Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Automatický převod programů napsaných v jazyce Java do dalších jazyků (C/C++/FreePascal)

Plzeň 2017 František Šimeček

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 9. května 2017

František Šimeček

Abstract

The main goal of this work is to design and implement a software that can translate Java source code into other languages. The languages are: C, C++, Free Pascal, and C#. The goal is not to translate any program written in Java, but only a small subset of basic Java functions, defined by the needs of programming competition PilsProg.

Abstrakt

Hlavním cílem této práce je navrhnout a implementovat program, který bude umět přeložit zdrojový kód Javy do jiných programovacích jazyků. Tyto jazyky jsou: C, C++, Free Pascal a C#. Cílem není aby tento program byl schopen převést libovolný program napsaný v Javě, ale pouze podmnožinu základních funkcí Javy, definovaných potřebami programovací soutěže PilsProg.

Obsah

1 Úvod **10**

2 Teorie překladu zdrojového kódu **11**

2.1 Pojmy 11

2.1.1 Programovací jazyk 11

2.1.2 Cílový jazyk 11

2.1.3 Vysokoúrovňový a nízkoúrovňový jazyk 11

2.1.4 Zdrojový kód 11

2.1.5 Kompilátor 12

2.1.6 Parsování 12

2.1.7 Notace výrazů 12

2.1.8 Regulární výraz 12

2.1.9 Gramatika 13

2.1.10 Statická analýza kódu 13

2.2 Parsování jazyka 13

2.2.1 Prediktivní parsování 14

2.2.2 Rekurzivní sestup 14

2.3 Tabulka symbolů 14

2.3.1 Implementace 14

2.4 Mezijazyk 15

2.5 Vyhodnocování výrazů 16

2.5.1 Nahrazení operátorů 16

2.5.2 Volání metod 16

2.5.3 Převod infixové notace do postfixové 16

2.5.4 Převod postfixové notace do infixové 17

2.5.5 Nebinární operátory 18

2.6 Nahrazování metod 18

2.7 Generování kódu 18

2.8 Garabage collector 19

2.9 Ověření správnosti generovaného kódu 19

3 Analýza cílových jazyků **21**

3.1 Příkazy 21

3.1.1 Příkaz switch 21

3.1.2 For cyklus 21

3.2 Výjimky 22

3.2.1 Try-catch blok 22
3.2.2 Příkaz throw 22
3.2.3 Volání metody vyhazující výjimku 23
3.2.4 Finally blok 23
3.3 Výrazy 23
3.3.1 Vyhodnocování argumentů 23
3.3.2 Řetězení řetězců 23
3.3.3 Automatická inicializace 24
3.3.4 Kopírování řetězců 24
3.3.5 Formátovací řetězec 24
3.3.6 Přiřazení do formálních parametrů v Pascalu 24
3.3.7 Operátory 24
3.3.8 Délka pole 25
3.4 Datové typy 25
3.4.1 Char v Pascalu 25
3.4.2 Číselné typy 25
3.4.3 Obalovací typy 25
3.5 Třídy 26
3.5.1 Objekty v C 26
3.5.2 Volání this konstruktoru 26
3.5.3 Dědičnost od třídy Object 26
3.5.4 Statická inicializace 26

4 Existující software pro překlad zdrojových kódů 27
4.1 Java2C 27
4.2 J2C 27
4.3 Java to C++ Converter 27
4.4 Java to C# Converter 27
4.5 Sharpen 28
4.6 Haxe 28
4.7 VARYCODE 28
4.8 XES 28
4.9 Souhrn 29
4.9.1 Souhrn nedostatků existujícího softwaru 29

5 Analýza řešení 30
5.1 Požadavky na řešení 30
5.2 Případy užití 30
5.3 Vlastní parsování vs. knihovna 31
5.3.1 Java Parser 31

5.3.2 Požadované funkce 31
5.4 Vymezení podmnožiny Javy 32
5.4.1 Objektově orientované programování 32
5.4.2 Lambda výrazy 32
5.4.3 Výjimky 32
5.4.4 Kolekce 33
5.4.5 Regulární výrazy 33
5.4.6 Char 33
5.4.7 Pole polí 34
5.4.8 Příkaz goto 34
5.4.9 Enum 34
5.4.10 Desetinný oddělovač 34
5.4.11 Anotace 34
5.5 Návrh řešení 35
5.5.1 Grafické rozhraní 35
5.5.2 Parsování a překlad 35
5.5.3 Struktura aplikace 35

6 Implementace **36**
6.1 Použité prostředky 36
6.2 Struktura aplikace 36
6.2.1 Architektura 36
6.2.2 UML diagram tříd 37
6.2.3 Datový tok 39
6.2.4 Soubory 39
6.3 Parsování Javy 40
6.3.1 Předzpracování 40
6.3.2 Parsování 40
6.4 Tabulka symbolů 40
6.4.1 Implementace 40
6.4.2 Využití při parsování 41
6.4.3 Využití při překladu 41
6.5 Zpracování výrazů 41
6.5.1 Zpracování unárních operátorů 42
6.5.2 Převod do postfixové notace 42
6.5.3 Tvorba stromu 43
6.6 Nahrazování knihovních metod 43
6.6.1 Přímé nahrazení 43
6.6.2 Přidání kódu v Javě 44
6.6.3 Přidání přeloženého kódu 44

6.6.4 Seznam nahrazovaných metod 44
6.6.5 Vlastní nahrazení 46
6.7 Substituce 46
6.8 Uvolňování paměti 47
6.8.1 Objekty 47
6.8.2 Pole 48
6.8.3 Řetězce 48
6.8.4 Neuvolněná paměť 49
6.9 Překlad tříd 49
6.9.1 C a C++ 49
6.9.2 Free Pascal 49
6.9.3 C# 50
6.10 Specifické problémy a jejich řešení 51
6.10.1 Substituce ve výrazech 51
6.10.2 Volání konstruktoru this 52
6.10.3 Konverze typu výsledného výrazu 53
6.11 Nahrazování kolekcí 53
6.12 Nahrazování tříd 54
6.13 Zmenšení výstupního kódu 54
6.14 Grafické rozhraní 55
6.14.1 Hlavní okno 55
6.14.2 Textový editor 55

7 Testování 56
7.1 Problém s unit testy 56
7.2 Testování pomocí programů 56
7.3 Implementace testů pomocí programů 57
7.4 Problémy v testovacích datech 57
7.5 Výsledné statistiky testů programů 57
7.6 Unit testy 58
7.7 Pokrytí kódu testy 59
7.8 Rychlosti výpočtů po překladu 59
7.8.1 Benchmarky Pascalu 59
7.9 Uvolňování paměti 61
7.10 Čitelnost 61
7.11 Časová náročnost 62

8 Nevyřešné problémy 63
8.1 Bitové posuny 63
8.1.1 Operátor >> 63

8.1.2 Operátor >>> 63
8.2 Nahrazené metody 63
8.2.1 Matematické funkce 63
8.2.2 Tisknutí reálných čísel 63
8.2.3 Metoda `Scanner.next()` 64
8.3 Finally blok 64
8.4 Metody `HashCode` a `compareTo` 64
8.5 Prázdné pole 64
8.6 Kolekce 65
8.7 Uvolňování paměti 65
8.8 Formátování 65
8.9 Dělení řetězce 66

9 Závěr 67
9.1 Budoucí práce 67

Literatura 68

A Uživatelská dokumentace 71
A.1 Sestavení 71
A.2 Spuštění 71
A.3 Hlavní okno 72
A.4 Nastavení 74
A.5 Textový editor 75
A.6 Přidání knihovní metody 76
A.6.1 Soubor replace.xml 76
A.6.2 Implementace metody v nativním jazyku 79
A.6.3 Implementace metody v Javě 79
A.7 Přidání knihovní třídy 80
A.8 Přidání kolekce 80
A.9 Rezervované symboly 80
A.9.1 Proměnné a metody 80
A.9.2 Třídy 81
A.9.3 Seznam rezervovaných jmen tříd 81
A.10 Seznam podporovaných metod a konstant 81
A.11 Seznam podporovaných tříd a kolekcí 85

1 Úvod

Katedrou informatiky a výpočetní techniky je každoročně pořádána soutěž v programování PilsProg [1]. Pro tuto soutěž je potřeba vytvořit zadání, které se musí otestovat naprogramováním řešení. Vzhledem k tomu, že je možné úlohu řešit v několika různých programovacích jazycích (C, C++, Free Pascal, Java, v budoucnu možná i C#), je třeba ověřit, zda jde tyto úlohy vyřešit běžným způsobem ve všech těchto jazycích. Přestože se může zdát, že je jasné že existuje pro tyto úlohy řešení ve všech jazycích, zkušenost ukázala, že existují drobné odlišnosti v těchto jazycích, jako například různé zpracování standardního vstupu, které mohou způsobovat problémy. Proto se zadavatel úlohy potřebuje přesvědčit, zda je úloha řešitelná (správně podmíněna) ve všech jazycích povolených v této soutěži.

Aby nemusel zadavatel tuto úlohu programovat ve všech těchto jazycích, vznikl tento projekt, který má za úkol zautomatizovat překlad jednoho řešení napsaného v jazyce Java do všech ostatních jazyků povolených v soutěži a případně jej upozornit na potenciální problémy, o kterých je pak možné informovat řešitele v zadání úlohy. To umožní zadavateli úlohy napsat její řešení jen jednou a jednoduše ověřit, zda úloha není špatně podmíněná.

V úlohách soutěže PilsProg jde vždy pouze o transformaci standardního vstupu na standardní výstup. Proto tento překladač nemusí umět práci se soubory a jiné komplikované konstrukce, ale stačí, když překladač bude umět jen funkce používané pro tento typ úloh. Jedná se zejména o metody pro práci s řetězci a matematické funkce, ale také se hodí podpora základních kolekcí. Cílem této práce tedy není vytvořit překladač libovolného kódu Javy, ale pouze jeho podmnožiny. Dalším cílem je používat v překladu standardní knihovní funkce jazyků, do kterých se překládá tak, jak by to dělal programátor daného jazyka. Cílem není obcházet tyto funkce vlastní nízkoúrovňovou implementací, protože program by měl být schopný upozornit na odlišnosti těchto jazyků a ne je obcházet.

2 Teorie překladu zdrojového kódu

2.1 Pojmy

2.1.1 Programovací jazyk

Programovací jazyk je formální jazyk počítače, nebo vytvořený jazyk navržený pro komunikaci s počítačem. Programovací jazyky se používají ke psaní programů, ovládání strojů nebo vyjádření algoritmů [2].

2.1.2 Cílový jazyk

V této práci bude zmiňován pojem cílový jazyk, čímž je myšlen jazyk, do kterého se překládá vstupní zdrojový kód (v tomto případě Javy).

2.1.3 Vysokoúrovňový a nízkoúrovňový jazyk

Programovací jazyky se dělí na vysokoúrovňové a nízkoúrovňové.

Vysokoúrovňový jazyk abstrahuje hardware od programátora. Vysokoúrovňový jazyk obsahuje konstrukce, které usnadňují programátorovi práci jako například objekty, dědičnost nebo rozsáhlé standardní knihovny funkcí. Nevýhoda těchto jazyků je ta, že programy v nich jsou typicky pomalejší, nebo omezují programátora v tom, co může v daném jazyku udělat.

Nízkoúrovňové jazyky typicky obsahují jen základní instrukce, kterými lze udělat cokoliv a velmi efektivně, ale vývoj programu v těchto jazycích je mnohem náročnější a pomalejší [3].

2.1.4 Zdrojový kód

Zdrojový kód je posloupnost počítačových instrukcí (případně komentářů) zapsaná programovacím jazykem. Zdrojový kód vzniká jako výsledek práce programátora a pokud se nejedná o interpretovaný jazyk (interpretované programovací jazyky se vykonávají rovnou ze zdrojového kódu), pak je možné jej přeložit do strojového kódu (binární kód čitelný počítačem), který může být následně vykonán počítačem [4].

2.1.5 Kompilátor

Kompilátor je program, který překládá zdrojový kód do strojového kódu, který může být následně vykonán počítačem [2].

2.1.6 Parsování

Syntaktická analýza neboli parsování je proces analýzy textu za účelem pochopení jeho obsahu. Pro účely vyvíjeného programu bude potřeba zparsovat zdrojový kód Javy, aby jej bylo možné následně přeložit do jiného jazyka [5].

2.1.7 Notace výrazů

Infixová notace

Infixová notace je zápis logického či aritmetického výrazu s operátory mezi operandy. Příkladem infixové notace je: $1 + 2 = 3$. Tuto notaci běžně používají lidé, ale pro počítač je obtížně zpracovatelná, protože precedence operátorů má komplikovaná pravidla a v některých případech vyžaduje závorky [6].

Postfixová notace

Postfixová notace je zápis logického či aritmetického výrazu s operátory za operandy. Příkladem postfixové notace je: $1\ 2 + 3 =$. Tato notace je lépe zpracovatelná pro počítač a navíc nevyžaduje závorky, protože precedence operátorů je jasně daná ze zápisu [6].

Prefixová notace

Prefixová notace je zápis logického či aritmetického výrazu s operátory před operandy. Příkladem prefixové notace je: $= + 1\ 2\ 3$. Tato notace je také dobře zpracovatelná pro počítač a nevyžaduje závorky, protože precedence operátorů je jasně daná ze zápisu, nicméně postfixová notace se pro tyto účely používá častěji [6].

2.1.8 Regulární výraz

Regulární výraz je posloupnost znaků definující vyhledávací vzor, za účelem hledání těchto vzorů v textu. Pro účely této práce se regulární výrazy hodí při parsování zdrojového kódu [2, 7]. Díky tomuto aparátu je možné například najít libovolný identifikátor, který je v jazyce Java definovaný jako

alfanumerický řetězec s případnými podtržítky nebo dolary, který nesmí začínat číslem. Regulární výraz pro takovýto identifikátor by mohl vypadat například takto: `([a-z]|[A-Z]|_|$)([a-z]|[A-Z]|[0-9]|_|$)*`. První část výrazu značí, že identifikátor musí začínat libovolným písmenem, podtržítkem nebo dolarem a druhá část výrazu značí, že může následovat to samé a navíc i číslice. Hvězdička za závorkou značí, že druhá část výrazu se může opakovat 0 až nekonečně krát.

2.1.9 Gramatika

Gramatika je sada pravidel, jak tvořit řetězce (věty) z abecedy jazyka tak, aby splňovaly syntaxi tohoto jazyka. Gramatiky jsou také používané při parsování programovacích jazyků [2]. Ukázka gramatiky se nachází ve Výpisu 2.1.

```
1 Program → Třída Program
2 Třída → Metoda Třída | Výraz Třída
3 Metoda → Hlavička Blok
4 Blok → { Příkazy }
5 ...
```

Výpis 2.1: Příklad podmnožiny gramatiky popisující program v jazyce Java

2.1.10 Statická analýza kódu

Statická analýza kódu se používá za účelem hledání chyb a potenciálních chyb ve zdrojovém kódu. Kompilátory většinou hledají pouze syntaktické chyby, které brání v kompilaci. Na statickou analýzu existuje řada nástrojů pro různé programovací jazyky. Příkladem chyby, kterou může analýza najít je neinicializovaná proměnná, která kompilátoru nemusí vadit [8].

2.2 Parsování jazyka

Úkolem parseru je zjistit, jak mohl být vstup odvozen z počátečního symbolu gramatiky. Toho je možné dosáhnout dvěma přístupy:

- Shora dolů – Parser se snaží převést počáteční symbol na vstup
- Zdola nahoru – Parser se snaží převést vstup na počáteční symbol

2.2.1 Prediktivní parsování

Prediktivní parser (parsování shora dolů) má schopnost předvídat, jaký symbol následuje a nepotřebuje se vracet. Například pokud bylo nalezeno klíčové slovo `if`, je jasné, že musí následovat výraz s podmínkou a nic jiného [7].

2.2.2 Rekurzivní sestup

Rekurzivní sestup je metoda prediktivního parsování, která je založená na tom, že každému neterminálnímu symbolu (například příkazu `if`) odpovídá metoda, která zparsuje následující element a rozhodne se, kterým neterminálním symbolem (kterou metodou) se bude pokračovat. Rekurzivní proto, že se některé prvky zdrojového kódu mohou do sebe zanořovat, což způsobí rekurzivní volání při parsování tohoto kódu [7, 9].

2.3 Tabulka symbolů

Tabulka symbolů slouží pro uchování jmen proměnných nebo metod. Pokud je ve zdrojovém kódu deklarována proměnná, je třeba ji uložit do tabulky symbolů, aby, až tato proměnná bude použita, bylo jasné, o jakou proměnnou jde a případně jakého typu byla definována [2, 7].

Vzhledem k tomu, že se parsuje jazyk Java, tabulka symbolů se musí řídit pravidly tohoto jazyka. Pokud je ukončen blok, proměnné definované v tomto bloku musí být z tabulky vyjmuty, aby nebyly zaměněny s proměnnými definovanými později. Protože se bude zdrojový kód Javy překládat i do jiných jazyků, nemůžou se ignorovat ani jejich pravidla a omezení. Například jazyk Pascal není case sensitive (nerozlišuje velká a malá písmena), proto tato tabulka symbolů také nesmí být. Dále v některých jazycích nesmí být stejně pojmenovaná proměnná a metoda, také tato tabulka symbolů musí zohlednit klíčová slova (slovo, které nemůže být použité jako identifikátor, protože má jiný význam v daném jazyce) všech jazyků, do kterých se bude zdrojový kód překládat.

2.3.1 Implementace

Existuje mnoho způsobů jak implementovat tabulku symbolů, ale dají se rozlišit dva základní přístupy – persistentní a imperativní.

U persistentní tabulky symbolů se nezahazují prvky, pokud se dostanou mimo rozsah platnosti. Tuto tabulku je možné implementovat například pomocí seznamu, kde se nové prvky vkládají na začátek seznamu a staré se

pouze posouvají dozadu [7].

U imperativní tabulky symbolů, se odstraňují staré prvky, které se dostanou mimo rozsah platnosti. Tuto tabulku je možné implementovat například pomocí zásobníku, kde při vystoupení z rozsahu platnosti je prvek zahozen, protože již není potřebný [7].

2.4 Mezijazyk

Při kompilaci (překladu do strojového kódu) programů se často používá mezijazyk (někdy objektová reprezentace), do kterého se přeloží vysokoúrovňový jazyk a ze kterého se následně generuje strojový kód. Účel tohoto mezijazyka je usnadnění překladu do strojového kódu, který je platformově závislý, tudíž je nutné napsat tento kompilátor pro každou platformu zvášť. Tento mezijazyk tedy musí být snadno přeložitelný do strojového kódu a typicky je nízkoúrovňový [2, 10]. Některé jazyky používají jako mezijazyk jiný již existující jazyk (například JavaScript), aby nemuseli psát vlastní interpret nebo kompilátor, který není jednoduchý. Tudíž stačí napsat překladač z jejich jazyka do JavaScriptu a použít již existující interpret na vykonání kódu jazyka. Příklad takového jazyka je TypeScript, který se překládá do JavaScriptu a ten je posléze interpretován.

V této práci bude mezijazyk potřeba, aby se nemusel vytvářet překladač pro každý cílový jazyk zvlášť. Potřebný mezijazyk však nemusí být nízkoúrovňový vzhledem k tomu, že jazyky, do kterých se bude překládat, jsou poměrně podobné překládanému jazyku Java.

Mezijazyk vůbec nemusí být uměle vytvořený jazyk s definovanou syntaxí, stačí objektová reprezentace původního kódu. Tato objektová reprezentace ušetří práci s opětovným parsováním kódu, protože stačí zdrojový kód Javy převést do této objektové reprezentace jen jednou a následně pro každý tento objekt napsat způsob překladu do jednotlivých jazyků. Výpis 2.2 ukazuje příklad objektové reprezentace příkazu větvení.

```
1 objekt If {
2   atribut: podmínka;
3   atribut: pozitivní_blok;
4   atribut: negativní_blok;
5 }
```

Výpis 2.2: Příklad objektové reprezentace příkazu `if`

2.5 Vyhodnocování výrazů

Aritmetické a logické výrazy nelze jednoduše přeložit po jednotlivých operátorech a operandech, některé operátory je nutné nahradit jinými a také je potřeba vyhodnotit volání metod.

2.5.1 Nahrazení operátorů

Příkladem operátoru, který je potřeba vyhodnotit je dělení. Jazyk Java má na celočíselné dělení i reálné dělení ten samý operátor /, ale Pascal má pro každé dělení různý operátor. Proto je třeba výraz převést do postfixové notace, aby bylo zjištěno, které operandy do dělení skutečně vstupují, zjistit jejich typ a pokud jsou oba celočíselné, nahradit operátor dělení operátorem celočíselného dělení.

2.5.2 Volání metod

Ve výrazech programovacího jazyka se mohou nacházet i volání metod. Aby bylo zjištěno, kterou metodu dané volání metody volá, musí být vyhodnoceny typy všech jeho argumentů. Toto je druhý důvod, proč je potřeba výrazy převést do postfixové notace. V postfixové notaci lze nad každou operací zjistit jakého typu bude výsledek, a postupně tak zjistit typ celého výrazu. Když jsou vyhodnocené typy všech výrazů v argumentech volání metody, může být nalezena metoda, kterou volání metody volá a lze jí nastavit příslušnou referenci. Tato reference je potřebná ze dvou důvodů – zjištění návratového typu pro vyhodnocení jiných výrazů a případného přejmenování volané metody kvůli kolizi s jiným identifikátorem.

2.5.3 Převod infixové notace do postfixové

Pro převod do postfixové notace se vytvoří zásobník na operátory a prochází se výraz zleva doprava. Pokud je nalezen operand, přepíše se do výstupu. Pokud je nalezena levá závorka, vloží se na zásobník. Pokud je nalezena pravá závorka, postupně se přesunou všechny operátory ze zásobníku do výstupu, dokud se nenarazí na levou závorku, která se ze zásobníku jen odstraní. Pokud je nalezen operátor, přesunou se ze zásobníku všechny operátory s vyšší nebo rovnou prioritou do výstupu a vloží se nalezený operátor do zásobníku. Až se dojde na konec výrazu, přesunou se všechny operátory ze zásobníku do výstupu [6]. Výpis 2.3 znázorňuje tento algoritmus zapsaný v pseudokódu.

```
for(Token t: expression){
  if(t instanceof Operand) output.add(t);
  else if(t.equals("(")) stack.push(t);
  else if(t.equals(")")){
    Token t2;
    while(!(t2 = stack.pop()).equals("(")) output.add(t2);
  }else{
    while(!stack.isEmpty() && stack.peek().getPriority() >=
      t.getPriority()) output.add(stack.pop());
    stack.push(t);
  }
}
while(!stack.isEmpty()) output.add(stack.pop());
```

Výpis 2.3: Algoritmus převodu infixového výrazu do postfixové notace

2.5.4 Převod postfixové notace do infixové

Všechny jazyky, do kterých se bude překládat, pracují s infixovou notací, proto je potřeba po vyhodnocení výrazu daný výraz převést zpět. Opět se využije zásobník, tentokrát ale na operandy. Postfixový výraz se prochází opět zleva doprava. Pokud je nalezen operand, vloží se do zásobníku. Pokud je nalezen operátor vyjmou se dva operandy ze zásobníku, otočí se jejich pořadí a vloží se mezi ně operátor, vzniklý infixový výraz se vloží zpět na zásobník. Po zpracování celého výrazu bude infixový výraz na vrcholu zásobníku [6]. Výpis 2.4 znázorňuje tento algoritmus zapsaný v pseudokódu. Stejným způsobem se i z postfixového výrazu vytvoří strom binárních operací, místo toho, aby se do zásobníku vracel výraz složený z dvou operandů a operátoru se do něj vloží objekt `BinaryOperation`. Převod tohoto stromu do infixové notce je jednoduchý, stačí zavolat překlad nad kořenem tohoto stromu, a každá operace přeloží pouze své operandy a operátor v infixovém tvaru, z čehož vznikne výraz v původním infixovém tvaru.

```
for(Token t: expression){
  if(t instanceof Operand) stack.push(t);
  else{
    Token a = stack.pop();
    Token b = stack.pop();
    stack.push(new Expression(b + t + a));
  }
}
```

Výpis 2.4: Algoritmus převodu postfixového výrazu do infixové notace

2.5.5 Nebinární operátory

Uvedené algoritmy převodů výrazů počítají pouze s binárními operátory, ale ve výrazech jazyka Java se mohou vyskytovat také unární i ternární operátory. Tyto operátory je třeba ještě před převodem detekovat a seskupit je s příslušným operandem. Nadále lze s vytvořenou skupinou zacházet jako s operandem.

2.6 Nahrazování metod

Knihovní metody Javy je třeba nějakým způsobem přeložit, což je možné udělat několika způsoby. Tím jednodušším způsobem je vzít zdrojový kód této knihovní metody a přidat jej ke zdrojovému kódu programu, který se překládá a přeložit jej jako součást tohoto programu. Toto ale není vždy možné, protože tato metoda nemusí mít implementaci ve zdrojovém kódu Javy, ale může být poskytována jako služba JVM (Java Virtual Machine). Případně může metoda obsahovat komplikované konstrukce, které ve vymezené podmnožině Javy nejsou podporovány. Proto v takovémto případě je potřeba metodu přeložit pomocí nativní metody jazyka, do kterého překládáme. Tyto metody však nemusí mít stejné argumenty, nebo se mohou lišit funkcionalitou. V těchto případech je potřeba volání metody obalit příkazy, které přetransformují vstup a výstup, aby volání metody odpovídalo tomu v Javě. Tyto nahrazené metody je třeba důkladně otestovat, zda se skutečně chovají stejně za všech okolností.

2.7 Generování kódu

Převod objektové reprezentace kódu zpět do zdrojového kódu se zdá jednoduché a pokud se objektová reprezentace příliš neliší od cílového jazyka tak tomu tak i je, ale tato podmínka není vždy splněna. Cílový jazyk nemusí totiž pro danou operaci vůbec mít jazykové prostředky, nebo se může mírně lišit ve funkcionalitě (sémantice). Toto typicky vede k tomu, že je nutné jeden původní příkaz přeložit pomocí více příkazů, nebo pomocí použití jiného příkazu. Příkladem je `for` cyklus jazyku Pascal, který není tak univerzální jako ten v jazyce Java. Výpis 2.5 ukazuje překlad `for` cyklu v Pascalu.

```
1 // Java
2 for(int i = 0; i < 10; i+=2)
3 {
4   // do something
5 }
```

```
(* Pascal *)
i := 0;
while(true) do
begin
  if(not(i < 10)) then break;
  (* do something *)
  i := i + 2;
end;
```

Výpis 2.5: Příklad překladu příkazu `for`

Příklad ve Výpisu 2.5 by bylo možné přeložit i elegantněji, ale toto je univerzálnější řešení, které se vypořádá s libovolnou podmínkou či změnou iterační proměnné. V tomto překladu je také potřeba vložit změnu iterační proměnné před každý příkaz `continue`.

2.8 Garabage collector

Java se stará o uvolňování paměti sama, tudíž ve zdrojovém kódu nejsou příkazy na příslušné uvolnění paměti. Některé jazyky, do kterých se bude překládat, však garbage kolekci nemají, tudíž je zapotřebí příkazy na uvolnění paměti vygenerovat pomocí analýzy kódu, nebo použít již implementovaný garbage collector pro daný jazyk [11]. Případně je možné uvolňování paměti neprovádět vůbec a doufat, že paměť bude vždy stačit. Protože se jedná jen o jednoduché programy, které transformují vstup na výstup a po dokončení uvolní paměť operační systém, tak by ve většině případů stačit měla. Vzhledem k tomu, že paměťová náročnost není prioritou, bude nejlepší kompromis a uvolňovat paměť jen částečně pomocí jednoduché analýzy zdrojového kódu. Tento přístup sice neuvolní veškerou nepotřebnou paměť, ale lepší než žádnou.

2.9 Ověření správnosti generovaného kódu

Vzhledem k tomu, že ke každé úloze jsou vytvořeny vstupy a očekávané výstupy, na kterých se testují řešení účastníků soutěže, je otestování kódu vygenerovaného vyvíjenou aplikací jednoduché. Stačí tento vygenerovaný kód spustit nad stejným validátorem, nad kterým se spouští úlohy řešitelů a ten již sám ověří, zda je program v daném jazyce řešitelný. Toto samozřejmě nutně neznamená, že je program správný a bezchybný. Nicméně pokud kód generuje stejné výstupy jako kód, ze kterého byl přeložený, pak tento překlad splnil požadovaný účel.

Další pomůckou ověření správnosti generovaného kódu je statická analýza kódu. Existuje celá řada nástrojů pro statickou analýzu kódu různých jazyků. Ani tato analýza nezaručí, že je vygenerovaný kód správný, ale pokud tento překlad dává stejné výsledky pro testovací vstupy a zároveň statická analýza nehlásí zásadní problémy, pak lze s vysokou pravděpodobností tvrdit, že byl překlad správný.

Statickou analýzu je však třeba brát s rezervou, protože generovaný kód často nesplňuje zásady programování v daném jazyce. Překlad pouze generuje kód, se stejnou funkcionalitou jako zdrojový jazyk. Pokud cílový jazyk nepodporuje některé konstrukce překládaného jazyka, pak se tyto konstrukce často přeloží ne úplně ideálním způsobem, protože typický způsob, jakým by se to v daném jazyce řešilo může mít mírně odlišnou funkcionalitu. Příkladem může být překlad příkazu `switch` posloupností příkazů `if-else`, nebo překlad příkazu `for` příkazem `while`.

3 Analýza cílových jazyků

Překlad programovacích jazyků sebou přináší různé problémy, zejména kvůli absenci některých programovacích konstrukcí v cílových jazycích či jejich odlišné funkcionalitě. Je proto potřeba provést analýzu jazyků, do kterých se překládá, a porovnat jejich chování s Javou [12, 13]. Následuje seznam významných odlišností jazykových konstrukcí.

3.1 Příkazy

3.1.1 Příkaz switch

Ekvivalent příkazu `switch` v jazyce Pascal (příkaz `case`) se nechová stejně jako v Javě. Není v něm možné projít několika větvemi současně. Proto je příkaz `switch` do Pascalu potřeba přeložit jako posloupnost příkazů `if` uvnitř `repeat until` cyklu, aby bylo možné použít příkaz `break`. Také je zavedena proměnná `lastCase`, která je nastavená na číslo posledně vykonaného `casu`, což umožní vykonat další větev, pokud nebyl proveden příkaz `break`. Výpis 3.1 ukazuje překlad příkazu `switch` v Pascalu.

```
1 repeat
2   lastCase := 0;
3   if(c = val1) then
4     begin
5       doSomething();
6       lastCase := 1;
7     end;
8   if((c = val2) or (lastCase = 1)) then doSomethingElse();
9 until(true);
```

Výpis 3.1: Příklad překladu příkazu switch

Obdobný problém má jazyk C#, ve kterém také není možné projít několika větvemi současně, nicméně je možné na konec každého `case` vložit příkaz `goto case x;`, což je jednodušší řešení. Dále pro zjednodušení překladu není možné příkaz použít nad proměnnou typu `String`.

3.1.2 For cyklus

V jazyce Java je možné do hlavičky `for` cyklu napsat i velmi komplikované konstrukce, které by ostatní jazyky nezvládly, nebo by z důvodu substituce

nebylo možné dané příkazy do hlavičky vložit. Proto je v některých případech potřeba for cyklus nahradit while cyklem.

3.2 Výjimky

Jazyk C nemá výjimky, proto je třeba vytvořit aparát, který umožní výjimky vyhazovat a odchytávat, i když jen s omezenou funkcionalitou.

3.2.1 Try-catch blok

Try-catch blok se dá přeložit v C pomocí dvou if příkazů a návěští. Blok try je jednoduchý příkaz if s podmínkou, která je vždy pravdivá. Blok catch se přeloží jako příkaz if s podmínkou, která testuje globální proměnnou, ve které je informace o tom zda byla vyhozena výjimka. Před catch blok se také vloží návěští, na které se skočí z try bloku v případě vyhozené výjimky. Uvnitř catch bloku se mimo jiné resetuje globální proměnná, která indikuje vyhozenou výjimku. Výpis 3.2 ukazuje překlad příkazu try v jazyce C.

```
1 if(1 /* try */)
2 {
3   result = foo(); // foo() může vyhodit výjimku
4   if(_exception_thrown) goto catch0;
5 }
6 catch0: if(_exception_thrown) {
7   _exception_thrown = 0;
8   printf("Exception caught\n");
9 }
```

Výpis 3.2: Příklad překladu příkazu try-catch

3.2.2 Příkaz throw

Příkaz throw se dá přeložit v C několika způsoby, závisejícími na kontextu. Pokud se nachází uvnitř try bloku, nastaví se globální proměnná, indikující vyhozenou výjimku a provede se skok příkazem goto do catch bloku. Pokud se nachází uvnitř metody, která má deklarováno throws Exception opět se nastaví globální proměnná, indikující vyhozenou výjimku a provede se return. Pokud se provádí return u metody, která má návratový typ, vrátí se hodnota null nebo její ekvivalent. Pokud neplatí ani jedna z předchozích možností, provede se chybový výpis a ukončení programu.

3.2.3 Volání metody vyhazující výjimku

Pokud se volá metoda, která má deklarováno, že by mohla vyhodit výjimku, je potřeba po jejím provedení zkontrolovat stav globální proměnné, indikující vyhozenou výjimku, a v případě, že byla výjimka vyhozena provést ekvivalent příkazu `throw` dle kontextu.

3.2.4 Finally blok

Blok `finally`, který může následovat za `try-catch` bloky v některých cílových jazycích neexistuje. Přeložit tento blok by nebylo až tak obtížné stačil by obyčejný blok s návěštím, na které by se mohlo skočit při nestandardním ukončením `try-catch` bloku. Problém je ale ten, že `finally` blok se provede vždy po `try-catch` bloku ať už je ukončen libovolným způsobem. Těchto způsobů je ale celá řada (běžné ukončení, příkaz `return`, příkaz `break`, příkaz `continue` a neodchycená výjimka) a podchytit je není jednoduché. Zároveň by bylo nutné se podle druhu ukončení na konci `finally` bloku příslušně zachovat.

3.3 Výrazy

3.3.1 Vyhodnocování argumentů

Java vyhodnocuje argumenty metod zleva doprava [14], ale C, C++ a Free Pascal mají pořadí vyhodnocování argumentů nedefinované [15, 16]. Tento problém lze vyřešit tak, že každý argument, který by mohl změnit stav programu (např. volání metody) se vyhodnotí před voláním metody a jeho výsledek se uloží do dočasné proměnné, která se následně použije místo tohoto výrazu jako argument.

3.3.2 Řetězení řetězců

V jazyce Java je zřetězování řetězců jednoduché, stačí mezi řetězce vložit operátor `+`, to ale není v C možné. Nejvhodnější metodou pro tento účel je metoda `sprintf`, protože je možné v ní řetězit libovolný počet operandů a zároveň je možné v ní řetězit i proměnné jiných typů (například číselných). Tato metoda, ale také potřebuje buffer, do kterého se výsledek uloží. Problémem je zjistit jak velký buffer je potřeba, buďto se zvolí jistý „dostatečně veliký“, nebo se poscítají délky všech argumentů. Problém je, že ne všechny argumenty jsou řetězce, u kterých lze zjistit délka.

3.3.3 Automatická inicializace

Jazyk Java automaticky inicializuje atributy tříd na jejich výchozí hodnoty jako je například `0` nebo `null` [17]. Některé cílové jazyky toto ale nedělají, tudíž je potřeba při inicializaci instance třídy automaticky provést i inicializaci atributů, které inicializované nebyly na jejich výchozí hodnotu.

3.3.4 Kopírování řetězců

Pokud je v jazyce Java přiřazená proměnná typu `String` do jiné, nebo je tato proměnná použita jako argument, vytvoří se její kopie (Copy on write = vytvoří se kopie teprve tehdy, je-li řetězec modifikován). V jazyce C je však pouze předán ukazatel, proto je potřeba si u každého přiřazení řetězců vytvořit kopii a přiřadit jí místo původní proměnné.

3.3.5 Formátovací řetězec

Formátovací řetězec, který je použit například v metodě `String.format()`, je třeba přetransformovat na formátovací řetězec cílového jazyka. Formátovací řetězce v jazycích C a C++ jsou s Javou téměř stejné, v Pascalu jsou velmi podobné, ale v C# jsou velmi odlišné [18]. Zároveň je potřeba vůbec rozeznat, že se jedná o formátovací řetězec, čehož lze dosáhnout například zjištěním, že se nachází v metodě, která požaduje formátovací řetězec jako argument.

3.3.6 Přiřazení do formálních parametrů v Pascalu

V jazyce Pascal není možné provést přiřazení do formálních parametrů metody v jejím těle viz Výpis 3.3. Je nutné vytvořit si kopii každého parametru, pokud by byl tento parametr někde v metodě modifikován.

```
1 void foo(int x){
2   x = x + 1;
```

Výpis 3.3: Příklad problémového přiřazení

3.3.7 Operátory

Některé operátory jazyka Java mají v ostatních jazycích odlišnou funkcionalitu, nebo v daném jazyku vůbec neexistují, proto je třeba operátor nahradit množinou jiných operátorů se stejnou funkcionalitou, nebo alespoň upozornit uživatele, že použitý operátor by se za určitých okolností nemusel chovat

stejně v jiných jazycích. Příkladem je operátor >>, který má v Pascalu funkcionalitu jako operátor >>> a operátor s funkcionalitou >> jazyk Pascal vůbec nemá.

3.3.8 Délka pole

U dynamických polí v jazycích C a C++ nelze zjistit jejich délka, tudíž je nutné si tuto délku pole někde uložit. Jednou z možností je uložit si délku pole na jeho začátek a díky ukazatelové aritmetice začátek pole posunout, tím pádem zůstanou přístupy na indexy tohoto pole nezměněny a délka pole tak bude uložena na indexu -1. Ukázka uložení délky pole je ve Výpisu 3.4.

```
float* array = (float*)calloc(10 * sizeof(float) +
    sizeof(int32_t), sizeof(char));
array = (float*)((char*)array + sizeof(int32_t));
((int32_t*)array)[-1] = 10;
```

Výpis 3.4: Příklad uložení délky pole v C

3.4 Datové typy

3.4.1 Char v Pascalu

Typ `char` se v jazyce Free Pacal nechová jako celočíselný typ, tudíž nad ním není možné provádět aritmetické operace tak jako v Javě. Je nutné chary v Pascalu obalit funkcemi `chr() a ord()` pokud je s nimi počítáno.

3.4.2 Číselné typy

Číselné typy v jazyce Java mají vždy stejnou velikost (počet bitů) [19], jiné jazyky však můžou mít odlišnou velikost těchto typů podle platformy, na které je program přeložen a spuštěn [20]. Je potřeba použít ty číselné typy, které mají stejně veliký rozsah na všech platformách (např. použitím typů v headeru `stdint.h` v C a C++).

3.4.3 Obalovací typy

Nejen pro použití v kolekcích existují v Javě obalovací typy těch primitivních. V jazyce Java lze s nimi pracovat stejně jako s těmi primitivními, což v některých ostatních jazycích není možné. Proto je třeba ve výrazech zjistit, zda je třeba primitivní typ převést na ten obalovací nebo naopak a přidat do výrazu potřebnou konverzi.

3.5 Třídy

3.5.1 Objekty v C

Jazyk C nemá objekty, má pouze struktury. Není tudíž možné nad strukturou zavolat metodu, ale je nutné do každé nestatické metody dodat referenci na strukturu jako argument.

3.5.2 Volání this konstruktoru

V jazyce Java je možné zavolat konstruktor té samé třídy zavoláním `this()`, což provede další inicializaci pomocí jiného konstruktoru, ale nevytvoří druhou instanci. V jazyku C tato konstrukce neexistuje, jedním možným řešením je zkopírovat obsah volaného konstruktoru namísto provádění tohoto volání. Před tímto zkopírováním obsahu konstruktoru je však potřeba vyhodnotit argumenty volání `this()`.

3.5.3 Dědičnost od třídy Object

V Javě každá třída implicitně dědí od třídy `Object`, čímž mimo jiné objekty získávají implicitní implementace metod `equals`, `hashCode` a `toString` [21]. Vzhledem k tomu, že se neplánuje podporovat dědičnost, je potřeba tyto implicitní implementace metod vložit ke každé třídě, která by je mohla potřebovat, pokud je tato třída neimplementuje sama.

3.5.4 Statická inicializace

Statická inicializace v Javě je provedena před prvním vytvořením instance či použitím statické proměnné nebo metody [22]. Takovýto mechanismus by byl poměrně obtížný implementovat, proto je jednodušší provést statickou inicializaci všech tříd před zahájením metody `main`. Toto řešení však nemusí skončit stejným výsledkem, pokud by statické inicializace tříd byly na sobě závislé. Pro účely vyvíjeného programu stačí, aby uživatel programu věděl, jak statická inicializace po překladu proběhne a nedopouštěl se závislostí mezi statickými inicializacemi tříd. Protože konstanty obalovacích typů jako je typ `Integer` jsou často používány ve statické inicializaci některých tříd, je vhodné tyto obalovací typy inicializovat jako první, tudíž je možné je použít ve statické inicializaci aniž by došlo k potížím.

4 Existující software pro překlad zdrojových kódů

4.1 Java2C

Java2C [23] je program na konverzi zdrojového kódu Javy do C zaměřený na embeded zařízení a realtime aplikace. Program je velice komplikované nakonfigurovat a použít, k této konfiguraci jsem nebyl schopen najít dokumentaci. Vzhledem k tomu, že je účel tohoto programu generovat kód pro embeded zařízení a ne programy na transformaci stadnardního vstupu na standardní výstup, lze usuzovat, že by tento program nebyl pro účel této práce vhodný.

4.2 J2C

J2C [24] je Eclipse plugin na konverzi zdrojového kódu Javy do C++(11). Program by měl zvládnout většinu konstrukcí Javy 1.6, avšak sám autor tvrdí, že negarantuje funkcionalitu a kvalitu kódu. Přeložený kód obsahuje všechny potřebné knihovní třídy Javy přeložené do C++, tudíž je výsledný zdrojový kód poměrně rozsáhlý a obsahuje velké množství souborů.

4.3 Java to C++ Converter

Java to C++ Converter [25] je program na konverzi zdrojového kódu Javy do C++. Program je placený, nicméně nabízí verzi zdarma, která je omezená délkou překládaného kódu. Vygenerovaný kód je čitelný a krátký, nicméně je potřeba s překládaným kódem přeložit stejným způsobem i použité knihovní třídy Javy, protože program si je sám z knihoven nevytáhne. Pokud jsou však v těchto třídách metody, které implementaci neobsahují, protože jí například provádí JVM, pak je tento překlad nemožný.

4.4 Java to C# Converter

Java to C# Converter [26] je program na konverzi zdrojového kódu Javy do C#. Program je placený, nicméně nabízí verzi zdarma, která je omezená

délkou překládaného kódu. Vygenerovaný kód je čitelný a krátký, nicméně je potřeba s překládaným kódem přeložit stejným způsobem i použité knihovní třídy Javy, protože program si je sám z knihoven nevytáhne. Pokud jsou však v těchto třídách metody, které implementaci neobsahují, protože jí například provádí JVM, pak je tento překlad nemožný.

4.5 Sharpen

Sharpen [27] je open source program na konverzi zdrojového kódu Javy do C#. Program funguje jen tehdy, existuje-li namapování použitých knihovních tříd Javy na knihovní třídy C#, což například u třídy `Scanner` neplatí. Třída `Scanner` je ale zásadní, tudíž je tento program nepoužitelný, pokud by nebyla třída `Scanner` v jazyce C# naprogramována a vložena do konfiguračního souboru.

4.6 Haxe

Haxe [28] je vysokoúrovňový programovací jazyk s překladačem do několika různých jazyků. Z potřebných jazyků umí C++, C# a Javu. Neumí však Free Pacal a C, zároveň by se musel uživatel naučit nový programovací jazyk, aby mohl nástroj použít. Vygenerovaný kód obsahuje spousty speciálních konstrukcí souvisejících s programovacím jazykem Haxe, tudíž je výsledný kód objemný a zároveň ne moc dobře čitelný.

4.7 VARYCODE

VARYCODE [29] je online aplikace na konverzi z C#, VB (Visual Basic) a Javy do C#, VB, Ruby a Python (webové stránky tvrdí, že umí i C++, ale je momentálně nedostupné 7.2.2017). Aplikaci je možné si zdarma vyzkoušet na `https://www.varycode.com/converter.html` v omezené funkcionalitě (délka překládaného kódu). Při testovacím překladu z Javy do C# aplikace přeložila pouze ty knihovní třídy a metody, na které existuje ekvivalent v C#, tudíž třídy jako je `Scanner` přeložené nejsou.

4.8 XES

XES [30] (XML Encoded Source) je program napsaný v Javě, který umí zparsovat zdrojový kód Javy do XML reprezentace, a následně pomocí skriptu

XML reprezentaci převést do jazyka C# nebo C++. Při pokusu o vyzkoušení programu na jednom z nejjednodušších příkladů z předchozích let soutěže, však vygeneroval úplně nesmyslný kód podobající se Javě, plný syntaktických chyb a to jak u překladu do C# tak do C++.

4.9 Souhrn

Nepodařilo se nalézt nástroj, který by zvládal konverze do všech požadovaných cílových jazyků. Ačkoliv na internetu byly nalezeny zmínky o existujícím konvertoru Javy do Pascalu, odkazy na tento projekt již nefungují. Nicméně i pokud by se našly nástroje pro jednotlivé konverze každé dvojice jazyků, lze na základě nalezených existujících nástrojů usuzovat, že by tyto nástroje nejspíš prováděly konverze tím způsobem, že by do cílového kódu vložily všechny potřebné knihovny i nativní funkce Javy přeložené do daného jazyka. Tím by nejspíš bylo dosaženo kódu se stejnou funkcionalitou, ale byly by ignorovány potenciální problémy (různé zpracování standardního vstupu, jiná funkcionalita bitových posunů, různé funkce na porovnávání řetězců), které je třeba překladem odhalit. Zároveň by takto vygenerovaný kód byl velmi rozsáhlý a pravděpodobně ne moc dobře čitelný.

Je tedy potřeba vytvořit nástroj, který by dokázal provést všechny požadované konverze jazyků a zároveň by upozornil uživatele o potenciálních problémech spojených s překladem konkrétního programu. Také by tento nástroj používal klasické nativní funkce jazyka, tak jak by to dělal programátor daného jazyka a ne použitím jejich doslovného nízkoúrovňového překladu.

4.9.1 Souhrn nedostatků existujícího softwaru

Pro přehlednost jsou zde shrunuty nalezené nedostatky existujícího softwaru.

- žádný z nalezených nástrojů neumí všechny požadované jazyky
- nebyl nalezen nástroj na konverzi Javy do Pascalu
- některé nalezené nástroje nejsou schopné přeložit některé knihovní třídy nebo metody
- nástroje typicky generují několik souborů (je požadován 1)
- nástroje nejsou schopné upozornit na potenciální problémy
- nástroje obchází standardní funkce jazyků vlastní implementací

5 Analýza řešení

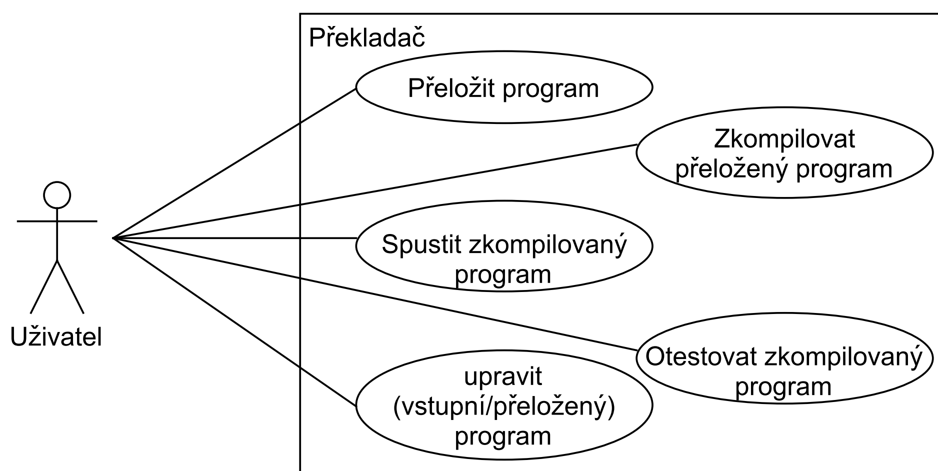
5.1 Požadavky na řešení

Je požadována desktopová aplikace napsaná v Javě s grafickým uživatelským rozhraním. Tato aplikace by měla být schopná zparsovat zdrojový kód Javy, který je dodán v jednom souboru a tento soubor se zdrojovým kódem přeložit do jazyků: C, C++ a Free Pascal opět jako jediný soubor se zdrojovým kódem. Takto vygenerovaný zdrojový kód musí být zkompilovatelný kompilátory na validačním serveru soutěže (pro C je to gcc 4.9.2, pro C++ dodržet standard C++11 (g++) a pro Free Pascal fpc 2.6.4). Tento zkompilovaný program by měl pro testovací vstupy generovat totožné výstupy jako překládaný vstupní kód Javy.

Program by si měl poradit s konstrukcemi, třídami a metodami běžně používanými v úlohách soutěže PilsProg. Jedná se zejména o matematické funkce, zpracování standardního vstupu, metody pro standardní výstup, metody pro práci s řetězci a základní kolekce.

5.2 Případy užití

Na Obrázku 5.1 je znázorněn diagram případů užití.



Obrázek 5.1: Diagram případů užití programu

Popis diagramu:

- Přeložit program – uživatel může přeložit vstupní program Javy do cílových jazyků C, C++, Free Pascal a C#.
- Zkompilovat přeložený program – uživatel může zkompilovat předem přeložený program a vygenerovat tak spustitelný soubor.
- Spustit zkompilovaný program – uživatel může předem zkompilovaný program spustit, a ověřit si tak jeho funkčnost.
- Otestovat zkompilovaný program – uživatel může z programu spustit otestování předem zkompilovaného programu a ověřit tak jeho funkčnost.
- upravit (vstupní/přeložený) program – uživatel může upravit zdrojový kód jak vstupního programu, tak vygenerovaného překladu.

5.3 Vlastní parsování vs. knihovna

Na zpracování zdrojového kódu Javy již existují nástroje, tyto nástroje by jednak ušetřili práci s psaním vlastního parseru, ale zároveň by mohly omezovat. Je třeba provést analýzu toho, co v je v parseru potřebné a toho co dostupné nástroje nabízí a vhodně se rozhodnout.

5.3.1 Java Parser

Java parser [31] je open source knihovna napsaná v jazyce Java, která umí automaticky ze zdrojového kódu Javy vygenerovat stromovou strukturu objektů (objektovou reprezentaci zdrojového kódu). Ačkoliv je Java Parser funkční, má z hlediska této práce nedostatky (viz dále).

5.3.2 Požadované funkce

Pro účely překladu zdrojového kódu je potřeba možnosti parseru a množinu generovaných objektů rozšířit, což by znamenalo komplikované modifikace parsovací knihovny. Příklad požadovaných odlišných funkcí:

- možnost nahradit generický typ v kolekcích za konkrétní
- reference na `catch` blok u volání metod, které mohou vyhodit výjimku
- reference příkazu `continue` na `for` cyklus, ve kterém se nachází

- reference na nahrazené knihovní metody
- odlišná reprezentace některých výrazů

Modifikace open source knihovny je sice možná, ale protože nejsem jejím autorem a nevyznám se v ní, tak by byla poměrně náročná. Proto by bylo vhodnější napsat si vlastní parser s objekty vytvořenými na míru potřebám překladu do požadovaných jazyků. Zároveň by byl parser schopen upozornit na nepodporované konstrukce ještě před překladem.

Také bude možno při tvorbě parseru se detailně seznámit se všemi konstrukcemi Javy, které se budou překládat, čímž se zmenší riziko, že by byl nějaký detail opomenut.

5.4 Vymezení podmnožiny Javy

Jazyk, ze kterého se překládá, je třeba omezit, aby byl překlad jednodušší a vůbec zvládnutelný. Omezit je třeba komplikované konstrukce, které by se obtížně překládali do jiných jazyků a které obecně nejsou potřebné k vyřešení úloh PilsProgu.

5.4.1 Objektově orientované programování

Z objektově orientovaného programování se odebere dědičnost, polymorsifmus a všechno s tím spojené. Důvodem je především jazyk C, který objekty a dědičnost vůbec nemá.

5.4.2 Lambda výrazy

Lambda výrazy jsou obecně komplikované konstrukce, které nejsou ve všech cílových jazycích podporovány, zároveň nejsou pro úlohy PilsProgu potřebné, proto se také odeberou.

5.4.3 Výjimky

Výjimky jsou občas potřebné, proto je nelze úplně odebrat, ale je možné je omezit pro zjednodušení. Důvody jsou jazyk C, který výjimky vůbec nemá a také fakt, že výjimky v Javě fungují odlišně než v jiných jazycích. Také je problém v tom, že jazyk Java vyhazuje výjimky i v situacích, kdy jiné jazyky ne.

Výjimky budou tedy omezené tak, že budou odchyceny pouze ty výjimky, které jsou ručně vyhozené příkazem `throw` a ty, které vyhazuje cílový jazyk

implicitně. `Try-catch` bloky budou odchytávat vždy obecnou výjimku, tudíž nelze reagovat na různé výjimky různým způsobem. Příkazy na výpis zprávy výjimky či stavu zásobníku se přeloží na generický výpis zprávy. Posledním problémem jsou runtime výjimky, které v Javě nemusí být přímo obklopeny `try-catch` blokem či deklarovat metodu s klíčovým slovem `throws`. V takovém případě je obtížné výjimku propagovat nahoru v přeloženém jazyku, proto může být program rovnou ukončen namísto propagace výjimky, která by eventuálně mohla být odchycena nadřazenou volající metodou. Z těchto důvodů není doporučené výjimky používat, pokud to není nezbytně nutné.

5.4.4 Kolekce

Generické kolekce jsou problém pro jazyky, které je neumí, a pokud je umí, nemusí se chovat stejným způsobem. Základní kolekce jsou nicméně potřebné. Možným řešením je vzít implementaci kolekce z Javy, vložit do ní metody implementované v supertypech a odebrat z ní externí závislosti, tak aby byla třída schopná fungovat samostatně. Pokud je tato kolekce potřebná při překladu, vloží se tato předzpracovaná třída do projektu s tím, že se nahradí generický typ za konkrétní typ, se kterým byla kolekce vytvořena. Tento způsob řešení sebou ale nese určitá omezení. Kolekci není možné vytvořit pro pole, ale pouze pro objekty, kterými ale mohou být i kolekce.

5.4.5 Regulární výrazy

Regulární výrazy jsou komplikované a v úlohách PilsProgu nepotřebné, protože jsou vstupy navržené tak, aby byly snadno parsovatelné. Nicméně některé standardní funkce Javy očekávají jako argument regulární výraz. Tyto funkce je možné omezit tak, že budou zpracovávat jen prostý řetězec, který by měl stačit, protože se typicky používají jen na dělení řetězců podle mezery.

5.4.6 Char

Typ `char` je v jazyce Java 16-ti bitový a je možné do něj zapsat unicode znaky. Typ `char` je ale v jazycích C a C++ pouze 8 bitový. Bylo by sice teoreticky možné reprezentovat `char` jazyka Java typem `short`, ale přineslo by to spoustu komplikací, proto se předpokládá práce pouze s ASCII znaky, pro které by měl překlad fungovat bez problému. Zároveň by se neměl `char` v jazyce Java používat na aritmetické operace, které by ovlivnily jeho horních 8 bitů.

5.4.7 Pole polí

Ačkoliv je možné pole různě dlouhých polí ve všech jazycích bez větších problémů implementovat, kvůli zjednodušení překladu a také z důvodu, že se tato konstrukce používá jen velmi zřídka nebude tato konstrukce podporována. Lze však stejného výsledku dosáhnout použitím pole `ArrayListů`.

5.4.8 Příkaz goto

Ačkoliv je tento příkaz podporován ve všech jazycích a neměl by být problém jej přeložit, mohl by při překladu způsobovat potíže a proto nebude v podmnožině jazyka povolen.

5.4.9 Enum

Výčtový typ `enum` nebude podporován, protože nebyl v žádné úloze z předešlých 5-ti let použit a lze se bez něj obejít. Je proto vynechán z důvodů usnadnění práce. Zda by byl překlad tohoto typu problematický nebo jednoduchý nebylo prozkoumáno.

5.4.10 Desetinný oddělovač

Ve výchozím nastavení jazyk Java používá jako desetinný oddělovač podle Locale z operačního systému, tedy u nás je to většinou desetinná čárka. Jazyky C a C++ však používají desetinnou tečku, ačkoliv lze v každém jazyku desetinný oddělovač změnit, bude při překladu oddělovač u všech jazyků nastavený na desetinnou tečku. Pokud je tedy potřeba číst desetinná čísla v Javě, je možné použít metodu `Scanner.setLocale(Locale.US)`, která nastaví desetinný oddělovač při čtení vstupu na desetinnou tečku. Druhou možností je použít metodu `Locale.setDefault(Locale.US)`, která nastaví oddělovač na desetinnou tečku, jak pro čtení vstupu, tak i pro výstup. Použití obou metod nebude mít vliv na překlad programu, přeložené programy budou vždy používat desetinnou tečku.

5.4.11 Anotace

Anotace nejsou podporovány v žádném cílovém jazyku, proto nebudou podporovány ani překladačem. V překládaném kódu se mohou vyskytovat, parser je však bude ignorovat.

5.5 Návrh řešení

5.5.1 Grafické rozhraní

Grafické rozhraní bude vytvořené na platformě JavaFX, protože s touto technologií mám již zkušenosti a jedná se o moderní dobře vypadající rozhraní. Dalším důvodem je, že alternativní grafické rozhraní `swing` bude touto technologií v budoucnu nahrazeno [32].

5.5.2 Parsování a překlad

Parsování zdrojového kódu bude provedeno vlastní implementací, z důvodů zmíněných v Kapitole 5.3. K tomuto parsování bude použit rekurzivní sestup, protože je to snadno pochopitelný a robustní algoritmus, který vede na čitelný a dobře dekomponovaný zdrojový kód. Překlad kódu je možné oddělit od objektové reprezentace kódu pomocí návrhového vzoru návštěvník.

5.5.3 Struktura aplikace

Projekt bude vyvíjen jako Maven projekt, protože s ním mám zkušenosti a je možné do něj snadno přidat případné užitečné knihovny. Jádro aplikace by bylo možné rozdělit do 5-ti balíků: grafické rozhraní, parser, objektová reprezentace zparsovaného kódu, překladač a data. Při vývoji však zřejmě nastane potřeba další balíky přidat.

6 Implementace

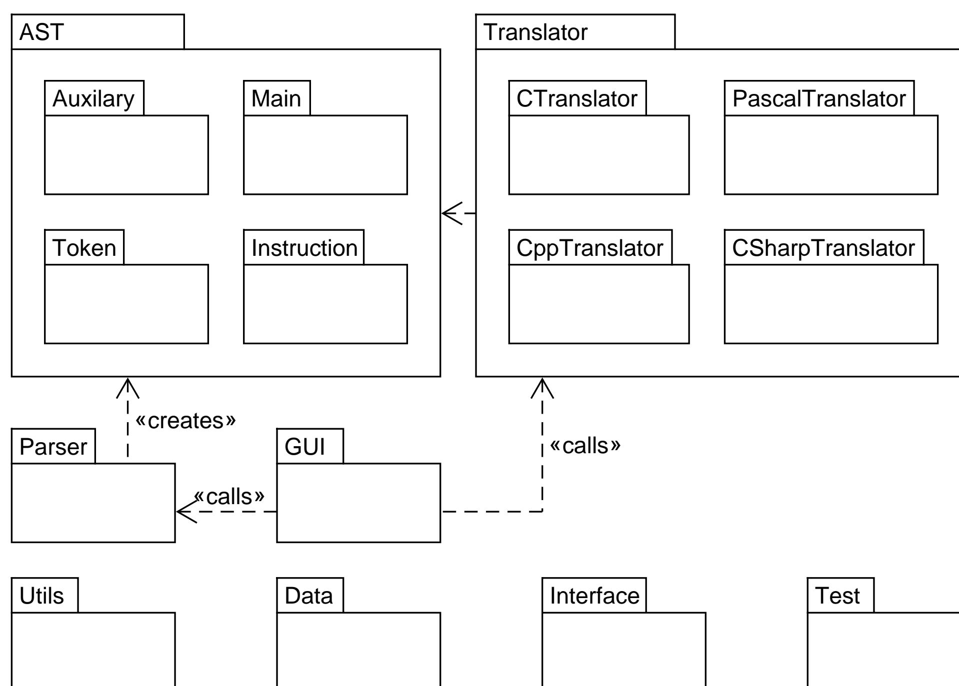
6.1 Použité prostředky

Výsledný program byl implementován v jazyce Java 1.8. Grafické rozhraní bylo vytvořeno technologií JavaFX. Na zobrazení zdrojového kódu v grafickém rozhraní byla použita knihovna `RichTextFX` [33]. Na analýzu výsledků testů (porovnání výstupu programu s očekávaným) byla použita knihovna `Java diff utils` [34]. Celý projekt byl vyvíjen jako Maven projekt ve vývojovém prostředí IntelliJ IDEA.

6.2 Struktura aplikace

6.2.1 Architektura

Aplikace je architektonicky rozdělená do 16-ti balíků znázorněných na Obrázku 6.1.



Obrázek 6.1: Balíky projektu

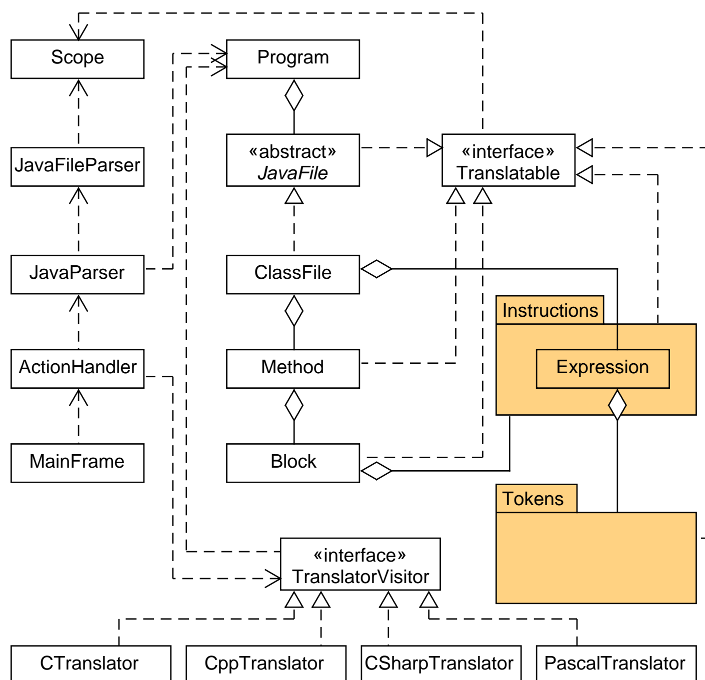
- AST(Abstract syntax tree) – Obsahuje zparsovaný zdrojový kód
- Main – Obsahuje hlavní objekty zparsovaného programu (program, třídy, metody, bloky, ...)
- Instruction – Obsahuje objekty reprezentující jednotlivé příkazy zparsovaného programu (if, while, for, ...)
- Token – Obsahuje jednotlivé elementy výrazů (operandy a operátory)
- Auxilary – Obsahuje pomocné třídy AST (datový typ, identifikátor, modifikátor, ...)
- Translator – Obsahuje balíky, které se starají o překlad programu (AST) do jednotlivých jazyků.
- CTranslator, CppTranslator, PascalTranslator, CSharpTranslator – Obsahují třídy vykonávající překlad do jednotlivých jazyků.
- Parser – Obsahuje logiku parseru zdrojového kódu Javy
- GUI – Obsahuje grafické rozhraní aplikace
- Utils – Obsahuje pomocné třídy (např. třída spouštěč programů a testů)
- Data – Obsahuje třídy, které čtou a zapisují soubory na disk
- Interface – Obsahuje všechny rozhraní aplikace
- Test – Obsahuje testy aplikace

6.2.2 UML diagram tříd

Kompletní UML diagram tříd lze najít na přiloženém CD. Zjednodušený diagram tříd je na Obrázku 6.2.

Popis významných tříd

- MainFrame – hlavní okno aplikace
- ActionHandler – třída obsluhující implementace akcí (stisk tlačítek) vyvolaných grafickým rozhraním
- JavaParser – Třída která předzpracuje projekt ke zparsování a řídí parsování projektu

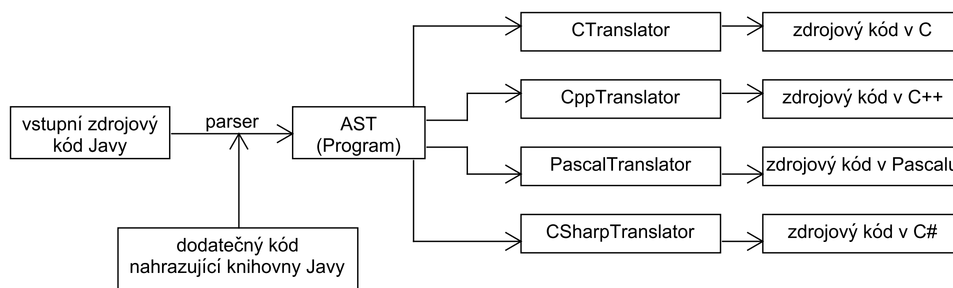


Obrázek 6.2: Zjednodušený UML diagram tříd

- JavaFileParser – Třída, která parsuje jednotlivé třídy
- Scope – tabulka symbolů
- Program – objekt zparsovaného programu
- ClassFile – objekt reprezentující zparsované třídy Javy
- Method – objekt reprezentující zparsovanou metodu
- Block – objekt reprezentující zprasovaný blok příkazů
- Expression – objekt reprezentující zprasovaný výraz či deklaraci proměnné
- Translatable – rozhraní reprezentující přeložitelný zparsovaný objekt
- TranslatorVisitor – rozhraní reprezentující překladač kódu (podle návrhového vzoru návštěvník)
- CTranslator, CppTranslator, PascalTranslator, CSharpTranslator – implementace rozhraní TranslatorVisitor. Provádí překlad do jednotlivých jazyků.

6.2.3 Datový tok

Poté co uživatel otevře zdrojový kód Javy, je tento kód automaticky zparsován spolu s dodatečnými potřebnými třídami, které nahrazují knihovní třídy Javy. Parser jako výsledek vrátí objekt typu `Program`, který obsahuje objektovou reprezentaci zparsovaného programu. Poté co se uživatel rozhodne provést překlad, je objekt `Program` přeložen do požadovaného jazyka a výsledný přeložený program je uložený na disk do pracovní složky nastavené uživatelem. Datový tok je znázorněn na Obrázku 6.3.



Obrázek 6.3: diaram průtoku dat aplikací

6.2.4 Soubory

Hlavní složka s programem obsahuje:

- Složka `Collections` – obsahuje upravené implementace kolekcí v Javě, tyto kolekce jsou v případě potřeby přidané k překládanému projektu a jsou s tímto projektem přeloženy do požadovaného jazyka.
- Složka `JavaClasses` – obsahuje upravené implementace některých knihovních tříd Javy, tyto třídy jsou v případě potřeby přidané k překládanému projektu a jsou s tímto projektem přeloženy do požadovaného jazyka.
- Složka `TranslatedMethods` – obsahuje implementace některých nahrazených metod v nativním kódu jazyka do kterého se překládá. Také obsahuje implementaci třídy `Scanner` v jazyce C#.
- Složka `testCases` – obsahuje programy napsané v Javě spolu se vstupy a očekávanými výstupy, používané pro testy.

- Soubor `JavaMethods.java` – obsahuje implementaci některých nahrazených metod v Javě. Tyto metody jsou v případě potřeby přidané k překládanému projektu a jsou s tímto projektem přeloženy do požadovaného jazyka.
- Soubor `replace.xml` – obsahuje seznam všech nahrazovaných metod spolu s informacemi, jak je nahradit. Tento soubor je používán programem při nahrazování metod.
- Soubor `settings.ini` – obsahuje nastavení programu
- Soubor `translator.jar` – spustitelný program překladače

6.3 Parsování Javy

6.3.1 Předzpracování

Vstupním souborem bývá vždy jen jeden soubor se zdrojovým kódem, který typicky obsahuje více tříd, proto se v předzpracování soubor rozdělí na jednotlivé třídy, které jsou následně parsovány zvlášť.

Komentáře mohou obsahovat kód Javy, který by se mohl při parsování zpracovat jako výkonný kód, také mohou být umístěné kdekoliv v kódu, což by značně komplikovalo proces parsování. Proto jsou obsahy komentářů nahrazeny pouze indexem do kolekce, kde je uložen jejich obsah. Zároveň jsou odstraněny komentáře na nevhodných místech, pro zjednodušení procesu parsování.

6.3.2 Parsování

Parsování kódu Javy je prováděno rekurzivním sestupem. Klíčová slova, identifikátory, hodnoty a jiné elementy jsou hledány pomocí regulárních výrazů. Výsledkem je objektová reprezentace parsované třídy.

6.4 Tabulka symbolů

6.4.1 Implementace

Tabulka symbolů byla implementována pomocí spojového seznamu. Položka tohoto seznamu má v sobě uložený identifikátor a `id` bloku, ve kterém byla vytvořena. Během parsování kódu se při vstupu do bloku zavolá metoda `enterBlock()`, která vytvoří v seznamu nový záznam s navýšeným `id` bloku

a identifikátorem `null`, který zabrání odstranění předchozího bloku, pokud v tom současném nebylo nic deklarováno. Při vytvoření nového identifikátoru se do seznamu vloží záznam se současným `id` bloku a zároveň se kontroluje, zda se již v seznamu nenachází identifikátor se stejným jménem. Pokud v seznamu totožný identifikátor existuje, ve smyčce je inkrementováno číslo, které se dodá za přidávaný identifikátor, dokud není unikátnost jména zaručena. Při vyvolání metody `exitBlock()` jsou ze seznamu vyjmuty všechny identifikátory se současným `id` bloku a `id` je zmenšeno o 1.

6.4.2 Využití při parsování

Pokaždé, když se při parsování narazí na deklaraci identifikátoru, je pro něj vytvořen záznam v tabulce symbolů, pokud se tento symbol objeví opakovaně, je na něj nalezena reference v tabulce symbolů, čímž se zjistí, o jakou proměnnou se jedná. Zároveň se tato tabulka symbolů stará o to, aby nenastala kolize. Tudíž, pokud nastane deklarace nového symbolu, který již je v tabulce symbolů, změní se jeho jméno v cílovém jazyku (Ve validním kódu Javy sice kolize být nemohou, ale například jazyk Pascal není case sensitive, proto ke kolizím docházet může).

6.4.3 Využití při překladu

Tabulku symbolů lze využít i při překladu, aby bylo možné vytvořit pomocné proměnné, které v původním kódu nebyly.

Dalším využitím je možnost sestavit deklarační část metody v jazyce Pascal. Jednoduše se provede překlad metody s tím, že se uvnitř metod nevytváří ani neukončují bloky a po překladu se z tabulky symbolů snadno zjistí seznam všech deklarovaných proměnných, ze kterých se sestaví deklarační část metody.

Posledním využitím je uvolňování paměti, které je popsané v Kapitole 6.8, kde je tabulka symbolů využita k získání seznamu proměnných, které je potřeba uvolnit.

6.5 Zpracování výrazů

Zpracování výrazů lze rozdělit na tři fáze – zpracování unárních operátorů, převod do postfixové notace a tvorby stromu.

6.5.1 Zpracování unárních operátorů

Algoritmus na převod do postfixové notace neumí pracovat s unárními operátory, proto je potřeba je nejdříve seskupit s jejich operandy a zacházet s nimi jako s jedním celkem. Prochází se výraz po jednotlivých elementech a vytváří se skupina, dokud se nenarazí na binární operátor, následně se ze skupiny vlevo od nalezeného binárního operátoru vytvoří stromová struktura objektů `UnaryOperation`. Po nalezení posledního binárního operátoru se samozřejmě musí vytvořit stromová struktura objektů `UnaryOperation` i z pravé strany binární operace. Tímto se odstraní všechny unární operátory z výrazu. Zároveň se v této fázi odstraní závorky, tím, že se jejich obsah nahradí samostatným výrazem (objektem `EnclosedExpression`).

```
for(Token t: expression){
  if(t.equals("(")){
    processed.add(new EnclosedExpression(extractBrackets()));
  }else if(t instanceof Operand){
    operandFound = true;
    group.add(t);    // přidá operand do skupiny
  }else{
    if(t.isPostUnary() || !operandFound)
      group.add(t); // přidá unární operáotr do skupiny
    else{
      // vytvoří ze skupiny jeden operand
      processed.add(new Operand(group));
      processed.add(t);         // přidá binární operátor
      operandFound = false;
      group.clear();
    }
  }
}
processed.add(new Operand(group));   // přidá pravou stranu
```

Výpis 6.1: Algoritmus zpracování unárních operátorů v pseudokódu

6.5.2 Převod do postfixové notace

Kromě použití standardního algoritmu na převod výrazu do postfixové notace je nutné se vypořádat ještě s ternárním operátorem. Pokud při průchodu výrazu je nalezen na operátor `?`, musí se najít začátek výrazu, určující podmínku ternárního operátoru. Poté se projde výraz dokud se nenarazí na odpovídající operátor `:`, který ukončuje pozitivní větev ternárního operátoru. Nesmí se však zapomenout na to, že by mohly být do sebe vnořené dva a více ternárních operátorů, proto to nutně nemusí být ten první nalezený. Nakonec je třeba najít konec negativní větve ternárního operátoru a ze

všech tří částí vytvořit instanci objektu `TernaryOperator` a následně s ní pracovat jako s jedním operandem.

Protože jazyk Pascal neumí operátory `+=, -=, *=,` atd., je potřeba výraz rozložit při jejich použití. Bylo by to sice možné až při překladu, ale vzhledem k tomu, že Pascal neumí ani vícenásobné přiřazení, tak by to působilo komplikace a je jednodušší tento operátor rozložit již v předzpracování výrazu. Pokud je tedy nalezen operátor tohoto typu, je do postfixové notace vložen výraz již v rozloženém tvaru.

6.5.3 Tvorba stromu

Kromě standardního postupu, jak vytvořit stromovou strukturu z postfixové notace je nutné se vypořádat s několika problémy. Při rozkladu operátorů `*=, /=,` atd., by mohlo dojít k problémům s prioritou operací, proto je třeba pravou část výrazu uzávorkovat. Z výrazu $x* = 5 + 3$ proto vznikne $x = x * (5 + 3)$.

Druhým problémem je zřetězování řetězců, které je problematické přeložit do jazyků C a C++. Proto je vhodné místo stromu binárních operací zřetězení vytvořit jeden objekt, kde budou všechny operandy účastnící se zřetězení. Obdobným způsobem je reprezentováno i vícenásobné přiřazení ($x = y = 5$), z důvodu jednoduššího překladu. Výsledkem tvorby stromu je tedy stromová struktura unárních, binárních, ternárních a řetězových operací. Každá z těchto operací je reprezentována objektem obsahující operandy vstupující do příslušné operace.

6.6 Nahrazování knihovních metod

Volané knihovní funkce Javy je třeba nahradit ekvivalentními v každém jazyku, do kterého se překládá. Ne vždy ale daný jazyk ekvivalentní metodu obsahuje, proto je třeba navrhnout více způsobů jak volání metody nahradit.

6.6.1 Přímé nahrazení

První možností, jak volání nahradit, je použít přímo knihovní funkci cílového jazyka. Toto je však možné jen tehdy, jestliže cílový jazyk obsahuje ekvivalentní metodu se stejnými argumenty.

6.6.2 Přidání kódu v Javě

Druhou možností, jak volání nahradit, je vzít metodu implementovanou v Javě, přiložit ji k překládanému kódu a přeložit ji stejně jako zbytek programu. Toto je možné pouze tehdy, nevolá-li metoda jinou nenahrazenou metodu.

6.6.3 Přidání přeloženého kódu

Poslední možností je implementovat metodu ve všech cílových jazycích a přiložit ji k již přeloženému kódu. Tato možnost však znamená implementovat metodu pro každý cílový jazyk zvlášť, což je podstatně náročnější a tím se zvyšuje riziko chyby.

6.6.4 Seznam nahrazovaných metod

Pro řešení tohoto problému byl zaveden soubor ve formátu XML, který obsahuje seznam všech knihovních metod a konstant Javy, které budou při překladu nahrazeny. Z tohoto souboru program pozná, zda volání metody má být nahrazeno a zároveň i čím. Z každého záznamu tohoto souboru je vytvořen objekt, který obsahuje všechny potřebné informace pro práci s tímto nahrazeným výrazem i pro jeho překlad. Pomocí tohoto souboru je možné snadno rozšířit množinu podporovaných knihovních metod Javy. Podrobná dokumentace ke všem značkám a atributům se nachází v Příloze A.6.

Přímé nahrazení

Ve Výpisu 6.2 je příklad pro přímé nahrazení metody `Math.sin(x)`.

```
1  <item name="Math.sin(" type="double" method="true" throws="false"
2           javamethod="false" volatile="false">
3    <arguments>
4      <argument type="double" />
5    </arguments>
6    <c replacement="sin(" ending=")" caller="no">
7      <imports>
8        <import imp="#include &lt;math.h&gt;" />
9      </imports>
10   </c>
11    ...
12 </item>
```

Výpis 6.2: Ukázka záznamu nahrazované metody

Nahrazení implementací v Javě

Pokud je potřeba provést nahrazení vlastní implementací metody v Javě, je to možné provést následovně. V hlavičce se nastaví atribut `javamethod` na `true` a vloží se značka `annotation` s unikátní anotací, která identifikuje metodu v souboru `JavaMethods.java`. Při překladu se tělo této metody přiloží k překládanému zdrojovému kódu a přeloží se spolu s ním. Ukázka záznamu nahrazované metody s implementací v Javě je ve Výpisu 6.3.

```
<item name="String.hashCode(" type="int" method="true"
  throws="false" javamethod="true" volatile="false">
  <arguments>
    <caller />
  </arguments>
  <annotation anot="@StringHash" />
</item>
```

Výpis 6.3: Ukázka záznamu nahrazované metody s implementací v Javě

Nahrazení implementací v nativním kódu

Pokud je potřeba nahradit metodu vlastní implementací, ale v nativním kódu, lze to udělat obdobně jako přímo s tím, že se přidá atribut `anot` a do něj unikátní anotace viz Výpis 6.4, která ukazuje na začátek metody v souborech `methods.<koncovka pro jazyk>`. Nemusí se takto nahradit metoda ve všech jazycích, jen v těch, ve kterých to nejde přímo. Atribut `replacement` musí obsahovat jméno metody tak, jak je v souboru `methods.<koncovka pro jazyk>`, aby překladač věděl, jak má metodu zavolat.

```
<item name="String.trim(" type="String" method="true"
  throws="false" javamethod="false" volatile="false">
  <arguments>
  </arguments>
  <c replacement="trim(" ending=")" caller="arg" anot="@trim">
    <imports>
      <import imp="#include &lt;ctype.h&gt;" />
    </imports>
  </c>
  ...
</item>
```

Výpis 6.4: Ukázka záznamu nahrazované metody s vlastní implementací v nativním kódu

Výpis 6.5 ukazuje obsah souboru `methods.cpp`. Anotace `@trim` ukazuje na začátek nahrazované metody, anotace `@end` na její konec. Tyto anotace slouží

k nalezení potřebné metody v souboru a snadnému zkopírování do výstupního souboru při překladu. Takto napsaná metoda smí uvnitř volat pouze nativní knihovní metody přístupné s importy ve značce `imports` konkrétního jazyka nebo metody výše definované mezi značkami `@trim` a `@end`. Pokud metoda vrací, nebo zjišťuje délku pole z argumentů, musí použít systém ukládání délky pole na index -1 popsaný výše a zároveň při vytváření pole vytvořit místo pro počítadlo referencí.

```
@trim
char *trim(char* input)
{
  char *start = input;
  while (isspace(*start) && *start != 0) start++;
  char *output = (char*)malloc(strlen(start) + 1);
  strncpy(output, start, strlen(start) + 1);
  char *end = output + strlen(output) - 1;
  while(end > output && isspace(*end)) end--;
  *(end+1) = 0;
  return output;
}@end
```

Výpis 6.5: Ukázka nativního kódu meotdy v souboru methods.cpp

6.6.5 Vlastní nahrazení

Ve fázi vyhodnocování výrazů se prochází jednotlivé volání metod a kontroluje se zda se signatura metody nachází v seznamu nahrazovaných metod. Pokud se zde vyskytuje, nastaví se reference volání metody na objekt vytvořený ze předem zmíněného seznamu nahrazovaných metod. Pokud se metoda v seznamu nenachází, hledá se v metodách, které byly zparsovány. Pokud není metoda nalezena ani tam, je vyhozeno chybové hlášení, které informuje uživatele o nepodporované knihovní metodě.

6.7 Substituce

V některých případech je potřeba namísto přímého zavolání metody uvnitř výrazu metodu vyhodnotit předem, její výsledek uložit do proměnné a následně použít uvnitř tohoto výrazu tuto proměnnou namísto volání metody. Důvodů substituce je však více. Proto metody rozhraní `Translatable` mají atribut `preceding` viz Výpis 6.7, což je seznam řetězců, do kterého se vloží příkazy, které patří před překládanou konstrukci. Tudíž v seznamu `preceding` se bude nacházet deklarace dočasné proměnné a přiřazení hodnoty z volané metody a návratová hodnota metody rozhraní `Translatable`

pouze vrátí jméno dočasné proměnné. Při překladu jednotlivých příkazů je tedy nutné nejdříve vyprázdnit seznam `preceding` a až poté vložit výsledek překladu příkazu.

```
if(foo()){
  doSomething();
}
```

```
bool tmp = foo();    // v preceding seznamu
if(tmp){        // tmp je návratová hodnota překladu výrazu foo()
  doSomething();
}
```

Výpis 6.6: Ukázka efektu substituce

```
res = ifStatement.toC(preceding);
for(String statement: preceding) {
  translation += statement;
}
translation += res;
```

Výpis 6.7: Překlad příkladu z Výpisu 6.6

6.8 Uvolňování paměti

Jak bylo již dříve řečeno, jazyky C a C++ neuvolňují paměť automaticky, proto by bylo vhodné příkazy na uvolňování paměti do výstupního kódu doplnit. Vzhledem k tomu, že toto je velmi komplexní problém, nepokoušel jsem se o kompletní uvolnění veškeré nepotřebné paměti, ale jen té, u které je možné jednoduchou analýzou kódu zaručit, že již nebude potřebná. Kód na uvolnění paměti je doplněn před každý konec bloku či příkaz `return`, kde je potřeba nějakou paměť uvolnit. Správně by měla být paměť uvolněna i před vyhozením výjimky či příkazy `break` a `continue`, ale tyto případy nebyly ošetřeny.

6.8.1 Objekty

Pro samotné uvolnění objektu z paměti bylo nutné vygenerovat metodu, která provede uvolnění ne-primitivních atributů.

U objektů bylo implementováno počítadlo referencí. Toto počítadlo se při každém přiřazení objektu zvýší a při každém ukončení bloku či přepsáním jinou hodnotou sníží. Pokud je po snížení počítadlo rovné 0, je objekt uvolněn z paměti.

Počítadlo referencí se nejlépe implementuje přetížením operátoru =, což způsobí to, že se před každým přiřazením provede dodatečný kód, který může například upravit počty referencí. Jazyk C však přetížit operátory neumí, tudíž bylo nutné každou operaci přiřazení objektů nahradit voláním metody viz Výpis 6.8.

```
Object* _assign_Object(Object* right, Object** left){
  if(right != NULL)right->_refCnt++;
  if(*left != NULL) _free_Object(*left);
  *left = right;
  return *left;
}
```

Výpis 6.8: Ukázka přiřazovací metody.

Použití metody místo operátoru však sebou přináší určité omezení. Aby bylo možné uvnitř metody provést samotné přiřazení, musí být předán ukazatel na ukazatel, tedy použít operátor &, ten ale nelze použít na nic jiného než proměnnou. Pokud by tedy bylo prováděno vícenásobné přiřazení (x = y = z), pak by bylo potřebné vytvořit ukazatel na ukazatel z výsledku metody přiřazení, což není možné. V tomto případě je tedy potřeba výraz rozložit a přeložit jej jako dva samostatné příkazy přiřazení. Dalším omezením je, že není možné tuto metodu použít při přiřazení u deklarace (Object *o = x;), proto místo toho, aby se výraz rozložil na deklaraci s přiřazením NULL a samotné přiřazení objektu, byla zavedena metoda _incRef_Object(Object *o), která provede pouze inkrementaci referencí u pravého operandu.

6.8.2 Pole

U polí bylo využito obdobného mechanismu jako na uložení délky pole, tudíž bylo na index -2 uloženo počítadlo referencí. Toto počítadlo se při každém přiřazení pole zvýší a při každém ukončení bloku či přepsáním jinou hodnotou sníží. Pokud je po snížení počítadlo rovné 0, je pole uvolněno z paměti. Z tohoto důvodu je pro každý typ pole, použitý v programu vygenerována metoda pro přiřazení tohoto pole, která kromě samotného přiřazení upravuje počítadla referencí a provádí případné uvolnění pole z paměti.

6.8.3 Řetězce

Vzhledem k tomu, že Java vytváří kopie řetězců při každém přiřazení, bylo toto chování přeneseno i do cílových jazyků. Z tohoto důvodu je možné

každý nově vytvořený řetězec, který byl přiřazen do lokální proměnné na konci bloku uvolnit, protože každé přiřazení tohoto řetězce vytvořilo kopii.

6.8.4 Neuvolněná paměť

Uvolňování paměti není stoprocentní, tudíž existuje několik případů, které vedou k tzv. „memory leakům", neboli nepotřebné paměti, která není uvolněna až do ukončení programu.

- Cyklické vazby – Pokud objekty ukazují sami na sebe v cyklu, paměť není uvolněna, protože počítadlo referencí nikdy neklesne na 0.
- Nepřiřazení – Pokud je alokována paměť, která není nikam přiřazena, ale je jen například předána jako argument metody, paměť není uvolněna, protože argumenty metod nejsou kvůli zjednodušení uvolňovány.

6.9 Překlad tříd

6.9.1 C a C++

Přestože jazyk C++ umí třídy, z důvodů znovupoužití a zjednodušení překladu byl použit totožný překlad tříd s překladem do C. Statické proměnné jsou přeloženy jako globální proměnné s prefixem jména třídy. Atributy tříd jsou přeložené jako struktury pojmenované podle třídy. Metody jsou přeložené s prefixem třídy, do které patří, a nestatické metody mají přidaný jako první argument strukturu, nad kterou je metoda volaná. Každá metoda celého programu je deklarovaná po definicích struktur, aby bylo možné je zavolat odkudkoli programu a nemuselo být řešené pořadí těchto metod. Po deklaraci metod jsou doplněné definice metod nahrazených nativním kódem, které nemusejí být deklarované, protože jsou vždy na začátku programu a nemohou se navzájem volat. Následují definice přeložených metod původních tříd programu a na konci programu jsou definované metody tříd dodaných do programu (např. kolekcí). Struktura vygenerovaných souborů je na Obrázku 6.4.

6.9.2 Free Pascal

Po deklaraci všech tříd a všech použitých typů polí jsou jednotlivé třídy definovány. Po definici tříd jsou deklarovány statické proměnné s prefixem jména třídy. Následují nativní nahrazené metody a za nimi definice jednotlivých metod tříd, nejprve těch původních, pak těch doplněných (např. kolekcí).

deklarace struktur
definice struktur
deklarace metod a statických proměnných
definice metod nahrazených nativním kódem
definice metod překládaného kódu
definice dodaných metod

Obrázek 6.4: Struktura vygenerovaných souborů v C a C++

Protože obdoba metody `main` v Pascalu musí být až na konci zdrojového kódu, následuje až za všemi doplněnými metodami. Struktura vygenerovaných souborů je na Obrázku 6.5.

deklarace tříd a typů polí
definice tříd
deklarace statických proměnných
definice metod nahrazených nativním kódem
definice metod překládaného kódu
definice dodaných metod
hlavní program (main metoda)

Obrázek 6.5: Struktura vygenerovaných souborů ve Free Pascalu

6.9.3 C#

Překlad tříd je v jazyce C# téměř totožný s původním kódem Javy. Za zmínění stojí akorát nahrazené nativní metody, které jsou uvnitř uměle vytvořené třídy `RequiredMethods`. Doplněné třídy (např. kolekce) a třída `Scanner` jsou vždy na konci zdrojového kódu.

6.10 Specifické problémy a jejich řešení

6.10.1 Substituce ve výrazech

Důvodů k provedení substituce ve výrazech je celá řada. Mezi ty nejdůležitější patří nedefinované pořadí vyhodnocování argumentů metod a nemožnost přiřazení uvnitř logických výrazů Pascalu. V některých případech však tato substituce může vést k vyhodnocení příkazů v nesprávném pořadí či k vyhodnocení výrazu, který se vyhodnotit vůbec neměl.

```
// puvodni kod
if((n = foo()) >= 0 && foo2(foo3(n))) doSomething();
// s provedenou substituci
tmp = foo3(n);
if((n = foo()) >= 0 && foo2(tmp)) doSomething();
```

Výpis 6.9: Příklad problémové substituce.

Jak je možné vidět na příkladu ve Výpisu 6.9, došlo ke substituci argumentu `foo3(n)`, protože se jedná o argument, který může změnit stav programu. Proto je nad argumentem preventivně provedena substituce, která zabrání problému s nedefinovaným pořadím vyhodnocení argumentů. Tato metoda však používá jako argument proměnnou `n`, která je vyhodnocená až v následujícím výrazu, způsobující potenciální chybu.

Zároveň se může stát to, že metoda `foo()` vrátí zápornou hodnotu, což by v původním kódu způsobilo nevyhodnocení druhé části výrazu, ale protože byla provedena substituce, je argument `foo3(n)` vyhodnocen nezávisle na první části výrazu.

Řešením je logický výraz spojený operátory `&&` nebo `||` rozdělit na více příkazů `if` jak je ukázáno ve Výpisech 6.10 a 6.11. Dělění výrazu není nutné provádět vždy, ale jen pokud nastala substituce v pravém operandu.

```
bool cond = false;
if((n = foo()) >= 0 ){
  tmp = foo3(n);
  if(foo2(tmp)){
    cond = true;
  }
}
if(cond) doSomething();
```

Výpis 6.10: Příklad rozkladu operace &&.

```
bool cond = false;
if((n = foo()) >= 0 ){
  cond = true;
}else{
  tmp = foo3(n);
  if(foo2(tmp)){
    cond = true;
  }
}
if(cond) doSomething();
```

Výpis 6.11: Příklad rozkladu operace ||.

K obdobnému problému však může dojít i ve výrazech bez logických spojek, pokud se ve výrazu nachází na sobě závislé operace viz Výpis 6.12.

```
int result = array[n = 5] + foo(foo2(n));
// po substituci
int tmp = foo2(n);
int result = array[n = 5] + foo(tmp);
```

Výpis 6.12: Příklad problémové substituce

K tomuto problému dochází tehdy, je-li v pravém operandu provedena substituce a levý operand by mohl změnit stav programu. Pokud toto nastane, lze tento problém vyřešit provedením substituce i na levém operandu, čímž se zachová správné pořadí operací viz Výpis 6.13.

```
int tmp = array[n = 5];
int tmp2 = foo2(n);
int result = tmp + foo(tmp2);
```

Výpis 6.13: Řešení problémové substituce

Z těchto důvodů může dojít k poměrně častým rozkladům výrazů, vedoucím k zhoršení čitelnosti přeloženého programu, ale to je pořád lepší, než omezovat programátora v tom, jaké smí a nesmí psát výrazy.

6.10.2 Volání konstruktoru this

V jazyce Java je možné v konstruktoru, zavolat konstrukcí `this()` jiný konstruktor té samé třídy, který nevytváří další instanci, nýbrž jen provádí určitou inicializaci té samé instance viz Výpis 6.14. Pokud tuto konstrukci cílový jazyk nepodporuje, nebo má nějaké omezení oproti Javě, je třeba přijít s jiným řešením. Tento problém byl vyřešen vložením inicializačního kódu volaného konstruktoru namísto volání `this()`. Před samotným vložením je však potřeba vyhodnotit případné argumenty volání a místo nich si pouze

deklarovat proměnné stejného typu viz Výpis 6.15.

```
public Object(int x){
  this.x = x;
}

public Object(int x, int y){
  this(x + 1);

  this.y = y;
}
```

Výpis 6.14: Volání this() v Javě

```
public Object(int x){
  this.x = x;
}

public Object(int x, int y){
  int x2 = x + 1;
  this.x = x2;
  this.y = y;
}
```

Výpis 6.15: Překlad volání this()

6.10.3 Konverze typu výsledného výrazu

Pokud je potřeba provést typovou konverzi uvnitř výrazu (např. `Integer` na `int`), lze to zjistit při jeho analýze. Pokud však ke konverzi dochází při předání výrazu jako argument, nebo v příkazu `return`, tak to není možné. Proto byl u objektu výraz zaveden atribut `expectedType`, který se nastaví na očekávaný typ výrazu (např. typ argumentu volané metody). Po překladu výrazu se provede kontrola zda je potřeba typ převést a pokud ano provede se potřebná konverze. Obdobný systém je také využit při volání metody `System.out.print()` s argumentem typu objekt, kterému se nastaví očekávaný typ na `String` a při překladu se provede konverze zavoláním metody `toString()`.

6.11 Nahrazování kolekcí

V konfiguračním souboru `settings.ini` jsou popsány třídy, které mají být nahrazeny a jakými třídami mají být nahrazeny. Zde jsou popsány obdobným způsobem i kolekce. Typicky je popsán i rodičovský typ kolekce, který se vždy přepíše na jeho potomka (např. `List` → `ArrayList`, `ArrayList` → `ArrayList`, `LinkedList` → `ArrayList`). Pokud je některá z těchto kolekcí použita v parsovaném kódu, je náhrada této kolekce zařazena do fronty ke zparsování. Jméno této kolekce je upraveno tak, že jsou typy elementů připojeny za jméno kolekce (např. `ArrayList<Float>` se stane `ArrayListFloat`). Zdrojové kódy těchto upravených kolekcí mají místo generických typů použity šablonu ve tvaru `#TYPE#` nebo `#TYPEX#`, kde X je pořadové číslo generického typu (např. `HashMap<String, Float>` `String` = `#TYPE0#` a `Float` = `#TYPE1#`). Tyto generické typy jsou již při parsování automaticky nahrazeny za ty konkrétní, se kterými byla kolekce vytvořena.

Pro zdrojové kódy těchto kolekcí platí několik pravidel. Nesmí být závislé na jiné nenahrazené třídě či metodě a nesmí obsahovat konstrukce mimo předem vymezenou podmnožinu jazyka Java. Aby bylo možné tuto kolekci proiterovat konstrukcí `for(Element e: kolekce)`, musí tato kolekce implementovat metodu `size()` na zjištění počtu elementů a metodu `get(int)` na získání prvku z kolekce na požadovaném indexu (u kolekce `Set` byla tato metoda vytvořena, je však neefektivní a doporučuje se použít metodu `toArray()` a iterovat přes pole místo setu).

6.12 Nahrazování tříd

Obdobným způsobem jako u kolekcí je prováděno nahrazování tříd. V konfiguračním souboru `settings.ini` je seznam tříd, které mají být nahrazeny. Jména některých tříd jsou odlišná kvůli kolizi s jinými datovými typy v cílových jazycích (např. typ `Integer`). Pokud je některá z těchto nahrazených tříd v parsovaném zdrojovém kódu použita, je z konfiguračního souboru zjištěno, jakou třídou má být tento typ nahrazen a soubor této upravené třídy se vloží do fronty ke zparsování. Až se na tuto třídu dostane řada, je zparsována a přidána k zpracovávanému programu. Pro tyto třídy platí obdobná pravidla jako pro kolekce, tudíž nesmí být závislé na jiné nenahrazené třídě či metodě a nesmí obsahovat konstrukce mimo předem vymezenou podmnožinu jazyka Java.

6.13 Zmenšení výstupního kódu

Během testování aplikace vedoucím práce bylo zjištěno omezení validátoru na maximální velikost odevzdávaného kódu na 50kB. Protože při použití kolekcí v překládaném kódu se kopíruje celá třída kolekce do výsledného kódu a to pro každý typ, se kterým byla kolekce použita zvlášť, může výsledný kód poměrně dost narůst. Proto byla implementována analýza kódu, která projde všechny výrazy v deklaracích atributů, statických proměnných a metodě `main`, a v každém tomto výrazu vyhledá všechny volané metody, které vloží do zásobníku. Pro každou metodu v zásobníku pak provede obdobou analýzu pro tělo této volané metody a označí tuto metodu jako použitou. Použité metody se znovu neanalyzují, aby nedošlo k zacyklení. Pomocí této analýzy se zjistí, která metoda je v kódu nepoužitá a nemusí se proto překládat, což vede ke kratšímu výstupnímu kódu. Pro ještě větší zkrácení je také možné v nastavení programu vypnout překládání komentářů nebo uvolňování paměti.

6.14 Grafické rozhraní

Pro jednodušší práci s programem, tak i pro jednodušší testování programu při vývoji, bylo vytvořeno grafické rozhraní programu. Grafické rozhraní bylo vyvinuté na platformě JavaFX. V tomto grafickém rozhraní je možné otevřít zdrojový kód Javy, přeložit jej do všech podporovaných jazyků, přeložený kód zkompilovat, spustit, otestovat i upravit. Grafické rozhraní se skládá ze tří oken:

- MainFrame – Hlavní okno, umožňuje otevřít zdrojový kód Javy, překlad, kompilaci, testování a spouštění zkompilovaného programu.
- CodeFrame – Okno s textovým editorem, ve kterém je možné upravit zdrojové kódy jak Javy, tak i všech překladů.
- OptionsFrame – Okno s nastavením programu, je možné v něm nastavit překlad a cesty k pracovní složce a kompilátorům.

6.14.1 Hlavní okno

Hlavní okno má v dolní části konzoli pro každou činnost zvlášť (parsování, překlad, kompilace, spuštění a testování) a je mezi nimi možné libovolně přepínat. Spustit zkompilovaný program je možné dvěma způsoby: poskytnutím souboru, jehož obsah bude přesměrován jako standardní vstup nebo zadat standardní vstup přímo do textového pole a spustit program s ním. Tlačítko `test`, provádí test tím způsobem, že se podívá do složky, ze které byl otevřen zdrojový kód Javy a hledá v ní textové soubory (.txt) se vstupy (obsahující ve jméně „_in-“) a očekávanými výstupy (obsahující ve jméně „_out-“). Program očekává, že ke každému vstupu existuje ekvivalentně pojmenovaný výstup a nedělá další kontroly. Posléze program spustí nad každým nalezeným vstupním souborem zkompilovaný program a testuje shodu s očekávaným výstupem. V případě neshody je vypsána do konzole příslušná chyba.

6.14.2 Textový editor

Textový editor v okně `CodeFrame` používá knihovnu `RichTextFX`, která dokáže mimo jiné zvýraznit syntaxi jednotlivých jazyků, jsou-li ji dodány regulární výrazy jazykových konstrukcí, které se mají zvýraznit. Styl zvýraznění je možné nastavit přes CSS soubor. Okno `CodeFrame` má dvě různá rozložení, které je možné přepínat: zobrazení zdrojového kódu jednoho jazyka přes celé okno a zobrazení jazyka Java vedle jednoho z cílových jazyků.

7 Testování

Aplikaci tohoto typu je třeba velmi důkladně otestovat. Proto byla aplikace otestována několika různými způsoby.

7.1 Problém s unit testy

Jednotlivé metody použité v parsování a překladu zdrojového kódu je velmi obtížné testovat pomocí unit testů. Důvodem je komplikované ověřování stromových struktur objektů vzniklých při parsování zdrojového kódu. Obdobně kontrola, zda vznikl při překladu očekávaný kód, by sice nebyla problematická, nicméně by vyžadovala konstrukci složité struktury objektů reprezentující kód, který se má přeložit. Ověřováním výsledného kódu na rovnost s tím očekávaným také není ideální, protože není cílem testovat, zda byl vygenerován kód v nějakém tvaru, ale to, zda se výsledný program chová stejně (dává stejné výstupy pro stejné vstupy) jako překládaný kód.

7.2 Testování pomocí programů

Vzhledem k tomu, že unit testy nejsou vhodné na testování tohoto překladače, je potřeba vymyslet lepší způsob. Nabízí se proto vytvořit množinu programů napsaných v Javě, kde každý z těchto programů může být zaměřený na určitou oblast překladu. K těmto programům se vytvoří množina vstupů a k nim odpovídajících očekávaných výstupů. Pak lze napsat jeden unit test, který postupně vezme každý z těchto programů, provede nad nimi překlad do každého jazyka, zkompiluje je a spustí s množinou připravených vstupů a porovnává jejich výstup s tím očekávaným. Tento způsob sice netestuje funkčnost konkrétní metody, nýbrž celého překládacího systému, ale testuje přesně to, co chceme, což je stejná funkcionalita přeložených programů. Pokud jsou navíc programy zaměřené na určitou programovací konstrukci, snadno ukáží, kde je v programu problém.

Vzhledem k tomu, že jsou k dispozici úlohy PilsProgu z předchozích let a k nim množina vstupů a očekávaných výstupů, lze obdobným způsobem otestovat program i nad nimi.

7.3 Implementace testů pomocí programů

Jako první se načte seznam všech programů, které se budou testů účastnit. Následně se spustí parametrizovaný test nad každým prvkem tohoto seznamu pro každý překládaný jazyk. Pro každý takto testovaný program se vytvoří instance objektu `TestCase`, který obsahuje cestu ke zdrojovému kódu programu, který se má překládat, a seznam cest k souborům se vstupy a očekávanými výstupy. Následně se nad tímto objektem zavolá překlad, kompilace, v cyklu spuštění nad jednotlivými vstupy a porovnání jejich výstupů. Pro jednodušší analýzu testů, které neprojdou, je nad výstupem programu a jeho očekávaným výstupem provedena funkce `diff`, která lépe zobrazí tu část výstupů, která se liší. Funkce `diff` byla získána z knihovny `Java diff utils`. Pokud jakákoliv z těchto částí selže, je vyhozena výjimka, která způsobí nesplnění testu a Výpis v jaké části testu chyba nastala. V případě, že chyba nastala v porovnání výstupů, je vypsán rozdíl těchto výstupů a zároveň celý výstup i ten očekávaný.

7.4 Problémy v testovacích datech

Některé programy z minulých ročníků soutěže, které byly k dispozici měly chyby v očekávaných výstupech. Chybami se rozumí to, že výstup, který poskytovala implementace v Javě, nebyl totožný s očekávaným výstupem v přiložených souborech. Nejednalo se však o špatné výpočty, nýbrž jen odlišné odsazení či absence nové řádky na konci výstupu. Protože však cílem bylo kontrolovat výstupy na úplnou shodu, byly tyto výstupy upravené tak, aby byly totožné s výstupem implementace v Javě.

Pět programů obsahovalo konstrukce či metody, které nejsou v předem vymezené podmnožině jazyka podporovány, a proto byly z testovací množiny vyloučeny.

7.5 Výsledné statistiky testů programů

Celkem bylo výše zmíněným způsobem otestováno 130 programů, z toho 93 jsou programy z předešlých ročníků soutěže a 37 jsou uměle vytvořené testy. Z úloh předešlých ročníků soutěže selhal jeden program, kvůli odlišným formátům Výpisů reálného čísla (nebyla použita formátovací metoda a v Pascalu došlo k zaokrouhlovací chybě). Z uměle vytvořených testů selhalo 8 s tím, že neselhaly vždy všechny jazyky. Tyto testy selhaly z důvodů popsaných v Kapitole 8, většinou se však jednalo o zaokrouhlovací chyby či

chyby formátování reálných čísel. Všechny tyto testy trvalo vykonat necelých 8 minut. Všechny testy jsou přiložené na CD ve složce `testCases` a je možné je spustit z testovací třídy `ProgramTests`. Následuje seznam testů, které selhaly.

- Velehory (C, C++, Free Pascal) – chyba formátování reálných čísel, zaokrouhlovací chyba
- bitops (všechny jazyky) – odlišná funkcionalita bitových posunů nad zápornými čísly
- math (všechny jazyky) – chyba formátování reálných čísel, zaokrouhlovací chyba
- parsing (C#) – zaokrouhlovací chyba
- scanner (Free Pascal, C#) – zaokrouhlovací chyba, v Pascalu je problém s metodou `Scanner.next()`
- string-arrays (Free Pascal) – neschopnost rozeznat pole nastavené na `null` od pole nulové délky.
- string-split (všechny jazyky) – odlišný výsledek při dělení prázdného řetězce podle mezery
- sysoutformat (všechny jazyky) – nefunkcionalita některých kombinací formátování
- sysoutprint (C, C++, Free Pascal) – chyba formátování reálných čísel

7.6 Unit testy

Jak již bylo zmíněno, unit testy parsování a překladu jsou nevhodné, ale v programu existuje řada jiných podpůrných metod, které tímto způsobem otestovat lze. Otestovány tímto způsobem byly třídy `Type`, `Scope` a `BinaryOperation`, protože jsou to jediné třídy s metodami dostatečně komplexními, aby se vyplatilo je testovat a všechny ostatní jsou již závislé na parsování, překladu kódu nebo grafickém rozhraní. Celkem bylo takových testů vytvořeno 46 a jsou přiloženy na CD. Těmito testy byly odhaleny některé chyby, které byly následně opraveny.

7.7 Pokrytí kódu testy

Po provedení všech programových testů i unit testů metod bylo pokryto 77% řádků kódu, pokud se pomine grafické rozhraní a třídy využívané pouze grafickým rozhraním, je to 89%. Většina nepokrytého kódu jsou debugovací metody (`toString()` metody u prvků AST) a chybové stavy jako například `catch` bloky.

7.8 Rychlosti výpočtů po překladu

Byly provedeny testy rychlosti výpočtu tří programů, aby bylo zjištěno, zda přeložené programy nepočítají déle než původní implementace v Javě viz Tabulka 7.1. Časy byly měřené příkazem `Measure-Command` v PowerShellu a zahrnují čas zavedení programu do paměti, jeho spuštění a případně i spuštění JVM. Programy byly zkompilovány se zapnutými optimalizacemi.

Tabulka 7.1: Časy výpočtů programů v různých jazycích

program	Java	C	C++	Pascal	C#
kodovani	183ms	11ms	12ms	34ms	51ms
seznam slov	357ms	280ms	288ms	328ms	272ms
zabezpeceni	5570ms	5690ms	5476ms	16021ms	8066ms

Z výsledků testů je možné vidět, že se můžou časy program od programu velmi lišit. V některých případech je po překladu výpočet rychlejší a v některých výrazně pomalejší. Zejména jazyk Pascal se zdá být problémovým, co se týká efektivity.

7.8.1 Benchmarky Pascalu

Protože z provedených testů výkonu překladu bylo zjevné, že má Pascal výkonnostní problémy, byla provedena řada benchmarků, na zjištění problémových operací. Bylo zjištěno, že je problém způsoben neefektivní prací s maticemi v Pascalu. Tato informace byla zjištěna po provedení uměle vytvořeného benchmarku viz Výpis 7.1, který si vytvoří matici a následně k jejím prvkům několikrát přistupuje. Tento benchmark po 5-ti násobném opakování konzistentně trval v Javě 3 sekundy a v Pascalu 20 sekund. Je nutné dodat, že překlad nebyl proveden optimálně ze dvou důvodů: `for` cykly jsou přeložené pomocí `while` cyklů a je použito dynamické pole, i když by bylo vhodnější použít statické. Proto byl změřen čas i s ideálním překladem a ten

trval 18,5 sekund, což je o něco lepší čas, nicméně je pořád 6× pomalejší než Java.

```java
int [][] array = new int [10000][10000];
long l = 0;
for(int i = 0; i < 10000; i++)
{
  for(int j = 0; j < 10000; j++){
    array[i][j] = i + j;
  }
}
for(int k = 0; k < 20; k++){
  for(int i = 1; i < 9999; i++)
  {
    for(int j = 1; j < 9999; j++)
    {
      l += array[i-1][j-1];
      l -= array[i+1][j];
      l += array[i+1][j+1];
      l -= array[i][j];
    }
  }
}
System.out.println(l);
```

Výpis 7.1: Benchmark testující výkon velkých matic

Pro ověření byly provedeny další bechmarky s obdobným kódem, jejich výsledky jsou v Tabulce 7.2. Ačkoliv není úplně jasné co zpomalení způsobuje, z provedených testů se lze usoudit, že by to mohla být špatná správa cache nebo neschopnost kompilátoru tento kód vhodně optimalizovat.

Tabulka 7.2: Srovnání všech provedených benchmarků s maticemi

typ pole	přístup	opakování	čas Java	čas Pascal
matice [10000][10000]	[i][j]	20	3026 ms	19931 ms
matice [10000][10000]	[j][i]	20	47940 ms	24648 ms
matice [5][5]	[i][j]	222222222	3780 ms	19481 ms
pole [10000 * 10000]	[i]	20	2978 ms	4067 ms
pole [10000 * 10000]	[i * 10000 + j]	20	2993 ms	20631 ms
pole [5 * 5]	[i * 5 + j]	222222222	5276 ms	18654 ms

Další problémovou oblastí se zdá být práce se 64-bitovými `integery`. Byl proveden benchmark viz Výpis 7.2, který v Javě trval přibližně 1,5 sekundy a v Pascalu s ideálním překladem 11,5 sekund. Pro ověření byl vykonán

totožný benchmark, kde byl vyměněn 64-bitový `integer` za 32-bitový, který byl výkonostně srovnatelný s Javou.

```
long l = 0;
for(int k = 0; k < 100_000_000; k++){
  for(int i = 1; i < 19; i++){
    l = l + i;
    l = l - k;
    l = l + 2;
  }
}
System.out.println(l);
```

Výpis 7.2: Benchmark testující výkon 64-bitových integerů

Oba benchmarky byly testovány na 2 různých počítačích (Windows 10, 64-bit notebook s Intel® Core™ i7-3610QM a eryx.zcu.cz) s obdobnými výsledky. Pascal byl překládán kompilátorem fpc 2.6.4. se zapnutými optimalizacemi (-O3). U obou problémům bylo provedeno hledání vysvětlení na internetu, ale neúspěšně.

7.9 Uvolňování paměti

Pro otestování uvolňování paměti byla vzata úloha `Zabezpeceni` soutěže PilsProg, která bez uvolňování paměti na pátém vstupu zabere kolem 3,7 GB paměti. Po zapnutí uvolňování paměti úloha zabrala přibližně 739 MB. Využití paměti bylo změřeno pomocí správce úloh.

7.10 Čitelnost

Vzhledem k tomu, že jsou přeložené programy jen otestovány, zda generují stejný vstup, není čitelnost vygenerovaného kódu prioritou. Nicméně, může nastat případ, že kvůli chybě v překladu bude potřeba zásah do vygenerovaného přeloženého kódu a dobrá čitelnost by v tomto případě pomohla.

U jazyků C a C++ je čitelnost kódu zhoršena o doplněné uvolňování paměti, a neobvyklou práci s poli. V Pascalu zhoršuje čitelnost kódu nahrazení `for` cyklů za `while` cykly. Dále čitelnost přeloženého kódu zhoršují substituce, a úpravy kódu vzniklé kvůli nim. Zároveň může vygenerovaný kód výrazně narůst o přidaný kód knihovních metod, tříd nebo kolekcí. Pokud ale tyto věci pomineme, je vygenerovaný kód značně podobný tomu původnímu napsanému v Javě.

7.11 Časová náročnost

Byl proveden test rychlosti parsování a překladu na rozsáhlejším programu. Tento program měl 518 řádek, obsahoval 5 původních tříd a během parsování mu byly dodány 2 další třídy a 2 kolekce. Parsování tohoto programu trvalo v průměru 363ms a jeho překlad do C (což je jazyk s nenáročnějším překladem) se zapnutým uvolňováním paměti trval v průměru 37ms. Vzhledem k tomu, že se jedná o jednorázovou činnost, jsou tyto časy velmi uspokojivé a dají se považovat za zanedbatelné.

8 Nevyřešné problémy

8.1 Bitové posuny

8.1.1 Operátor >>

V jazyce Pascal se operátor >> chová jako operátor >>> v Javě. Tudíž, pokud je použit na záporném čísle, dojde k jinému výsledku. Například v Javě $-1 >> 1 = -1$ a v Pascalu $-1 >> 1 = 2147483647$.

8.1.2 Operátor >>>

Operátor >>> v jazycích C, C++ a C# neexistuje, je proto překládán jako operátor >>, který dává jiné výsledky na záporných číslech. Například v Javě $-1 >>> 1 = 2147483647$ a v C $-1 >> 1 = -1$.

8.2 Nahrazené metody

8.2.1 Matematické funkce

Matematické funkce s reálnými čísly mohou dospět k mírně odlišným výsledkům s určitou zaokrouhlovací chybou. Dalším problémem je způsob vypořádání s nevhodnými argumenty. V některých jazycích je vrácena hodnota `NaN` (not a number), případně `Infinity` a v některých jazycích je vyhozena výjimka. Je tedy potřeba se ujistit, že argumenty metod jsou správné a počítat s tím, že výsledek se může s drobnou přesností lišit.

8.2.2 Tisknutí reálných čísel

Metoda `System.out.print/ln()` se nedoporučuje na tisknutí reálných čísel, protože každý jazyk je může vypsat v různém formátu. Doporučuje se tedy použít metoda `System.out.format()`, kde je možné požadovaný formát nadefinovat. Opět však může nastat zaokrouhlovací chyba vedoucí k mírně odlišným výsledkům.

Obdobně je tomu tak při řetězení reálných čísel, nedoporučuje se pouze zřetězit proměnnou s reálným číslem, ale použít metodu `String.format()`, v ní si naformátovat reálné číslo a výsledek této metody zřetězit.

8.2.3 Metoda `Scanner.next()`

Nebyla překládána třída `Scanner` z jazyka Java, místo toho byly překládány samostatné metody pro čtení vstupů různých typů. Proto v jazyce Pascal byla ekvivalentní metoda implementována pomocí čtení jednotlivých znaků. Tento fakt způsobuje to, že je přečten i bílý znak, ukončující slovo, které má být přečteno, což může způsobit problém při čtení dalšího vstupu. Použití této metody se tedy nedoporučuje, je lepší použít metodu na přečtení celé řádky a tu následně rozdělit dle potřeby.

8.3 Finally blok

Finally blok v jazyce Java může následovat za `try-catch` blokem a je vykonán vždy po ukončení tohoto `try-catch` bloku a to ať už je ukončený jakýmkoliv způsobem. Těchto způsobů ukončení je překvapivě mnoho (normální ukončení, vyhození vyjímky, příkaz `return`, příkaz `break`, příkaz `continue`, neodchycená výjimka). Řešením tohoto problému by bylo zkopírovat obsah `finally` bloku před každé ukončení `try-catch` bloku, což by zároveň vyžadovalo netriviální analýzu kódu, zda například příkazy `break` nebo `continue` skutečně `try-catch` blok ukončí. Vzhledem k tomu, že hlavní využití `finally bloku` je uzavírání streamů, souborů nebo připojení, které úlohy PilsProgu nepotřebují, není `finally` blok přeložen ideálním způsobem, ale je přeložen pouze jako blok příkazů následující `try-catch` blok.

8.4 Metody `HashCode` a `compareTo`

Metody `hashCode` a `compareTo` tříd `Float` a `Double` používají v Javě metodu JVM, proto byly tyto metody upraveny a nemusí se chovat stejně. Metoda `hashCode` generuje jiný hash a metoda `compareTo` může mít problémy s porovnáváním hodnot: 0, -0, NaN, Infinity, -Infinity.

8.5 Prázdné pole

Dynamická pole v jazyce Pascal nedokáží rozlišit pole nulové délky a pole nastavené na `null`. Tudíž je možné aby výraz $arr == null$ byl pravdivý, i když se jedná o pole délky 0. Pokud je zavolána metoda `Arrays.toString()` s polem nulové délky, metoda vrátí řetězec "null" namísto řetězce "[ ]".

8.6 Kolekce

Kromě omezení, které neumožňuje vytvářet kolekci polí, nastávají i jiné problémy s kolekcemi. Protože již samotné zmínění kolekce v parsovaném kódu automaticky přidá požadovanou kolekci do fronty ke zparsování, může samotné zmínění typu kolekce uvnitř jiné znamenat nekonečnou smyčku přidávání nových kolekcí ke zparsování. Z tohoto důvodu nebylo možné aby implementace metod `keySet(), values(), entrySet` kolekce `Map` vracely kolekce typu `Set`. Aby je však bylo možné použít, byly implementace těchto metod změněny tak, aby vracely pole. Proto nesmí v překládaném kódu nastat přiřazení návratových hodnot těchto metod do proměnné typu `Set`. Aby s těmito metodami však bylo možné vůbec něco udělat, je možné je projít pomocí příkazu `for`, jak je ukázáno ve Výpisu 8.1. Případně je možné s nimi pracovat jako s polem, což může vést na nevalidní kód v Javě, ale překladači by to nevadilo.

```
1 for(Object o: map.keySet())
2 {
3   doSomething(o);
4 }
```

Výpis 8.1: Průchod setu z mapy

Z obdobného důvodu a zároveň kvůli nepodpoře dědičnosti byly implementace metod `addAll(), containsAll(), removeAll(), retainAll()` upraveny tak, že pracují pouze s kolekcí stejného typu, nad jakou metodu voláme.

8.7 Uvolňování paměti

Kromě případů, ve kterých paměť uvolněná není a dochází k tzv. „memory leakům“, nastává ještě jiný problém. Pokud by byl uvolňován z paměti například spojový seznam, který by byl velmi dlouhý, způsobilo by to rekurzivní volání metody uvolňující prvek tohoto seznamu pro každý prvek tohoto seznamu, což by mohlo vést k přetečení zásobníku. Proto je potřeba se buďto vyhnout používání velmi dlouhých spojových struktur, nebo je možné uvolňování paměti vypnout v nastavení programu.

8.8 Formátování

Ačkoliv jsou metody `System.out.format()` a `String.format()` podporovány, nebylo dost dobře možné přeložit veškerou jejich funkcionalitu. Ex-

plicitní zobrazení znaménka před číslem i když je kladné nefunguje, protože není v Pascalu podporováno. Dále nefungují některé kombinace formátování, jako je například zobrazení nul před reálným číslem, které má naformátováno počet desetinných míst. Takovýchto kombinací existuje zřejmě více, ale všechny nebyly otestovány.

8.9 Dělení řetězce

Metoda `String.split()` a zároveň i třída `StringTokenizer`, která tuto metodu využívá, nepoužívá na dělení řetězce regulární výraz, ale jen prostý řetězec. Z tohoto důvodu se může v některých situacích chovat odlišně. Při testování této metody došlo k odlišným výsledkům při dělení prázdného řetězce podle mezery.

9 Závěr

Cílem této práce bylo vytvořit program, který by byl schopný přeložit zdrojový kód napsaný v omezené podmnožině jazyka Java do jazyků C, C++ a Pascal tak, aby tento překlad používal standardní knihovní funkce těchto jazyků, jako kdyby je programoval programátor těchto jazyků.

Výsledný software byl podle požadavků implementován v jazyce Java a zadání splňuje. Je schopný přes jednoduché grafické rozhraní otevřít zdrojový kód programu napsaného v jazyce Java a postupně jej přeložit do všech požadovaných jazyků. Dále byl nad rámec zadání implementován i překlad do jazyka C#, který by se mohl v budoucnu stát dalším z povolených jazyků soutěže PilsProg. Zároveň implementovaný software dokáže přímo z grafického rozhraní program i zkompilovat, spustit a otestovat. Pokud by při parsování či překladu nastala chyba, bude vypsána do konzole v grafickém uživatelském rozhraní, a je možné tuto chybu i z grafického rozhraní opravit v jednoduchém textovém editoru.

Díky vhodné implementaci je možné vytvořený software i snadno rozšířit o další knihovní metody či celé třídy včetně kolekcí jazyka Java.

Vzhledem k tomu, že je vyvinutý software schopný přeložit všechny úlohy z předešlých 5-ti let soutěže PilsProg se zdá, že řešení je poměrně robustní a kvalitní. Pokud budou dodržována stanovená omezení podmnožiny jazyka Java, program by měl být schopný přeložit budoucí úlohy soutěže PilsProg a výrazně tak usnadnit zadavatelům úloh práci.

9.1 Budoucí práce

Ačkoliv se zdá, že je vymezená podmnožina jazyka Java pro účely soutěže PilsProg dostačující, je možné, že v budoucnu nastane potřeba práci rozšířit. Bylo by žádoucí vyřešit nevyřešené problémy popsané v Kapitole 8. Dále by bylo možné rozšířit podmnožinu jazyka Java o další jazykové konstrukce, které nejsou současně podporované. Další možností je program rozšířit o další programovací jazyk.

Literatura

[1] Pilsprog. https://pilsprog.fav.zcu.cz/index.php. Accessed: 2016-12-11.

[2] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[3] Defining high, mid and low-level languages. http://www.codecommit.com/blog/java/defining-high-mid-and-low-level-languages. Accessed: 2016-12-11.

[4] Source code definition. http://www.linfo.org/source_code.html. Accessed: 2016-12-11.

[5] Parsing. https://www.thoughtco.com/parsing-grammar-term-1691583. Accessed: 2016-12-11.

[6] Charles L Hamblin. Translation to and from polish notation. *The Computer Journal*, 5(3), 1962.

[7] Torben Agidius Mogensen. *Basics of Compiler Design.* lulu.com, 2010.

[8] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE software*, 25(5), 2008.

[9] Niklaus Wirth, Niklaus Wirth, Niklaus Wirth, Suisse Informaticien, and Niklaus Wirth. *Compiler construction*, volume 1. Citeseer, 1996.

[10] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction.* Springer, 2002.

[11] A garbage collector for c and c++. https://www.hboehm.info/gc/. Accessed: 2017-01-18.

[12] Lili Qiu. Programming language translation. Technical report, Cornell University, 1999.

[13] Ron Rockhold James Peterson, Glenn Downing. Translating java programs into c++. Technical report, IBM Austin Research Laboratory Austin, Texas, 1998.

[14] Argument lists are evaluated left-to-right. http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.7.4. Accessed: 2016-07-11.

[15] Order of evaluation. http://en.cppreference.com/w/cpp/language/eval_order. Accessed: 2016-07-11.

[16] Order of parameter evaluation. http://wiki.freepascal.org/Code_Conversion_Guide#Order_of_parameter_evaluation. Accessed: 2016-07-11.

[17] Initial values of variables. http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.12.5. Accessed: 2016-07-11.

[18] Metoda string.format. https://msdn.microsoft.com/cs-cz/library/system.string.format(v=vs.110).aspx. Accessed: 2016-07-11.

[19] Primitive data types. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html. Accessed: 2016-07-11.

[20] Fundamental types. http://en.cppreference.com/w/cpp/language/types. Accessed: 2016-07-11.

[21] Object as a superclass. https://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html. Accessed: 2016-07-11.

[22] When initialization occurs. http://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html#jls-12.4.1. Accessed: 2016-07-11.

[23] Java2c. http://www.vishia.org/Java2C/. Accessed: 2016-07-10.

[24] J2c. https://bitbucket.org/arnetheduck/j2c/overview. Accessed: 2016-07-10.

[25] Java to c++ converter. http://www.tangiblesoftwaresolutions.com/product_details/java_to_cplusplus_converter_details.html. Accessed: 2016-07-10.

[26] Java to c# converter. http://www.tangiblesoftwaresolutions.com/product_details/java_to_csharp_converter.html. Accessed: 2016-07-10.

[27] Sharpen. https://github.com/mono/sharpen. Accessed: 2016-07-10.

[28] Haxe. http://haxe.org/manual/introduction-what-is-haxe.html. Accessed: 2016-07-11.

[29] Varycode. https://www.varycode.com/. Accessed: 2016-07-11.

[30] Xes. http://www.euclideanspace.com/software/language/xes/userGuide/index.htm. Accessed: 2016-07-11.

[31] Java parser. https://github.com/javaparser/javaparser. Accessed: 2016-07-11.

[32] Is javafx replacing swing as the new client ui library for java se? http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6. Accessed: 2016-08-10.

[33] Richtextfx. https://github.com/TomasMikula/RichTextFX. Accessed: 2016-08-10.

[34] java-diff-utils. https://github.com/is/java-diff-utils. Accessed: 2016-08-10.

Přílohy

A Uživatelská dokumentace

A.1 Sestavení

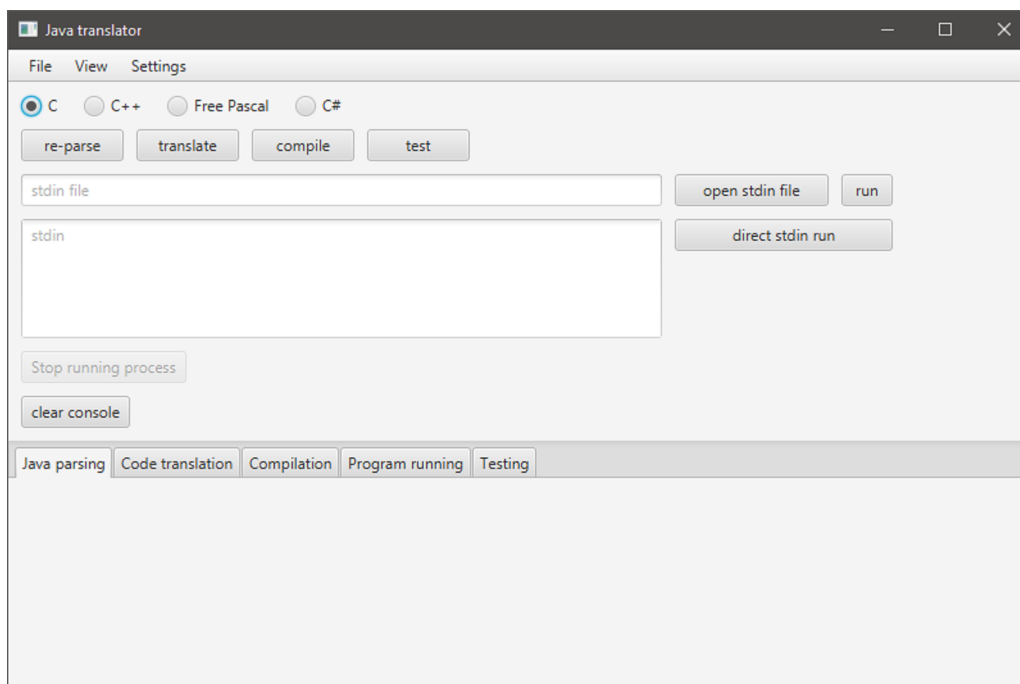
Jedná se o Maven projekt, tudíž je možné program zkompilovat a sestavit `jar` soubor příkazem `mvn clean install -DskipTests` ze složky, kde se nachází soubor `pom.xml` (parametr -DskipTests přeskočí testy, které trvají přibližně 8 minut. Zároveň se zde nacházejí testy, které neprojdou a způsobí nesestavení projektu). Samotný sestavený `jar` soubor však nestačí, program vyžaduje ke svému běhu složky: Collections, JavaClasses, TranslatedMethods a soubory: JavaMethods.java, replace.xml a settings.ini. Alternativně je možné projekt importovat do libovolného programovacího prostředí, které podporuje Maven a sestavit program z něj. Třetí možností je spustit skript `build.bat`, který provede sestavení, a přesun výsledku včetně všech potřebných souborů do nově vytvořené složky `bin` (tento skript však vyžaduje mít Maven na systémové cestě).

A.2 Spuštění

Program lze spustit dvojklikem na soubor `translator.jar`, případně příkazem `java -jar translator.jar`. Program vyžaduje nainstalovanou Javu 8 nebo novější a zároveň všechny požadované soubory a složky popsané v sestavení. Program byl otestován a funguje jak na operačních systémech Windows tak i Linux, ale na Linuxu samozřejmě nebude fungovat kompilace a spuštění programů v C#, protože tento jazyk je podporován pouze na operačním systému Windows. Dále byla na operačním systému Linux zjištěna nekompatibilita knihovny `Java diff utils`, která se při zavolání zasekla, proto podpora této funkce je na Linuxu vypnuta a vrací pouze řetězec „diff not supported“.

A.3 Hlavní okno

Obrázek A.1 ukazuje hlavní okno aplikace, které se objeví po spuštění programu.



Obrázek A.1: Hlavní okno aplikace

V horní části okna se nachází hlavní menu. Hned pod ním je možné si vybrat jazyk, se kterým se bude zrovna pracovat. Tlačítka pod tímto výběrem jazyka jsou na tomto vybraném jazyku závislá. Pod těmito tlačítky se nachází textové pole, do kterého je možné zadat cestu k souboru, který bude přesměrován jako standardní vstup při spuštění zkompilovaného programu. Pod tímto textovým polem je další (větší), do kterého je možné přímo zadat vstup, který bude přesměrován jako standardní vstup při spuštění zkompilovaného programu. V dolní části okna se nachází konzole, do které jsou vypisovány informace o průběhu parsování, překladu, kompilace, spuštění a testování. Každá z těchto kategorií má vlastní konzoli, mezi kterými lze libovolně přepínat.

Hlavní menu

- `File` → `open java file` – otevře dialogové okno na výběr zdrojového kódu k překladu (celý zdrojový kód musí být v jediném souboru).
- `File` → `exit` – ukončí program.

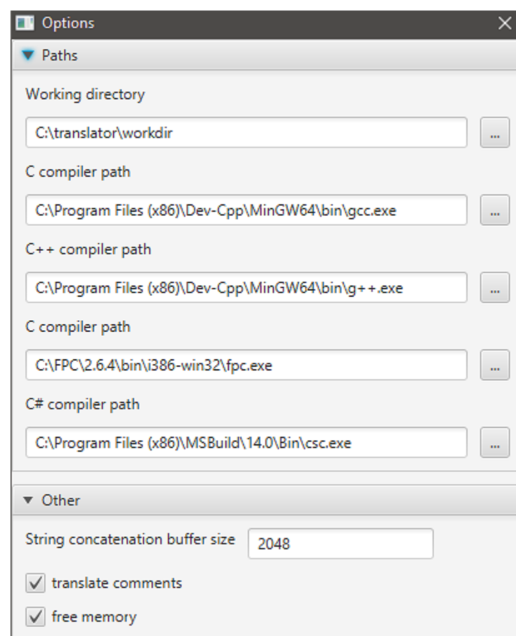
- `View` → `source code` – otevře textový editor, ve kterém je možné upravovat a podívat se na překládaný zdrojový kód a jeho překlad.
- `Settings` → `options` – otevře okno s nastavením programu.

Tlačítka

- `re-parse` – znovu zparsuje naposledy otevřený zdrojový kód Javy (pro případ, že by byl upraven).
- `translate` – přeloží naposledy zparsovaný zdrojový kód Javy do jazyka vybraného v přepínači.
- `compile` – zkompiluje naposledy přeložený kód jazyka, který je vybrán v přepínači (vyžaduje, aby byl zdrojový kód již ve vybraném jazyce přeložený a vyžaduje nastavenou cestu ke kompilátoru v nastavení programu).
- `test` – spustí testy nad naposledy zkompilovaným programem v jazyku vybraném v přepínači (vyžaduje aby v adresáři, kde se nachází překládaný zdrojový kód Javy existovaly soubory (.txt) se vstupy (obsahující ve jméně „_in-“) a očekávanými výstupy (obsahující ve jméně „_out-“)).
- `open stdin file` – otevře dialogové okno pro výběr souboru, který bude přesměrován jako standardní vstup, při spuštění programu tlačítkem `run`.
- `run` – spustí naposledy zkompilovaný program v jazyce vybraném na přepínači a přesměruje mu soubor v textovém poli vlevo od tlačítka jako standardní vstup.
- `direct stdin run` – spustí naposledy zkompilovaný program v jazyce vybraném v přepínači a přesměruje mu obsah textového pole vlevo od tlačítka jako standardní vstup.
- `Stop running process` – přeruší právě vykonávanou kompilaci programu, spuštěný program či test pro případ, že by se zasekl.
- `clear console` – vymaže obsah všech konzolí.

A.4 Nastavení

Po otevření nastavení přez hlavní menu → `Settings` → `options`, se objeví okno ukázané na Obrázku A.2.



Obrázek A.2: Okno s nastavením aplikace

V horní části okna se nachází nastavení cest. Vedle každé této cesty je tlačítko, které vyvolá dialogové okno, které umožní cestu nastavit. V dolní části jsou nastavení týkající se samotného překladu.

Cesty

- `Working directory` – složka, do které se budou ukládat přeložené programy a jejich zkompilované spustitelné soubory (nutné nastavit před překladem).
- `C compiler` – cesta ke kompilátoru jazyka C (parametry kompilátoru jsou nastavené pro gcc, při použití jiného kompilátoru nemusí kompilace fungovat).
- `C++ compiler` – cesta ke kompilátoru jazyka C++ (parametry kompilátoru jsou nastavené pro g++, při použití jiného kompilátoru nemusí kompilace fungovat).

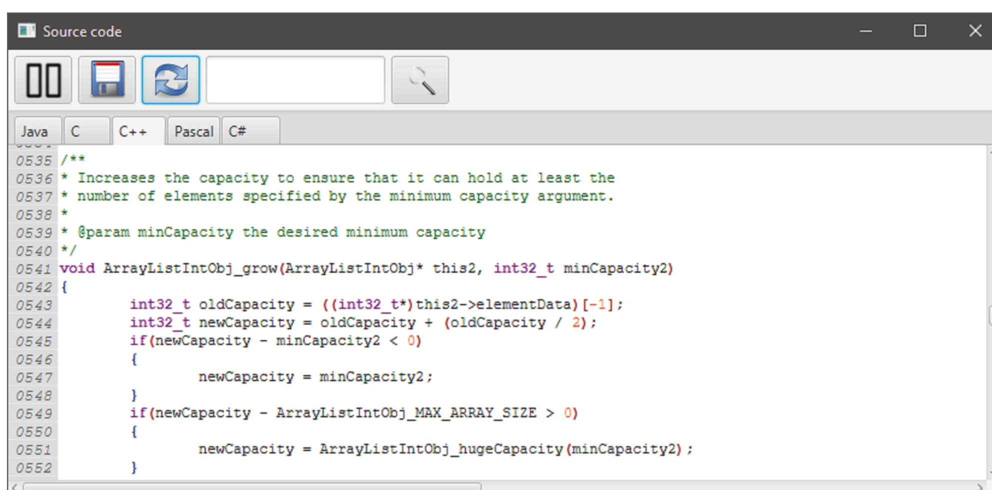
- `Free Pascal compiler` – cesta ke kompilátoru jazyka Pascal (parametry kompilátoru jsou nastavené pro fpc 2.6.4, při použití jiného kompilátoru nemusí kompilace fungovat).
- `C# compiler` – cesta ke kompilátoru jazyka C# (parametry kompilátoru jsou nastavené pro csc.exe, při použití jiného kompilátoru nemusí kompilace fungovat).

Ostatní nastavení

- `String concatenation buffer size` – nastaví velikost bufferu (počet znaků), která je použita při zřetězování řetězců v C a C++. Pokud by byl překládán program, který zřetězuje dlouhé řetězce, je potřeba velikost tohoto bufferu zvětšit.
- `translate comments` – pokud je zaškrtnuto, do přeloženého kódu se přenesou i některé (ne všechny) komentáře z původního kódu.
- `free memory` – pokud je zaškrtnuto, do jazyků C a C++ se budou generovat příkazy na uvolnění paměti. Kvůli problému popsanému v Kapitole 8.7 může být potřebné uvolňování paměti v některých případech vypnout.

A.5 Textový editor

Po otevření textového editoru přez hlavní menu → `View` → `source code`, se objeví okno ukázané na Obrákzu A.3.



Obrázek A.3: Okno textového editoru

V horní části okna se nachází panel nástrojů s tlačítky. Pod panelem nástrojů se nachází listy se zdrojovými kódy jednotlivých jazyků, mezi kterými lze libovolně přepínat.

Tlačítka

- první tlačítko – přepne rozložení okna mezi zobrazením 1 jazyka a 2 jazyků vedle sebe.
- tlačítko s disketou – uloží právě zobrazený list se zdrojovým kódem.
- tlačítko s šipkami – znovu načte obsah zobrazovaného souboru (lze použít na zobrazení kódu, který byl přeložen až po otevření textového editoru).
- tlačítko s lupou – vyhledá v právě zobrazeném zdrojovém kódu obsah textového políčka vlevo od tlačítka (na velikosti znaků nezáleží).

A.6 Přidání knihovní metody

A.6.1 Soubor replace.xml

Pokud je potřeba přidat podporu knihovní funkce do překladače, lze to udělat následujícím způsobem. Soubor `replace.xml`, který se nachází ve složce se spustitelnou aplikací, je potřeba otevřít v textovém editoru, vytvořit v něm nový záznam jako značku `item` viz Výpis A.1 a nastavit její atributy:

- name – jméno volané metody včetně přístupu třídy, ze které je volaná. Pokud se nejedná o statickou metodu, doplní se třída stejně jako kdyby statická byla (např. „String.equals(“). Závorka na konci je povinná, a naznačuje, že se jedná o metodu. Pokud se nahrazuje statickoá konstanta, závorka se nepíše.
- type – návratový typ metody či konstanty ve formátu Javy. Lze použít #TYPE#, pokud má metoda více přetížených verzí s různým návratovým typem, typ konkrétní verze se pak specifikuje ve značce `version`.
- method – nastaví se na `true`, pokud se jedná o metodu, `false` pokud se jedná o konstantu.

- throws – nastaví se na `true`, může-li metoda vyhodit výjimku v jazyku C nahrazené verze, jinak `false`. Vyhození výjimky by se provedlo nastavením proměnné `_exception_thrown` na 1 a provedením příkazu `return`.

- javamethod – nastaví se na `true` pokud bude metoda nahrazena zdrojovým kódem v jazyce Java v souboru JavaMethods.java.

- volatile – nastaví se na `true`, může-li metoda ovlivnit stav programu (např. Scanner.nextLine()), jinak `false`.

```
<item name="Math.sin(" type="double" method="true" throws="false"
        javamethod="false" volatile="false">
```

Výpis A.1: Ukázka značky item

Pokud je třeba specifikovat více přetížených verzí metody vloží se dovnitř značky `item` značka `version` a nastaví se její atribut `type`, určující návratový typ metody. Následně do značky `version` nebo `item`, pokud se specifikuje pouze jedna verze, se vloží značka `arguments` viz Výpis A.2. Pro každý argument metody se nyní vloží do značky `attributes` značka `attribute` a nastaví se její atribut `type` specifikující typ argumentu. Pokud se jedná o konstantu či metodu bez argumentů, poslední krok se přeskočí. Pokud se nahrazuje metoda pomocí implementace v Javě a je potřeba, aby proměnná, nad kterou se metoda volá, byla vložena jako argument, vloží se na příslušné místo do argumentů značka `caller`.

```
<arguments>
  <caller />
  <argument type="String"/>
</arguments>
```

Výpis A.2: Ukázka značky arguments

Pokud se nenahrazuje pomocí zdrojového kódu v Javě, tak se za značku `arguments` nyní vloží značky pro každý podporovaný jazyk, tedy: c, cpp (C++), pascal a csharp (C#) viz Výpis A.3. U každé z těchto značek se specifikují atributy:

- replacement – čím bude volání nahrazeno, to jest jméno metody či konstanty. U metody se opět vloží i počáteční závorka.

- ending – znaky následující za vyjmenováním všech argumentů (typicky ukončovací závorka).

- caller – atribut určující, co se provede s proměnnou, nad kterou byla metoda volána, pokud není statická. Možnosti: `call` = provede volání nad danou proměnnou (`var.method()`), `callwo` = opět volání nad proměnnou, ale bez přístupového operátoru (`varmethod()`, využito u `String[x]` nahrazující `String.charAt(x)`), `arg` = vloží proměnnou, nad kterou byla metoda volána jako první argument (`method(var)`), `no` = jedná se o statickou metodu či konstantu.

- anot – nepovinný atribut, specifikuje anotaci v souboru s nativním zdrojovým kódem metody, která se použije jako nahrazení. Anotace musí začínat znakem @ a její jméno musí být pro každý jazyk unikátní (na velikosti znaků záleží).

```
<c replacement="trim(" ending=")" caller="arg" anot="@trim">
```

Výpis A.3: Ukázka značky c

Pokud nahrazená metoda vyžaduje nějaký import v daném jazyku, lze vložit do jazyka značku `imports` a do ní seznam značek `import` s atributem `imp` viz Výpis A.4, ve kterém se vloží nativní kód pro import (pozor na znaky, které je potřeba v XML escapovat).

```
<imports>
  <import imp="#include &lt;math.h&gt;" />
</imports>
```

Výpis A.4: Ukázka značky imports

Pokud se nahrazuje pomocí zdrojového kódu v Javě, místo jednotlivých jazyků se specifikuje značka `annotation` a v ní atribut `anot` viz Výpis A.5, který se nastaví na jméno anotace, specifikující implementaci metody v souboru JavaMethods.java. Tato anotace musí začínat znakem @ a musí být unikátní v celém tomto souboru (na velikosti znaků záleží).

```
<annotation anot="@ArraySort" />
```

Výpis A.5: Ukázka značky annotation

Pokud byl v návratovém typu či v argumentech definován typ jako `#TYPE#`, je nutné specifikovat ve značce `replacetype` atribut `arg` na index argumentu volané metody, specifikující typ, za který bude `#TYPE#` nahrazen (např. pro Arrays.sort(array) je argument na indexu 0 ten, který určí, za co se má `#TYPE#` nahradit. S tímto typem je pak i metoda v souboru `JavaMethods.java` implementována).

```
<replacetype arg="0" />
```

Výpis A.6: Ukázka značky replacetype

Pokud je cokoliv nejasné, stačí se podívat na záznamy jiných metod a provést vyplnění obdobným způsobem.

A.6.2 Implementace metody v nativním jazyku

Pokud byl specifikován atribut `anot` u značky jazyka v souboru `replace.xml`, musí se v souboru s implementacemi těchto metod vytvořit záznam s implementací metody, kterou bude volání nahrazeno. Implementace těchto metod se nachází ve složce `TranslatedMethods` a souboru methods s koncovkou příslušnou danému jazyku. Dovnitř těchto souborů lze vložit záznam tak, že se uvede anotace, která byla specifikována v souboru `replace.xml` a anotace `@end`. Mezi tyto dvě anotace se pak vloží zdrojový kód v daném jazyku. Tento zdrojový kód nemůže volat jiné takto definované metody, protože není zaručené, že by je viděl, může však volat knihovní metody, které jsou přístupné v daném jazyku s importy specifikovanými v `replace.xml` u dané metody. Pokud definovaná metoda v jazycích C a C++ přijímá pole jako argument, lze zjistit jeho délku přetypováním pole na `int32_t*` a přístupem na index -1, pokud by metoda vracela nově vytvořené pole, je nutné na index -1 vložit jeho délku, a na index -2 počet referencí (0, zvětší se až při přiřazení návratové hodnoty). Výpis A.7 ukazuje jak vytvořit takové pole.

```
float *array = (float*)calloc(length * sizeof(float) +
    2*sizeof(int32_t), sizeof(char));
array = (float*)((char*)array + 2*sizeof(int32_t));
((int32_t*)array)[-1] = length;
```

Výpis A.7: Ukázka, jak vytvořit pole v C/C++

A.6.3 Implementace metody v Javě

Pokud byl nastaven atribut `javamethods` na `true` u metody v souboru `replace.xml`, pak je nutné aby v souboru `JavaMethods.java` byla implementace metody, která bude použita jako náhrada. Před začátkem metody musí být specifikována anotace, která byla uvedena v `replace.xml`. Za ní následuje implementace jedné metody v Javě, která bude přiložena ke zdrojovému kódu, který se překládá a bude přeložena s ním. Jakýkoliv kód, který by se nacházel za koncem této jedné metody bude ignorován. Uvnitř této metody je možné psát kód v Javě se stejnými omezeními jako kód, který se

bude překládat. Navíc je možné v této metodě použít typ `#TYPE#`, pokud bylo v souboru `replace.xml` definováno, za co se má nahradit.

A.7 Přidání knihovní třídy

Pokud je potřeba přidat podporu celé třídy do překladače, je možné to provést následujícím způsobem. Do složky `JavaClasses` se vloží třída, která se použije jako náhrada. Tato třída musí splňovat ta samá omezení jako překládaný kód Javy. Následně je potřeba do souboru `settings.ini` vložit záznam ve tvaru „jméno_třídy;“ na konec řádky začínající „javaclasses=“.

A.8 Přidání kolekce

Pokud je potřeba přidat podporu kolekce do překladače, je možné to provést následujícím způsobem. Do složky `Collections` se vloží třída, která se použije jako náhrada. Tato třída musí splňovat ta samá omezení jako překládaný kód Javy. Místo generických typů je potřeba použít `#TYPE#`, pokud je jich více tak lze za `TYPE` přidat číslo (např. `#TYPE1#`). Při použití vlastního typu kolekce se nespecifikují generické typy, ale jen jméno kolekce (tzn. ne `ArrayList<#TYPE#>`, ale jen `ArrayList`). Následně je potřeba do souboru `settings.ini` vložit záznam ve tvaru „jméno kolekce která bude nahrazená:jméno nahrazené kolekce;“ na konec řádky začínající „collections=“. Záznamů je možné vložit více, a doporučuje se vložit jak nahrazení rodičovského typu tak i konkrétního (např. `List` i `ArrayList`).

A.9 Rezervované symboly

A.9.1 Proměnné a metody

Přestože jsou do tabulky symbolů před překladem do každého jazyka přidána klíčová slova tohoto jazyka, může se stát, že nastane kolize nějakého symbolu s nativní knihovní metodu, proměnnou či konstantou. Proto je doporučené dát si pozor na jména proměnných, která by mohla být v cílových jazycích rezervována. U jmen metod by neměl nastat problém, protože jsou vždy členem nějaké třídy, nebo mají přidaný prefix se jménem třídy a podtržítkem.

A.9.2 Třídy

Z implementačních důvodů není možné použít některá jména uživatelem definovaných tříd. Jedná se o jména knihovních tříd jazyka Java a uměle generovaných tříd.

A.9.3 Seznam rezervovaných jmen tříd

- `List, ArrayList, LinkedList, Set, HashSet, Map, HashMap`.
- `RequiredMethods, JavaMethods, Scanner, StringTokenizer, Arrays`.
- `BoolObj, ByteObj, DoubleObj, FloatObj, CharObj, IntObj, LongObj, ShortObj`.
- `String, Character, Exception, Math, Boolean, Byte, Double, Float, Integer, Long, Short`.
- Také by teoreticky mohly nastat kolize s jinými datovými typy v jazycích, do kterých překládáme.

A.10 Seznam podporovaných metod a konstant

- `Exception.getMessage()` – pozn. nevrací stejný řetězec jako v javě
- `Exception.printStackTrace()` – pozn. netiskne stejný řetězec jako v javě
- `Math.E`
- `Math.PI`
- `Math.round(float/double)`
- `Math.toDegrees(double)`
- `Math.sin(double)`
- `Math.cos(double)`
- `Math.tan(double)`
- `Math.atan(double)`

- `Math.atan2(double, double)`
- `Math.sinh(double)`
- `Math.cosh(double)`
- `Math.tanh(double)`
- `Math.exp(double)`
- `Math.pow(double, double)`
- `Math.sqrt(double)`
- `Math.log(double)`
- `Math.log10(double)`
- `Math.asin(double)`
- `Math.acos(double)`
- `Math.ceil(double)`
- `Math.floor(double)`
- `Math.abs(int/long/float/double)`
- `Math.min(any number type)`
- `Math.max(any number type)`
- `Byte.parseByte(String)` – pozn. nemusí vyhodit výjimku
- `Byte.valueOf(String)` – pozn. nemusí vyhodit výjimku
- `Short.parseShort(String)` – pozn. nemusí vyhodit výjimku
- `Short.valueOf(String)` – pozn. nemusí vyhodit výjimku
- `Integer.parseInt(String)` – pozn. nemusí vyhodit výjimku
- `Integer.valueOf(String)` – pozn. nemusí vyhodit výjimku
- `Long.parseLong(String)` – pozn. nemusí vyhodit výjimku
- `Long.valueOf(String)` – pozn. nemusí vyhodit výjimku
- `Float.parseFloat(String)` – pozn. nemusí vyhodit výjimku

- `Float.valueOf(String)` – pozn. nemusí vyhodit výjimku
- `Double.parseDouble(String)` – pozn. nemusí vyhodit výjimku
- `Double.valueOf(String)` – pozn. nemusí vyhodit výjimku
- `System.exit(int)`
- `System.out.print(any type)`
- `System.out.println()`
- `System.out.println(any type)`
- `System.out.format(String, ...)`
- `System.in`
- `Locale.setDefault(Locale)`
- `Locale.US`
- `Scanner.useLocale(Locale)`
- `Scanner.close()`
- `Scanner.nextLine()`
- `Scanner.next()` – problémy v Pascalu, nedoporučuje se používat
- `Scanner.nextByte()`
- `Scanner.nextShort()`
- `Scanner.nextInt()`
- `Scanner.nextLong()`
- `Scanner.nextFloat()`
- `Scanner.nextDouble()`
- `String.length()`
- `String.replace(char, char)`
- `String.replace(String, String)`
- `String.charAt(int)`

- `String.trim()`
- `String.substring(int)`
- `String.substring(int, int)`
- `String.indexOf(char)`
- `String.indexOf(String)`
- `String.indexOf(char, int)`
- `String.indexOf(String, int)`
- `String.lastIndexOf(char)`
- `String.lastIndexOf(String)`
- `String.lastIndexOf(char, int)`
- `String.lastIndexOf(String, int)`
- `String.toCharArray()`
- `String.toLowerCase()`
- `String.toUpperCase()`
- `String.split(String)` – pozn. nefunguje s regulárními výrazy
- `String.equalsIgnoreCase(String)`
- `String.isEmpty()`
- `String.hashCode()`
- `String.concat(String)`
- `String.getBytes()`
- `String.getChars(int, int, char [ ], int)`
- `String.compareToIgnoreCase(String)`
- `String.contains(String)`
- `String.endsWith(String)`
- `String.startsWith(String)`

- `String.startsWith(String, int)`
- `new String(char [ ])`
- `Character.isLetter(char)`
- `Character.isDigit(char)`
- `Arrays.toString(any type [ ])`
- `Arrays.fill(any type [ ], any type)`
- `Arrays.fill(any type [ ], int, int, any type)`
- `Arrays.copyOf(any type [ ], int)`
- `Arrays.sort(any type [ ])`

A.11 Seznam podporovaných tříd a kolekcí

Následuje seznam podporovaných tříd a kolekcí. Zmíněné třídy nemají implementovány úplně všechny metody.

- `Boolean`
- `Byte`
- `Short`
- `Integer`
- `Long`
- `Float` – pozn. hashCode() a compareTo() se nemusí chovat stejně jako v Javě
- `Double` – pozn. hashCode() a compareTo() se nemusí chovat stejně jako v Javě
- `StringTokenizer` – pozn. nefunguje s regulárními výrazy
- `List, LinkedList, ArrayList` – pozn. LinkedList se přeloží jako ArrayList
- `Set, HashSet`
- `Map, HashMap`
- `Entry, MapNode` – pozn. využívané kolekcí HashMap