

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY
FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

ZPĚTNÝ PŘEKLADAČ JAZYKA JAVA

JAVA DECOMPILER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ZDENĚK ŽAMBERSKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN ROUPEC, Ph.D.

BRNO 2015

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Ústav automatizace a informatiky
Akademický rok: 2014/15

ZADÁNÍ DIPLOMOVÉ PRÁCE

student(ka): Bc. Zdeněk Žamberský

který/která studuje v **magisterském studijním programu**

obor: **Aplikovaná informatika a řízení (3902T001)**

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Zpětný překladač jazyka Java

v anglickém jazyce:

Java decompiler

Stručná charakteristika problematiky úkolu:

Cílem je vytvořit zpětný překladač bytového kódu virtuálního stroje Javy verze 6 a 7 do textu programu v jazyce Java. K realizaci bude použit jazyk Java.

Cíle diplomové práce:

1. Popište problematiku překladu a zpětného překladu.
2. Popište jazyk Java s důrazem na bytový kód.
3. Realizujte zpětný překladač a funkčnost ověřte na příkladech.

Seznam odborné literatury:

Venners, B.: Inside the Java Virtual Machine. Mc Graw-Hill, 2000.

Vedoucí diplomové práce: Ing. Jan Roupec, Ph.D.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2014/15.

V Brně, dne 1. 12. 2014



L.S.

Ing. Jan Roupec, Ph.D.
ředitel ústavu

doc. Ing. Jaroslav Katolický, Ph.D.
děkan

Abstrakt
Práce se zabývá vytvořením zpětného překladače pro jazyk Java. Cílem zpětného překladače je co nejlépe rekonstruovat zdrojový kód jazyka Java z class souborů. V práci je nejdříve rozebírán jazyk Java jako takový, kompilace Javy a struktura souborů class (zkompilovaná podoba javy). Je také přiblíženo fungování virtuálního stroje Javy a jeho instrukční sada. Poté se přejde k problematice zpětného překladu a popisu algoritmů navržených a použitých pro realizaci zpětného překladače. V práci jsou uvedeny příklady dekompilovaného kódu.

Summary
The goal was to create decompiler for Java programing language. Decompiler should reconstruct original Java source code from class files, representing its compiled form. First part of thesis focuses on Java langage, its compilation and structure of class file. Then Java Virtual Machine and its instruction set is discussed. After that thesis focuses on decompilation and algoritms designed and used for decompiler realization. Examples of decompiled code are presented.

Klíčová slova
Java, zpětný překladač, bytecode

Keywords
Java, decompiler, bytecode

ŽAMBERSKÝ, Z.*Zpětný překladač jazyka Java*. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2015. 75 s. Vedoucí diplomové práce Ing. Jan Roupec, Ph.D..

Prohlašuji, že práci jsem vytvořil samostatně, pod vedením Ing. Jana Roupce, Ph.D. a za použití citované literatury.

Bc. Zdeněk Žamberský

Obsah

1 Úvod **3**

2 Jazyk Java **5**
2.1 Balíčky . . . 5
2.2 Kompilační jednotka . . . 5
2.3 Importy . . . 5
2.4 Primitivní datové typy . . . 6
2.5 Operátory . . . 7
2.6 Řídící struktury . . . 8
2.7 Referenční typy . . . 9
2.8 Členové (Members) . . . 10
2.9 Fieldy . . . 11
2.10 Metody . . . 11
2.11 Vnořené třídy (Nested Classes) . . . 13

3 Kompilace jazyka java **15**
3.1 Java Development Kit (JDK) . . . 16
3.2 Projekty v Javě . . . 16

4 Soubor class **17**
4.1 Struktura class souboru na nejvyšší úrovni . . . 17
4.2 Constant pool . . . 19
4.3 Fieldy . . . 22
4.4 Metody . . . 23
4.5 Atributy . . . 24

5 Virtální stroj javy a jeho instrukční sada **31**
5.1 Virtuální stroj javy . . . 31
5.2 Typy rozlišované virtuálním strojem javy . . . 33
5.3 Instrukce virtuálního stroje javy (java bytecode) . . . 34
5.4 Vertifikace bytecodu . . . 41
5.5 Příklad vykonávání bytecodu . . . 42

6 Dekompilace javy a algoritmy používané dekompilerem **45**
6.1 Algoritmus rekonstrukce statementů . . . 46
6.2 Tvorba control flow grafu . . . 48
6.3 Kombinace algoritmů na tvorbu CFG a statementů . . . 49
6.4 Seřazení uzlů CFG . . . 53
6.5 Převod seřazených uzlů CFG do strukturovaného kódu . . . 57
6.6 Optimaliace řídících struktur . . . 67
6.7 Generování kódu javy . . . 68
6.8 Ukázka kódu generovaného dekompilerem . . . 68

7 Závěr **71**

8 Seznam použitých zkratek a symbolů **75**

1. Úvod

Tato práce se zabývá problematikou zpětného překladu jazyka Java a realizací zpětného překladače. Zpětný překlad (dekompilace) představuje proces, při kterém se z binárních souborů vzniklých kompilací rekonstruuje zdrojový kód. Proces dekompilace je však obecně obtížný, protože u binární podoby zkompilovaných souborů (s kódem přeloženým do podoby instrukcí) se zpravidla nepočítá se zpětným překladem. Těžiště práce bude tedy navržení a realizace algoritmů, jež budou zpětný překlad provádět.

Při řešení problému je však nutná nejen znalost jazyka Java, ale také jeho binární podoby po zkompilování. Protože tyto znalosti nejsou považovány za obecně známé, je v práci nejdříve rozebrán jazyk Java (jeho typy a pod.), a poté jeho kompilace. Dále je rozebírána vnitřní struktura souborů class, vznikajících kompilací Javy. Nakonec je přiblíženo fungování virtuálního stroje Javy (JVM) a jeho instrukční sada (označovaná také jako java bytecode).

V další části se již přejde k problematice dekompilace a popisu fungování zpětného překladače v té formě, v jaké bylo realizováno. Toto spočívá především v popisu algoritmů použitých pro dekompilaci. Realizace se zaměřovala především na problematiku rekonstrukce řídících struktur, protože tyto jsou považovány za klíčové z hlediska dekompilace.

2. Jazyk Java

V této kapitole bude rozebrán jazyk Java po syntaktické stránce. V rámci toho je jazyk srovnáván s jazyky C a C++, které lze v jistém smyslu považovat za předchůdce Javy.

2.1. Balíčky

Balíčky v Javě mají podobnou funkci jako namespace v C++. Balíčky tvoří stromovou strukturu s nepojmenovaným kořenem. Zdrojové soubory javy se umisťují do adresářové struktury odpovídající struktuře balíčků. Jako oddělující znak ve jménech vnořených balíčků se požívá tečka. Ve jménech balíčků se podle konvence používají pouze malá písmena. Aby se zamezilo kolizím jmen, je doporučeno, aby společnosti jména svých balíčků uvozovali jejich internetovou doménou v obráceném pořadí [4]. Takže například všechny balíčky společnosti s internetovou doménou company.com by začínaly com.company. Balíčky, které jsou přímo součástí jazyka java, jsou umístěny v balíčcích začínajících java a javax.

2.2. Kompilační jednotka

Jeden zdrojový soubor Javy představuje jednu kompilační jednotku. Zdrojové soubory jazyka Java mají příponu .java. Tyto musí být umístěny v adresářové struktuře v závislosti na jejich příslušnosti do balíčků. Každý soubor se zdrojovým kódem Javy začíná specifikací balíčku pomocí klíčového slova package (viz obr. 2.1). Výjimku tvoří zdrojové soubory umístěné v kořenovém (nepojmenovaném) balíčku. Zpravidla se používá jeden soubor pro jeden typ Javy (vyjma vnořených tříd).

```
package com.company.project;
```

Obrázek 2.1: použití klíčového slova package

2.3. Importy

Importy se umisťují na začátek kompilační jednotky (po specifikaci balíčku), rozsah platnosti představuje celou kompilační jednotku. I když se způsob použití podobá způsobu, jakým se v jazycích C a C++ požívá direktiva #include, jejich fungování blíží spíše klíčovému slovu using z jazyka C++. Importy slouží k tomu, aby nebylo nutno typy jazyka (třídy apod.) a jejich členy odkazovat plným jménem. Takže například pokud se provede import třídy java.awt.Graphics, je možné v rámci kompilační jednotky odkazovat třídu pouze jako Graphics. Lze také využít "divoké karty"tj. znaku * a importovat všechny třídy v balíčku (ne však rekurzivně). Java implicitně importuje balíček java.lang, proto je možné třídy jako Object a String odkazovat zkráceným jménem. Dále je možno použít využít statických importů, které umožňují importovat statické členy tříd a tyto pak odkazovat pouze jejich jménem.

```
import java.awt.Graphics;
import java.io.*;
import static java.util.Map.Etry;
```

Obrázek 2.2: příklad importů

2.4. Primitivní datové typy

Java definuje 8 primitivních typů. Mezi tyto patří celočíselné typy. Těmi jsou byte, char, short, int a long. Celočíselné typy v Javě jsou znaménkové (signed). Java nepodporuje neznaménkové varianty těchto typů (unsigned). Výjimkou je typ char, ten se pro aritmetické operace chová jako neznaménkový. (Ten se však většinou se však používá k ukládání znaků.) Kromě celočíselných typů Java podporuje 2 typy v plovoucí desetinné čárce (floating point). Těmito jsou typy float a double. Tyto jsou definovány standardem IEEE 754. Posledním primitivním typem je typ boolean. Ten se používá k uložení logické hodnoty true nebo false. Shrnutí typů a jejich rozsahy lze vidět v tabulce 2.1.

typ	velikost v bytech	rozsah
boolean	-	hodnoty true a false
byte	1	-128 až 127
short	2	-32768 až 32768
char	2	'\u0000' až '\0uffff'
int	4	-2^31 až 2^31-1
long	8	-2^63 až 2^63-1
float	4	dle IEEE 754
double	8	dle IEEE 754

Tabulka 2.1: Primitvní datové typy javy

Tyto typy mají na rozdíl o jazyků C a C++ přesně danou velikost, bez ohledu na platformu.

2.5. Operátory

Operátory v javě se velmi podobají těm z jazyků C a C++ a to i z hlediska priority a asociativity. Hlavní rozdíl je v absenci operátorů nad ukazateli v Javě. (Java neumožňuje přímou práci s ukazateli.) Přibyl však operátor instanceof, kterým lze testovat, jestli instance je příslušného typu (nebo jeho potomka). Přehled operátorů lze vidět v tabulce 2.2. Priorita je zde definována tak, že přednostně jsou vykonávány operace, mající nižší číslo ve sloupci priorita.

priorita	operátor	popis	asociativita
1	[]	přístup k prvku pole	zleva
1	()	volání funkce	zleva
1	.	přístup k členovi	zleva
2	++ - -	pre inkrementace dekrementace	zprava
2	++ - -	post inkrementace dekrementace	zleva
2	+ -	unární + a -	zprava
2	~	bitová negace	zprava
2	!	logická negace	zprava
2	(type)	přetypování	zprava
2	new	vytvoření nové instance	zprava
3	* / %	násobení, dělení, modulo	zleva
4	+ -	sčítání a odčítání	zleva
5	<< >>	bitový posun vlevo a vpravo	zleva
5	>>>	neznaménkový bit. posun vpravo	zleva
6	< <= > >=	menší (nebo rovno), větší (nebo rovno)	zleva
6	instanceof	test objektu na typ	zleva
7	== !=	rovnost, nerovnost	zleva
8	&	bitové and	zleva
9	^	bitové xor	zleva
10	|	bitové or	zleva
11	&&	logické and	zleva
12	||	logické or	zleva
13	? :	podmíněný operátor	zprava
14	=	přiřazení	zprava
14	*= /= += -= %= <<= >>= >>>= &= ^= |=	kombinovaný operátor a přiřazení	zprava

Tabulka 2.2: Operátory javy

2.6. Řídící struktury

Stejně jako operátory jsou i řídící struktury podobné jazkům C a C++. Java tedy obsahuje běžné řídící struktury jako if (if-else), switch a cykly while, do-while a for (viz obr. 2.3).

```
If(a){
   ...
} else if (!b){
   ...
} else {
  ...
}

switch(value){
   case 1: ... ; break;
   case 2: ... ; break;
   default: ... ;
}
```

```
while(x<10) {
   ...
}

do {
  ...
} while (true);

for(int i =0; i<5; ++i){
   ...
}
```

Obrázek 2.3: Řídící struktury struktury v jazyce java

V jazyce však nelze využívat goto. Java ale naproti tomu rozšiřuje možnosti statementů break a continue. V Javě lze s pomocí jmenovek (label) tyto statementy používat kupříkladu na nepřímo nadřazený cyklus. Statement break je pak možno použít také na pojmenovaný scope. (Příklady lze vidět na obr. 2.4.)

```
while(...){
   if(...) break;
   if(...) continue;
}

outerloop:
for(...) {
   for(...){
      if(...) break outerloop;
      if(...) continue outerloop;
   }
}

scope1:
{
   if(...) break scope1;
}
```

Obrázek 2.4: Možnosti statementů break a continue

2.7. Referenční typy

Kromě primitivních datových typů obsahuje Java řadu referenčních typů. Tyto typy jsou třída (class), rozhraní (interface), výčtový typ (enum), anotační typ (annotation type) a pole. Všechny tyto typy jsou objekty (ve smyslu OOP) a všechny jsou přímo nebo nepřímo potomky třídy java.lang.Object. (Tímto se odlišují od primitivních datových typů.) Instance těchto typů se předávají referencí.

Třída (Class)

Třídy jsou prvním z referenčních typů Javy. Na třídy, jakožto na referenční typ se uplatňuje dědičnost, jak bylo řečeno výše. Třídy v Javě nemohou mít, na rozdíl od C++, více než jednoho předka. K vyjádření dědičnosti se využívá klíčového slova extends. Třídu lze dále pomocí klíčového slova abstrakt učinit abstraktní. Instance abstraktních tříd není možno vytvářet a taková třída pak slouží pouze jako bázová třída pro jiné třídy (tedy aby se od ní dědilo).

```
public class Trida1 extends Trida2 {
    ...
}
```

Obrázek 2.5: Třída

Rozhraní (Interface)

Interface (rozhraní) představují alternativu k dědičnosti tříd. (Obě formy dědičnosti lze však kombinovat.) Třída (rozhraní) může implementovat více než jedno rozhraní a rozhraní tedy představují jakousi omezenou formu vícenásobné dědičnosti. K tomuto se využívá klíčové slovo implements. Metody rozhraní na rozdíl od tříd postrádají tělo (jsou abstraktní, viz dále). Na rozdíl od tříd také nelze vytvářet instance rozhraní, ale k třídám implementujícím rozhraní lze pomocí něj přistupovat. Dědičnost může existovat i mezi rozhraními samotnými, zde se ale na rozdíl od tříd užívá klíčové slovo extends. Následuje příklad dvou rozhraní a třídy implementující tato rozhraní.

```
public interface Rozhrani1 {

    public int getA();
}
```

Obrázek 2.6: Rozhraní 1

```
public interface Rozhranni2 {

    public int getB();
}
```

Obrázek 2.7: Rozhraní 2

```
public class Trida implements Rozhrani1, Rozhrani2 {
    int a = 1;
    int b = 2;

    public int getA(){
        return a;
    }

    public int getB(){
        return b;
    }
}
```

Obrázek 2.8: Třída implementující rozhraní 1 a rozhraní 2

Dále je uveden příklad použití rozhraní. Nejde sice o příklad praktického vyžití, ale demonstruje možnost přistupovat k metodám getA a getB skrze rozhraní Rozhrani1 a Rozhrani2, a tedy možnost, pracovat s třídami implementující daná rozhraní jednotným způsobem.

```
Trida trida = new Trida();
Rozhrani1 rozhrani1 = (Rozhrani1) trida;
rozhrani1.getA(); // vrátí hodnotu 1

Rozhrani1 rozhrani2 = (Rozhrani2) trida;
rozhrani2.getB(); // vrátí hodnotu 2
```

Obrázek 2.9: Příklad použití rozhraní

Enum

V Javě je výčtový typ (Enum) referenčním typem. Přestože z hlediska syntaxe jde o samostatný typ, ve skutečnosti jde o speciální třídu, která je potomkem třídy java.lang.Enum [2]. Na rozdíl od třídy jsou všechny instance typu enum předem vytvořeny a za běhu již nelze další vytvářet. K těmto instancím se poté přistupuje prostřednictvím statických fieldů. V ostatních ohledech se však Enum chová jako třída. Může mít metody, fieldy apod.

Anotační typ

Anotačním typem je možné anotovat. Instance tohoto typu se pak používají jako reprezentace anotací za běhu programu. Tento typ implicitně implementuje rozhraní java.lang.annotation.Annotation [1]. Žádnou další dědičnost však nelze na tento typ aplikovat. Metody tohoto typu nemají tělo, avšak mohou mít defaultní návratovou hodnotu, kterou tyto metody vrací, nejsou li specifikovány anotací.

2.8. Členové (Members)

Členy mohou být metody, fieldy a deklarace typů (vnořené třídy). Členové vždy přísluší nějakému referenčnímu typu. V kódu se členové nachází uvnitř těla referenčního typu, k němuž přísluší (jehož jsou členy). Vlastnosti členů lze ovlivňovat pomocí různých klíčových slov. Viditelnost členů lze modifikovat pomocí těchto klíčových slov:

public - přístupné odkudkoli

protected - přístupné pouze v rámci balíčku a z potomka

private - přístupné pouze v referenčního typu do něhož přísluší

Dále lze lze modifikovat, jestli jsou členvé vázáni ke konkrétní instanci. Při použití klíčového slova static se členové nevážou k instancím. Lze je tedy používat bez vytváření instance typu.

2.9. Fieldy

Fieldy jsou datové položky referenčního typu. Mohou být primitivních nebo referenčních typů. Každá třída má svoje vlastní hodnoty fieldů. Výjimkou statické fieldy, jež přísluší typu a pro přístup k nim není potřeba vytvářet instance. Fieldy dále mohou být finální (klíčové slovo final), jejichž hodnotu po jejich inicializaci již nelze měnit. Pro fieldy, jež představují konstanty, se v Javě využívá kombinace klíčových slov public, static a final.

2.10. Metody

Jazyk java nepodporuje funkce, jak jsou známy v jazyce C a C++. V Jazyce java se využívá metod. Každá metoda musí být deklarována uvnitř nejakého typu javy (tedy třídy, enum, apod.). Metoda je volaná pro určitou instanci příslušného typu (tedy typu, uvnitř kterého je deklarovaná nebo jeho potomka).

statické metody

Výjimku tvoří statické metody. Ty se deklarují pomocí klíčového slova static a jsou to metody, pro jejichž zavolání není potřeba instance typu. Při volání těchto metod se místo instance použije jméno třídy (nebo jinného typu), uvnitř které jsou deklarovány. Statické metody mohou přistupovat pouze ke statickým členům svého typu a nedědí se. Statické metody se proto podobají funkcím v jazycích C a C++. Využívají se i třídy které mají pouze statické metody a třída zde má pouze funkci namespace. (například třída java.lang.Math) Takovéto třídy dále mívají private konstruktor, aby nebylo možno vytvářet jejich instance.

abstraktní metody

Metody mohou být dále abstraktní, ty jsou deklarovány s užitím klíčového slova abstract a postrádají tělo. Tyto mohou být deklarovány pouze uvnitř abstraktní třídy (a rozhranní). Je požadováno přepsání (overide) těchto metod potomky pokud i oni nejsou abstraktní nebo již nebyly přepsány mezilehlým předkem. Všechny metody rozhranní jsou implicitně abstraktní. Abstraktní metoda nemůže být zároveň static, final nebo private.

finální metody

Opak k abstraktním metodám tvoří metody finální. (klíčové slovo final) Takovéto metody naopak nemohou být přepsány potomkem třídy. Metody deklarovány jako static nebo private se implicitně chovají jako finální.

ostatní metody

Metody které nejsou abstraktní nebo finální mohou (ale nemusí) být přepsány potomkem.

konstruktory

Dalším speciálním typem metody je konstruktor, který lze využít u tříd a typů enum. Tento je volán při vytváření nové instance. Konstruktor vypadá jako metoda se stejným názvem jako typ, v němž je deklarován a postrádající návratový typ. Pokud je konstruktor vynechán, vytvoří se implicitní konstruktor (tj. konstruktor, který je public, bez argumentů a s prázdným tělem). Pokud však třída obsahuje nějaký konstruktor, žádný implicitní se nevytváří. Kontruktor také implicitně zavolá konstruktor rodičovské třídy bez argumentů, pokud takový existuje a není specifikováno jinak. Konstruktor rodičovské třídy může být explicitně volán (odkazován jako super). Konstruktor může také volat jiný konstruktor třídy (ten je odkazován jako this). Je zde omezení, že konstruktor předka a/nebo jiný konstruktor třídy musí být volán na začátku konstruktoru.

```
class A extends B {

    // konstruktor je private může být volán pouze uvnitř třídy.
    private A(int x) {
        super(x);   // volání konstruktoru třídy B (rodičovské)
                    // s jedním argumentem typu int
    }

    public A(){
        this(1); // volání druhého konstruktoru
    }

}
```

Obrázek 2.10: Konstruktory

bloky static

Přestože blok static ve zdrojovém kódu jako metoda příliš nevypadá, jde vlastně také o metodu. Je to blok kódu ve složených závorkách uvnitř třídy, předcházený klíčovým slovem static. (V rámci jednoho typu smí být maximálně jeden takový blok.) Kód uvnitř static bloku se vykoná v rámci načítání typu, tedy při prvním "použití"referenčního typu. Vykoná se tedy před prvním voláním metody, přístupu k fieldu nebo vytvoření instance příslušného typu apod., jež způsobí načtení třídy virtuálním strojem Javy.

```
public class Trida {
    static int[] pole = new int[1000];

    static {
        for(int i=0; i < pole.length; ++i){
            pole[i] = i;
        }
    }
}
```

Obrázek 2.11: Užití static bloku

2.11. Vnořené třídy (Nested Classes)

V Javě je také možné vytvářet vnořené třídy. Třídy mohou být vnořeny v třídě nebo v metodě. Třídy mohou být vnořeny i vícenásobně. Typy vnořených tříd jsou vnitřní třídy, statické vnořené třídy, anonymní třídy a lokální třídy.

```
public class Enclosing {

   // inner class
   class Nested1 {
      ...
   }

   // static nested class
   static class Nested2 {
      ...
   }

}
```

Obrázek 2.12: Vnitřní a statická vnořená třída

Vnitřní třídy (Inner classes)

Vnitřní třídy jsou třídy, vnořené uvnitř jiné třídy. Instance takovýchto tříd lze vytvářet pouze v rámci obklopující třídy. Z vnitřní třídy lze volat metody obklopující třídy. To protože instance vnitřní třídy má skrytou referenci na instanci obklopující třídy, jež ji vytvořila.

Statické vnořené třídy (Static nested classes)

Ze statických vnitřních tříd již nelze volat metody obklopující třídy (kromě statických samozřejmě), avšak instance takovéto třídy lze vytvářet i mimo obklopující třídu. Obklopující třída má zde podobnou funkci jak balíček.

Lokální třídy (Local classes)

Lokální třídy jsou deklarované uvnitř metody. Instance takovéto třídy lze vytvářet pouze uvnitř metody.

```
public class Enclosing {

  public enclosingMethod(){

    // local class
    class LocalClass {
      ...
    }
  }

}
```

Obrázek 2.13: Lokální třída

Anonymní třídy (Anonymous classes)

Anonymní třídy jsou nepojmenované třídy, které lze použít v případě, kdy instance třídy je vytvářena pouze na jednom místě v kódu a třídu tedy není nutno pojmenovat. Anonymní třídy lze využít uvnitř třídy (přiřazení fieldu) anebo uvnitř metody. U dekalrace anonymní třídy se jako jméno použije jméno rodičovské třídy (nebo rozhranní), které je následované parametry konstruktoru v závorkách. Konstruktor anonymní třídy se vytváří implicitně na základě parametrů. Konstruktor pouze volá příslušný konstruktor (podle parametrů) rodičovské třídy. Konstruktor není možno deklarovat explicitně.

```
public class Enclosing {

  // přiřazení instance anonymní třídy fieldu
  Runnable anonymous1 = new Runnable() {

    public void run(){
      ...
    }
  }


  public void enclosingMethod() {

    // vytvoření instance anonymní třídy uvnitř metody
    Runnable anonymous2 = new Runnable() {

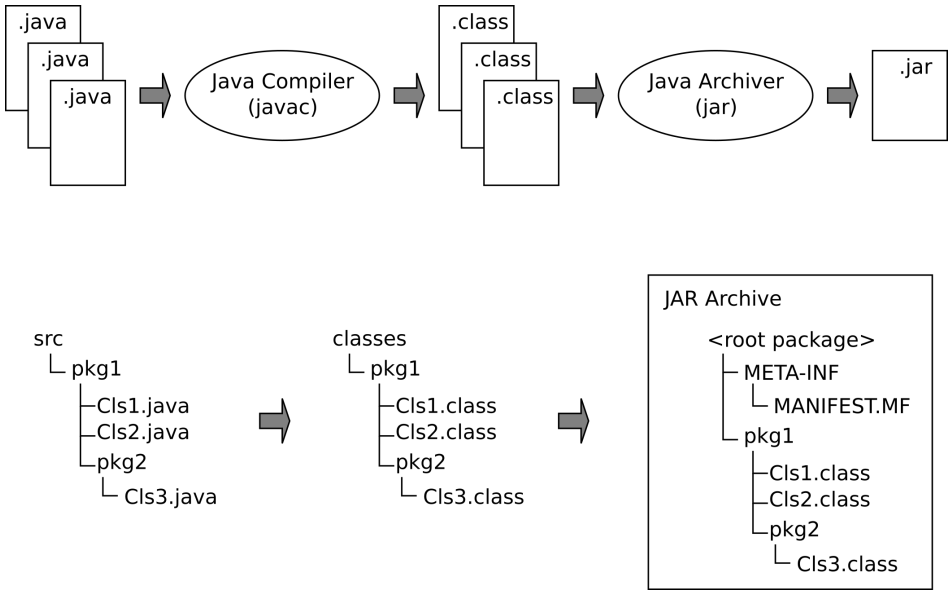
      public void run(){
        ...
      }
    }
  }

}
```

Obrázek 2.14: Anonymní třídy

3. Kompilace jazyka java

Aby bylo možno programy psané v jazyce java spustit, je nutné je zkompilovat. Protože však Java klade velký důraz na přenositelnost, není Java překládána pro konkrétní architekturu, nýbrž pro virtuální stroj Javy (JVM, viz dále). Po přeložení má program formu class souborů. Pro každý referenční typ Javy (tedy třídu, rozhraní, enum nebo anotační typ) se vytvoří jeden class soubor. Neplatí tedy obecně, že jednomu souboru se zrojovým kódem Javy, odpovídá jeden class soubor (např. každá vnořená třída má svůj vlastní class soubor). Class soubory jsou pak překladačem Javy umisťované do adresářové struktury odpovídající balíčkům. JVM sice umí s takto rozmístěnými class soubory pracovat přímo, většinou se však class soubory balí do JAR archivu. Kódování tohoto formátu je shodné se zip formátem. Class soubory se uvnitř archivu opět umisťují do adresářové struktury odpovídají balíčkům. Do JAR archivu je dále přidán manifest, což je textový soubor, obsahující páry název hodnota. Tento se používá k uložení cesty k main class (pro spustitelný soubor), informaci o použitém JDK, elektronickým podpisům apod.



Obrázek 3.1: Kompilace jazyka java

Class soubory samotné neobsahují pouze spustitelný kód, ale také různá metadata, která umožňují využívat kód jako knihovnu, při debugování apod. Díky tomu se např. v Javě nepoužívají hlavičkové soubory jako u jazyků C a C++, ale kompiler si může potřebné informace vyčíst z class souborů použitých knihoven.

3.1. Java Development Kit (JDK)

Pro kompilaci kódu psaného v jazyce java je nutno mít nainstalován JDK (java development kit), ne tedy pouze JRE (java runtime environment), protože to obsahuje pouze běhové prostředí. JDK obsahuje, kromě běhového přostředí, také řadu nástrojů využívaných při vývoji v jazyce java. Mezi tyto patří i javac (java compiler), jar (java archiver), jdb (java debugger). JDK dále obsahuje hlavičkové soubory JNI(java native interface) a nástroj javah pro generování hlavičkových souborů k nativním metodám, které lze využít pokud je nutné volat nativní kód z Javy nebo obráceně. Avšak je zde také nástroj javap (java dissasembler), který umí zobrazit bytecode zkompilovaných metod v lidsky čitelné podobě. JDK je však pouze jakousi standardizovanou sadu nástrojů. Existuje vícero alternativních implementací JDK. Lze tedy využít JDK od společností Oracle, IBM nebo třeba projekt OpenJDK.

3.2. Projekty v Javě

JDK však nedefinuje žádnou standartní podobu projektů Javy. Pokud tedy nechceme volat Java compiler (a jiné nástroje JDK) přímo, je vhodné využít dalších nástrojů, s pomocí kterých lze projekty vytvářet, jako jsou např. Apache Ant a Apache Maven. Tyto mívají podporu přímo v rámci IDE (Netbeans, Eclipse). Maven navíc umožňuje automatické stažení knihoven závislostí (označované jako artefakty) z jeho oficiálního repozitáře. (U Antu je možné k tomuto využít správce závislostí Ivy)

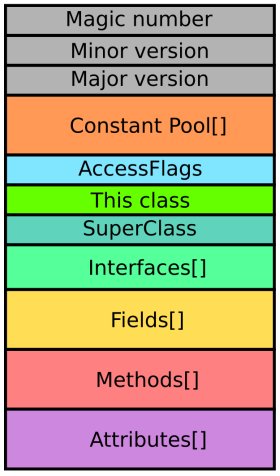
4. Soubor class

V této sekci je popsána struktura class souboru a nastíněn jeho vztah k původnímu zdrojovému kódu v jazyce Java. Z důvodu přehlednosti a srozumitelnosti je zde popsána pouze obecná struktura a není zabíháno do detailů reprezentace dat v souboru. Detailní specifikaci class souboru lze nalézt ve specifikaci [7].

Pro vyjádření vícero položek stejného typu uvnitř souboru, předcházených hodnotou určující jejich počet, je použita dvojice hranatých závorek. Slovní spojení "index na"je používáno v souvislosti s indexováním položky (konstanty) v constant poolu, pokud není řečeno jinak.

4.1. Struktura class souboru na nejvyšší úrovni

Strukturu class souboru na nejvyšší úrovni lze vidět na 4.1.




```java
package example;

import example.annotations.SomeAnnotation;
import java.util.Date;

@SomeAnnotation
public class ExampleClass extends Object
    implements Comparable<ExampleClass> {

    private int toCompare;
    public static final float one = 1.0f;
    public static final String hello = "hello";
    protected Date date = new Date();

    public ExampleClass(int i) {
        this.toCompare = i;
    }

    public static int add(int a, int b) {
        return a + b;
    }

    @Override
    public int compareTo(ExampleClass ec) {
        return toCompare - ec.toCompare;
    }
}
```

Obrázek 4.1: Soubor class

magic - číselná hodnota indentifikující class soubor. Ta je 0xCAFEBABE a představuje první 4 byty class souboru.

minor version, major version - hodnoty určující verzi class souboru

ConstantPool[] - konstanty používané v rámci class souboru, podklad pro tvorbu runtime constant poolu (viz dále)

access flags - bitová maska access flagů. Určuje zda class file představuje třídu, rozhraní, enum nebo anotační typ a další vlastnosti. Jednotlivé access flagy lze vidět v tab. 4.1.

jméno	popis
ACC_PUBLIC	deklarována public
ACC_FINAL	deklarována final
ACC_SUPER	pro javu vždy nastaven
ACC_INTERFACE	class file představuje rozhranní
ACC_ABSTRACT	deklarována abstract
ACC_SYNTHETIC	syntetická třída (není přítomna ve zdrojových kódech)
ACC_ANNOTATION	soubor class představuje anotační typ
ACC_ENUM	soubor class představuje enum

Tabulka 4.1: Access flags třídy

this class - index na CONSTANT_Class představující třídu

super class - index na CONSTANT_Class představující rodičovskou třídu

interfaces[] - výčet indexů na CONSTANT_Class představující implementovaná rozhraní

fields[] - fieldy (datové položky) třídy

methods[] - metody třídy

attributes[] - attributy třídy

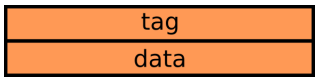
4.2. Constant pool

Constant pool je umístěn na začátku souboru. Je tvořen položkami (konstantami) různých typů. Tyto představují mimo jiné textové řetězce kódované v UTF-8, kontanty primitivních datových typů javy, dvojice jméno-typ, reference na fieldy, metody a třídy. Některé typy položek jsou složeny z více jednodušších položek na které odkazují např. CONSTANT_NameAndType odkazuje na dvě položky typu CONSTANT_Utf8 představující jméno a typ, ten je poté sám odkazován dalšími typy položek.

Na položky v konstant poolu je poté odkazováno ve zbytku soboru. Constant pool v class souboru také slouží jako základ pro vytvoření runtime constant poolu. Do tohoto se poté odkazuje z bytecodu (spustitelného kódu JVM). V něm jsou poté na příslušných pozicích hodnoty příslušných konstant a symbolické odkazy na metody a třídy jsou při linkováni nahrazeny příslušnými referencemi. (Podrobněji o runtime constant poolu později.)

Typy položek (konstant) v constant poolu

Všechny konstanty v constant poolu mají hlavičku obsahující tag (1 byte), určující typ konstanty. Velikost a typ dat se poté odvíjí od typu konstanty.



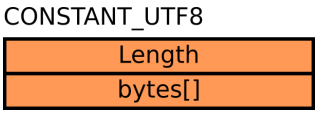
Obrázek 4.2: položka constant poolu

Typy konstant které se mohou objevit v constant poolu a jim příslušící tagy lze vidět v tab. 4.2. Protože tag je přítomen u všech typů konstant, je dále vynecháván.

typ konstanty	popis
CONSTANT_Class	představuje třídu Javy
CONSTANT_Fieldref	symbolický odkaz na field
CONSTANT_Methodref	symbolický odkaz na metodu
CONSTANT_InterfaceMethodref	symbolický odkaz na metodu rozhraní
CONSTANT_String	konstanta typu String
CONSTANT_Integer	konstanta typu int nebo kraších (short, char, byte a boolean)
CONSTANT_Float	konstanta typu float
CONSTANT_Long	konstanta typu long
CONSTANT_Double	konstanta typu double
CONSTANT_NameAndType	popisující představující dvojici jméno-typ
CONSTANT_Utf8	textový řetězec (používající kódování UTF-8)
CONSTANT_MethodHandle	pro dynamicky typové jazyky
CONSTANT_MethodType	pro dynamicky typové jazyky
CONSTANT_InvokeDynamic	pro dynamicky typové jazyky

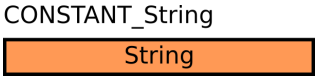
Tabulka 4.2: Tabulka typů položek v konstant poolu

CONSTANT_UTF8 - jde o textový řetězec kódovaný pomocí UTF-8 a maximální délkou 65535 bytů. Tento se používá k reprezentaci všech textových řetězců v rámci class souboru.



Obrázek 4.3: CONSTANT_UTF8

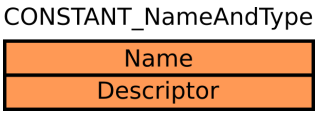
CONSTANT_String reprezentuje konstantní textový řetězec v Javě. Představuje tedy instanci třídy java.lang.String. Tato se vytvoří v rámci načítání class souboru JVM, viz dále.



Obrázek 4.4: CONSTANT_String

String - index na Constant_UTF8, která bude použita pro vytvoření objektu string.

CONSTANT_NameAndType představuje dvojici jméno a typ.

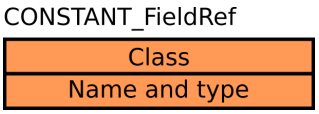


Obrázek 4.5: CONSTANT_NameAndType

Name - index na Constant_UTF8, reprezentující jméno

Type - index na Constant_UTF8, reprezentující typ

CONSTANT_Fieldref je symbolická reference na field.

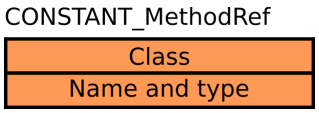


Obrázek 4.6: CONSTANT_Fieldref

Class - index na Constant_Class, reprezentující třídu v níž je field deklarován.

Name and type - index na položku typu Constant_Name_And_Type se jménem a typem fieldu

CONSTANT_Methodref je symbolická reference na metodu.

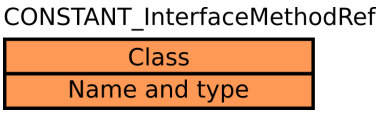


Obrázek 4.7: CONSTANT_Methodref

Class - index na Constant_Class, reprezentující třídu v níž je metoda deklarována.

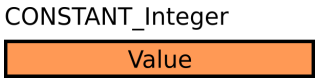
Name and type - index na Constant_Name_And_Type se jménem a deskriptorem (řetězec popisující datový typ parametrů a návratový typ) metody.

CONSTANT_InterfaceMethodref je symbolická reference na metodu rozhranní. Shoduje se s CONSTANT_Methodref_info s tím rozdílem, že Class je index na Constant_Class, reprezentující rozhranní.



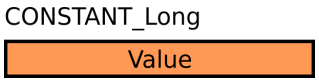
Obrázek 4.8: CONSTANT_InterfaceMethodref

CONSTANT_Integer reprezentuje 32-bitovou celočíselnou konstantu. Používá se pro reprezentaci konstant typů int, short, char, byte a boolean.



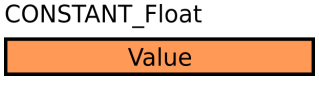
Obrázek 4.9: CONSTANT_Integer

CONSTANT_Long reprezentuje konstantu typu long, tedy 64-bitovou celočíselnou konstantu.



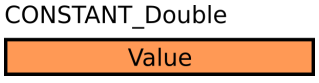
Obrázek 4.10: CONSTANT_Long

CONSTANT_Float reprezentuje konstantu typu float. (32-bit, IEEE 754 single)



Obrázek 4.11: CONSTANT_Float

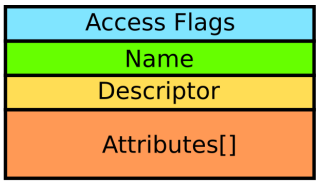
CONSTANT_Double reprezentuje konstantu typu double. (64-bit, IEEE 754 double)



Obrázek 4.12: CONSTANT_Double

4.3. Fieldy

Fieldy jsou reprezentovány stejnojmenou strukturou uvnitř class souboru. Třída může obsahovat více fieldů stejného jména, dvojice Name a Descriptor musí být v rámci třídy unikátní.




```
public static int someField;
private String s;
```

Obrázek 4.13: fieldy

Access Flags - bitová maska určující vlastnosti fieldu. Seznam access flagů fieldů lze vidět v tabulce 4.3

jméno	popis
ACC_PUBLIC	deklarován public
ACC_PRIVATE	deklarován private
ACC_PROTECTED	deklarován protected
ACC_STATIC	deklarován static
ACC_FINAL	deklarován final
ACC_VOLATILE	deklarován volatile
ACC_TRANSIENT	deklarován transient
ACC_SYNTHETIC	syntetický field (není přítomen ve zdrojových kódech)
ACC_ENUM	jde o element enum

Tabulka 4.3: Access flags fieldů

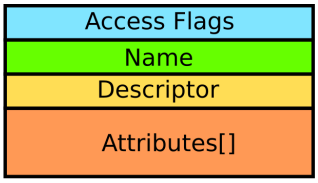
Name - index na CONSTANT_Utf8 představující jméno fieldu

Descriptor - index na CONSTANT_Utf8 s deskriptorem, to je textový řetězec, reprezentující typ fieldu. Descriptory fieldů v příkladu by vypadaly následovně: "I"a "Ljava/lang/String;". Podrobnější informace o tvorbě desckriptoru lze nalézt ve specifikaci.

Attributes[] - atributy fieldu (viz dále)

4.4. Metody

Pro reprezentaci metod uvnitř class souboru je opět užita stejnojmenná struktura. Třída může obsahovat více metod stejného jména, avšak dvojice Name, Descriptor musí být v rámci třídy unikátní.




```
public final int someMethod(String s, int a) {
    ...
}
```

Obrázek 4.14: methody

Access Flags - bitová maska určující vlastnosti metody. Seznam access flagů metod lze vidět v tabulce 4.4.

jméno	popis
ACC_PUBLIC	metoda deklarována public
ACC_PRIVATE	metoda deklarována private
ACC_PROTECTED	metoda deklarována protected
ACC_STATIC	metoda deklarována static
ACC_FINAL	metoda deklarována final
ACC_SYNCHRONIZED	metoda deklarována synchronized
ACC_BRIDGE	bridge metoda (generována kompilátorem)
ACC_VARARGS	deklarována s proměnným počtem argumentů
ACC_NATIVE	metoda deklarována static (provádí nativní kód)
ACC_ABSTRACT	metoda deklarována abstract
ACC_STRICT	metoda deklarována strictfp
ACC_SYNTHETIC	syntetická metoda (není přítomna ve zdrojových kódech)

Tabulka 4.4: Access flags method

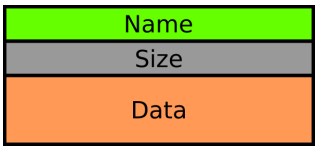
Name - index na CONSTANT_Utf8 představující jméno metody

Descriptor - index na CONSTANT_Utf8 s deskriptorem, to je textový řetězec, který popisuje datový typ parametrů a návratový datový typ. Pro ukázkovou metodu by vypadal následovně: "(Ljava/lang/String;I)I". Podrobnější informace o tvorbě desckriptoru lze nalézt ve specifikaci.

Attributes[] - atributy metody (viz dále)

4.5. Atributy

Atributy lze přiřazovat fieldům, metodám nebo celému typu (souboru). Atribut je tvořen názvem a hodnotou udávající velikost jeho dat a data. Názvem se určuje druh atributu. Hodnota udávající velikost dat umožňuje přeskočit neznámé atributy. Protože takovouto hlavičku mají všechny atributy nebude již dále uváděna. Uváděný název atributů se shoduje s jejich jménem v class souboru.



Obrázek 4.15: attribut

Name - index na CONSTANT_Utf8 představující jméno atributu

Size - hodnota udávající velikost dat atributu

Data - data atributu (liší se podle typu)

Atribut ConstantValue

Atribut ConstantValue je určený pro fieldy. Používá se k inicializaci statických fieldů s konstantami.



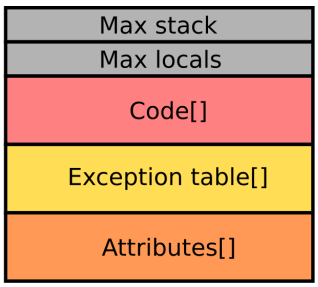

```
public static final int TEN = 10;
public static final String HELLO = "hello"
```

Obrázek 4.16: attribut ConstValue

Conant value - index na CONSTANT_Long, CONSTANT_Float, CONSTANT_Double, CONSTANT_Integer nebo CONSTANT_String v constant poolu

Atribut Code

Atribut Code je používaný pro metody. Každá metoda, která není abstraktní nebo nativní metodou musí mít právě jeden tento atribut. Tento atribut obsahuje zkompilovaný kód metody pro virtuální stroj Javy.




```
public final int someMethod(String s, int a) {
    ...
}
```

Obrázek 4.17: attribut Code

Max stack - určuje maximální hloubku operand stacku této metody

Max locals - vyjadřuje počet lokálních proměnných (v bytecodu), které daná metoda využívá

Code[] - posloupnost bytů reprezentující zkompilovaný kód methody pro virtuální stroj javy

Exception table[] - tabulka výjimek. Určuje pro každou zachytávanou výjimku v metodě typ, rozsah v němž je zachytávána a adresu první instrukce handleru (uvnitř kódu), který se provede při vyhození výjimky.

Attributes[] - atributy code atributu

Atribut Exceptions

Atribut Exceptions je používaný pro metody. Určuje, jaké kontrolované výjimky (checked exceptions) může metoda vyhazovat. Tyto se ve zdrojovém kódu specifikují pomocí klíčového slova throws.




```
public Image readImage(File f) throws IOException,
  FileNotFoundException {
  ...
}
```

Obrázek 4.18: attribut Exceptions

Exceptions[] - výčet indexů na CONSTANT_Class reprezentující třídy vyhazovaných vyjímek

Atribut InnerClasses

InnerClass je atribut, který lze použít na třídu. Obsahuje položku pro každou vnořenou třídu (nested class) používanou nebo deklarovanou v rámci třídy. Vnořené třídy mohou být buď vnitřní třídy (Inner class) nebo statické vnořené třídy (static nested class).




```
import java.util.Map;

public class ThisClass {

    public Map.Entry entry;

    public class InnerClass {
      ...
    }

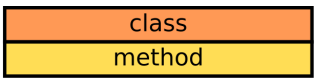
    public static class StaticNestedClass {
      ...
    }

}
```

Obrázek 4.19: attribut Exceptions

Atribut EnclosingMethod

Atribut EnclosingMethod je použitelný u třídy. Tento atribut se používá u anonymních a lokálních tříd a obsahuje informaci o třídě a metodě ve které byla třída ve zdrojovém kódu deklarována. Uvažovaná třída (v jejímž class souboru se tento atribut používá) je v příkladech označena zelenou barvou.




```
public class OuterClass1 {

    public void someMethod(){
        class LocalClass {
            ...
        }
        ...
    }
}

public class OuterClass2 {

    public void someMethod(){
        Runnable annonymous = new Runnable() {
            ...
        }
        ...
    }
}

public class OuterClass3 {

    Runnable ananymous = new Runnable(){
        ...
    }
    ...
}
```

Obrázek 4.20: attribut Exceptions

class - index na CONSTANT_Class reprezentující obklopující třídu (nejvnitřenšjí)

method - index na CONSTANT_Methodref reprezentující přímo obklopující metodu nebo 0 pokud třída není přímo obklopena metodou

Atribut Synthetic

Atribut Synthetic může být použit na třídu, metodu nebo field. Je generován překladačem javy a vyjadřuje, že třída, field, nebo metoda není přítomna v původním zdrojovém kodu. Takovéto "pomocné" fieldy a metody se generují například u vnořených tříd.

Atribut LineNumberTable

Atribut LineNumberTable je využívaný code attributem. Vyjadřuje mapování instrukcí bytecodu na čísla řádků v původním zdrojovém kódu. Využívá se při debugování. S pomocí něho je pak možné krokovat program po řádcích, vkládat breakpointy apod.

Atribut LocalVariableTable

LocalVariableTable je atribute používáný code atributem a obsahuje informace o lokálních proměnných funkcí v původním zdrojovém kódu, jejich rozsahu platnosti a mapování na lokální proměnné bytecodu. Tento atribut se používá při debugování, aby bylo možné sledovat změny hodnot proměnných při krokování programu.

Atribut LocalVariableTypeTable

Struktura tohoto atributu je shodná LocalVariableTable, tento atribut však obsahuje informace genericích.

Atribut Deprecated

Atribut Deprecated může být použit na třídu, metodu nebo field. Generuje se pro třídy, fieldy a metody anotované ve zrojovém kódu jako @Deprecated. Toho se používá například u zastaralých metod, které byly nahrazeny jinou metodou a metoda už by se tedy neměla v novém kódu používat.

```
@Deprecated
public class SomeClass {

   @Deprecated
   private String someField;

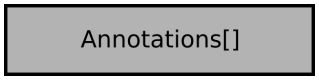
   @Deprecated
   public void someMethod(){
      return data;
   }

}
```

Obrázek 4.21: anotování pomocí @Deprecated

Atributy RuntimeVisibleAnnotations a RuntimeInvisibleAnnotations

Atributy RuntimeVisibleAnnotations a RuntimeInvisibleAnnotations mohou být použity na třídu, metodu nebo field. Jsou pomocí nich kódované anotace. Rozdíl mezi RuntimeVisibleAnnotations a RuntimeInvisibleAnnotations je ten, že anotace reprezentované pomocí RuntimeVisibleAnnotations lze za běhu zjistit pomocí reflection api. Anotace reprezentované pomocí RuntimeInvisibleAnnotations naopak pomocí reflection API zjistitelné být nesmějí.



Obrázek 4.22: attribut RuntimeVisibleAnnotations a RuntimeInvisibleAnnotations

Annotations[] Obsahuje položky představující jednotlivé anotace.

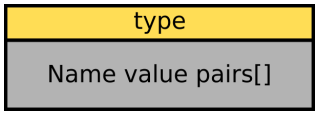
```
@Entity
@Table(name = "USERS")
public class User {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long Id;

    ...
}
```

Obrázek 4.23: anotace ve zdrojovém kódu Javy

Položka annotation Strukturu položky annotation lze vidět na obr. 4.24.

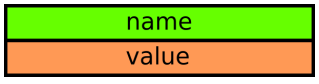


Obrázek 4.24: položka annotation

type Index na CONSTANT_Class reprezentující typ anotace.

Name value pairs Páry jméno hodnota pro příslušnou anotaci.

Dvojice jméno hodnota Strukturu dvojice jméno hodnota lze vidět na obr. 4.25.



Obrázek 4.25: dvojice jméno hodnota

name Index na CONSTANT_Utf8 se jménem.

element value - hodnota může jednoho z následujících typů: primitivní datový typ Javy, String, enum, Class (tedy objekt reprezentující třídu), anotační typ nebo pole s prvky některého z těchto typů těchto typů. Podrobnosti o tom, jak se příslušné typy kódují, lze nalézt ve specifikaci.

Atributy RuntimeVisibleParameterAnnotations a RuntimeInvisibleParameterAnnotations

Atributy RuntimeVisibleParameterAnnotations a RuntimeInvisibleParameterAnnotations mohou být použit u metod a reprezentují anotace parametrů metod. Rozdíl mezi RuntimeVisibleParameterAnn a RuntimeInvisibleParameterAnnotations opět spočívá v zjistitelnosti anotací pomocí reflection api.

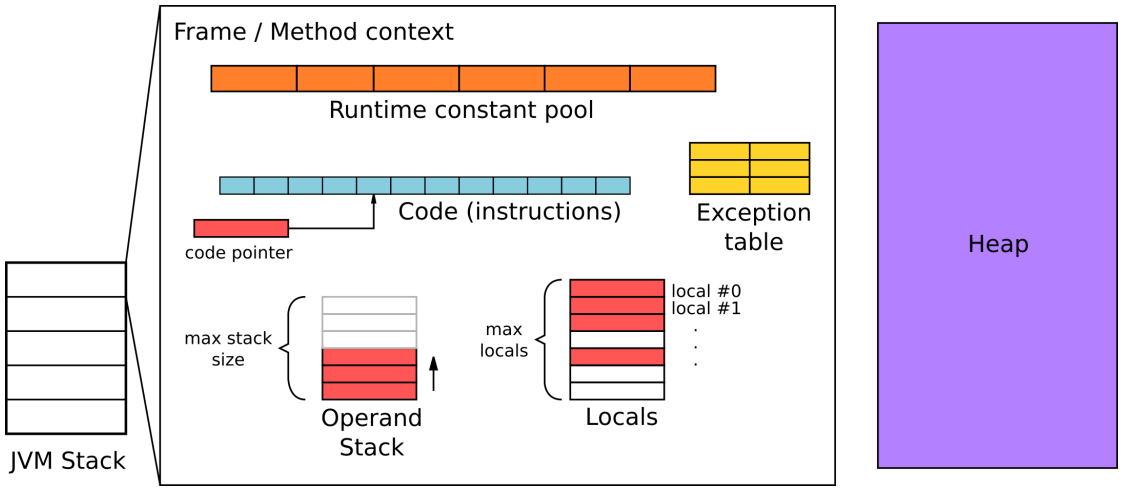
Atribut AnnotationDefault

Atribut AnnotationDefault se používá u metod a to v class souborech reprezentujících anotační type. Reprezentuje defaultní hodnotu vrácenou metodou. (tj. pokud není specifikovaná anotací)

5. Virtální stroj javy a jeho instrukční sada

5.1. Virtuální stroj javy

Virtuální stroj javy (JVM) je fiktivní architektura, jež umí vykonávat bytecode. Jde o tzv. zásobníkovou architekturu (tj. většina instrukcí operuje na zásobníku). Tato fiktivní architektura byla navržena tak, aby bylo možno její kód efektivně vykonávat na nejrůznějších architekturách. A to s využitím interpreteru a/nebo JIT (tj. přeložením bytecode do instrukční sady příslušné architektury za běhu).



Obrázek 5.1: Virtuální stroj javy

JVM stack

Každé vlákno v rámci jvm má svůj JVM stack. JVM stack je složen z rámců (frame). Vždy při volání metody se vytvoří nový rámec a ten se umístí vrchol zásobníku, při návratu z metody se rámec odebere.

Rámec (Frame)

Rámec obsahuje Operand Stack, lokální proměnné, referenci na runtime constant pool a další data vážící se ke kontextu volané metody. Kontext metody je zde myšlen "svět v němž se pohybujeme"v rámci vykonávání jedné metody (z pohledu instrukcí JVM). Kontext metody a rámec jsou úzce spjaty, proto zde nejsou striktně rozlišovány.

Zásobník operandů (operand stack)

Jak bylo řečeno výše, na zásobníku (operand stack) se vykonává většina instrukcí JVM. To probíhá tak, že instrukce odebírají operandy z vrcholu zásobníku a výsledek umísťují opět na vrchol zásobníku. Na zásobník se také umisťují parametry před voláním funkce a návratová hodnota funkce je poté opět umístěna na vrchol zásobníku. Maximální hloubka, jaké může zásobník příslušné metody dosáhnout, je omezena hodnotou v Code Atributu.

Lokální proměnné (local variables)

Lokální proměnné slouží k ukládání mezivýsledků. Hodnoty ze zásobníku lze ukládat do lokálních proměnných pomocí store instrukcí. K umístění hodnoty z lokální proměnné na vrchol zásobníku slouží load instrukce. Skrze lokální proměnné jsou dále předávány parametry volané funkci. Ty jsou po zavolání funkce v prvních n lokálních proměnných.

Runtime constant pool

Runtime constant pool se vytvoří v rámci načítání class souboru pro každou třídu. Je vytvořen podle informací obsažených v constant poolu. A to tak, že v runtime constant poolu jsou na příslušných pozicích umístěny hodnoty příslušných konstant. Pro každou položku CONSTANT_STRING je vytvořena instance objektu java.lang.String a reference na něj umístna na příslušnou pozici v runtime konstant poolu. V rámci linkování jsou dále na příslušné pozice v runtime constant poolu umístěny reference na třídy a metody a to podle symbolických refernencí CONSTANT_Class a CONSTANT_MethodRef. Hodnoty z runtime constant poolu lze umisťovat na zásobník pomocí instrukcí k tomu určených, avšak hodnoty v něm uložené nelze pomocí instrukcí Java bytecodu modifikovat.

Code

Kód představuje instrukce volané metody.

Exception table

Tabulka výjimek se využívá v případě, že byla vyhozena výjimka. Tato tabulka obsahuje seznam výjimek, rozsahy na kterých jsou tyto výjimky zachytávány a pozice v kódu na které začíná obslužný kód pro danou výjimku (handler). Tabulka výjimek se prochází postupně a hledá se první vyhovující položka. Pokud v tabulce neexistuje položka, která by vyhovovala vyhozené výjimce, pokračuje se v hledání v tabulce výjimek volající metody, metody volající tuto metodu atd. Běh poté pokračuje v příslušné metodě na instrukci definované příslušnou položkou v tabulce výjimek. Tabulka výjimek je přečtena z code atributu.

Heap

Heap představuje paměť, v níž jsou alokovány všechny objekty. Instrukce JVM však pracují s heapem pouze nepřímo. Nové objekty se v heapu alokují pomocí instrukce new, která poté umístí referenci na nově alokovaný objekt na vrchol zásobníku. Objekty se však explicitně nedealokují, o toto se stará garbage collector. Heap je sdílen všemi vlákny JVM.

5.2. Typy rozlišované virtuálním strojem javy

JVM rozlišuje datové typy boolean, byte, char, short, int, float, double, reference a returnAddress. Lze vidět, že kromě typů, známých z jazyka Java, jsou zde typy reference a returnAddress. Reference představuje referenci na objekt Javy (instance refernčních typů). ReturnAddress je speciální datový typ používaný instrukcemi jsr a ret (tyto se používaly pro implementaci finally bloků). Přehled typů lze vidět v tabulce 5.1.

Actual type	Computational type	Category
boolean	int	1
byte	int	1
char	int	1
short	int	1
int	int	1
float	float	1
reference	reference	1
returnAddress	returnAddress	1
long	long	2
double	double	2

Tabulka 5.1: Typy JVM

Přestože JVM rozlišuje všechny typy původního jazyka Java, existují zde určité odlišnosti. Některé typy totiž existují pouze ve velmi omezené formě. Konkrétně jde o typy boolean, byte, short a char (tj. typy kratší než 32-bitů). Se všemi těmito typy se totiž pracuje jako s inty (vyjádřeno jako computional type v tabulce). Rozlišují se pouze pole těchto typů. Po umístění hodnot těchto typů na zásobník s nimi však JVM dále pracuje jako s inty. V případě booleanů se hodnota true reprezentuje hodnotou 1, false hodnotou 0. Další věcí stojící za zmínku je, že ne všechny typy zabírají pouze 1 položku na operand stacku nebo jednu lokální proměnnou. Konkrétně je řeč o 64-bitových typech long a double. Hodnoty těchto typů zabírají 2 položky na zásobníku operandů a zabírají také 2 lokální proměnné (Vyjádřeno v tabulce jako category). To proto, aby bylo možno JVM efektivně implementovat na 32-bitových architekturách.

5.3. Instrukce virtuálního stroje javy (java bytecode)

Instrukční sada JVM, označovaná také jako java bytecode, je to soubor instrukcí, které mohou být vykonávané virtuálním strojem javy. Ten je umístěn v atributu Code u metod v class souboru. Java bytecode se kromě jiného liší oproti instrukčním sadám běžných procesorů jednoduchostí dekódování. Toto je důležité proto, aby šlo bytecode efektivně vykonávat interpretery. Instrukce lze od sebe rozlišit jednoduše pomocí jejich prvního bytu. Pro ten se používá označení opcode. V závislosti na tom, o jakou instrukci jde, mohou dále následovat operandy. Instrukce však mohou být tvořeny pouze opcodem. Pro menší objem kódu je v java bytecodu také mnoho instrukcí s implicitními operandy (např. instrukce iconst_0, která vloží kontantu 0 typu int na zásobník, avšak instrukce jako taková operandy nemá). Java bytecode obsahuje jediný prefix a to je instrukce wide (používá se v kombinaci s load a store instrukcemi, pokud je potřeba přistupovat k více než 256 lokálním proměnným). Seznam instrukcí JVM řazených vzestupně podle opcodu lze vidět v tabulce na obr. 5.2 . Pro každou instrukci je zde uveden opcode v desítkové a šestnáckové soustavě a její název. Použité jsou standardizované názvy. Instrukce jsou barevně rozděleny do skupin tak, jak jsou rozděleny ve specifikaci [5]. Kromě instrukcí zde zmíněných jsou rezervovány některé opcody pro interní účely JVM, breakpointy apod. Tyto se však běžně v bytecodu uvnitř class souborů nevyskytují, proto zde nebyly uvedeny.

dec	hex	mnemonic	dec	hex	mnemonic	dec	hex	mnemonic	dec	hex	mnemonic
0	0	nop	51	33	baload	102	66	fsub	153	99	ifeq
1	1	aconst_null	52	34	caload	103	67	dsub	154	9a	ifne
2	2	iconst_m1	53	35	saload	104	68	imul	155	9b	iflt
3	3	iconst_0	54	36	istore	105	69	lmul	156	9c	ifge
4	4	iconst_1	55	37	lstore	106	6a	fmul	157	9d	ifgt
5	5	iconst_2	56	38	fstore	107	6b	dmul	158	9e	ifle
6	6	iconst_3	57	39	dstore	108	6c	idiv	159	9f	if_icmpeq
7	7	iconst_4	58	3a	astore	109	6d	ldiv	160	a0	if_icmpne
8	8	iconst_5	59	3b	istore_0	110	6e	fdiv	161	a1	if_icmplt
9	9	lconst_0	60	3c	istore_1	111	6f	ddiv	162	a2	if_icmpge
10	0a	lconst_1	61	3d	istore_2	112	70	irem	163	a3	if_icmpgt
11	0b	fconst_0	62	3e	istore_3	113	71	lrem	164	a4	if_icmple
12	0c	fconst_1	63	3f	lstore_0	114	72	frem	165	a5	if_acmpeq
13	0d	fconst_2	64	40	lstore_1	115	73	drem	166	a6	if_acmpne
14	0e	dconst_0	65	41	lstore_2	116	74	ineg	167	a7	goto
15	0f	dconst_1	66	42	lstore_3	117	75	lneg	168	a8	jsr
16	10	bipush	67	43	fstore_0	118	76	fneg	169	a9	ret
17	11	sipush	68	44	fstore_1	119	77	dneg	170	aa	tableswitch
18	12	ldc	69	45	fstore_2	120	78	ishl	171	ab	lookupswitch
19	13	ldc_w	70	46	fstore_3	121	79	lshl	172	ac	ireturn
20	14	ldc2_w	71	47	dstore_0	122	7a	ishr	173	ad	lreturn
21	15	iload	72	48	dstore_1	123	7b	lshr	174	ae	freturn
22	16	lload	73	49	dstore_2	124	7c	iushr	175	af	dreturn
23	17	fload	74	4a	dstore_3	125	7d	lushr	176	b0	areturn
24	18	dload	75	4b	astore_0	126	7e	iand	177	b1	return
25	19	aload	76	4c	astore_1	127	7f	land	178	b2	getstatic
26	1a	iload_0	77	4d	astore_2	128	80	ior	179	b3	putstatic
27	1b	iload_1	78	4e	astore_3	129	81	lor	180	b4	getfield
28	1c	iload_2	79	4f	iastore	130	82	ixor	181	b5	putfield
29	1d	iload_3	80	50	lastore	131	83	lxor	182	b6	invokevirtual
30	1e	lload_0	81	51	fastore	132	84	iinc	183	b7	invokespecial
31	1f	lload_1	82	52	dastore	133	85	i2l	184	b8	invokestatic
32	20	lload_2	83	53	aastore	134	86	i2f	185	b9	invokeinterface
33	21	lload_3	84	54	bastore	135	87	i2d	186	ba	invokedynamic
34	22	fload_0	85	55	castore	136	88	l2i	187	bb	new
35	23	fload_1	86	56	sastore	137	89	l2f	188	bc	newarray
36	24	fload_2	87	57	pop	138	8a	l2d	189	bd	anewarray
37	25	fload_3	88	58	pop2	139	8b	f2i	190	be	arraylength
38	26	dload_0	89	59	dup	140	8c	f2l	191	bf	athrow
39	27	dload_1	90	5a	dup_x1	141	8d	f2d	192	c0	checkcast
40	28	dload_2	91	5b	dup_x2	142	8e	d2i	193	c1	instanceof
41	29	dload_3	92	5c	dup2	143	8f	d2l	194	c2	monitorenter
42	2a	aload_0	93	5d	dup2_x1	144	90	d2f	195	c3	monitorexit
43	2b	aload_1	94	5e	dup2_x2	145	91	i2b	196	c4	wide
44	2c	aload_2	95	5f	swap	146	92	i2c	197	c5	multianewarray
45	2d	aload_3	96	60	iadd	147	93	i2s	198	c6	ifnull
46	2e	iaload	97	61	ladd	148	94	lcmp	199	c7	ifnonnull
47	2f	laload	98	62	fadd	149	95	fcmpl	200	c8	goto_w
48	30	faload	99	63	dadd	150	96	fcmpg	201	c9	jsr_w
49	31	daload	100	64	isub	151	97	dcmpl			
50	32	aaload	101	65	lsub	152	98	dcmpg			

Constants	Stores	Math	Comparsions	References
Loads	Stack	Conversions	Control	Extended

Obrázek 5.2: Instrukční sada JVM

Dělení instrukcí JVM podle datového typu

Instrukce však lze dělit také podle typu dat. U instrukcí které se vstahují k nějakému konkrétnímu typu lze tento typ z jejího názvu jednoduše zjistit z počátečního písmena jejich názvu. Tyto jsou b - byte, s - short, i - int, l - long, f - float, d - double, c - char a a - reference. Malý počet instrukcí pro typy byte, short a char je dán tím, že tyto jsou podporovány jen ve velmi omezené formě (jako výpočetní typ používají int), jak bylo řečeno výše. Dále si lze všimnout úplné absence typu boolean. Pro čtení z a ukládání do polí bytů a booleanů se totiž používají stejné instrukce, a to baload a bastore.

opcode	byte	short	int	long	float	double	char	reference
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_TcmpOP			if_icmpOP					if_acmpOP
Treturn			ireturn	lreturn	freturn	dreturn		areturn

Tabulka 5.2: Instrukce JVM podle typu

Konkrétní skupiny instrukcí

Zde je rozebráno fungování jednotlivých skupin instrukcí. Protože detailní rozebírání všech instrukcí by bylo příliš obsáhlé a nemělo by ani význam, snaží se popis zde spíše přiblížit princip fungování jednotlivých skupin instrukcí. (Detailní popis lze nalézt ve specifikaci [8].) Pokud existuje varianta instrukce pro více typů, nejsou tyto explicitně vyjmenovávány, ale je použito velké písmeno T. Existující varianty lze pak dohledat v tabulce výše.

nop - no operation. Neprovede žádnou operaci.

Tconst*, Tipush - umístí konstantu na zásobník. Ta může být buď implicitní (Tconst_*) nebo součástí instrukce jakožto její operand (Tipush).

ldc* - tyto instrukce slouží na umístění konstanty z constant poolu na vrchol zásobníku. Součástí instrukce jakožto její operand je index do constant poolu.

Tload, Tstore - tyto instrukce umožnují umisťovat hodnoty z lokálních proměnných na zásobník operanů (Tload) a naopak (Tstore). Index lokální proměnné je buď implicitní (Tload_*, Tstore_*) a nebo součást instrukce, jako její operand (Tload, Tstore).

Taload, Tastore - slouží k ukládání hodnot do polí (Tastore) nebo jejich čtení. (Taload) Jako operandy se na zásobník umístí reference na pole a index (int) a pokud jde o zápis tak ukládaná hodnota. Při čtení je na zásobník umístěna hodnota z pole.

Tadd, Tsub, Tmul, Tdiv, Trem, Tshl, Tshr, Tushr, Tand, Tor, Txor - toto jsou instrukce, které reprezentují běžné binární aritmetické operace, jako sčítání násobení apod. Instrukce s této skupiny odebírají dva operátory ze zásobníku operandů a výsledek opět umisťují na zásobník. Tyto jsou Tadd (sčítání), Tsub (odčítání), Tmul (násobení), Tdiv (dělení), Trem (zbytek po dělení), Tshl (bitový posun vlevo), Tshr (bitový posun vpravo), Tushr (neznaménkový bitový posun vpravo), Tand (bitové and), Tor (bitové or) a Txor (bitové xor).

iinc - tato instrukce jako jedna z mála operuje na lokální proměnné, místo na zásobníku. Umožňuje přičíst k lokální proměnné s intem hodnotu v rozsahu bytu (-128 až 127).

Tneg - odebere hodnotu ze zásobníku a jako výsledek na zásobník umístí číslo opačné.

T2T - toto jsou konverzní operace. Odebírají hodnotu prvního typu ze zásobníku, zkonvertují ji na hodnotu druhého typu, a tu poté umístí na zásobník.

pop*, dup*, swap - jsou speciální operace na manipulaci se zásobníkem operandů. Manipulace představuje odstranění (pop), duplikování (dup) nebo prohození hodnot na vrcholu zásobníku (swap). Instrukce pop2 a dup2* pracují s dvojicí hodnot. Operace se nevztahují k žádnému konkrétnímu typu a dají se použít na hodnoty libovolných typů. U typů, které představují 2 položky na zásobníku, však není povoleno s nimi pracovat odděleně. (tj. například odebrat 1 vrchní položku zásoníku, pokud je na vrcholu zásobníku hodnota složená ze 2 položek, tedy long nebo double)

Tcmp* - odebere 2 hodnoty ze zásobníku a na zásobník umístí hodnotu -1, 0 nebo 1, v závislosti na tom, jestli hodnota druhého operandu je mešní, stejně velká nebo větší než hodnota operandu prvního.

if* - jde o instrukce podmíněných skoků. Instrukce ifeq, ifne, iflt, ifle, ifgt, ifge porovnávají (==, !=, <, <=, >, >=) hodnotu typu int na vrcholu zásobníku s nulou, a v závilosti výsledku porovnání, provedou/neprovedou skok. Instrukce ifnull a ifnonnull pak porovnávají referenci oproti hodnotě null. Součástí instrukce jako její operand je offset, který udává cíl skoku.

if_*cmp* - tyto fungují podobně jako if*, ale porovnávají 2 hodnoty na vrcholu zásobníku.

goto, goto_w - jsou nepodmíněné skoky. Součástí instrukce jako její operand je offset, který udává cíl skoku.

tableswitch, lookupswitch - jsou instrukce sloužící k implementaci statementu switch. Operandy instrukce jsou hodnoty u jednotlivých case a offsety skoků (pro případy case a default).

return, Treturn - provede návrat z metody. Jako návratová hodnota je použita hodnota na vrcholu zásobníku (Treturn). V případě instrukce return provede pouze návrat (void metoda).

getstatic, putstatic - provádí čtení nebo zápis do statického fieldu. Součástí instrukce (jako její operand) je index do constant poolu na symbolickou referenci na field. Hodnota je odebrána ze zásobníku operandů nebo na něj umístěna.

getfield, putfield - provádí čtení nebo zápis do fieldu, který není statický. Instrukce fungují podobně jako getstatic a putstatic, navíc však odebírají ze zásobníku (kromě případné ukládané hodnoty) také referenci na objekt, jehož field je čten nebo zapisován.

invokevirtual, invokespecial, invokestatic, invokeinterface, invokedynamic - provádějí volání metod. V závislosti na typu metody a na tom, jestli se volá prostřednictvím rozhraní se použije příslušná instrukce. Součástí instrukce je index na symbolickou referenci na volanou funkci v constant poolu. Instrukce invokedynamic přidává JVM podporu pro dynamicky typové jazyky, Java tuto instrukci nevyužívá.

new - vytvoří nový objekt (instanci referenčního typu) a umístí referenci na něj na zásobník. Instrukce obsahuje index do konstant poolu na symbolikou referenci dané třídy.

newarray - vytvoří pole primitivního datového typu (v heapu) a referenci na něj umístí na zásobník. Typ pole je součást instrukce jako operand. Instrukce odebere hodnotu typu int ze zásobníku, která určuje velikost pole.

anewarray - podobně jako newarray, akorát vytvoří pole referenčního typu. Instrukce obsahuje index do konstant poolu na symbolickou referenci na typ prvků pole (jako operand).

multianewarray - podobně jako předchozí, ale vytvoří vícerozměrné pole. (alokuje i pole nižších dimenzí)

instanceof - odebere ze zásobníku referenci a zkontroluje, jestli objekt je instancí třídy (nebo jejího potomka). Součástí instrukce, jako její operand, je index do constant poolu na symbolikou referenci na třídu (rozhranní, pole). Výsledek 0 (false) nebo 1 (true) je poté umístěn na zásobník.

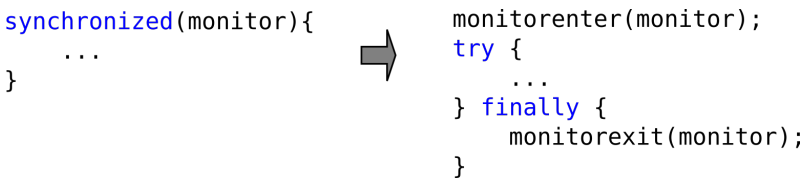
checkcast - užívá se pro implementaci dynamického přetypování. Instrukce provede kontrolu jestli je přetypování možné. (stejným způsobem jako instanceof) Pokud přetypování není možné vyhodí výjimku ClassCastException. Součástí instrukce je opět index do constant poolu na položku, představující příslušný typ, jako v případě instrukce instanceof.

wide - je jediný prefix v instrukční sadě Javy. Instrukce Javy běžně používají byte pro indexaci lokální proměnné. Tímto je pak dán limit 256 používaných lokálních proměnných. Pokud je potřeba indexovat více lokálních proměnných je možné použít instrukce s wide prefixem, které indexují dvěma byty a zvyšují limit na 65536 lokálních proměnných. Wide prefix jde aplikovat na instrukce Tload, Tstore, ret a iinc.

athrow (a výjimky) - instrukce athrow vyhodí výjimku. Výjimka musí být typu Throwable a být umístěna na vrchol zásobníku. Výjimka příslušného typu však může být vyhozena i jinými instrukcemi, například při čtení prvku pole, se záporným indexem nebo indexem větším nebo rovným délce pole. Také při selhání dynamického přetypování. Výjimku mohou dále vyhazovat instrukce které provádí dereferencovaní reference (např. čtení fieldu), aplikované na null apod. Také volání metody může vyhodit výjimku. Po vyhození výjimky je vyprázdněn zásobník a v tabulce výjimek (tabulka výjimek je součástí Code atributu) metody se hledá vyhovující položka (handler) a pokračuje se příslušnou instrukcí. Pokud vyhovující položka není nalezena, je tato výjimka vyhozena příslušnou instrukcí volání metody (ve volající metodě). Proces se poté opakuje.

jsr, jsr_w, ret, ret (a finally bloky) - instrukce jsr a ret se používaly pro implementaci finally bloků v Javě do verze 1.5 (5). Instrukce jsr pak provedla skok na začátek finally bloku přičemž uložila návratovou adresu (onen typ returnAddress) na zásobník. Instrukcí ret se poté šlo vrátit na původní adresu po provedení finally bloku. Toho se využívalo protože finally blok se musí spustit vždy při opuštění try bloku. Instrukce jsr se tedy umisťovala před všechny instrukce vedoucí k opuštění finally bloku (např. return) a také do všech případných exception handlerů. Ne všechny takové handlery nutně musejí existovat v původním zdrojovém kódu. Spuštění finally bloku se musí zajistit např. i v případech, kdy je výjimka vyhozena třeba při čtením z pole nebo fieldu, voláním metody apod. (viz výše) Protože však instrukce jsr a ret značně komplikovali vertifikaci (viz dále), od jejich používání se upustilo. Místo toho nyní java kompiler kopíruje kódu finally bloku na všechna místa původní jsr instrukce.

monitorenter, monitorexit (a synchronizace) - synchronized bloky jsou do bytecodu přeloženy s pomocí těchto instrukcí. Instrukce monitorenter pak představuje vstup do synchronizovaného bloku a monitorexit jeho opuštění. K provedení instrukce monitorexit však musí dojít při opuštění synchronized bloku libovolným způsobem. Pokud by tedy Java podporovala explicitní volaní monitorenter a monitorexit, kód ekvivalentní synchronized blocku by tedy vypadal tak, jak je vidět na obr. 5.3.



Obrázek 5.3: Pseudokód finally bloků

A to se všemi důsledky. Toto samozřejmě představuje komplikaci při jeho dekompilaci. Pokud je však synchronizovaná celá metoda (metoda má access flag ACC_SYNCHRONIZED v class souboru), nepoužívají se již instrukce monitorenter a monitorexit, ale JVM provádí potřebné akce implicitně při volání metody.

5.4. Vertifikace bytecodu

Vertifikace je process, který je standadně prováděn pro každý class soubor načtený JVM. Kromě některých strukturálních kontrol celého souboru je prováděna vertifikace bytecodu metod. To, že je u bytecodu standartě prováděna vertifikace, umožňuje zvýšené požadavky na kvalitu bytecodu a tím i na jeho robustnost a bezpečnost. Potenciálně nebezpečný kód je tak zachycen ještě před jeho spuštěním. To mimo jiné umožňuje JVM např. vyžadovat od kódu typovou bezpečnost, přičemž tuto typovou kotrolu, s vyjímkou dynamického přetypování (instrukce checkcast) lze provést již při vertifikaci. Zachyceny budou také nebezpečné metody u kterých by hrozilo, že provedou skok mimo rozsah metody, že dojde k přetečení nebo podtečení zásobníku s operandy apod. Následuje výčet některých z požadavků, které musí bytecode splňovat. (Jde nejdůležitější body vyjádřené vlastními slovy, úplný seznam s přesným zněním v angličtině lze nalézt ve specifikaci [9].)

- Hloubka operand staku i datové typy jeho položek musí být před vykonáním kterékoli instrukce jednoznačné, bez ohledu na to po jaké cestě se k instrukci přijde.
- Pokud instrukce java bytecodu čte z lokální proměnné, musí být do této proměnné nejdříve přiřazena hodnota. Typ této hodnoty musí být jednoznačný a to bez ohledu na to, po jaké cestě se k instrukci přijde.
- V důsledku instrukce nesmí hloubka zásobníku klesnout pod 0 nebo být vyšší než maximální hodnota uvedená v Code atributu.
- Instrukce smí přistupovat pouze k lokálním proměnným s rozsahem indexů od 0 do hodnoty definované v Code atributu.
- Instrukce nesmí provádět skok, jehož cíl by ležel mimo rozsah metody.
- Nesmí být dosaženo konce metody (pokračováním z poslední instrukce).
- Byty kódu mohou tvořit pouze validní instrukce JVM.
- Cíl každého skoku musí končit na opcodu instrukce (tj. jejím prvním bytu).
- Cíl skoku nemůže být instrukce modifikovaná pomocí wide instrukce, ale může být wide instrukce samotná.

Tímto je vertifikace zajímavá také z hlediska dekompilace, kdy lze vycházet z předpokladu, že tyto požadavky jsou u kódu splněny (protože toto musí platit pro všechny korektní class soubory).

5.5. Příklad vykonávání bytecodu

Pro lepší představu o fungování JVM následuje příklad. Fungování JVM je zde demonstrováno na příkladu velmi jednoduché metody. Zdrojový kód metody lze vidět na obr. 5.4. Jak je vidět ze zdrojového kódu metoda vynásobí první 2 opernandy mezi sebou a přičte k nim třetí. Výsledná hodnota je návratovou hodnotou metody.

```
public static int mulAdd(int x, int y, int z){
    return x * y + z;
}
```

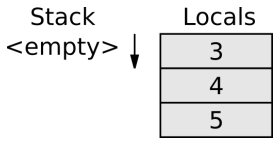
Obrázek 5.4: Instrukční sada JVM

Na obr. 5.5 je poté vidět bytecode metody po zkompilování.

```
public static int mulAdd(int, int, int);
  Code:
     0: iload_0
     1: iload_1
     2: imul
     3: iload_2
     4: iadd
     5: ireturn
```

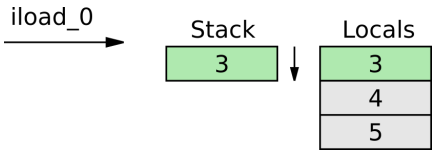
Obrázek 5.5: Instrukční sada JVM

Následuje samotný příklad. V tomto příkladu bude metoda volána s parametry 3, 4 a 5. Protože, jak bylo řečeno, parametry jsou funkci předány prvními n lokálními proměnnými, počáteční stav po zavolání metody bude vypadat tak, jak je vidět na obr. 5.6. Změny nebo hodnoty, se kterými se pracuje budou dále zvýrazňovány zeleně.



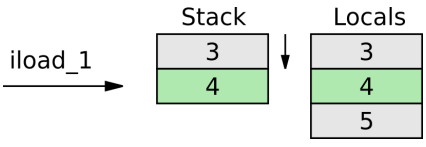
Obrázek 5.6: Parametry předané v lokálních proměnných

Instrukce iload_0 umístí hodnotu typu int z lokální proměnné s indexem 0 na zásobník operandů.



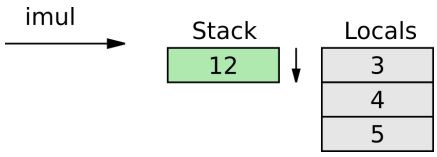
Obrázek 5.7: Hodnota 1. parametru funkce umístěna na zásobník operandů

Instrukce iload_1 umístí hodnotu typu int z lokální proměnné s indexem 1 na zásobník operandů.



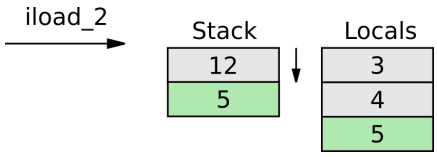
Obrázek 5.8: Hodnota 2. parametru funkce umístěna na zásobník operandů

Instrukce imul odebere ze zásobníku operandů 2 hodnoty typu int a na zásobník poté umístí hodnotu typu int, reprezentující výsledek operace násobení.



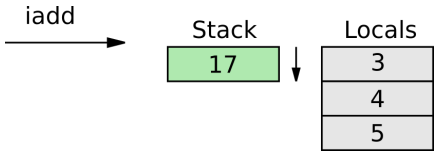
Obrázek 5.9: Vynásobeny hodnoty na vrcholu zásobníku

Instrukce iload_2 umístí hodnotu typu int z lokální proměnné s indexem 2 na zásobník operandů.



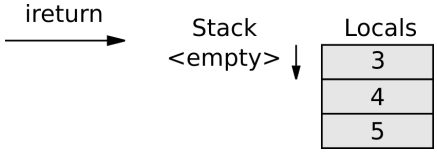
Obrázek 5.10: Hodnota 3. parametru funkce umístěna na zásobník operandů

Instrukce iadd odebere ze zásobníku operandů 2 hodnoty typu int a na zásobník poté umístí hodnotu typu int, reprezentující výsledek operace sčítání.



Obrázek 5.11: Sečteny hodnoty na vrcholu zásobníku

Instrukce iret provede návrat z funkce, přičemž návratová hodnota typu int je odebrána z vrcholu zásobníku.



Obrázek 5.12: Navratová hodnota odebrána ze zásobníku

6. Dekompilace javy a algoritmy používané dekompilerem

Dekompilace je proces inverzní ke kompilaci. Dekompilace se tedy snaží z binárních souborů vzniklých kompilací souborů zpětně rekonstruovat (co nejlépe) podobu původních zdrojových kódů. Tento proces je obtížný, protože překladače (kompilery) se zpětným překladem nepočítají. Ani binární podoba zkompilovaných souborů většinou není navržena s ohledem na možnost zpětného překladu. Mnoho informací o původním zdrojovém kódu je kompilací nenávratně ztraceno a ze zkompilované formy kódu je již nelze zjistit (toto mohou být např. jména proměnných, řídící struktury použité v kódu apod). Binární soubor je také často obecnější a může mít podobu, která do zdrojových kódů není převoditelná (binární soubor může např. vzniknout kompilací z různých jazyků).

Co se týče dekompilace Javy (tedy rekonstrukce kódu jazyka Java z class souborů), ta je až po úroveň metod poměrně přímočará (i když i zde mohou vzniknout problémy). To je dáno tím, že class soubory obsahují všechny potřebné informace nejen pro potřeby liknování, ale také informace potřebné pro kompiler, k tomu, aby zkomilovaný kód šlo využívat jako knihovnu dalším kódem (jak bylo řečeno, java nepoužívá žádné další hlavičkové soubory apod.). Z class souborů lze tak přímo vyčíst informace o jménech tříd i informace o jejich členech (jako jsou fieldy a metody). Podobu anotací a informace o vnořených třídách lze z atributů uvnitř class souborů také rekonstruovat.

Dekompilace kódu metod je však mnohem složitější. Zde již neexistuje jednoznačný přímočarý postup rekonstrukce původního kódu. To je dáno tím, že kód metod je kompilován do podoby java bytecodu. Informace o původní podobě statementů uvnitř metody jsou tedy nenávratně ztraceny a decompiler se může pouze pokusit rekonstruovat jejich přibližnou původní podobu. Problém je dále v tom, že podmíněné skoky používané v bytecode jsou obecnější než řídící struktury používané v původním jazyce Java. (Toto platí také o způsobu zpracování výjimek a synchronizaci) Může tedy existovat bytecode jehož převedení do kódu Javy bude velmi obtížné až nemožné.

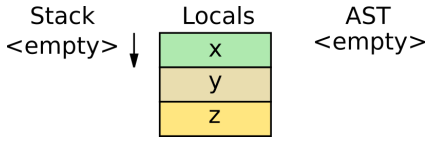
Protože dekompilace je proces obtížný, je dekompilace bytecodu prováděna v mnoha krocích. Pro potřeby dekompileru bylo vytvořeno množství algoritmů, jejichž kombinací je realizován zpětný překlad.

6.1. Algoritmus rekonstrukce statementů

Algoritmus rekonstrukce statementů rekonstruuje statementy z bytecodu metody. Ty jsou rekonstruovány do podoby abstraktních syntaktických stromů (AST). Výsledkem algoritmu je posloupnost statementů s touto reprezentací. Použití algoritmu samostatně je však limitováno na kód neobsahující větvení (skoky). Algoritmus funguje podobně jako interpreter java bytecodu, avšak místo hodnot pracuje s uzly stromu, které postupně spojuje a rekonstruuje tak statementy v podobě AST. Do každé lokální proměnné (JVM), která předává parametr funkce se vytvoří, jedna proměnná pro dekompilovaný kód. Nová proměnná pro dekompilovaný kod se dále generuje s každým zápisem do lokální proměnné bytecodu. Přesněji provede se přiřazení do této nové proměnné a uzel, představující tuto proměnnou se poté vloží do příslušné lokální proměnné. Nový statement je hotov ve chvíli, kdy je ze zásobníku odebrán jeho AST a stal by se tak nedosažitelný. To může být např. v důsledku return instrukce, uložení hodnoty do lokální proměnné nebo fieldu apod.

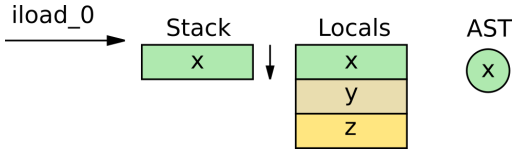
Příklad

Algoritmus je zde předveden na stejném příkladu, na jakém bylo demonstováno fungování virtuálního stroje Javy. Protože funkci jsou předány 3 parametry, jsou vytvořeny 3 uzly představující ony proměnné a ty jsou umístěny do lokálních proměnných.



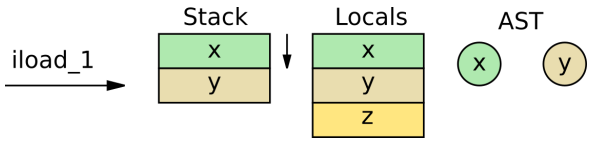
Obrázek 6.1: hodnoty parametrů předané lokálními proměnnými

Uzel z lokální proměnné s indexem 0 je umístěn na zásobník (proměnná x).



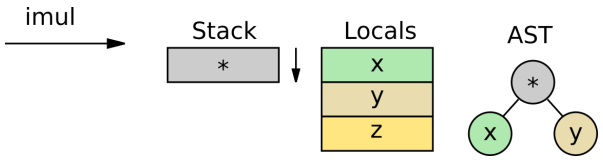
Obrázek 6.2: proměnná x (uzel) vložena na stack

Uzel z lokální proměnné s indexem 1 je umístěn na zásobník (proměnná y).



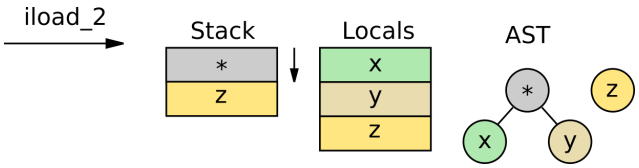
Obrázek 6.3: proměnná y (uzel) vložena na stack

Horní 2 uzly jsou odebrány ze zásobníku, je vytvořen uzel reprezentující operaci násobení a oba odebrané uzly jsou k němu připojeny. Uzel samotný je poté umístěn na zásobník.



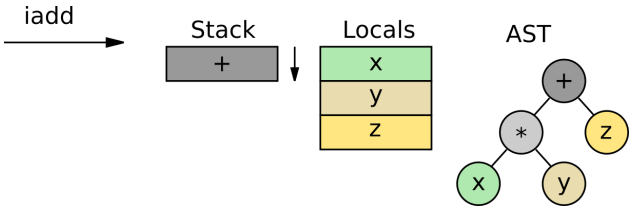
Obrázek 6.4: vytvoření uzlu násobení

Uzel z lokální proměnné s indexem 2 je umístěn na zásobník (proměnná z).



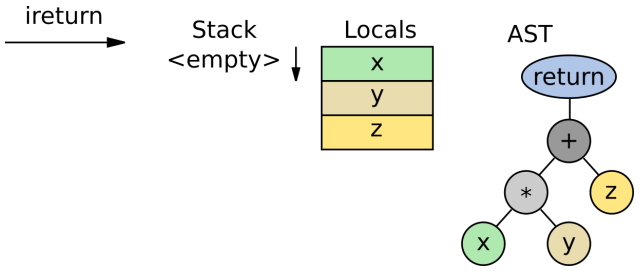
Obrázek 6.5: proměnná z (uzel) vložena na stack

Horní 2 uzly jsou odebrány ze zásobníku, je vytvořen uzel reprezentující operaci sčítání a oba odebrané uzly jsou k němu připojeny. Uzel samotný je poté umístěn na zásobník.



Obrázek 6.6: vytvoření uzlu sčítán

Uzel z vrcholu zásobníku je odebrán a je připojen k uzlu představujícímu return statement.



Obrázek 6.7: statement return hovov

Limitace

Algoritmus v této formě však nezaručuje, že pořadí vyhodnocování výrazů bude shodné s původním kódem (např. vlivem instrukce swap, hodnot "přežívajících"na stacku tvorbu statementů apod.). Pro řešení tohoto by musel být zařazen další krok, který by toto zkontroloval a statementy případně rozdělil. Toto však zatím nebylo implementováno.

6.2. Tvorba control flow grafu

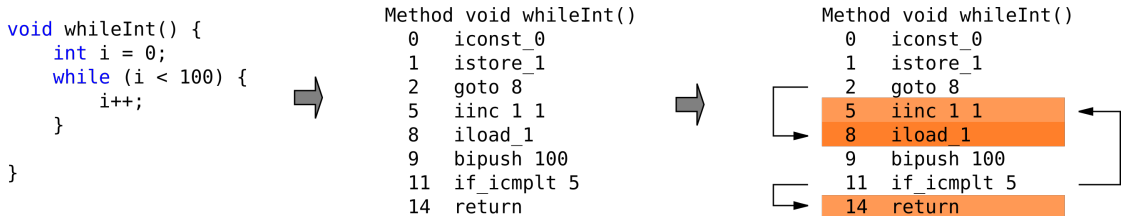
Zatím byl uvažován pouze jednoduchý kód. Reálný kód však často obsahuje řídící struktury (if, while, for, ...), které se poté do bytecodu promítnou jako podmíněné a nepodmíněné skoky, instrukce tableswitch (prozatím neuvažována) a další. Proto je, před započetím rekonstrukce statementů, nutné provézt ještě některé další kroky. Prvním z nich je tvorba CFG.

Control flow graph

Control flow graph (dále CFG) je graf, který popisuje jakými cestami se může ubírat provádění programu. Uzly tohoto grafu představují části kódu, který neobsahuje instrukce skoků (s výjimkou poslední instrukce) a instrukce, které jsou cílem skoku (s výjimkou první instrukce). Hrany reprezentují trasy, po kterých se může provádění programu dále ubírat.

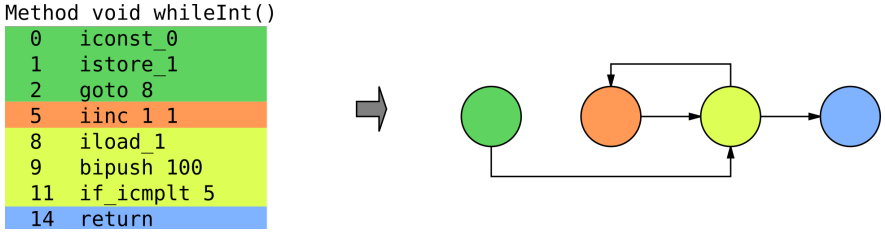
Tvorba CFG

V první fázi projde algoritmus kód metody a nalezne všechny instrukce, které jsou cílem nějakého skoku. (V případě, že jde o podmíněný skok, tak se jako cíl skoku považují obě větve) Cíle skoků jsou v příkladu na obr. 6.8 zobrazeny oranžově.



Obrázek 6.8: Vyhledání instrukcí které jsou cíle skoků

Uzel CFG je vytvořen pro každý blok kódu začínající buď první instrukcí metody nebo instrukcí, která je cílem skoku. S touto instrukcí se asociuje. Uzly končí před dalším cílem skoku, poslední instrukcí metody nebo instrukcí return apod. Poté jsou vytvořeny hrany. Začíná se na uzlu asociovaném s první instrukcí metody, hrany se poté vytváří k uzlům asociovaným s instrukcemi, jež představují cíle skoků z tohoto CFG uzlu. Pokud uzly ještě nebyly navštíveny, algoritmus tvorby hran se na ně rekurzivně aplikuje. Tyto kroky lze vidět na obr. 6.9 . Nalevo jsou zobrazeny bloky kódu, příslušících do jednotlivých uzlů CFG, napravo je CFG vzniklý vytvořením příslušných hran.



Obrázek 6.9: Vytvoření CFG

Pořadí uzlů CFG (dané posloupností v jaké se vyskytují v metodě), však nemusí být optimální pro rekonstrukci řídících struktur. Pokud bychom například obarvily části původního zdrojového kodu v tomto příkladu odpovídající jednotlivým uzlům CFG, zjistili

bychom, že pořadí je odlišné. (Toto přeskupení kódu je v tomto případě určitá optimalizace kompileru, díky které se v rámci jedné iteraci cyklu provádí pouze 1 podmíněný skok, namísto jednoho podmíněného a jednoho nepodmíněného skoku v případě, že by bylo pořadí zachováno)

6.3. Kombinace algoritmů na tvorbu CFG a statementů

Algoritmy na tvorbu CFG a statementů jsou v dekompileru kombinovány. To tak, aby vznikl algoritmus produkující uzly CFG, jejichž obsahem je posloupnost statementů. Nejdříve se tedy provede provede algoritmus na tvorbu CFG a na instrukce příslušící uzlu se poté provede algoritmus pro tvorbu statementů. Toto však přináší komplikace, které je nutné řešit. Těmito komplikacemi jsou hodnoty, které jsou obsažené v lokálních proměnných a na stacku při přechodu mezi instrukcemi, spadajících do rozdílných CFG uzlů. Tyto hodnoty budou dále označovány jako hodnoty předávané lokálními proměnnými a hodnoty předávané stackem. Protože uzly CFG mohou mít obecně více než jednoho předchůdce, můžou být i tyto hodnoty předány z více různých míst.

Hodnoty předávané lokálními proměnnými

Pro algoritmus pro tvorbu statementů to znamená, že jedné lokální proměnné mohou být od různých předchůdců předány uzly představující různé proměnné. Při řešení tohoto problému hraje důležitou roli požadavek na bytecode z hlediska vertifikace. Ten říká, že pokud je instrukcí čteno z lokální proměnné, musí být do ní nejdříve zapsáno a typ musí být jednoznačný. To bez ohledu na to po jaké cestě se k instrukci došlo. Je však třeba si uvědomit, že to také znamená, že pokud již z lokální proměnné není dále čteno, typ jednoznačný být nemusí. Nelze proto jednoduše pro každou lokální proměnnou jednoduše sloučit všechny proměnné (nyní myšleny ty, vytvořené algoritmem pro tvorbu statementů) ze kterých tato proměnná mohla vzniknout do jediné a tu poté použít.

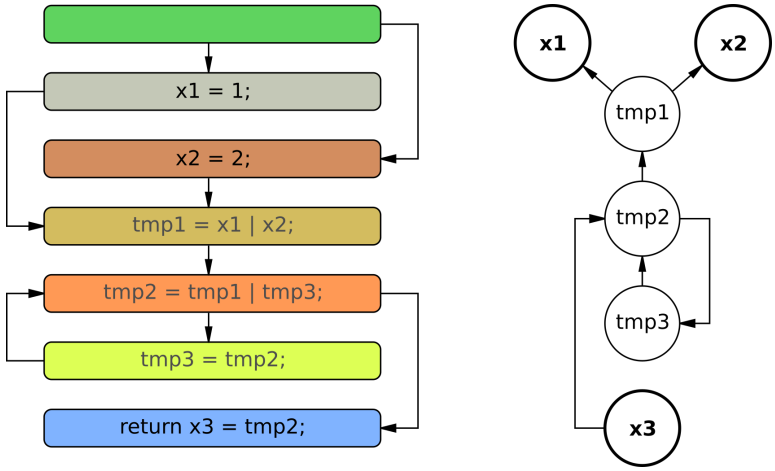
Algoritmus slučování lokálních proměnných Algoritmus tvorby statementů problém řeší tak, že v lokálních proměnných vytvoří nové proměnné. Pro každou takovou proměnnou jsou postupně určeni všichni předci ze kterých proměnná mohla vzniknout. Protože u žádné takové nové proměnné není zatím známo jestli z ní bude čteno, bude brána jako neaktivovaná. Proměnné vzniklé přímým přiřazením budou brány jako aktivované. Poté, co je tímto způsobem zpracován všechen kód, je takto vytvořený graf závislostí zpětně projit směrem od proměnných, o kterých je známo, že z nich je čteno. Při průchodu dochází k postupné aktivaci procházených proměnných. Skupiny aktivovaných proměnných, které společně tvoří souvislý graf, jsou poté sloučeny. Proměnné které zůstanou neaktivované se "zahodí". Následuje ukázka algoritmu na příkladu. Nalevo na obr. 6.10 je vidět zdrojový kód ukázkového příkladu. Zde je důležitá proměnná x. Zápis do ní může být proveden na dvou různých místech, na konci metody je z ní čteno. Napravo pak lze vidět bytecode zkompilované metody s nalezenými uzly CFG zvýrazněnými barevně a instrukcemi provádějící zápis a čtení do proměnné x zvýrazněnými červeně.

```
public int variableMeger(boolean b) {
    int x;
    if (b) {
        x = 1;
    } else {
        x = 2;
    }
    int i = 0;
    while (i < 10) {
        ++i;
    }
    return x;
}
```

```
public int variableMeger(boolean);
    0: iload_1
    1: ifeq          9
    4: iconst_1
    5: istore_2
    6: goto          11
    9: iconst_2
   10: istore_2
   11: iconst_0
   12: istore_3
   13: iload_3
   14: bipush        10
   16: if_icmpge     25
   19: iinc          3, 1
   22: goto          13
   25: iload_2
   26: ireturn
```

Obrázek 6.10: Zdrojový a zkompilovaný kód příkladu

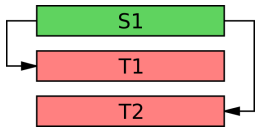
Na 6.11 obrázku vlevo už lze vidět CFG, v jehož uzlech jsou vidět proměnné tak, jak jsou postupně vytvořeny algoritmem pro tvorbu statementů (tyto proměnné však nemusí být reálně použity). Neaktivované proměnné používají šedou barvu a jejich název začíná tmp. U proměnných je dále uvedeno ze kterých proměnných mohly vzniknout. Operátor | se zde vyjadřuje "nebo". Napravo je poté zobrazen graf závislostí lokálních proměnných. Je zřejmé, že postupem z proměnné x3 (ze které je čteno) po orientovaných hranách a aktivací proměnných budou nakonec aktivovány všechny proměnné v grafu. Ty jsou následně sloučeny do jediné proměnné.



Obrázek 6.11: Závislosti proměnných

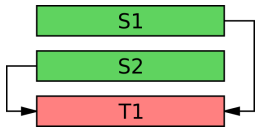
Hodnoty předávané operand stackem

I když předávání hodnot operand stackem používá kompiler javy spíše vyjímečně, je nutné ho řešit. Situace zde je navíc komplikovaná tím, že v kódu javy neexistuje alternativa pro předávání hodnot stackem. Zde je opět důležitý požadavek na kód z hlediska vertifikace. Ten říká, že hloubka operand stacku i typ jeho hodnot musí být před vykonáním každé instrukce jednoznačné, bez ohledu na to, po jaké cestě se k instrukci přijde. Základní myšlenka pro řešení problému přenášení hodnot operand stackem je ta, že algoritmus pro tvorbu statementů vytvoří pomocné proměnné, které budou předávání realizovat. Algoritmus tvorby statementů tedy vytvoří statementy přiřazující předávané výrazy (tj. uzly AST, které se nacházejí na operand stacku, při dosažení konce CFG) těmto pomocným proměnným. U CFG uzlů následníku se poté uzly představující tyto proměnné umístí na zásobník a poté se provede rekonstrukce statementů. Ovšem vytvořit tyto proměnné není tak jednoduché, jak by se mohlo zdát. Problém bude rozveden na příkladech. Uvažujeme CFG na obr. 6.12, kde uzel S1 má dva následníky T1, T2 a předává hodnoty operand stackem. V tomto případě je zřejmé, že uzel S1 provede zápis do pomocných proměnných, které budou použity k předání hodnoty uzlům T2 nebo T3. Aby vše fungovalo použije se samozřejmě stejná sada proměnných v obou případech.



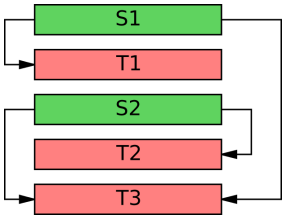
Obrázek 6.12: Předávání hodnot operand stackem dvěma následníkům

Další je příklad, kdy CFG uzel T1 má 2 předky S1 a S2, které předávají hodnoty operand stackem. (Situace je na obr. 6.13.) Zde provedou uzly S1 a S2 zápis do pomocných proměnných, které budou použity k předání hodnot uzlu T1. Uzly S1 a S2 samozřejmě použijí stejnou sadu pomocných proměnných.



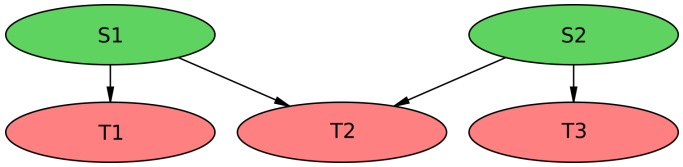
Obrázek 6.13: Předávání hodnot operand stackem společnému následníkovi

Poslední příklad je kombinace obou případů. Uvažovaná situace je na obr. 6.14 . Podle prvního příkladu tedy platí, že uzel S1 použije stejnou sadu proměnných pro předání hodnot uzlům T1 a T3 a že uzel S2 použije stejnou sadu proměnných pro předání hodnot uzlům T2 a T3. Dále podle druhého příkladu platí, že uzly S1 a S2 používají stejnou sadu proměnných pro předávání hodnot uzlu T3. Z tohoto plyne, že v tomto příkladu, musí být užita stejná sada pomocných proměnných pro realizaci všech předávání hodnot operand stackem.



Obrázek 6.14: Kombinovaný případ

Alternativní vyobrazení příkladu je vidět na obr. 6.15.



Obrázek 6.15: Kombinovaný případ

Algoritmus vytváření pomocných proměnných V použitím algoritmu je na situaci nahlíženo z pohledu cílů (v příkladech růžové). Cíle jsou řazeny do skupin, které vyjadřují sdílení sady pomocných proměnných. Nejdříve se vytvoří skupiny s tím, že pokud dva cíle sdílí předka (v příkladech zelené) patří do stejné skupiny (tímto se aplikuje pravidlo z prvního příkladu). Až jsou takto vytvořeny všechny skupiny, projdou se cíle znovu. Pokud je nalezen cíl patřící do více než jedné skupiny, všechny skupiny do kterých patří jsou sloučeny do jediné (aplikace pravidla z druhého příkladu). Poté platí, že pro předávání hodnot všem cílům patřícím do jedné skupiny se používá jedna (sdílená) sada pomocných proměnných.

6.4. Seřazení uzlů CFG

Aby se zajistilo vhodné pořadí uzlů CFG, pro následné algoritmy, rekonstruující řídící struktury je nejdříve zařazen algoritmus pro seřazení uzlů.

Topologické řazení

Jedním z cílů řazení uzlů CFG je, aby se vyeliminovali všechny "zbytečné"zpětné skoky, které by bytecode mohl obsahovat. K tomuto lze využít tzv. topologické řazení, které zajistí, že každý uzel bude v seřazené posloupnosti uzlů předcházet všechny své následníky. Topologické řazení má však jedno velké omezení. Graf jím řazený nemůže obsahovat cykly. Přesněji řečeno nesmí obsahovat tzv. silně souvislé komponenty.

Silně souvislé komponenty (strongly connected components)

Silně souvislé komponenty (dále označované jako SCC) jsou takové části grfu (podgrafy), u kterých platí, že z každého jeho uzlu se lze dostat (po orientovaných hranách) na každý další. V algoritmu pro hledání SCC se uplatňuje prohledávání do hloubky, přičemž první navštívený uzel SCC se nazývá kořen (root) SCC.

Algoritmus vytvořený pro decompiler

Alogoritmus řazení CFG uzlů využitý v dekompileru, využívá oba zmíněné algoritmy (tedy topologické řazení a vyhledávání SCC) a funguje následujícím způsobem. Na CFG se použije algoritmus pro nalezení SCC. Každý SCC je odebrán a nahrazen jediným uzlem, přičemž všechny hrany končící na uzlech, jež jsou součástí tohoto SCC, jsou odebrány a nahrazeny hranami, končícími v tomto novém uzlu. Všechny hrany vycházející z uzlů jež jsou součástí SCC jsou odebrány a nahrazeny hranami vycházejícími z nového uzlu. Dále jsou odebrány všechny hrany vedoucí do kořene daného SCC (tímto není porušena souvislost podgrafu, který býval SCC). Na výsledný graf (s novým uzlem) je aplikován algoritmus topologického řazení (pokud jde o graf vzniklý z SCC začíná se na kořenovém uzlu). Algoritmus je poté rekurzivně aplikován na každý graf vzniklý z SCC (odebráním příslušných hran, jak bylo popsáno výše). Názorně je algoritmus demonstrován na následujících příkladech.

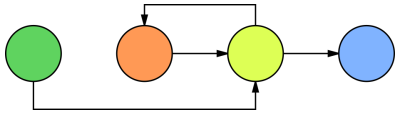
př. 1 cyklus while

Jako první příklad bude použita opět metoda s while cyklem. Na obr lze vidět zkompilovaný kód a z něho vytvořený CFG s uzly v pořadí v jakém se vyskytují ve zkompilovaném kódu.

```
void whileInt() {
    int i = 0;
    while (i < 100) {
        i++;
    }
    return;
}
```

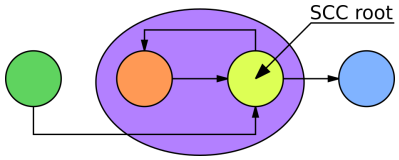
```
Method void whileInt()
  0   iconst_0
  1   istore_1
  2   goto 8
  5   iinc 1 1
  8   iload_1
  9   bipush 100
  11  if_icmplt 5
  14  return
```

Obrázek 6.16: Metoda s while cyklem a její zkompilovaná podoba



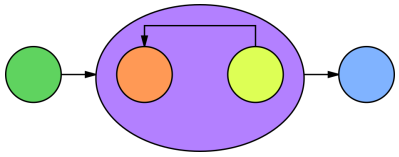
Obrázek 6.17: Původní CFG s neseřazenými uzly

Nejdříve jsou nalezeny SCC (v tomto případě pouze jeden). Kořen (root) je označen šipkou.



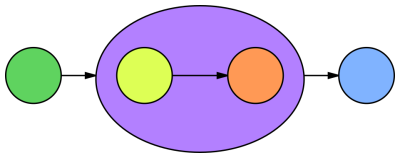
Obrázek 6.18: CFG s nalezeným SCC

SCC je uvažován jako jediný uzel (fialový), hrany vedoucí do něj a z něj jsou přemístěny na nový uzel. Dále jsou odstraněny všechny hrany končící na kořenu SCC. Graf je topologicky seřazen. (v tomto případě zůstává pořadí nezměněno)



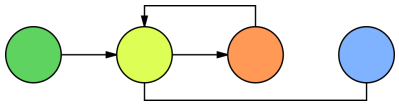
Obrázek 6.19: CFG s ostraněnými hranami

Protože SCC z odstraněnými příslušnými hranami neobsahuje další SCC, je pouze topologicky seřazen. (začíná se kořenem)



Obrázek 6.20: Seřazení grafu vzniklého z SCC

Zde už je CFG se seřazenými uzly. Lze vidět, že pořadí CFG uzlů nyní odpovídá pořadí konstruktů v povodním zdrojovém kódu.



Obrázek 6.21: Seřazený CFG

př. 2 vnořené while cykly

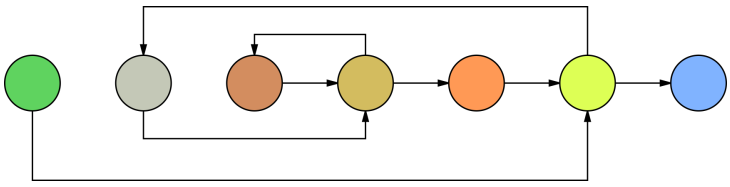
Zdrojový a zkompilovaný kód tohoto příkladu lze vidět na obr. 6.22.

```
void nestedLoops() {
    int i = 0;
    while (i < 100) {
        int j = 0;
        while (j < 100) {
            j++;
        }
        i++;
    }
    return;
}
```

```
public void nestedLoops();
    0: iconst_0
    1: istore_1
    2: goto          28
    5: iconst_0
    6: istore_2
    7: goto          16
   10: iinc_w        2, 1
   16: iload_2
   17: bipush        100
   19: if_icmplt     10
   22: iinc_w        1, 1
   28: iload_1
   29: bipush        100
   31: if_icmplt     5
   34: return
```

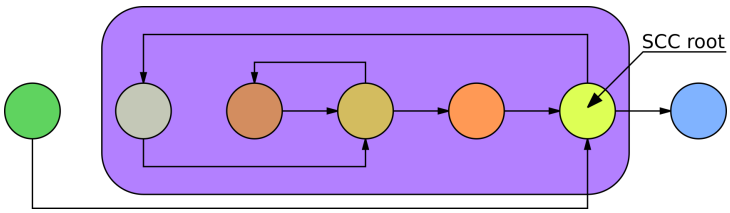
Obrázek 6.22: Metoda se dvěma while cykly a její zkompilovaná podoba

Následuje CFG s uzly v pořadí, v jakém se vyskytují ve zkompilovaném kódu.



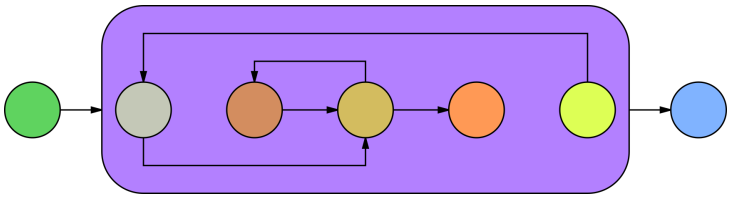
Obrázek 6.23: Původní CFG s neseřazenými uzly

Nejdříve jsou nalezeny SCC. Kořen (root) je označen šipkou.



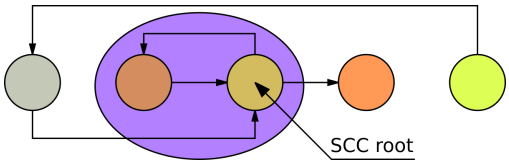
Obrázek 6.24: CFG s nalezeným SCC

SCC je uvažován jako jediný uzel (fialový), hrany vedoucí do něj a z něj jsou přemístěny na nový uzel. Dále jsou odstraněny všechny uzly končící v kořenu SCC. Graf je topologicky seřazen. (pořadí zůstává nezměněno)



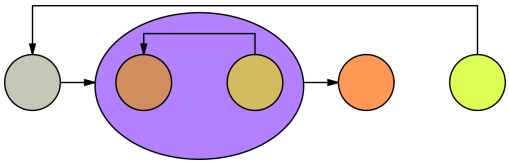
Obrázek 6.25: Graf vzniklý po odstranění příslušných hran a topologickém seřazení

Algoritmus se rekurzivně aplikuje na graf vzniklý z SCC (odstraněním příslušných hran), jsou zde tedy opět hledány SCC.



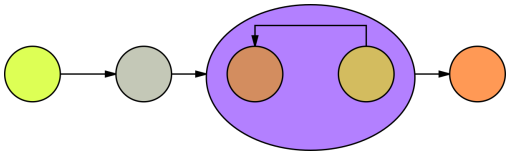
Obrázek 6.26: SCC nalezený rekurzivní aplikací algoritmu

Odebrány příslušné hrany.



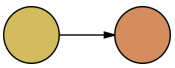
Obrázek 6.27: Graf vzniklý z SCC odebráním příslušných hran

Graf je topologicky seřazen. (Začíná se v uzlu který byl kořenem SCC)



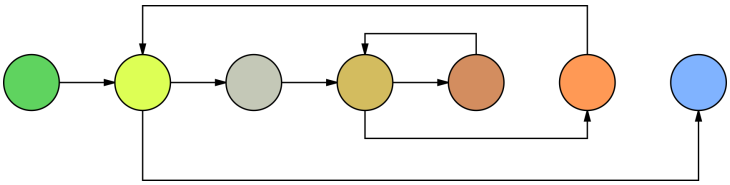
Obrázek 6.28: Topologicky seřazení podgrafu

Algoritmus se podruhé rekurzivně aplikuje na graf vzniklý z SCC (odstraněním příslušných hran), nyní už tedy dojde pouze k topologickému seřazení.



Obrázek 6.29: Druhá rekurzivní aplikace algoritmu na zbývající podgraf

Zde už CFG po seřazení. Stejně jako v prvním příkladu je vidět, že pořadí CFG uzlů nyní odpovídá pořadí konstruktů v povodním zdrojovém kódu.



Obrázek 6.30: CFG se seřazenými uzly

6.5. Převod seřazených uzlů CFG do strukturovaného kódu

Ani seřazené uzly CFG, s vytvořenými statementy, však stále není možno ve zdrojovém kódu javy přímo reprezentovat. Kód je nyní nutno strukturovat. (Java neumožňuje použití goto) Tento krok je jeden z nejkritičtějších. Je zřejmé, že ne každý kód lze rozumně strukturovat. Právě proto by měl být použitý algoritmus co nejuniverzálnější. Převod by měl být proveden do co nejobecnějších řídících struktur, přičemž zároveň by bylo vhodné, aby počet druhů těchto řídících struktur, byl co nejnižší. To aby si algoritmus zachoval dostatečnou jednoduchost a univerzálnost. V Javě běžně používané řídící struktury jako if, if-else, for, while, do-while apod. tedy nepřicházeli v úvahu, protože jsou příliš konkrétní a je jich příliš mnoho druhů.

Obecný scope

Cílem snahy vytvoření vhodné řídící struktury bylo zavedení něčeho, co bud dále označováno jako "obecný scope". Obecný skope si lze představit jako pojmenovaný scope, na který lze navíc aplikovat continue. Avšak obecný scope není cyklem (po dosažení jeho konce dojde k jeho opuštění). Continue aplikované na obecný skope provede skok na začátek, break na konec. Break a continue aplikované na obecný scope navíc mohou být podmíněné.

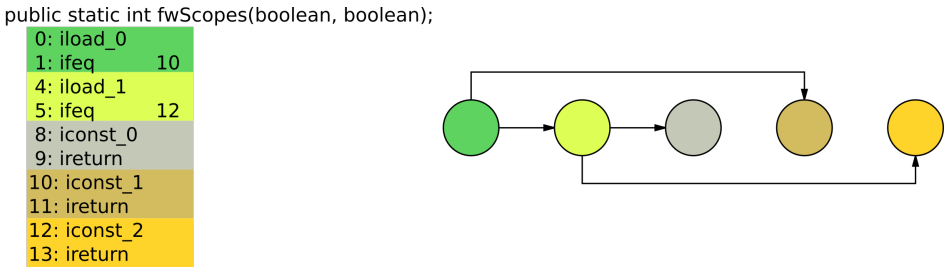
Převod obecných scopů do jazyka java

Pokud obecný scope neobsahuje statementy continue, lze jej reprezentovat jednoduše pomocí pojmenovaného scopu. V obecné formě je převoditelný do Javy jako pojmenovaný nekonečný cyklus, který je implicitně zakončen breakem. Protože takovéto značení by bylo značně nepřehledné, budou u příkladů demonstrujících fungování algoritmů vytvářejících obecné scopy, tyto zobrazeny jako pojmenovaný scope. Pro podmíněné příkazy break a continue bude využito statementu if.

Tvorba obecných scopů pro dopředné skoky

Zde rozebírán případ, že kód obsahuje pouze dopředné skoky. Algoritmus vytvoří scope metody, a poté pracuje postupně "od zhora dolů"a do metody postupně přidává statementy. Skoky nahradí kombinací obecného scopu a statementu break. Myšlenka je ta, že je vytvořen obecný scope, který obsahuje "místo ze kterého je proveden skok"a sahá až před "místo doskoku". Na ten je poté aplikován break. Pokud je proveden skok z místa, které je vnořeno v jednom nebo více scopech, které ale nedosahují k místu doskoku, jsou i všechny tyto scopy zahrnuty do nového scopu (tak, aby se předešlo "křížení"scopů). Pokud už vhodný scope existuje nový se nevytváří a break se aplikuje na tento scope. Následuje demonstace na příkladu.

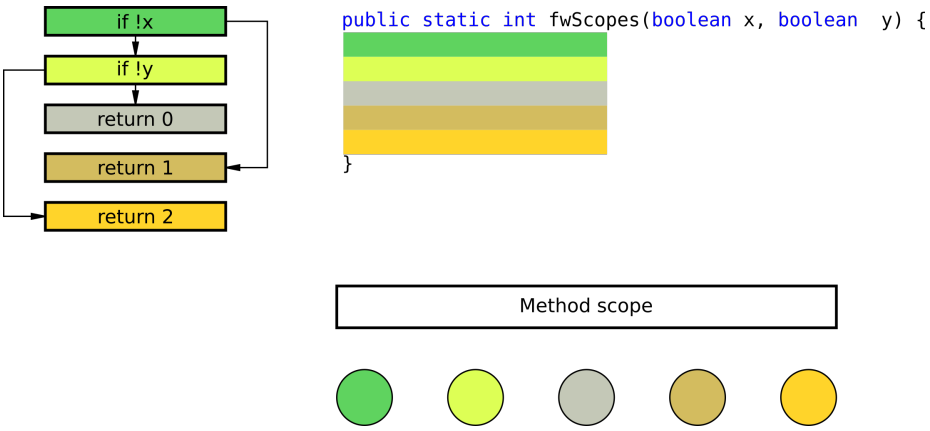
Demonstrace algoritmu na příkladu Na obr. 6.31 lze vidět původní zkompilovaný kód a vytvořený CFG, na kterém bude fungování algoritmu demonstrováno.



Obrázek 6.31: Kód příkladu a vytvořený CFG

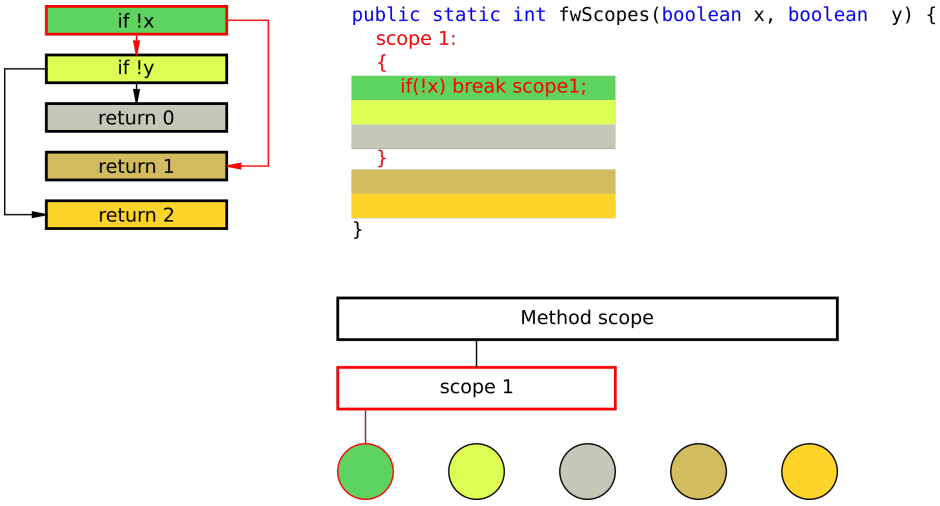
Následuje demonstrace samotného algoritmu. Na levé straně jsou seřazené prvky CFG (od shora dolů). Ty obsahují statementy (a podmínky skoků) metody. V pravé části je poté vyobrazeno postupné umisťování kódu do scopů. (nahoře tak jak by vypadal dekompilovaný kód, dole ve stromové reprezentaci)

Nejdříve je vytvořen scope metody.



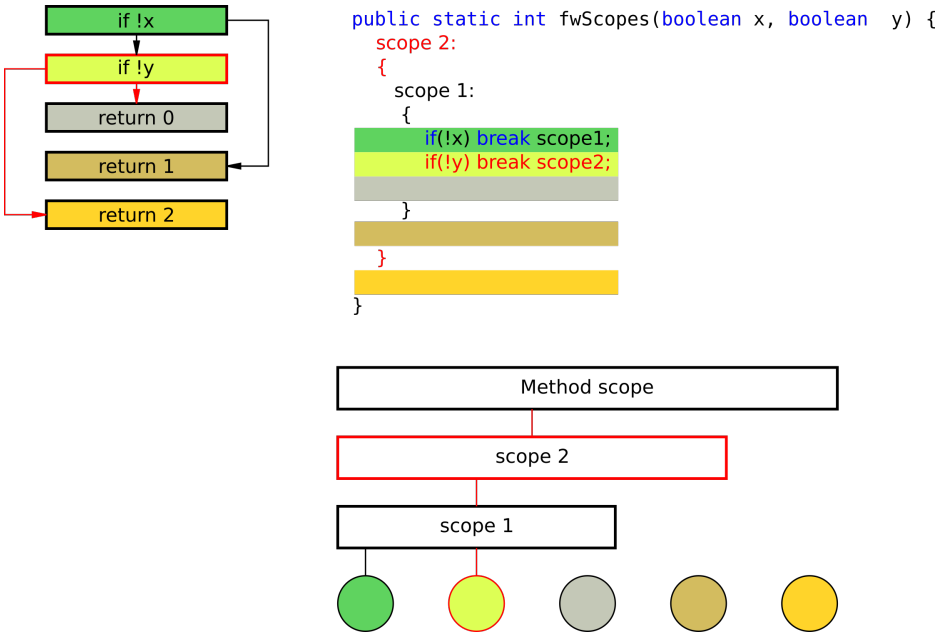
Obrázek 6.32: Vytvořen scope metody

Místo skoku je zabeleno do scopu tak, že scope sahá až před místo doskoku. Na scope je aplikován podmíněný break s příslušnou podmínkou.



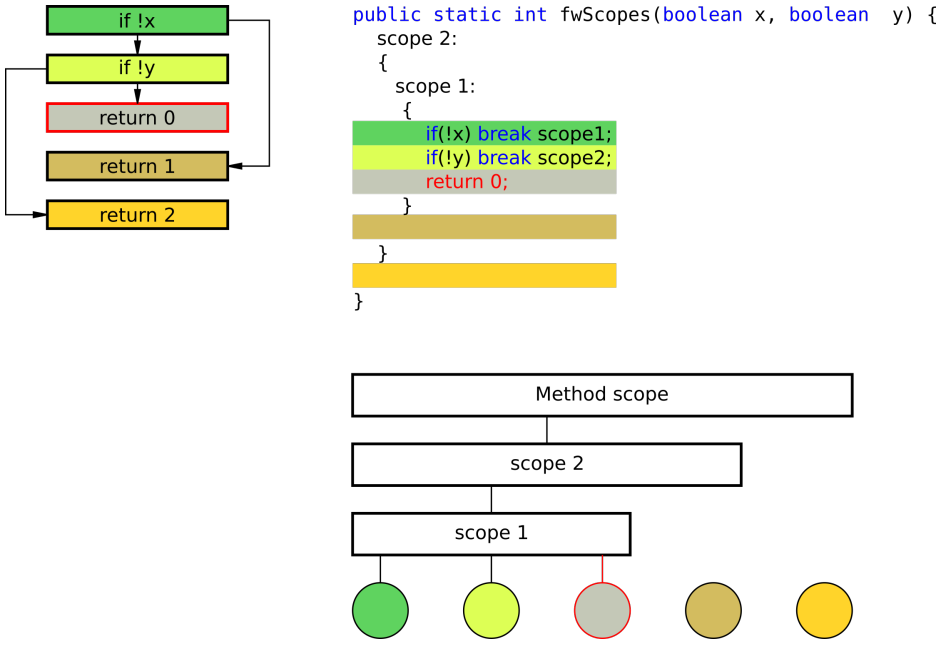
Obrázek 6.33: Zpracován skok

Místo skoku již ve scopu je, avšak tento scope nedosahuje až před místo doskoku. Je proto vytvořen nový scope, který celý tento scope obsahuje. Na skope je aplikován podmíněný break s příslušnou podmínkou.



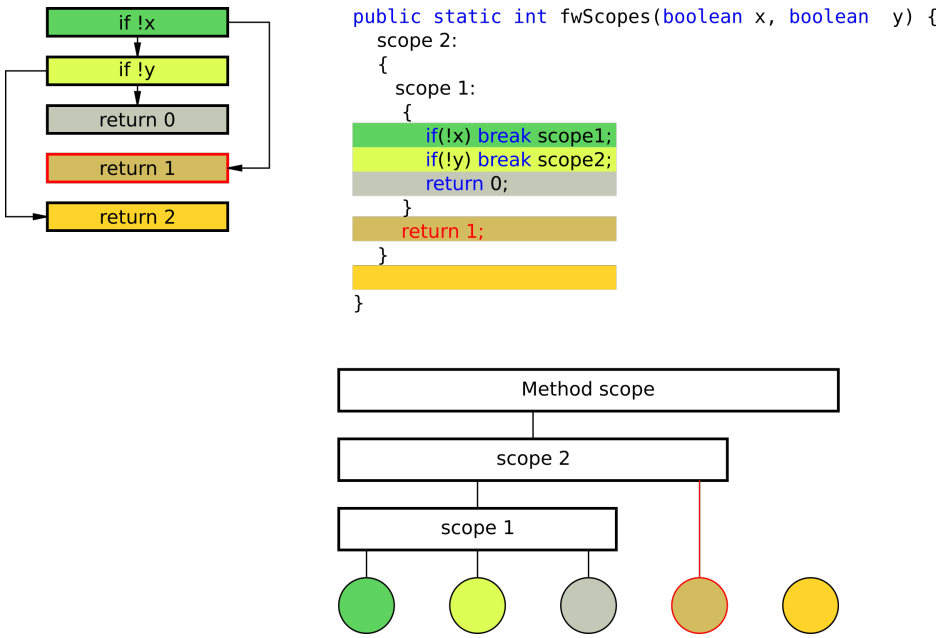
Obrázek 6.34: Zpracován skok

První statement return je vložen na příslušnou pozici.



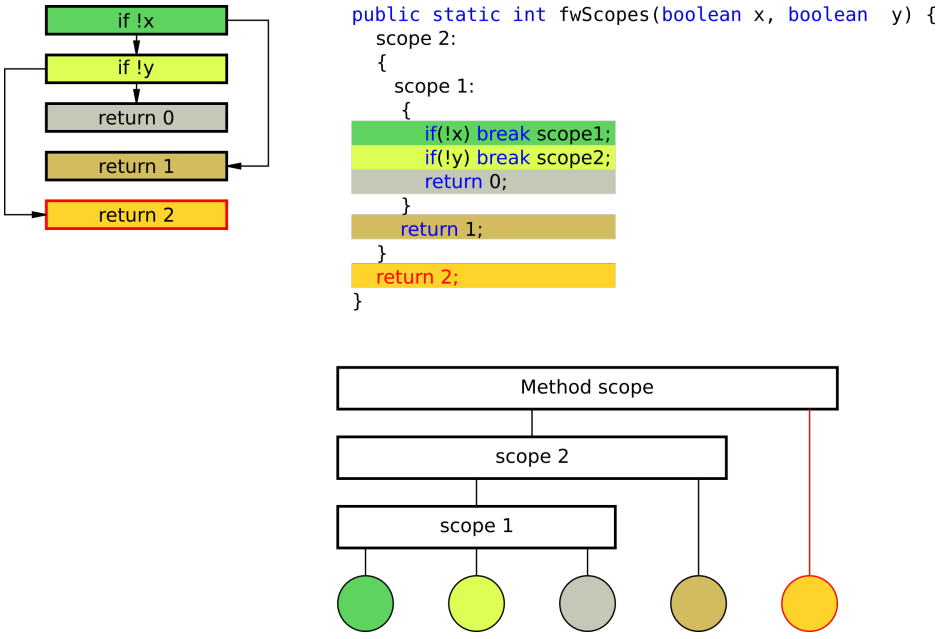
Obrázek 6.35: Vložení prvního statementu return

Druhý statement return je vložen na příslušnou pozici.



Obrázek 6.36: Vložení druhého statementu return

Třetí statement return je vložen na příslušnou pozici.

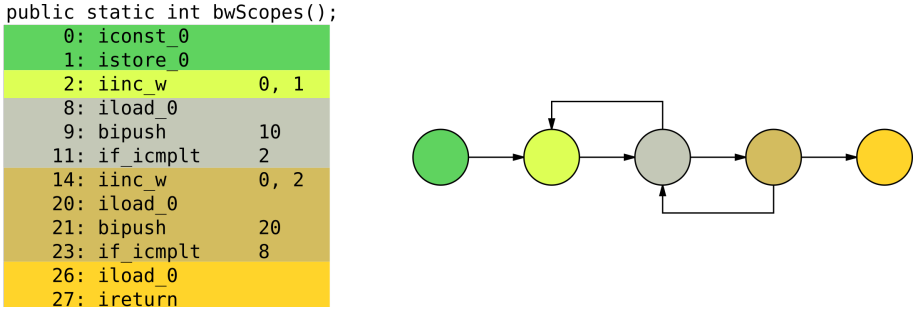


Obrázek 6.37: Vložení třetího statementu return

Tvorba obecných scopů pro zpětné skoky

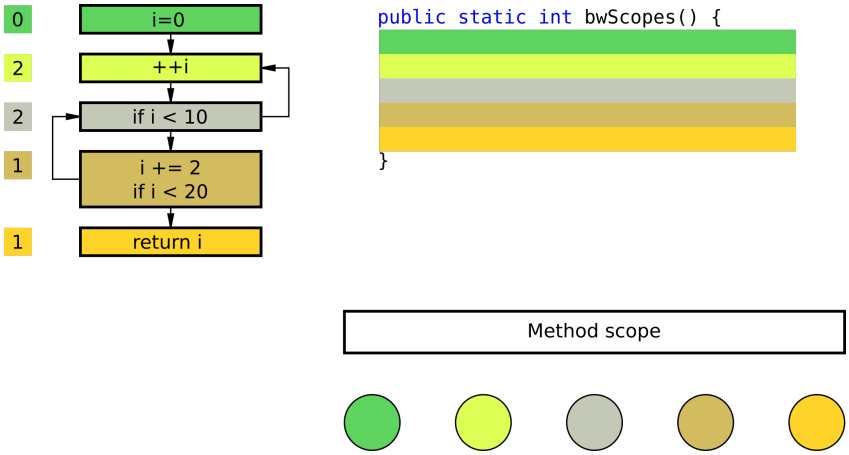
Na zpětné skoky by šel aplikovat algoritmus podobný, jako u dopředných skoků, pouze by se postupovalo od konce a místo breaku by se používal continue. Avšak aby bylo možné algoritmy později sloučit i zde se bude postupovat dopředným směrem. Algoritmus tedy nejdřív vytvoří scope metody a poté zpracovává uzly CFG v tom pořadí, do jakého byly seřazeny algoritmem výše. Umisťuje přitom statementy v nich obsažené do aktuálního scopu. Problém však je, že při postupu dopředným směrem není jasné, kdy by se měly otevírat nové scopy, ani kdy by se měly uzavírat. Toto je řešeno tím, že každý uzel CFG opatřen čítačem, jehož počáteční hodnota odpovídá počtu hran, po kterých se na tento uzel CFG dá dostat. Pokud je při vstupu na uzel CFG hodnota jeho čítače nenulová, znamená to, že je cílem zpětného skoku (viz dále) a je otevřen nový obecný skope. Vždy při opouštění uzlu CFG jsou čítače všech uzlů CFG, které jsou jeho přímými následníky (po hranách CFG) sníženy o 1. Zároveň se také vytvoří statement continue, pokud je z uzlu proveden zpětný skok. Pokud některý čítač dosáhne nuly, je scope začínající na tomto cíli (uzlu CFG) uzavřen (jestliže takový scope existuje). Pokud není možné scope uzavřít z důvodu, že již byl otevřen jeden nebo více vnořených skopů, pozdrží se uzavření do chvíle, kdy jsou všechny vnořené skopy uzavřeny.

Demonstrace algoritmu na příkladu Algoritmus je opět demonstrován na příkladu. Na obr. 6.38 lze vidět původní zkompilovaný kód a k němu vytvořený CFG. Kvůli přehlednosti budou obecné scopy (viz výše) generované algoritmem zobrazeny jak obyčejné pojmenované scopy. Následuje algoritmus samotný.



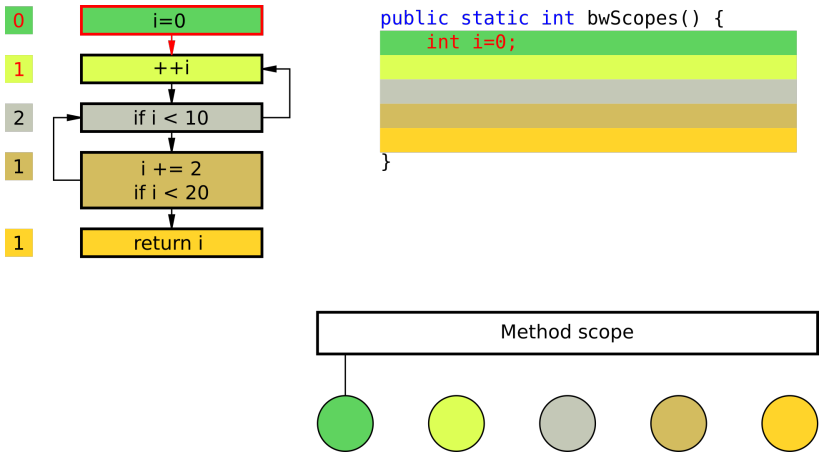
Obrázek 6.38: Kód příkladu a vytvořený CFG

Algoritmus začíná vytvořením scopu metody.



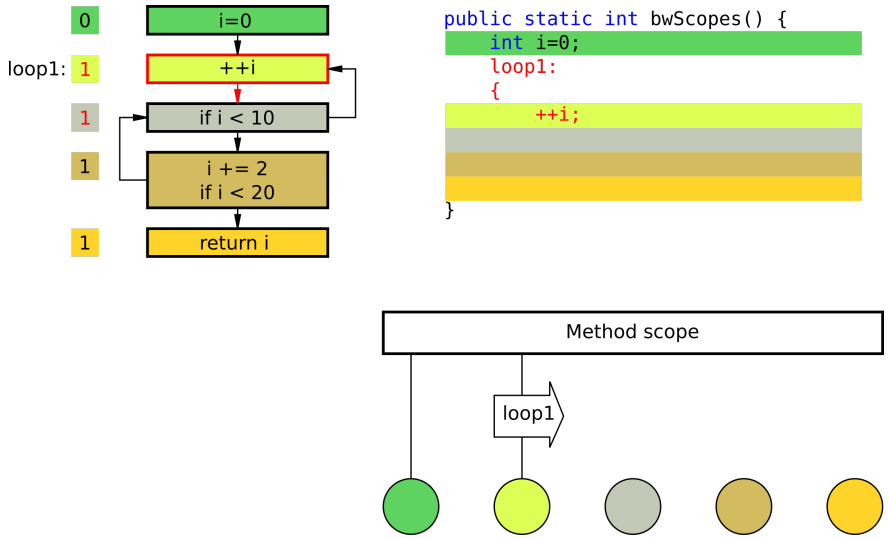
Obrázek 6.39: Vytvořen scope metody

Do aktuálního scopu se umístí všechny statementy z prvního uzlu CFG (jeho čítač má hodnotu 0, nový scope se tedy nevytváří). Hodnota čítače následníka (po šipce) je snížena o 1.



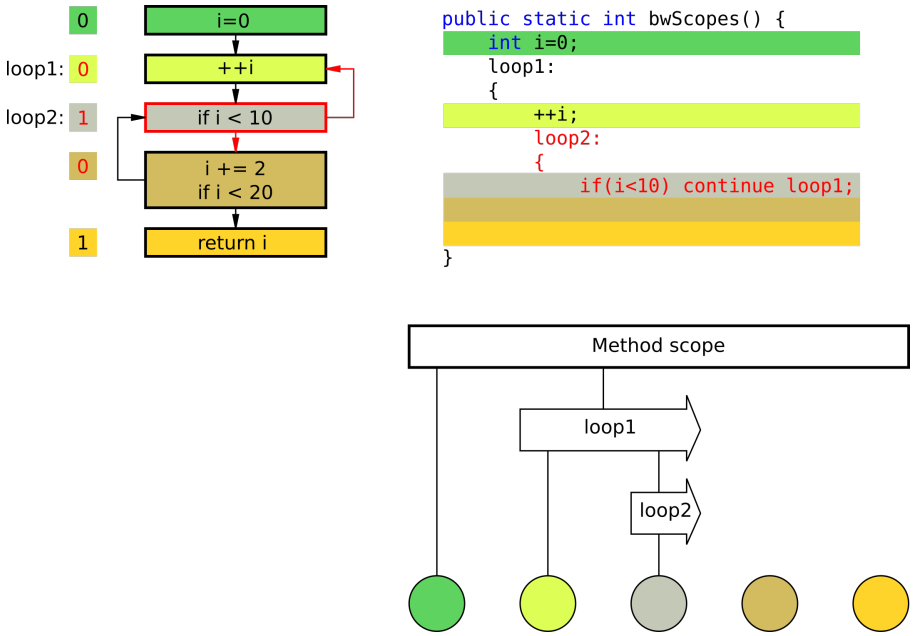
Obrázek 6.40: Vložen první statement

Přejde se na další uzel CFG. Protože hodnota jeho čítače je nenulová, otevírá se nový scope. Do něj se umístí statement z uzlu CFG. Opět se sníží čítače následníků.



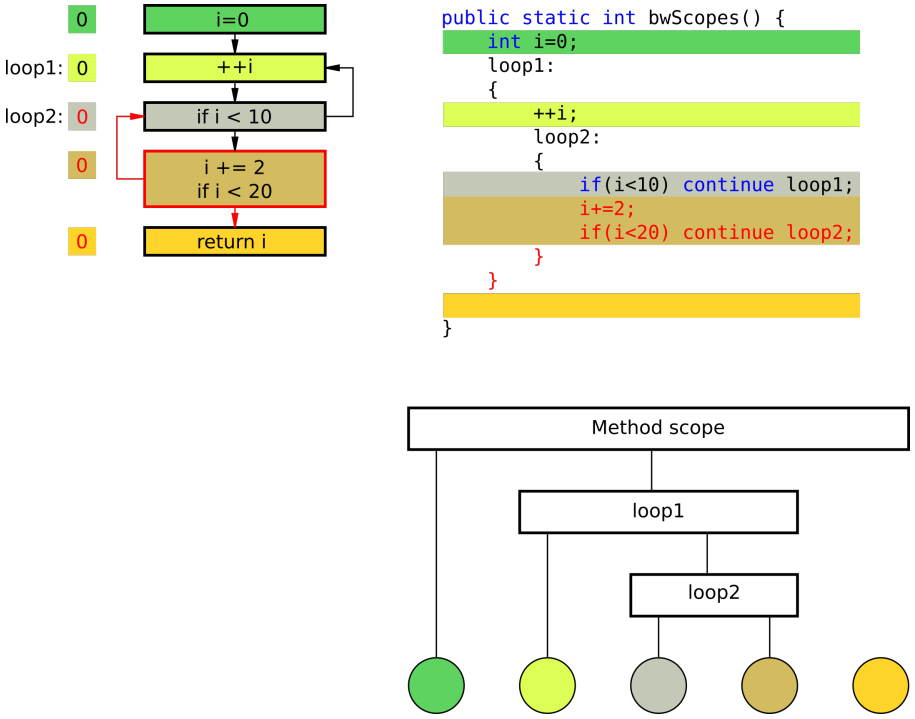
Obrázek 6.41: Otevřen scope

Přejde se na další uzel CFG. Hodnota čítače je opět nenulová, vytváří se tedy nový scope. CFG provádí podmíněný zpětný skok, je tedy vytvořen příslušný podmíněný statement continue. Sníží se čítače následníků. Hodnota čítače pro první scope dosáhla 0. Uzavření však není možné, protože je otevřen vnořený scope. Uzavření se tedy pozdrží do chvíle, kdy to bude možné.



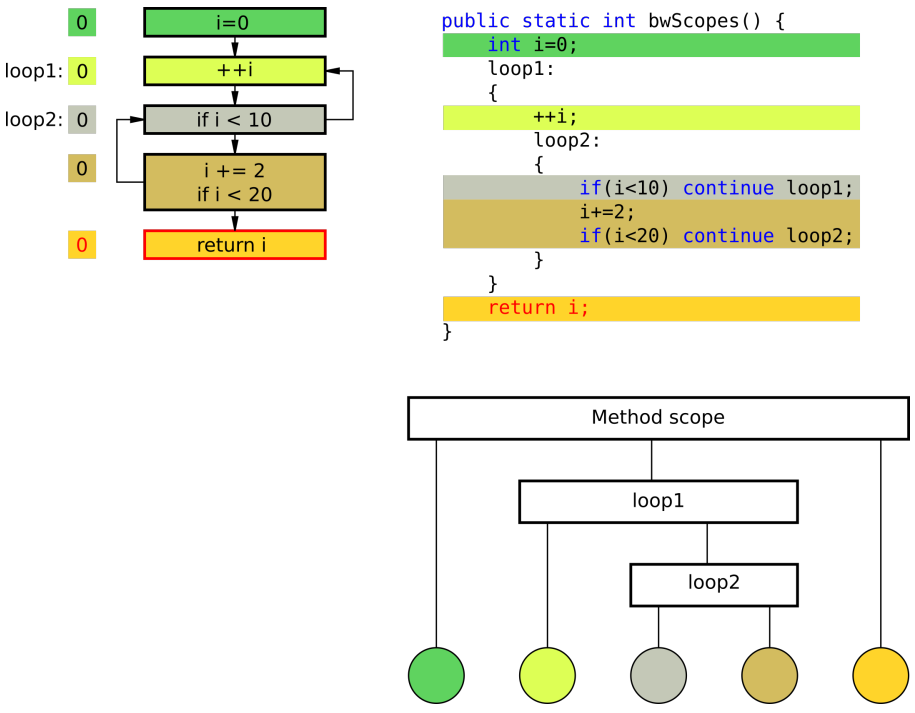
Obrázek 6.42: Otevřen 2. scope

Přejde se na další uzel CFG. Hodnota čítače je nulová, nový scope se nevytváří. Do aktuálního scopu se vloží statement z uzlu CFG a vytvoří se příslušný podmíněný statement continue. Sníží se čítače následníků. Čítač druhého otevřeného scopu dosáhl 0, je tedy uzavřen. Nyní je již možné uzavřít také první vytvořený scope, je tedy uzavřen.



Obrázek 6.43: Vloženy statementy, uzavřeny scopy

Přejde se na další uzel CFG. Zde už dojde pouze k vložení statementu return a algoritmus končí.



Obrázek 6.44: Vložení statementu return

Algoritmus vzniklý sloučení obou algoritmů

Výsledný algoritmus vzniká sloučením obou algoritmů. Zpracovává tedy jak dopředné, tak zpětné skoky. Uvnitř algoritmu jsou oba algoritmy stále do značné míry rozlišitelné. Vytváří se dvě sady obecných scopů. Jedny jsou označované jako dopředné, na ty na tyto se aplikují pouze statementy break, druhé jako zpětné, na tyto jsou aplikované pouze statementy continue. Sloučení spočívá především v tom, že při tvorbě dopředných skoků jsou brány v úvahu i scopy vytvořené kvůli zpětným skokům. Při pozdržování uzavření zpětných scopů jsou naopak brány v úvahu i scopy vytvořené dopřednými skoky. Sloučením algoritmu však také přibyl stav, který algoritmus nedokáže řešit. (toto je důsledek nutnosti přechodu z instrukcí skoků na méně obecné strukturované řídící struktury, příklad kódu (bytecodu) neřešitelného tímto algoritmem bude uveden dále).

Vstupem do algoritmu je seřazená posloupnost uzlů CFG, náležících do CFG u něhož je strukturování požadováno. Uzly CFG jsou zde uvažovány třech druhů.

- uzly bez následníka (zakončené statementem return)
- uzly s jedním následníkem (uzel CFG byl při vytvoření zakončen instrukcí goto nebo následující intrukce byla cíl skoku)
- uzly se dvěma následníky (uzel byl při vytvoření zakončen instrukcí z rodiny if*)

Jediný následník (u uzlů s jedním následníkem) nebo ten následník, jež nepředstavuje původní podmíněný skok (u uzlů se dvěma následníky), bude označován jako defaultní následník. Druhý následník u uzlů se dvěma následníky bude označován jako podmíněný následník.

V rámci algoritmu je s každým uzlem CFG asociován čítač. Mimo to může být s každým uzlem CFG asociován obecný scope vytvořený dopředným skokem (označován dopředný scope) a obecný skope vytvořený kvůli zpětným skokům (označován zpětný scope). V algoritmu pak také figurují aktuální scope a aktuální uzel CFG. Následují kroky algoritmu.

Kroky algoritmu

1. Pro každý uzel CFG se vytvoří čítač s počáteční hodnotou odpovídající počtu hran, přes které je daný uzel CFG dosažitelný. Jako aktuální uzel CFG se nastaví první ze seznamu. Dále se vytvoří scope celé metody a nastaví se jako aktuální. Přejde se na bod 3 (bod 2 by byl zbytečný).

2. Pokud je s aktuálním uzlem CFG asociován dopředný skope, zjistí se jestli odpovídá aktuálnímu scope. Pokud ano, uzavře se, jako aktuální se nastaví jemu přímo nadřazený scope, provede se algoritmus uzavírání scopů a pokračuje se bodem 3. Pokud ne, algoritmus skončí chybou (tj. je otevřen vnořený scope -> dopředný scope nelze uzavřít).

3. Je-li hodnota čítače aktuálního CFG uzlu nenulová, vytvoří se nový obecný skope a nastaví se jako zpětný skope aktuálního uzlu CFG. Nově vytvořený skope se dále vloží do aktuálního scopu a nastaví se jako aktuální.

4. Do aktuálního scopu se vloží statementy z aktuálního uzlu CFG.

5. Čítače všech přímých následníků aktuálního CFG (po hranách CFG) jsou sníženy o 1.

6. Pokud má aktuální uzel CFG následníky tak se pro podmíněného následníka (pokud ho má) vykoná algoritmus pro zpracování následníka, poté se stejný algoritmus provede pro defaultního následníka. Nakonec se provede algoritmus uzavírání scopu. Algoritmus poté ze seznamu seřazených uzlů CFG vybere následující uzel a pokračuje bodem 2. Pokud takový neexistuje a již byl zpracován poslední, algoritmus končí.

Algoritmus zpracování následníka

1. Pokud jde o defaultního následníka, který i v seřazené posloupnosti uzlů CFG následuje (přímo) za aktuálním uzlem CFG, algoritmus končí. Pokud ne, pokračuje se bodem 2.

2. Otestuje se jestli jde o dopředný skok (následník, je následníkem aktuálního uzlu také v seřazené posloupnosti uzlů CFG). Pokud ano, provede se algoritmus pro získání scopu pro dopředný skok a do aktuálního scopu se vloží statement break, cílený na získaný scope (ten je podmíněný příslušnou podmínkou, v případě podmíněného následníka). Pokud ne pokračuje se bodem 3.

3. Do aktuálního scopu se vloží statement continue, cílený zpětný skoupe následníka (ten je podmíněný příslušnou podmínkou, v případě podmíněného následníka).

Algoritmus získání scopu pro dopředný skok

1. Pokud již s cílovým uzlem CFG je asociován dopředný scope algoritmus končí přičemž vrátí tento scope. Pokud ne pokračuje se bodem 2

2. Vytvoří se nový obecný scope. Tento se asociuje s cílovým uzlem CFG, jakožto dopředný scope. Poté se zjistí, jestli platí, že mezi aktuálním scopem a jeho obalujícími scopy, existují takové, které jsou uzavřené (v případě zpětných scopů) a končí před cílem doskoku. Jestli ano, nový scope "obalí"nejvyšší takový (aby se předešlo "křížení"scopů). Pokud ne, nový scope je vložen do aktuálního a poté se nastaví jako aktuální. V obou případech je vrácen nový scope.

Algoritmus uzavírání scopu

1. pokud je aktuální scope čekající na uzavření uzavře se, jako aktuální se nastaví scope jemu přímo nadřazený a tento algoritmus se rekurzivně provede, pokud ne algoritmus končí

6.6. Optimaliace řídících struktur

Poté, co je kód strukturován do podoby obecných scoupů, je zařazena řada optimalizačních kroků. Tyto kroky mají za cíl vytvoření lidsky lépe čitelného kódu s použitím běžných řídících struktur Javy. Optimalizační kroky mají zpravidla velmi jednoduchou formu. Termínem "scope, který nepředstavuje cyklus"je myšlen obecný scope, na něhož není aplikován žádný statement continue,"scope, který představuje cyklus"je naopak scope, na který nějaké continue aplikované je. V obou případech jde ale stále o obecné scopy. Následuje příklad některých takových optimalizačních kroků. Jde však spíše o přiblížení způsobu, jakým optimalizace řídících struktur funguje. Nemusí tedy nutně přesně odpovídat těm aktuálně použitým v dekompileru (s tímto se stále experimentuje).

- Jsou nalezeny scopy, které představují cyklus a obsahují pouze scope, nepředstavující cyklus. Vnořený scope může být vypuštěn a příkazy break aplikované na něj jsou přesměrovány na obklopující scope.
- Jsou nalezeny obecné scopy, které které nepředstavují cyklus a jejich tělo začíná podmíněným breakem, který na ně cílí. Tyto jsou převedeny na statement if, který používá invertovanou podmínku původního podmíněného breaku.
- Jsou nalezeny scopy, které představují cyklus a jejich tělo končí statementem continue, který na ně cílí. Tyto jsou převedeny na cyklus do-while, v případě podmíněného skoku, nebo převedeny na nekonečný cyklus v případě nepodmíněného skoku. (tento může být reprezentován jako for(;;) nebo dále optimalizován)
- Cykly které jsou nekonečnými cykly a začínají podmíněným statementem break na ně cíleným jsou převedeny do podoby while cyklu používajícím invertovanou podmínkou původního podmíněného breaku.

Tyto optimalizační kroky jsou prováděny postupně a rekurzivně na všechny "vnořené"scopy (popř. později také na jiné řídící struktury). Jejich jednoduchost je dána dobrými vlastnostmi výchozích zavedených struktur (tj. obecný scope), dále také tím, že výstup algoritmu pro tvorbu obecných scopů je velmi předvídatelný (nedělá tedy žádné "chytáky"). Součástí této fáze je také potřebné převádění intů zpět na booleany (v podmínkách). Implementace tohoto je však zatím zjednodušená. Dále jsou deklarovány proměnné. Deklarace je přitom umisťována v co "nejhlubším"scopu a co nejpozději v jeho těle tak, aby však svým rozsahem platnosti pokryla všechny svoje použití.

6.7. Generování kódu javy

Po provedení optimalizačních kroků je již vygenerován kód jazyka Java. Na úrovni typů se zdrojové kódy generují s pomocí informací přečtených z class souboru. Na úrovni metod je pak kód generován z abstraktních syntaktických stromů statementů, vytvořených v předchozích krocích. Při generování musí být brána v úvahu priorita a asociativita operátorů. Přidáním závorek je zajištěno, aby pořadí vyhodnocování výrazů v generovaném kódu odpovídalo tomu, které je dáno AST. Algoritmus se snaží přidávat pouze nezbytné závorky aby se dosáhlo správného pořadí vyhodnocování. Přidání závorek může být nezbytné také v některých speciálních případech. To je například unární operátor mínus aplikovaný na zápornou hodnotu, vícenásobné použití těchto operátorů nebo použití tohoto operátoru na druhý operand operace odčítání apod. Závorky se musí přidat tak, aby se předešlo vzniku operátoru pre-dekrementace (např. -(-5)). Dalším takovým příkladem by mohlo být čtení z nově vytvořeného pole, kde by mohlo místo toho dojít k vytvoření vícerozměrného pole (tj. (new int[5])[1] a ne new int[5][1]) apod. Dále musí být také vygenerovány literály a potřebné escape sekvence v textových řetězcích pro znaky, které nelze do řetězců umístit přímo (tj. například znak nového řádku). Generátor zároveň provádí odsazování pro zvýšení přehlednosti generovaného kódu.

6.8. Ukázka kódu generovaného dekompilerem

Zde již následují praktické ukázky dekompilace kódu dekompilerem. Protože práce se zaměřuje na dekompilaci metod, bude zde uveden pouze kód metod (bez obklopující třídy). Na příkladech bude také ukázán vliv optimalizace.

statement if Na tomto příkladu je ukázána dekompilace statementu if. Původní bytecode vzniklý kompilací příslušného if statementu vypadá tak, jak je vidět na obrázku 6.46. Kód je zobrazen v podobě vypsané nástrojem javap (dissassemblerem) z JDK.

```
public static boolean simpleIf(boolean);
Code:
   0: iload_0
   1: ifeq          6
   4: iconst_1
   5: ireturn
   6: iconst_0
   7: ireturn
```

Obrázek 6.45: Metoda v podobě bytecodu

Zde už je dekompilovaný příklad do podoby obecných scoupů bez dalších optimalizací. Ten představuje řešení s použitím scope a podmíněného breaku.

```
public static boolean simpleIf(boolean p1) {
    scope1:
    {
        if (!p1)
            break scope1;
        return true;
    }
    return false;
}
```

Obrázek 6.46: Metoda po strukturování do podoby obecných scopů

Protože si lze všimnout, že scope začíná podmíněným breakem na něj cílícím, je scope převeden na statement if (jež používá invertovanou podmínku).

```
public static boolean simpleIf(boolean p1) {
   if (p1)
      return true;
   return false;
}
```

Obrázek 6.47: Metoda po optimalizaci řídících struktur

cyklus while Na tomto příkladu je ukázána dekompilace statementu while. Původní bytecode vypadá tak jak je vidět na obrázku 6.48.

```
void whileInt();
 Code:
    0: iconst_0
    1: istore_1
    2: iload_1
    3: bipush        100
    5: if_icmpge     14
    8: iinc          1, 1
   11: goto          2
   14: return
```

Obrázek 6.48: Metoda v podobě bytecodu

Kód dekompilovaný do podoby obecných scopů lze vidět na obr. 6.49 . Zde si lze také povšimnout reprezentace obecného scopu v jeho nejobecnější formě. (Tedy for(;;) s implicitním breakem na konci.) Je zde také vidět, že algoritmus pro tvorbu scopů generuje oddělenou sadu scopů pro dopředné a zpětné skoky.

```
void whileInt() {
   int v1 = 0;
   scope1:
   for (;;) {
      scope2:
      {
         if (v1 >= 100)
            break scope2;
         v1 = v1 + 1;
         continue scope1;
      }
      break;
   }
}
```

Obrázek 6.49: Metoda po strukturování do podoby obecných scopů

Je zřejmé, že oba obcné scopy lze sloučit a výsledný scope bude mít nepodmíněný statement continue na konci. (Implicitní break je zde jako reprezentace obecného scopu a nepočítá se.) Tím by šlo o nekonečný cyklus, tento jde však dále optimalizovat na cyklus while, protože prvním statementem v nekonečném cyklu by byl podmíněný statement break, cílící na tento cyklus. Výsledný dekompilovaný kód lze tedy vidět na obr. 6.50.

```
void whileInt() {
   int v1 = 0;
   while (v1 < 100) {
      v1 = v1 + 1;
   }
}
```

Obrázek 6.50: Metoda po optimalizaci řídících struktur

7. Závěr

V rámci práce bylo vytvořeno množství algoritmů, které byly poté použity ve zpětném překladači. Úsilí bylo věnováno především snaze rekonstruovat řídící struktury Javy (což bylo považováno za klíčové pro dekompilaci). V tomto bylo dosaženo úspěchu v podobě funkční realizace algoritmů k tomu navržených. Pro potřeby testování byly vytvořeny třídy s řadou testovacích metod, na kterých byl překladač (algoritmy) testován. Některé tyto třídy byly psány v jazyce Java, jiné obsahovaly ručně psaný bytecode s pomocí nástroje Jasmin. Možnosti vytváření tříd s metodami obsahujícími ručně psaný bytecode se využívalo ke zvýšení kontroly na kódem. Toto umožňovalo snadněji testovat některé speciální a potenciálně problematické konstrukce v bytecodu. Testování ověřilo, že algoritmy splnily očekávání na ně kladená. Použití zpětného překladače mimo oblast rekonstrukce řídících struktur je zatím dosti omezená (také zatím chybí podpora switch instrukcí), avšak algoritmy byly navrženy tak, aby mohly být v budoucnu rozšířeny o podporu dalších konstrukcí Javy.

Literatura

[1] ORACLE. *Annotation (Java Platform SE 7)* [online]. 2014, 2014 [cit. 2015-05-29]. Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Annotation.html

[2] ORACLE. *Enum (Java Platform SE 7)* [online]. 2014, 2014 [cit. 2015-05-29]. Dostupné z: http://docs.oracle.com/javase/7/docs/api/java/lang/Enum.html

[3] VENNERS, Bill. *Inside the Java Virtual Machine*. McGraw-Hill Companies, 1997. 579 s. ISBN: 978-0079132482.

[4] ORACLE. *Naming a Package* [online]. 2015, 2015 [cit. 2015-05-29]. Dostupné z: https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html

[5] LINDHOLM, Tim; YELLIN, Frank; BRACHA Gilad; BUCKLEY, Alex. *Opcode Mnemonics by Opcode* [online]. 2013, 28.2.2013 [cit. 2015-05-29]. Dostupné z: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-7.html

[6] SEDGEWICK, Robert; WAYNE, Kevin. *Operator Precedence in Java* [online]. 2012, 18.1.2015 [cit. 2015-05-29]. Dostupné z: http://introcs.cs.princeton.edu/java/11precedence/

[7] LINDHOLM, Tim; YELLIN, Frank; BRACHA Gilad; BUCKLEY, Alex. *The class File Format* [online]. 2013, 28.2.2013 [cit. 2015-05-29]. Dostupné z: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html

[8] LINDHOLM, Tim; YELLIN, Frank; BRACHA Gilad; BUCKLEY, Alex. *The Java Virtual Machine Instruction Set* [online]. 2013, 28.2.2013 [cit. 2015-05-29]. Dostupné z: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html

[9] LINDHOLM, Tim; YELLIN, Frank; BRACHA Gilad; BUCKLEY, Alex. *Verification of class Files* [online]. 2013, 28.2.2013 [cit. 2015-05-29]. Dostupné z: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10

8. Seznam použitých zkratek a symbolů

JDK	Java development kit
JRE	Java runtime evironment (běhové prostředí Javy)
JVM	Java Virtual Macheine (virtuální stroj Javy)
CFG	control flow graph
SCC	strongly connected component (silně souvislý komponent)