod instrukcije se brinu o ispravnom tipu. Naravno, analiza tipa podataka prestaje biti zadovoljavajuća što se više rukuje sa složenijim tipovima podataka. Tada se zahtijeva uvođenje modula analizatora tipa podataka.

Analiziranje kontrole toka spada među najzahtjevnije zadatke dekompajlera, zato modul analize kontrole toka treba imati dovoljno dobre strukture podataka. Pomoću tih struktura podataka zaključuje se na kojem mjestu nastaju više mogućih putanja kroz niz instrukcija i na kojim mjestima se skače na već posjećene instrukcije odnosno kada nastaju petlje. Zatim se rekonstruira prikladna naredba kontrole toka. Međutim, dekompajleri s jednim modulom za analizu imaju poteškoća s takvim naredbenim struktura, pogotovo ako se nizovi instrukcija svake metode analiziraju u jednom prolazu. Ovakav dekompajler koristi znanje o adresama skoka instrukcija kontrole toka. Drugim riječima za svaku instrukciju skoka analizirat će se gdje se nalazi adresa odredišta i pokušati otkriti o kojem se naredbenom konstruktu radi. Jedan takav primjer prikazan je u ispisu 4.6.

```
fun main() {
	var ref0 = Scanner(System.in)
	var int0: Int = ref0.nextInt()
	if(int0 >= 5) {
		var str0: String = ">= 5"
		var int1: Int = 0
		println(str0)
		if(int0 == 5) {
			var str1: String = "= 5"
			var int2: Int = 0
			println(str1)
		}
	}
}
```

Ispis 4.6 Dekompajlirani kod ugniježđenih *if* naredbi. Originalni izvorni kod prikazan je u ispisu 4.7.

```
fun main() {
    var sc = Scanner(System.`in`)
    var n = sc.nextInt()
    if (n >= 5) {
        println(">= 5")
        if (n == 5) {
            println("= 5")
        }
    }
}
```

Ispis 4.7 Originalni izvorni kod ugniježđenih *if* naredbi. Program ispituje relaciju unesenog broja s brojem 5.

Analize *if* naredbi, kao i više njih ugniježđenih, različita je od analize *if-else* strukturi. *If* strukture bez pripadnih *else if* ili *else* uvjeta specifični su po odsutnosti naredbe bezuvjetnog skoka *goto*. Sav posao grananja preuzimaju instrukcije oblika *if_<tip>cmp<relacija>*. Ako je uvjet ispunjen preskače se na prvu instrukcija izvan tijela *if* naredbe. Ako postoji i naredba *else* tada će se, ako se izvršavaju instrukcije tijela naredbe *if*, naredbom *goto* preskočiti niz instrukcija naredbe *else*. To znanje se koristi prilikom nailaska na instrukcije grananja i analizira se okolina odredišta skoka. Ako ne postoji instrukcija *goto* neposredno prije odredišta skoka, to znači da se radi o jednostavnoj naredbi *if* bez ostalih uvjeta. Također, odredište skoka instrukcije pohranjuje se u podatkovnu strukturu koja sadrži sva odredišta skokova kako bi se označili krajevi tijela naredbe *if*. Nakon analize instrukcije grananja prelazi se na sljedeću instrukciju koja će se nalaziti u tijelu naredbe *if*. Kada se naiđe na instrukciju koja se nalazi na nekoj odredišnoj adresi spremljenoj u podatkovnoj strukturi sa svim odredišnim adresama strukturira se kod za završavanje tijela *if* naredbe.

U ispisu 4.6 primjećuje se nepostojanje indirekcije vrijednosti 5 u uvjetu *if* naredbe. Drugim riječima, vrijednost 5 se nigdje ne pohranjuje stoga ne postoji ni idiom za definiciju nove

varijable. Da je u izvornom kodu definirana varijabla, naprimjer, `var pet = 5` i uvjet postavljen poput `if (n >= pet)`, instrukcijama bi se stvorila vrijednost 5, pohranila u lokalnu varijablu čime nastaje idiom definicije varijable i učitala ponovno na stog neposredno prije instrukcije grananja. S druge strane, razine indirekcije i dalje postoje u korištenju metode *println*.

Daljnje analize složenijih izraza kontrole toka zahtijevaju metode opisane u poglavlju 3.3.2. pomoću podatkovnih struktura kao što su osnovni blokovi i pomoću odgovarajućih CFG grafova.

Za dekompajliranje izraza stvaranja objekta pojedinih klasa i pozivanje metoda na tim objektima analiziraju se instrukcije za stvaranje i rukovanje objekata. Razmotrimo poziv metode *println*. Metoda *println* dostupna je u osnovnom Kotlin paketu *kotlin.io*. Ta metoda poziva se nad objektom tipa *PrintStream*. Jedan od atributa (*field*) klase *System* je atribut *out* koji je tipa *PrintStream*. Kao što se metoda u Javi poziva izrazom *System.out.println()*, poziva se i u Kotlinu. Međutim Kotlin dozvoljava izostavljanje objekta *System.out*. Učitavanje statičkog atributa neke klase obavlja instrukcija *getstatic*. Instrukcija *getstatic* indeksom tablice konstanti dohvaća ime atributa pojedine klase. Tu vrijednost stavlja na stog. Nakon te instrukcije pozivaju se instrukcije za učitavanje podataka na stog. Ti stavljeni podaci predstavljaju argumente pozivajuće metode. Zatim se poziva metoda instrukcijama za poziv metoda kao što je, naprimjer, instrukcija *invokevirtual*. Metoda *invokevirtual* indeksom tablice konstanti dohvaća naziv metode koju poziva. Analizom instrukcije poziva metodi dobivamo naziv imena metode i s vrha stoga učitavamo parametre, a nakon parametara i referencu na objekt. Primjer poziva metode *println* prikazan je u ispisu 4.6. U tom primjeru učitani parametar s vrha stoga je varijabla tipa *String*, a referenca na objekt je *System.out* kojeg se smije izostaviti u jeziku Kotlin.

Nakon pretvorbe Dalvik izvršne datoteke u skup Java izvršnih datoteka translatorom *dex2jar*, dekompajliraju se dobivene izvršne datoteke s ovim dekompajlerom. Primjer dekompajliranja datoteke *MainActivity.class* jednostavne Android aplikacije prikazan je u ispisu 4.8. U tom primjerku koda vide se uspješne i neuspješne generacije konstrukta izvornog koda. Metoda *findViewById* parametrizirana tipom *TextView* nije uspješno pozvana s pripadnim argumentom *R.id.primitive_text_view* niti je takav izraz pridružen varijabli. Svakoj se komponenti za prikaz teksta (*TextView*) može pristupiti svojim jedinstvenim identifikatorom. Identifikator je varijabla klase *R* definiranoj u istom direktoriju kao i *MainActivity.class* (prikazano u ispisu 4.1.). Međutim, identifikatori se ne predaju u obliku podataka tipa *String* već kao tip *Int*. Što otežava cijeli proces jer se identifikator zapisan kao broj mora pretvoriti u čitljiv oblik identifikatora *R.id.primitive_text_view*. Treba se također saznati kako i gdje se pohranjuju brojevne vrijednosti identifikatora. Također postoje idiomi za koje nije uspješno pronađena procedura strukturiranja izvornog koda. U ispisu 4.8 neprepoznati izrazi su: pozivi *super.onCreate()* i *setContentView()* kao i postavljanje teksta komponente za ispis teksta *src.text*. Vidljiv je i nepoznati konstrukt provjere *Null* vrijednosti nad varijablom *src*.

```
fun onCreate(savedInstanceState: android.os.Bundle) {

      findViewById()

      var double0: Double = 1.0
      var int0: Int = 2
      var int1: Int = 0
      var double1: Double = 3.0
      println(double0)
      var ref0 = StringBuilder()

      checkExpressionValueIsNotNull(src)
      valueOf(2)
      var long0: Long = 3
      var int2: Int = 1
}
```

Ispis 4.8 Dekompajlirani izvorni kod jednostavne Android aplikacije. Originalni kod prikazan je u ispisu 4.9.

```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        var src = findViewById<TextView>(R.id.primitive_text_view);

        var a = 1.0
        var b = 2
        var c = a + b
        println(a)
        var sc = StringBuilder()

        src.text = (a + b).toString();
    }
```

Ispis 4.9 Android aplikacija koja zbraja dva definirana broja i prikazuje rezultat u vizualnoj komponenti *TextView*.

Opisani dekompajler uspješno rekonstruira definicije i deklaracije varijabli, stvaranje objekta, neke pozive metoda nad objektima i aritmetičke operacije unutar klasa Android aplikacije. Međutim, postoje veće skupine instrukcija koje nisu prepoznate i samo preskočene, kao i krivo generirani konstrukti. Isto tako, da s dijelovi koda koji se mogu dekompajlirati pomiješaju s dijelovima koda koji se ne mogu, redoslijedi takvih pomiješanih instrukcija ne bi tvorile ispravne korištene idiome. Uz to, složen sustav klasa Android aplikacije stvara problem za dekompajler ovakve složenosti odnosno ovakve jednostavnosti.

5. Zaključak

Sve veća korištenost programskog jezika Kotlin u razvoju Android aplikacija zahtijeva prilagodbu razvijatelja aplikacija novom jeziku. Jedna od prilagodbi bi bila i proizvodnja većeg izbora sofisticiranijih alata za analizu takvih aplikacija. Kao i ostali alati reverznih tehnika i dekompajleri imaju svoju svrhu za analizu ponajprije sigurnosti aplikacije.

Razmatranje ovoga rada je kako dekompajlirati izvršne datoteke Android aplikacije i dobiveni su rezultati dekompajliranja jednostavnijih programa. Opisana je struktura unutrašnjosti aplikacije i izgled jezika Kotlin i njegove izvršne datoteke. Kroz analizu Java bajt koda navedeni su ključni pojmovi i problemi koji tijekom implementacije dekompajlera zahtijevaju različite prilagodbe. Objašnjeni su različiti aspekti analize i dekompajliranja kao što su deklaracije i definicije varijable, kontrole toka, tipovi podataka i stvaranje objekata i pozivanje metoda nad njima. Česti modeli dekompajliranja koriste više-modularnost i koriste tehnike optimizacije procedura kako bi se generirao što precizniji i čitljiviji izvorni kod ciljnog programskog jezika. Također, opisane se tehnike obrade kontrole toka koje se koriste u modernijim dekompajlerima koje služe kao prijedlog u budućoj nadogradnji. Opisan je postupak prevođenja izvršnih datoteka Android aplikacije u Java izvršnu datoteku koju dekompajler uzima kao ulaznu datoteku. Kroz postupak implementacije programa dekompajlera naveli smo probleme i potencijalna rješenja tih problema. Opisani dekompajler je strukturiran u dva modula. Modul za parsiranje izvršne datoteke i modul za iteriranje po metodama i njihovim instrukcijama koji ujedno i provodi analizu tih instrukcija.

Razvijeni dekompajler koristi jedan modul za analizu izvršne datoteke, i analizira metode i niz instrukcija kroz jedan prolazak. Posljedica toga je loša analiza složenijih kontrola toka, analiza tipova podataka i analiza toka podataka. Zato budući postupci unaprjeđenja ovog dekompajlera bi bili stvaranje dodatnih modula za provođenje specijaliziranih analiza kao i stvaranja dodatnog modula za pohranu zajedničkih obrađenih podataka.

6. Literatura

[1] O. Lysne, „Reverse Engineering of Code“ in *The Huawei and Snowden Questions*, ch. 6. Cham, CH: Springer Nature, 2018. Available: https://link.springer.com/content/pdf/10.1007%2F978-3-319-74950-1_6.pdf.

[2] Z. Budimlić, K. Kennedy, Optimizing Java: Theory and Practice. Houston, TX: John Wiley & Sons, 1998. Available: https://core.ac.uk/reader/204877990.

[3] Kotlin Foundation, „Using Kotlin for Android Development“, Kotlin documentation, Available: https://kotlinlang.org/docs/reference/android-overview.html. [Accessed May 18, 2020].

[4] Kotlin Foundation, „Comparison to Java Programming Language“, Kotlin documentation, Available: https://kotlinlang.org/docs/reference/comparison-to-java.html. [Accessed May 19, 2020].

[5] Strategoxt website, List of java decompilers, Available: http://strategoxt.org/Transform/JavaDecompilers. [Accessed May 10, 2020].

[6] C. Toombs, „Meet ART, Part 1: The New Super-Fast Android Runtime“, Android Police, Nov. 2013., Available: https://www.androidpolice.com/2013/11/06/meet-art-part-1-the-new-super-fast-android-runtime-google-has-been-working-on-in-secret-for-over-2-years-debuts-in-kitkat/. [Accessed May 11, 2020].

[7] S. Thornton, „What are compilers, translators, interpreters, and assemblers“, Feb. 2017., MicrocontrollerTips. Available: https://www.microcontrollertips.com/compilers-translators-interpreters-assemblers-faq/. [Accessed May 10, 2020].

[8] Kotlin Foundation, „Calling Java code from Kotlin“, Kotlin documentation, Available: https://kotlinlang.org/docs/reference/java-interop.html. [Accessed May 18, 2020].

[9] Oracle Corp., „Oracle Java SE Support Roadmap“, Available: https://www.oracle.com/java/technologies/java-se-support-roadmap.html. [Accessed May 18, 2020].

[10] Oracle Corp., Java SE Documentation, The Java® Virtual Machine Specification, ch. 4.1. „The ClassFile Structure“, Available: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html. [Accessed May 23, 2020].

[11] Oracle Corp., Java SE Documentation, The Java® Virtual Machine Specification, ch. 2.11.1. „Types and the Java Virtual Machine“, Available: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.11.1. [Accessed May 23, 2020].

[12] pxb1988, „dex2jar“, Available: https://github.com/pxb1988/dex2jar.

[13] Android Studio documentation, „Enable multidex“, Available: https://developer.android.com/studio/build/multidex. [Accessed May 13, 2020].

[14] Oracle Corp., Java SE Documentation, The Java Class File Disassembler, Available: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html. [Accessed May 20, 2020].

[15] S. Chong, Lecture „Data-flow analysis“, Harvard University, School of Engineering and Applied Sciences, 2010., Available: https://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec02-Dataflow.pdf.

[16] J. A. Paulson, Lecture „Control-Flow Graph and Dana Flow Analysis“, Harvard University, School of Engineering and Applied Science, Available: https://www.seas.harvard.edu/courses/cs153/2018fa/lectures/Lec17-CFG-dataflow.pdf.

[17] Apache Software Foundation, Maven Project, Available: http://maven.apache.org/. [Accessed May 10, 2020].

[18] Apache Commons BCEL, Byte Code Engineering Library, Available: https://commons.apache.org/proper/commons-bcel/, https://github.com/apache/commons-bcel/. [Accessed May 10, 2020].

Sažetak

Dekompajliranje aplikacija pisanih u programskom jeziku Kotlin

Dekompajler je alat za rekonstrukciju izvornog koda ne temelju izvršne datoteke. Alat s tom mogućnošću koristi se za dobivanje uvida u unutarnje strukture i ponašanje nekog programa. Ovaj rad fokusira se na dekompajliranje aplikacija operacijskog sustava Android pisanih u programskom jeziku Kotlin. Opisana je struktura aplikacije i predočena je osnovna sintaksa jezika Kotlin. Za potrebe dekompajliranja definirane su izvršne datoteke koje se dekompajliraju. Naveden je proces dekompajliranja i programsko ostvarenje dekompajlera koji ima određeni opseg točnosti generiranja izvornog koda. Pomoću primjera ispisa Java bajt kod instrukcija opisana je analiza pojedinih instrukcija i njihovih grupa ili idioma. Kroz primjere generiranog izvornog koda programski ostvarenog dekompajlera prokomentirani su dobiveni izrazi i strukture.

Ključne riječi: dekompajliranje, Android aplikacija, Kotlin, Java izvršna datoteka, analiza Java bajt kod instrukcija, JVM, analiza idioma, analiza kontrole toka, CFG, strukturiranje izvornog koda

Summary

Decompilation of applications written in Kotlin programming language

Decompiler is a tool for reconstructing source code from an executable file. A tool with that capability is used to gain an insight into the internal structures and behaviour of a program. This paper focuses on decompiling Android operating system applications written in Kotlin programming language. The structure of the application is described and the basic syntax of the Kotlin language is presented. For decompiling purposes appropriate executables files are defined which are to be decompiled. The decompilation process is described and software realization of the decompiler, with limited accuracy in generating source code, is presented. Using the examples of Java bytecode instructions of certain simple programs, the analysis of each instruction and their groups or idioms are described. Through the examples of the generated source code of the presented decompiler in this paper, different expressions and structures are discussed.

Keywords: decompilation, Android application, Kotlin, Java executable, Java bytecode instructions analysis, JVM, analysis of idioms, control-flow analysis, CFG, code structuring